Sistemas Operativos

1era Parte

Moreira Zambrano, Cesar Armando
Zambrano Romero, Walter Daniel
Mero García, Alejandro Fabián
Chauca Chicaiza, José Luis
Mendoza Briones, Duglas Antonio
Cotera Ramírez, Gabriel Agustín
López Navarrete, Maryuri Lisbeth
Pita Valencia, Josselyn Stefania
Martínez Falcones, Víctor Alfonzo

© Autores

Cesar Armando Moreira-Zambrano
Docente de la Universidad Técnica de Manabí, Portoviejo, Manabí, Ecuador.

Walter Daniel Zambrano-Romero
Docente de la Universidad Técnica de Manabí, Portoviejo, Manabí, Ecuador.

Alejandro Fabián Mero-García
Docente de la Universidad Técnica de Manabí, Portoviejo, Manabí, Ecuador.

José Luis Chauca-Chicaiza
Docente de la Universidad Técnica de Manabí, Portoviejo, Manabí, Ecuador.

Duglas Antonio Mendoza-Briones
Docente de la Universidad Técnica de Manabí, Portoviejo, Manabí, Ecuador.

Gabriel Agustín Cotera-Ramírez
Docente de la Universidad Técnica de Manabí, Portoviejo, Manabí, Ecuador.

Maryuri Lisbeth López-Navarrete
Docente de la Universidad Técnica de Manabí, Portoviejo, Manabí, Ecuador.

Josselyn Stefania Pita-Valencia
Docente de la Universidad Técnica de Manabí, Portoviejo, Manabí, Ecuador.

Víctor Alfonzo Martínez-Falcones
Docente de la Universidad Técnica de Manabí, Portoviejo, Manabí, Ecuador.

Casa Editora del Polo - CASEDELPO CIA. LTDA.
Departamento de Edición

Editado y distribuido por:

Editorial: Casa Editora del Polo
Sello Editorial: 978-9942-816
Manta, Manabí, Ecuador. 2019
Teléfono: (05) 6051775 / 0991871420
Web: www.casedelpo.com
ISBN: 978-9942-816-88-7

Impreso en Ecuador

Revisión, Ortografía y Redacción:
Lic. Jessica Mero Vélez

Diseño de Portada:
Michael Josué Suárez-Espinar

Diagramación:
Ing. Edwin Alejandro Delgado-Veliz

Director Editorial:
Dra. Tibisay Milene Lamus-García

Comité Científico Académico

Constancia de Arbitraje

La Casa Editora del Polo, hace constar que este libro proviene de una investigación realizada por los autores, siendo sometido a un arbitraje bajo el sistema de doble ciego (peer review), de contenido y forma por jurados especialistas. Además, se realizó una revisión del enfoque, paradigma y método investigativo; desde la matriz epistémica asumida por los autores, aplicándose las normas APA, Sexta Edición, proceso de anti plagio en línea Plagiarisma, garantizándose así la cientificidad de la obra.

Comité Editorial

CONTENIDO

PRÓLOGO

Los sistemas operativos son una parte esencial de cualquier sistema informático. Del mismo modo, un curso sobre sistemas operativos es una parte esencial de cualquier carrera de Informática. Este campo está cambiando muy rápidamente, ya que ahora las computadoras se encuentran prácticamente en cualquier aplicación, desde juegos para niños hasta herramientas de planificación extremadamente sofisticadas para los gobiernos y las grandes multinacionales.

Sin embargo, los conceptos fundamentales siguen siendo bastante claros y es en ellos en los que se basa este libro. Hemos escrito esta obra como libro de texto para un curso de introducción a los sistemas operativos para estudiantes universitarios de primer y segundo ciclo. Esperamos asimismo que los profesionales también lo encuentren útil, ya que proporciona una clara descripción de los conceptos que subyacen a los sistemas operativos. Como prerrequisitos, suponemos que el lector está familiarizado con las estructuras de datos básicas, la organización de una computadora y algún lenguaje de alto nivel, como por ejemplo C.

En los capítulos se han incluido los conceptos necesarios para poder comprender los sistemas operativos. Los conceptos se presentan mediante descripciones intuitivas. El libro aborda algunos resultados teóricos importantes. Se han utilizado figuras y ejemplos para indicar por qué debemos esperar que el resultado en cuestión sea cierto. Los conceptos y algoritmos fundamentales cubiertos en el libro están basados, a menudo, en aquéllos que se emplean en los sistemas operativos comerciales existentes. Nuestro objetivo ha sido presentar estos conceptos y algoritmos para una configuración general que no estuviera ligada a un sistema operativo concreto. Hemos presentado gran cantidad de ejemplos que pertenecen a los más populares e innovadores sistemas operativos,

INTRODUCCIÓN

No existe una definición única de sistema operativo. Los sistemas operativos existen porque son una vía razonable para resolver los problemas que crea un sistema informático. El hardware por sí solo no es fácil de utilizar. Es necesario ayudar tanto al programador como al usuario a abstraerse de la complejidad del hardware. La forma de hacerlo es colocando una capa de software, por encima del hardware con el fin de presentar al usuario del sistema y a las aplicaciones una interfaz de máquina virtual que facilite la comprensión y utilización del sistema. Esta capa de software es lo que se denomina sistema operativo. El sistema operativo integra un conjunto de funciones responsables de controlar el hardware que son comunes a la mayoría de las aplicaciones, como las funciones de control de los dispositivos y las rutinas de servicio a interrupciones, ocultando al programador los detalles del hardware y ofreciéndole una interfaz cómoda para utilizar el sistema.

Los sistemas operativos, también llamados núcleos o kernels, suelen ejecutarse de manera privilegiada respecto al resto del software, sin permitir que un programa cualquiera realice cambios de importancia sobre él que puedan comprometer su funcionamiento. El sistema operativo es el protocolo básico de operatividad del computador, que coordina todas sus demás funciones de comunicaciones, de procesamiento, de interfaz con el usuario.

Los sistemas operativos consisten en interfaces gráficas, entornos de escritorio o gestores de ventanas que brindan al usuario una representación gráfica de los procesos en marcha. También puede ser una línea de comandos, es decir, un conjunto de instrucciones ordenado según su prioridad y que funciona en base a órdenes introducidas por el usuario.

Las primeras versiones de las computadoras no tenían sistemas operativos. En la década de los sesenta los ordenadores usaban

procesamientos por lotes y fue durante estos años cuando comenzaron a desarrollarse los sistemas operativos.

El en presente libro, está estructurado en cuatro capítulos, el primero de ellos describe la estructura de los sistemas operativos, conceptos, funciones, componentes, funciones. El segundo capítulo se centra describe los procesos e hilos, el estado de los procesos, procesos ID, la multiprogramación. El capítulo tres se refiere a la Concurrencia, exclusión mutua y sincronización, problemas, condiciones de carrera y el capítulo cuatro Gestión de memoria y memoria virtual.

Se busca que los lectores puedan reconocer los sistemas operativos como aquellos que permiten que otros programas puedan utilizarlos de apoyo para poder funcionar. Por eso, a partir del sistema utilizado pueden ser instalados ciertos programas y otros no.

Son parte esencial del funcionamiento de los sistemas informáticos y la pieza de software central en la cadena de procesos, ya que establecen las condiciones mínimas para que todo funcione: la administración de los recursos, el método de comunicación con el usuario y con otros sistemas, las aplicaciones adicionales.

CAPITULO I

ESTRUCTURA DE LOS SISTEMAS OPERATIVOS

Introducción

En la sociedad actual, altamente automatizada, las computadoras u ordenadores desempeñan una función vital a través de los cuales se desenvuelve la dinámica cotidiana de los ciudadanos y las actividades de las múltiples organizaciones a escala mundial y en este entramado, sin duda alguna, los sistemas operativos constituyen el elemento clave para el funcionamiento del ordenador, pues es el encargado de que cualquier software pueda hacer uso del hardware necesario para la ejecución del equipo. En tal sentido, (Villarreal, 2017) destaca "un sistema operativo es un programa que controla la ejecución de los programas de aplicación y que actúa como interfaz entre el usuario de un computador y el hardware de la misma (p.7).

Un sistema operativo es un software que administra el hardware de un computador. Además de proveer un conjunto de aplicaciones básicas que actúan como un intermediario entre el usuario y el hardware de la computadora. Un aspecto interesante de los sistemas operativos es que se encuentran en todas partes, desde un automóvil, y equipos que incluyen dispositivos de "Internet de las Cosas (IoT), hasta teléfonos inteligentes, computadores personales, entornos de informática de la nube, etc., y el cómo cumplen sus tareas en esos entornos tan diversos.

Para entender el papel de los sistemas operativos en un entorno informático actual, es esencial que primero se comprenda la organización y arquitectura del hardware computacional, esta incluye procesador (CPU Unidad de Procesamiento Central), memorias, dispositivos de E/S y almacenamiento. Donde la principal responsabilidad de un sistema operativo es asignar esos recursos a los programas.

Debido a la complejidad de un sistema operativo, este debe ser creado pieza por pieza, donde cada una de estas piezas debe ser diseñada cuidadosamente con entradas, salidas y funciones bien definidas.

A eta razón, a lo largo de este compendio se revisa de forma general los componentes principales de un sistema informático actual, así como las funciones que aporta el sistema operativo.

1.1.- Visión General del Sistema Operativo

El sistema operativo es el único programa que interactúa directamente con el hardware de la computadora, y sus funciones principales, según (Wolf, Ruiz, Bergero, & Meza, 2015) son:

Figura 1. Funciones Primarias del Sistema Operativo

Fuente: (Wolf, Ruiz, Bergero, & Meza, 2015)

Abstracción: El sistema operativo se encarga de proporcionar una serie de abstracciones para que los programadores puedan enfocarse en resolver las necesidades particulares de sus usuarios (Wolf, Ruiz, Bergero, & Meza, 2015). Como forma de ilustrar lo expresado, estos autores indican como ejemplo de tales abstracciones es que la información está organizada en archivos y directorios (en uno o muchos dispositivos de almacenamiento).

Administración de Recursos: Un sistema de cómputo puede tener a su disposición una gran cantidad de recursos (memoria, espacio de almacenamiento, tiempo de procesamiento, etc.), y los diferentes procesos

que se ejecuten en él compiten por ellos (Wolf, Ruiz, Bergero, & Meza, 2015). Al gestionar toda la asignación de recursos, el sistema operativo puede implementar políticas que los asignen de forma efectiva y acorde a las necesidades establecidas para dicho sistema (Wolf, Ruiz, Bergero, & Meza, 2015).

Aislamiento: En un sistema multiusuario y multitarea cada proceso y cada usuario no tendrá que preocuparse por otros que estén usando el mismo sistema. Idealmente, su experiencia será la misma que si el sistema estuviera exclusivamente dedicado a su atención (aunque fuera un sistema menos poderoso) (Wolf, Ruiz, Bergero, & Meza, 2015). De acuerdo con los citados autores, para implementar correctamente las funciones de aislamiento hace falta que el sistema operativo utilice hardware específico para dicha protección.

Hardware, es toda la parte física del computador, quiere decir aquello que se puede ver y tocar, ej.: procesador, tarjeta madre, tarjetas de sonido, vídeo, memorias, discos duro, etc. (Ocampo & Sánchez, 2015)

1.2.- Objetivos y Funciones de los Sistemas Operativos

Los sistemas informáticos en general, según considera (Fernández Fernández, 2015) constituyen una simbiosis de recursos de hardware (CPU, memorias y periféricos) y de software (programas que se ejecutan en la CPU) y, dentro de los sistemas informáticos los sistemas operativos tienen un papel vital, debido a que son la estructura que soporta y maneja todos los programas y partes de la computadora u ordenador.

Un sistema informático puede ser dividido en aproximadamente cuatro componentes: hardware, sistema operativo, programas de aplicación y usuario.

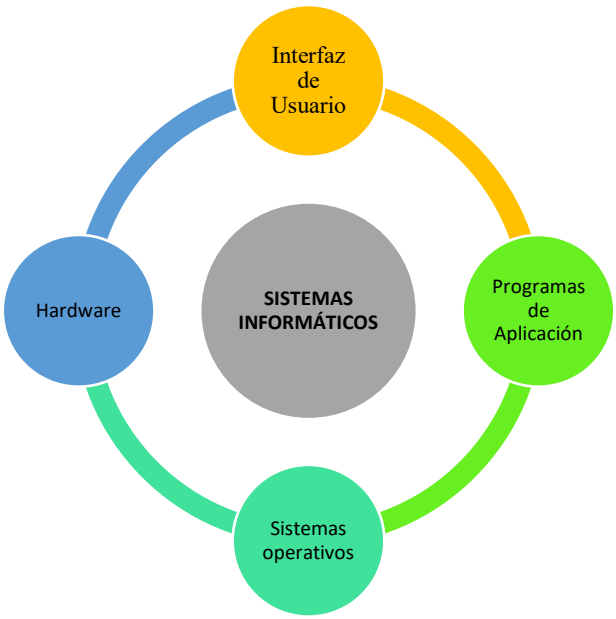


Figura 2. Componentes de un Sistema Informático

Fuente: (Chacón, 2008)

El sistema operativo: Es un conjunto de programas que constituyen la "inteligencia básica" del ordenador. También se le conoce con el nombre de software del sistema (Chacón, 2008). El sistema operativo de un ordenador tiene dos objetivos fundamentales. El primero es, convertir el conjunto de dispositivos interconectados en un ordenador capaz de dialogar con el mundo exterior, con el usuario. El segundo de los objetivos es gestionar de la forma más eficaz los recursos físicos del ordenador (Chacón, 2008). El sistema operativo, controla el hardware y coordina su uso entre los varios programas de aplicación para varios usuarios.

El sistema operativo es el software que controla la operación general de una computadora, proporciona los medios por los que un usuario puede almacenar y recuperar archivos, provee la interfaz por la que un usuario

puede solicitar la ejecución de programas y provee el ambiente necesario para que los programas solicitados se ejecuten (Glenn Brookshear, 2007).

Hardware: En informática hardware es el conjunto de elementos físicos, ya sean eléctricos, electrónicos, mecánicos o magnéticos que integran un ordenador (Chacón, 2008). Se refiere, por tanto, a los componentes materiales de un sistema informático. La función de estos componentes puede dividirse en tres categorías principales: Entrada, Salida y Almacenamiento (Chacón, 2008). Los componentes de esas categorías están conectados a través de un conjunto de cables o circuitos llamado bus con la unidad central de proceso (CPU) del ordenador (Chacón, 2008).

Así, el hardware, está compuesto por la unidad de procesamiento central (CPU), las memorias, y los dispositivos de entrada y salida (E/S), los cuales proveen los recursos informáticos esenciales para el sistema.

Interfaz de usuario: este componente permite la interacción con el usuario, lo que puede ocurrir a través de iconos gráficos y un escritorio o mediante una línea de comandos (IONOS, 2017).

El programa de interfaz de usuario, shell o GUI, es el nivel más bajo del software en modo usuario y permite la ejecución de otros programas, como un navegador Web, lector de correo electrónico o reproductor de música. Estos programas también utilizan en forma intensiva el sistema operativo (Villarreal, 2017).

Programas de Aplicación: Forman el software de aplicaciones los programas que realizan tareas concretas para los usuarios: procesadores de textos y de documentos, hojas de cálculo, navegadores web, reproductores multimedia, gestores de bases de datos, etc (Fernández Fernández, 2015).

Los programas de aplicación son los recursos utilizados para resolver los problemas informáticos de los usuarios, estos pueden ser los procesadores

de palabras, las hojas de cálculo, navegadores web, juegos, etc., los actuales además definen cual es el uso principal de sistema informático.

Estos programas de aplicaciones se apoyan en otros cuya función no es ya una tarea concreta, sino proporcionar servicios generales. A estos otros se les llama «software de sistemas» (o «software del sistema») (Fernández Fernández, 2015).

En suma, también se puede considerar que un sistema informático consta de hardware, software y datos. Entonces el sistema operativo provee los medios para el uso apropiado de estos recursos en el funcionamiento del sistema informático. Por lo cual un sistema operativo cumple funciones similares a la de un administrador, porque al igual que este, no realiza funciones por si mismo, sino que provee un entorno en el cual otros programas pueden realizar su trabajo

1.3.- Sistema Operativo Desde las Perspectivas del Usuario y del Sistema

Para dar una idea más clara del papel que tiene un sistema operativo, a continuación, se presenta un análisis desde las perspectivas del usuario y del sistema.

a.- Perspectiva del usuario

Desde la perspectiva del usuario, la vista del computador varía según la interfaz que se utilice y el equipo que se utilice, sea este un computador portátil o de escritorio, el cual está diseñado para que sólo un usuario esté en control total del dispositivo y sus recursos. De ahí que su objetivo es maximizar el uso que le esté dando el usuario, sea este para trabajar o jugar. Para el caso, el sistema operativo está diseñado principalmente para facilitar su uso, prestando cierta atención al rendimiento y la seguridad y ninguna a la utilización de recursos de cómo se comparten los diversos recursos de hardware y software.

Para los usuarios finales la máquina tiene que estar dotada, junto con el sistema operativo, de una interfaz gráfica, de algunas utilidades (como un programa de respaldo) y, por supuesto, de los programas de aplicación

necesarios para este tipo de usuarios. Es, por tanto, frecuente que bajo el nombre de «sistema operativo» se incluyan estos programas (Fernández Fernández, 2015).

En la actualidad el incremento en el uso de dispositivos móviles como los teléfonos inteligentes, las tabletas, las consolas de videojuegos, etc., por parte de los usuarios, los cuales están reemplazando a los sistemas informáticos de escritorio y portátiles. Dispositivos que están continuamente conectados a redes celulares o de tecnologías inalámbricas. Cuya interfaz de usuario es típicamente una pantalla táctil, donde el usuario interactúa con el sistema por presionar y deslizar los dedos a través de la pantalla y rara vez utilizan un teclado o un ratón físico, adicionalmente estos dispositivos permiten al usuario interactuar con ellos a través del reconocimiento de voz como Siri de Apple, Cortana de Microsoft, Alexa de Amazon, y el Asistente de Google.

De igual forma muchos sistemas pueden o no tener pequeñas interfaces de usuarios como por ejemplo los computadores embebidos de los dispositivos del hogar o automóviles, los cuales muchas veces son solo un teclado con pocos números (ejemplo el control de Chevystar), los cuales al ser activados pueden encender un indicador luminoso o emitir un sonido, para indicar su estado, donde sus aplicaciones y sistemas operativos, están diseñados para ejecutarse sin que el usuario intervenga.

De esta forma, la mayoría de las computadoras tienen dos modos de operación: modo kernel y modo usuario. El sistema operativo es la pieza fundamental del software y se ejecuta en modo kernel (también conocido como modo supervisor). En este modo, el sistema operativo tiene acceso completo a todo el hardware y puede ejecutar cualquier instrucción que la máquina sea capaz de ejecutar. El resto del software se ejecuta en modo usuario, en el cual sólo un subconjunto de las instrucciones de máquina es permitido (Villarreal, 2017).

b.- Perspectiva del Sistema

Desde la perspectiva del sistema, el sistema operativo es el programa (porque es un programa) que está más íntimamente involucrado con el hardware. En este contexto el sistema operativo actúa como un gestor de recursos, ya que un sistema informático consta de muchos recursos los cuales pueden ser requeridos para resolver un problema: tiempos del CPU, espacio de memoria, espacio de almacenamiento, dispositivos de E/S, y demás, por lo cual el sistema operativo actúa como un administrador, gestionando esos recursos, al hacer frente a numerosas y posibles solicitudes conflictivas de recursos, el sistema operativo debe decidir qué hacer y como asignarles programas y usuarios específicos para que el sistema informático pueda operar de manera eficiente y correcta.

Según el Diccionario de la Lengua Española (DRAE), un sistema operativo es un programa o conjunto de programas que realizan funciones básicas y permiten el desarrollo de otros programas (Fernández Fernández, 2015). En una definición más completa y técnica el Free On-Line Dictionary Of Computing (FOLDOC) indica que un sistema operativo es un software de bajo nivel que gestiona la interfaz con el hardware periférico, planifica tareas, asigna almacenamiento y presenta al usuario una interfaz por defecto cuando no se está ejecutando ningún programa de aplicación (Fernández Fernández, 2015).

Una vista ligeramente diferente de un sistema operativo enfatiza la necesidad de controlar los diversos dispositivos de E/S y programas de usuario. Un sistema operativo es un programa de control, el cual gestiona la ejecución de los programas del usuario para evitar errores y un uso inadecuado de la computadora, además se ocupa especialmente del funcionamiento y control de dispositivos de E/S.

1.4.- Concepciones de Sistema Operativo

Hasta ahora se puede apreciar que se han adelantado algunas concepciones sobre el termino sistema operativo A continuación se definen

algunos otros conceptos, abordado desde varios autores, en primera instancia, (Ocampo & Sánchez, 2015) considera que "Es el programa o conjunto de programas que efectúan la gestión de los procesos básicos de un sistema informático, y permite la normal ejecución del resto de las operaciones" (p.13).

En segunda instancia, en el documento de la Biblioteca de la Universidad de Alicante (BUA, 2015), se indica que el sistema operativo es el software (programa o conjunto de programas) que en un sistema informático gestiona los recursos de la máquina y provee servicios básicos a los programas de aplicación. El sistema operativo siempre se ejecuta en modo privilegiado (p.4).

Asimismo, (Jaramillo, 2015) subraya "El sistema operativo es el principal programa que se ejecuta en toda computadora de propósito general. Los hay de todo tipo, desde muy simples hasta terriblemente complejos, y entre más casos de uso hay para el cómputo en la vida diaria, más variedad habrá en ellos (p. 5).

Por otro lado, (Roa, 2017) manifiesta que un Sistema Operativo (SO) se presenta como un intermediario entre un sistema de cómputo y el usuario, cuya tarea es suministrar un buen servicio a estos y alcanzar un uso eficiente de este (p.7). Por su parte, (Candela, García, Quesada, Santana, & Santos, 2007), sostienen que "Sistema Operativo es un programa que actúa como intermediario entre el usuario y el hardware de un sistema de cómputo. El propósito de un Sistema Operativo es ofrecer un ambiente en el que el usuario pueda ejecutar programas de una forma cómoda y eficiente".

Así también, (O´Brien, 2006), refiere que: "un sistema operativo es un software de sistema, es decir, un conjunto de programas de computadora destinado a permitir una administración eficaz de sus recursos. Comienza a trabajar cuando se enciende el computador, y gestiona el hardware de

la máquina desde los niveles más básicos, permitiendo también la interacción con el usuario"

Lo señalado conduce a la conclusión de que el sistema operativo es la pieza nuclear de un sistema informático para el procesamiento de la información y la obtención de los resultados.

1.5.- Evolución de los Sistemas Informáticos

El termino sistema operativo abarca muchas funciones y roles, lo cual es en parte, y esto, debido al gran número de diseños y usos de las computadoras, ya que estas se encuentran presentes en tostadoras, cafeteras, automóviles, barcos, aviones, satélites, naves espaciales, hogares y empresas. De hecho, son la base de las consolas de videojuegos, sintonizadores de canales, sistemas de control industrial, etc.

Para explicar esta diversidad, de debe regresar en la historia de las computadoras, pues a pesar de tener una muy corta historia, estas se han desarrollado muy rápido, de hecho, la informática comenzó como un experimento para determinar lo que se podía hacer y rápidamente se trasladó a sistemas de propósito específicos en área de la milicia, descifrando códigos, trazando trayectorias de misiles, y el uso gubernamental para realizar censos, luego se volvió de uso general, en grandes computadores llamadas mainframes que eran multifuncionales, y es ahí donde nace el sistema operativo.

Tabla 1. Evolución Histórica de los Sistemas Operativos

Década de 1940	Década de 1950
Sale la primera generación de computadoras, se accedía directamente a la consola de la computadora desde la cual se actuaba sobre una serie de micro interruptores que permitían	Con el objeto de facilitar la interacción de persona y computadora, los sistemas operativos hacen una aparición discreta y bastante simple, con conceptos tales como monitor

introducir directamente el programa en la memoria de la computadora	residente, el proceso por lotes y el almacenamiento temporal
Década de 1960	**Década de 1970**
Se producen varios cambios notorios en el campo de la informática, con la aparición del circuito integrado la mayoría orientados a seguir incrementando el potencial de los ordenadores. Para ello se utilizaban técnicas de lo más diversas	BDOS (Basic Disk Operating System): traductor de las instrucciones en llamados a la BIOS. Surgió a raíz de Multics a principios de la década de 1970.
Década de 1980	**Década de 1990**
En el año 1981, nace el Sistemas Operativos MS-DOS (Micro Soft Disk Operating System), creado por Microsoft para IBM PC. Desde esta fecha iniciaron con las diferentes generaciones, desde la versión PC DOS 1.0. En el año 1984, nace el Sistema Operativo Mac OS cuya característica principal era una GUI (Graphic User Interface), multitareas y mouse; a finales de la década, para el año 1987 surge el Sistema Operativo OS/2 de IBM que intentó reemplazar a DOS como Sistema Operativo.	Diversas versiones de Windows. En el año 1990, nacen dos grandes Sistemas operativos, SunOS y BeOS. En 1992, sale al mercado el Sistema Operativo Solaris de tipo Unix. En el año 1993 nace Windows NT, el cual pertenece a la familia de sistemas operativos producidos por Microsoft, de aquí en adelante lanzan las diferentes versiones de Windows: Windows 98 (1998), Windows 10 (Beta), Windows Server (2000),Windows XP (2001), Windows Vista (2007), Windows 7 (2009) y Windows 8.x (2012)
Década de 2000	**Década de 2010-2018**

En el año 2001, nace el Sistema Operativo MAC OS X, el cual está basado en el entorno operativo Unix, este SO es desarrollado, comercializado y vendido por Apple Inc. Durante los años 2005 – 2010, emerge OpenSolaris como Sistema Operativo libre, a partir de la versión privativa de Solaris de Sun Microsystems, a partir de esta generación han nacido otras versiones como IlluOS, OpenIndiana.	El mundo de los celulares abarca a la sociedad con diversos sistemas operativos móviles compitiendo por ser el mejor en el mercado. En 2012 sale Windows 8. El cual promete una mejor experiencia con el usuario. En el año 2015 nace Windows 10. Esta versión del sistema operativo, se caracteriza por ir lanzando actualizaciones masivas periódicamente, añadiendo mejoras y nuevas funciones. Es un sistema operativo maduro, y entre sus características destacadas podemos citar la vuelta del menú de inicio, el nuevo navegador Microsoft Edge, el asistente de voz Cortana, o la sincronización con la nube (Redondo, 2021). De Windows 10, se dio a conocer su último update, Windows 10 April 2018 Update (Carmona, 2018)

Fuente: (Roa, 2017)

Hoy, en los sistemas operativos para dispositivos móviles, se observa que una vez más aumenta el número de funciones que constituyen el sistema operativo. Los sistemas operativos móviles a menudo incluyen no solo un núcleo central, sino también middleware, un conjunto de software frameworks que brindan servicios adicionales a los desarrolladores de aplicaciones. Por ejemplo, cada uno de los dos sistemas operativos móviles más destacados, iOS de Apple y Android de Google, presenta un núcleo

central junto con middleware que admite bases de datos, multimedia y gráficos (por nombrar solo algunos).

El sistema operativo incluye el núcleo siempre en ejecución, middleware frameworks que facilitan el desarrollo de aplicaciones y proporcionan funciones y programas del sistema que ayudan a administrar el sistema mientras está en ejecución. La mayor parte de este texto se ocupa del núcleo de los sistemas operativos de propósito general, pero se analizan otros componentes según sea necesario para explicar completamente el diseño y la operación del sistema operativo.

1.6.- Funciones de los Sistemas Operativos

Un **middleware** es un software **framework** ejecutándose entre una aplicación y otra plataforma. Un middleware puede proveer una abstracción de los sistemas para el desarrollador de aplicaciones de tal forma que pueda desarrollar una aplicación fácilmente (Castellanos, 2018)

La función principal de un Sistema Operativo es admitir la ejecución de los programas del usuario para asegurar su conveniencia y el uso eficiente de los recursos (Roa, 2017).

Un sistema operativo gestiona: procesos, memoria, sistemas de archivos o ficheros, almacenamiento masivo, caché y dispositivos de E/S. De acuerdo con (BUA, 2015) un proceso es, sencillamente, un programa en ejecución que necesita una serie de recursos para realizar su tarea: tiempo de CPU (Central Process Unit o Unidad de Proceso Central, es decir, el procesador principal del ordenador), memoria, archivos y dispositivos de E/S (entrada/salida) (p.5).

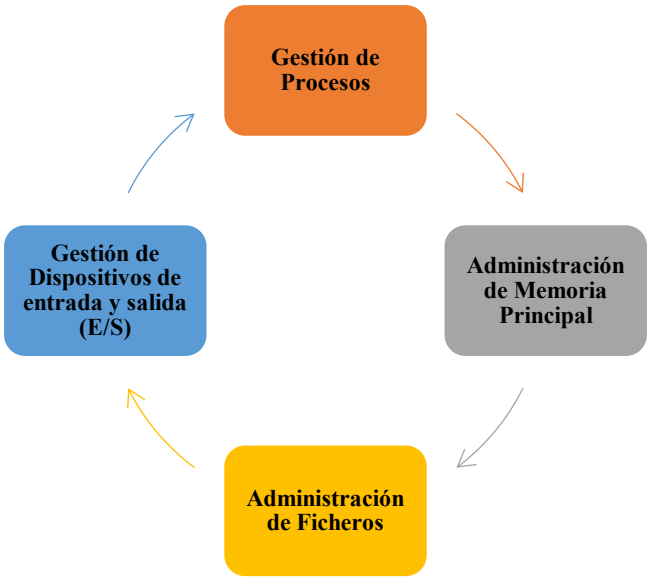


Ilustración 3. Componentes de un Sistema Operativo

Fuente: (Roa, 2017)

a. Gestión de Procesos

Un proceso es, sencillamente, un programa en ejecución que necesita una serie de recursos para realizar su tarea: tiempo de CPU (Central Process Unit o Unidad de Proceso Central, es decir, el procesador principal del ordenador), memoria, archivos y dispositivos de E/S (entrada/salida) (Villarreal, 2017).

Un programa en ejecución es un proceso, sea este un juego, un compilador, un navegador web, etc., mientras se esté ejecutando es un proceso, así como una aplicación de un dispositivo móvil, cuando se está ejecutando, esta es un proceso. Por ahora, se puede considerar un proceso como una instancia de un programa en ejecución, pero más adelante se revisará nuevamente este concepto y se verá que este es más general. Como se describe en la siguiente unidad, es posible proporcionar llamadas al sistema que permitan a los procesos crear subprocesos para ejecutarse simultáneamente.

Un proceso necesita ciertos recursos, incluido el tiempo de CPU, la memoria, los archivos y los dispositivos de E/S, para realizar su tarea. Estos recursos generalmente se asignan al proceso mientras se ejecuta. Además de los diversos recursos físicos y lógicos que obtiene un proceso cuando se crea, se pueden pasar varios datos de inicialización (entrada). Por ejemplo, considere un proceso que ejecuta un navegador web cuya función es mostrar el contenido de una página web en una pantalla. El proceso recibirá la URL como entrada y ejecutará las instrucciones apropiadas y las llamadas al sistema para obtener y mostrar la información deseada en la pantalla. Cuando finaliza el proceso, el sistema operativo recuperará los recursos reutilizables.

Un programa es una entidad pasiva, como el contenido de un archivo almacenado en el disco, mientras que un proceso es una entidad activa. Un proceso de un solo subproceso tiene un contador de programa que especifica la siguiente instrucción a ejecutar. (Los temas se tratan en la siguiente unidad) La ejecución de dicho proceso debe ser secuencial. La CPU ejecuta una instrucción del proceso tras otra, hasta que el proceso se completa. Además, en cualquier momento, se ejecuta una instrucción como máximo en favor del proceso.

El sistema operativo es responsable de las siguientes actividades en relación con la gestión de procesos:

- Crear y eliminar procesos de usuario y del sistema
- Programar procesos e hilos en las CPU
- Suspender y reanudar procesos
- Proporcionar mecanismos para la sincronización de procesos.
- Proporcionar mecanismos para la comunicación de procesos.

A tono con lo anterior (BUA, 2015), señala que es función del sistema operativo:

- Planificación de procesos: decide qué proceso emplea el procesador en cada instante de tiempo.

- Mecanismos de comunicación entre procesos: permiten comunicar a dos procesos del sistema operativo.
- Mecanismos de sincronización: permiten coordinar a procesos que realizan accesos concurrentes a un cierto recurso.

b. Gestión de Memoria

Es importante conocer que la memoria principal es fundamental para el funcionamiento de un sistema informático moderno. La memoria principal es un gran arreglo de bytes, que varían en tamaño desde cientos de miles hasta miles de millones. Cada byte tiene su propia dirección. La memoria principal es un depósito de datos de acceso rápido compartidos por la CPU y los dispositivos de E/S.

El CPU lee instrucciones de la memoria principal durante el ciclo de búsqueda de instrucciones y lee y escribe datos de la memoria principal durante el ciclo de búsqueda de datos (en una arquitectura de von Neumann). La memoria principal es generalmente el único dispositivo de almacenamiento grande al que la CPU puede direccionar y acceder directamente. Por ejemplo, para que la CPU procese datos del disco, esos datos deben transferirse primero a la memoria principal mediante llamadas de E/S generadas por la CPU. De la misma forma, las instrucciones deben estar en la memoria para que la CPU las ejecute.

Para que se ejecute un programa, debe asignarse a direcciones absolutas y cargarse en la memoria. A medida que el programa se ejecuta, accede a las instrucciones y datos del programa desde la memoria generando estas direcciones absolutas. Finalmente, el programa termina, su espacio de memoria se declara disponible y el siguiente programa se puede cargar y ejecutar.

Para mejorar tanto la utilización de la CPU como la velocidad de respuesta de la computadora a sus usuarios, las computadoras de uso general deben

mantener varios programas en la memoria, lo que crea la necesidad de administrar la memoria. Se utilizan muchos esquemas de administración de memoria diferentes. Estos esquemas reflejan varios enfoques y la efectividad de cualquier algoritmo dado depende de la situación. Al seleccionar un esquema de administración de memoria para un sistema específico, debemos tener en cuenta muchos factores, especialmente el diseño de hardware del sistema. Cada algoritmo requiere su propio soporte de hardware.

El sistema operativo es responsable de las siguientes actividades relacionadas con la gestión de la memoria:

- Realizar un seguimiento de qué partes de la memoria se están utilizando actualmente y qué proceso las está utilizando.
- Asignar y desasignar espacio de memoria según sea necesario.
- Decidir qué procesos (o partes de procesos) y datos se moverán hacia y desde la memoria.

Relacionado con lo anteriormente expresado (La Red Martínez, 2008), la Memoria (informática) es una gran tabla de palabras o bytes que se referencian cada una mediante una dirección única. Este almacén de datos de rápido accesos es compartido por la CPU y los dispositivos de E/S, son volátiles y pierden su contenido en los fallos del sistema. Asimismo, el mencionado autor indica: el Sistema Operativo es el responsable de:

- Conocer qué partes de la memoria están utilizadas y por quién.
- Decidir qué procesos se cargarán en memoria cuando haya espacio disponible.
- Asignar y reclamar espacio de memoria cuando sea necesario.

c. Gestión del sistema de archivos

Para que el sistema informático sea conveniente para los usuarios, el sistema operativo proporciona una vista lógica y uniforme del almacenamiento de información. El sistema operativo se abstrae de las

propiedades físicas de sus dispositivos de almacenamiento para definir una unidad lógica de almacenamiento, el sistema de archivos. El sistema operativo asigna archivos a medios físicos y accede a estos archivos a través de los dispositivos de almacenamiento.

La gestión de archivos es uno de los componentes más visibles de un sistema operativo. Las computadoras pueden almacenar información en varios tipos diferentes de medios físicos. El almacenamiento secundario es el más común, pero también es posible el almacenamiento terciario. Cada uno de estos medios tiene sus propias características y organización física. La mayoría están controlados por un dispositivo, como una unidad de disco, que también tiene sus propias características únicas. Estas propiedades incluyen la velocidad de acceso, la capacidad, la tasa de transferencia de datos y el método de acceso (secuencial o aleatorio).

Un archivo es una colección de información relacionada definida por su creador. Por lo general, los archivos representan programas (fuentes y formas de objeto) y datos. Los archivos de datos pueden ser numéricos, alfabéticos, alfanuméricos o binarios. Los archivos pueden ser de formato libre (por ejemplo, archivos de texto) o pueden tener un formato rígido (por ejemplo, campos fijos como un archivo de música mp3). Claramente, el concepto de archivo es extremadamente general.

El sistema operativo implementa el concepto abstracto de un archivo al administrar los medios de almacenamiento masivo y los dispositivos que los controlan. Además, los archivos normalmente se organizan en directorios para facilitar su uso. Finalmente, cuando varios usuarios tienen acceso a archivos, puede ser conveniente controlar qué usuario puede acceder a un archivo y cómo ese usuario puede acceder a él (por ejemplo, leer, escribir, agregar).

El sistema operativo es responsable de las siguientes actividades relacionadas con la gestión de archivos:

- Crear y eliminar archivos.
- Crear y eliminar directorios para organizar archivos.
- Soporta primitivas para manipular archivos y directorios.
- Asignación de archivos al almacenamiento masivo.
- Hacer copias de seguridad de archivos en medios de almacenamiento estables (no volátiles)

En concordancia con lo antes expuesto, (Roa, 2017) señala que la gestión de archivos, es conocida también como gestión de ficheros, y carpetas respectivamente. En tal sentido, el sistema de archivo proporciona servicios que permite al usuario crear archivos, asignarles nombres con sentido, manipularlos y especificar la manera en que serán compartidos con otros usuarios del sistema (Roa, 2017).

Igualmente, el sistema de archivo provee las estructuras de directorio que permiten que un usuario organice sus datos en grupos lógicos de archivos. Por ejemplo, un usuario podrá preferir separar datos personales de datos profesionales y estructurar los datos profesionales según las actividades (Roa, 2017).

El sistema de archivo proporciona protección contra acceso ilegal a los archivos y reglas para compartirlos concurrentemente. También asegura que los datos se almacenen de manera confiable, es decir, que ningún dato se pierda cuando ocurra una caída del sistema (Roa, 2017).

El sistema operativo es el responsable de crear y mantener un archivo, para asegurar que a él acceden los usuarios solo conforme a los privilegios de acceso especificados para el archivo, y para eliminar el archivo cuando así los solicite su propietario. Estas funciones son realizadas mediante el empleo de la interfaz del sistema de archivos. El verdadero acceso a los archivos, es decir, la lectura o escritura de registros, se implementan utilizando los Indicadores de Compromiso (IOCS).

d. Gestión de almacenamiento masivo

El sistema informático debe proporcionar almacenamiento secundario para respaldar la memoria principal. La mayoría de los sistemas informáticos modernos utilizan unidad de discos duros (HDD Hard Disk Drive) y dispositivos de memoria no volátil (NVM Non-Volatile Memory) como el principal medio de almacenamiento en línea para programas y datos.

La mayoría de los programas, incluidos compiladores, navegadores web, procesadores de texto y juegos, se almacenan en estos dispositivos hasta que se cargan en la memoria. Luego, los programas utilizan los dispositivos como fuente y destino de su procesamiento.

Por tanto, la gestión adecuada del almacenamiento secundario es de vital importancia para un sistema informático. El sistema operativo es responsable de las siguientes actividades en relación con la gestión del almacenamiento secundario:

- Montaje y desmontaje
- Gestión del espacio libre
- Asignación de almacenamiento
- Planificación de disco
- Fraccionamiento o particionamiento
- Protección

Debido a que el almacenamiento secundario se usa con frecuencia y de manera extensiva, debe usarse de manera eficiente. Toda la velocidad de funcionamiento de una computadora puede depender de las velocidades del subsistema de almacenamiento secundario y de los algoritmos que manipulan ese subsistema.

Al mismo tiempo, hay muchos usos para el almacenamiento que son más lentos y de menor costo (y, a veces, de mayor capacidad) que el almacenamiento secundario. Las copias de seguridad de los datos del disco, el almacenamiento de datos poco utilizados y el almacenamiento de

archivos a largo plazo son algunos ejemplos. Las unidades de cinta magnética y sus cintas y las unidades y platos de CD, DVD y Blu-ray son dispositivos de almacenamiento terciario típicos.

El almacenamiento terciario no es crucial para el rendimiento del sistema, pero aún debe administrarse. Algunos sistemas operativos se encargan de esta tarea, mientras que otros dejan la gestión del almacenamiento terciario a los programas de aplicación. Algunas de las funciones que pueden proporcionar los sistemas operativos incluyen el montaje y desmontaje de medios en dispositivos, la asignación y liberación de dispositivos para uso exclusivo de los procesos y la migración de datos del almacenamiento secundario al terciario.

e. Gestión de caché

El almacenamiento en caché es un principio importante de los sistemas informáticos. De acuerdo con (Aguilar & Sumoza, 2007), la memoria cache, es una memoria temporal, generalmente de existencia oculta y automática para el usuario, que proporciona acceso rápido a los datos de uso más frecuente (p.2). También señalan estos autores "la forma de trabajar de la memoria cache es simple, los datos pedidos por el procesador son buscados primero en la memoria cache, y después, si no están disponibles en ella, se buscan en la memoria principal" (p.2).

Así es como funciona. La información normalmente se guarda en algún sistema de almacenamiento (como la memoria principal). A medida que se utiliza, se copia en un sistema de almacenamiento más rápido, el caché, de forma temporal. Cuando se requiere una información en particular, primero se verifica si está en la caché. Si es así, se usa la información directamente del caché. Si no es así, se utiliza la información de la fuente, poniendo una copia en el caché bajo el supuesto de que se puede necesitar nuevamente pronto.

Además, los registros programables internos proporcionan una caché de alta velocidad para la memoria principal. El programador (o compilador)

implementa la asignación de registros y los algoritmos de reemplazo de registros para decidir qué información conservar en los registros y cuál en la memoria principal.

Otros cachés se implementan totalmente en hardware. Por ejemplo, la mayoría de los sistemas tienen un caché de instrucciones para almacenar las instrucciones que se espera que se ejecuten a continuación. Sin esta caché, la CPU tendría que esperar varios ciclos mientras se recupera una instrucción de la memoria principal. Por razones similares, la mayoría de los sistemas tienen una o más cachés de datos de alta velocidad en la jerarquía de memoria.

A tono con esto, (Aguilar & Sumoza, 2007) destacan que existen una serie de mecanismos para la administración y gestión de la memoria cache que son utilizados por los distintos tipos de plataformas distribuidas (memoria compartida, memoria distribuida, etc.), los cuales influyen directamente en el rendimiento y el costo de este recurso. Entre los mecanismos está el denominado Write Through o Write Update en el cual la data es escrita en la memoria principal al mismo tiempo que es almacenada en la memoria cache.

Siguiendo con los aportes de los mencionados autores, la distribución de la memoria cache en sistemas con memoria principal distribuida, significa que cada procesador posee su propia cache y su propia memoria principal. Los mecanismos presentados previamente establecen una relación entre la memoria principal y su cache, sin considerar a los otros nodos del sistema. Es en estos casos donde el problema de coherencia deviene crucial. El uso de la Memoria Cache Distribuida es importante en ambientes donde hay muchos accesos a datos remotos por aplicaciones, en los cuales las actualizaciones de los mismos ocurren con muy poca frecuencia. Por ejemplo, en los Grid de Datos donde las consultas remotas son importantes (Aguilar & Sumoza, 2007).

Debido a que las cachés tienen un tamaño limitado, la administración de la caché es un problema de diseño importante. La selección cuidadosa del tamaño de la caché y de una política de reemplazo puede resultar en un rendimiento mucho mayor, como puede ver al examinar la Tabla 2.

Tabla 2. Características de varios tipos de almacenamientos

Nivel	1	2	3	4	5
Nombre	registros	caché	Memoria principal	Disco de estado sólido (SSD)	Disco magnético
Tamaño típico	< 1KB	>16 MB	< 65 GB	< 1TB	< 10 TB
Implementación tecnológica	memoria personalizada con múltiples puertos CMOS	CMOS SRAM en chip o fuera de chip	CMOS SRAM	memoria tipo flash	disco magnético
Tiempo de acceso (ns)	0.25 – 0.50	0.50 – 25	80 – 250	25,000 – 50,000	5'000,000
Ancho de banda (MB/sec)	20,000 – 100,000	5,000 – 10,000	1,000 – 5,000	500	20 – 150
Gestionado por	compilador	hardware	sistema operativo	sistema operativo	sistema operativo
Respaldado por	cache	memoria principal	disco	disco	disco o cinta

Fuente: Los autores (2022)

El movimiento de información entre niveles de una jerarquía de almacenamiento puede ser explícito o implícito, según el diseño del hardware y el software del sistema operativo de control. Por ejemplo, la transferencia de datos de la caché a la CPU y los registros suele ser una función de hardware, sin intervención del sistema operativo. Por el contrario, la transferencia de datos del disco a la memoria suele estar controlada por el sistema operativo.

En una estructura de almacenamiento jerárquica, los mismos datos pueden aparecer en diferentes niveles del sistema de almacenamiento. Por ejemplo, suponga que un entero A qué se va a incrementar en 1 está ubicado en el archivo B y el archivo B reside en el disco duro. La operación de incremento procede emitiendo primero una operación de E/S para copiar el bloque de disco en el que A reside a la memoria principal. A esta operación le sigue la copia de A en la caché y en un registro interno. Así, la copia de A aparece en varios lugares: en el disco duro, en la memoria principal, en la caché y en un registro interno (ver figura 4). Una vez que se produce el incremento en el registro interno, el valor de A difiere en los distintos sistemas de almacenamiento. El valor de A se convierte en el mismo sólo después de que el nuevo valor de A se escribe desde el registro interno de nuevo al disco duro.

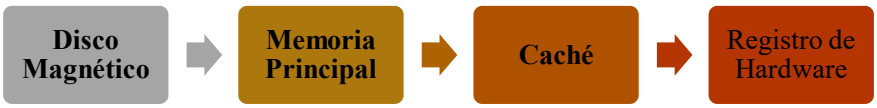


Figura 4. Migración del entero A desde el disco al registro

Fuente: Los autores (2022)

En un entorno informático donde solo se ejecuta un proceso a la vez, esta disposición no plantea dificultades, ya que un acceso al entero A siempre será a la copia en el nivel más alto de la jerarquía. Sin embargo, en un entorno multitarea, donde la CPU se alterna entre varios procesos, se debe tener sumo cuidado para garantizar que, si varios procesos desean acceder a A, cada uno de estos procesos obtendrá el valor actualizado más recientemente de A.

La situación se vuelve más complicada en un entorno multiprocesador donde, además de mantener registros internos, cada una de las CPU también contiene una caché local (será tratado en el siguiente compendio). En tal entorno, una copia de A puede existir simultáneamente en varias cachés. Dado que las distintas CPU pueden ejecutarse todas en paralelo, debemos asegurarnos de que una actualización del valor de A en una caché se refleje inmediatamente en todas las demás cachés donde A reside. Esta situación se denomina coherencia de caché y suele ser un problema de hardware (que se maneja por debajo del nivel del sistema operativo).

En un entorno distribuido, la situación se vuelve aún más compleja. En este entorno, se pueden guardar varias copias (o réplicas) del mismo archivo en diferentes computadoras. Dado que se puede acceder y actualizar las diversas réplicas al mismo tiempo, algunos sistemas distribuidos garantizan que, cuando una réplica se actualiza en un lugar, todas las demás réplicas se actualizan lo antes posible, para lograrlo hay varias formas de lograr esta garantía (este tema no será tratado en esta asignatura).

f. Gestión del sistema de Entradas y Salidas (E/S)

El Sistema Operativo contiene gestores de periféricos, que son rutinas de E/S encargadas de controlar dispositivos, estos gestores son los únicos que deberán tener en cuenta las peculiaridades concretas de los dispositivos. La rutina de entrada/salida realiza las comprobaciones necesarias sobre

la operación que se va a realizar, y hace la petición de servicio al gestor del periférico correspondiente, que será el que efectúe la operación (Roa, 2017). El subsistema de E/S consta de varios componentes:

- Un componente de administración de memoria que incluye almacenamiento en búfer, almacenamiento en caché y cola
- Una interfaz general de controlador de dispositivo
- Controladores para dispositivos de hardware específicos

Solo el controlador del dispositivo conoce las peculiaridades del dispositivo específico al que está asignado.

Para esto, el sistema operativo se apoya en la manipulación de las interrupciones y los controladores de dispositivos los cuales son usados para la construcción de subsistemas de E/S eficientes.

A este propósito, (Roa, 2017) destaca que las operaciones de E/S pueden realizarse de tres formas:

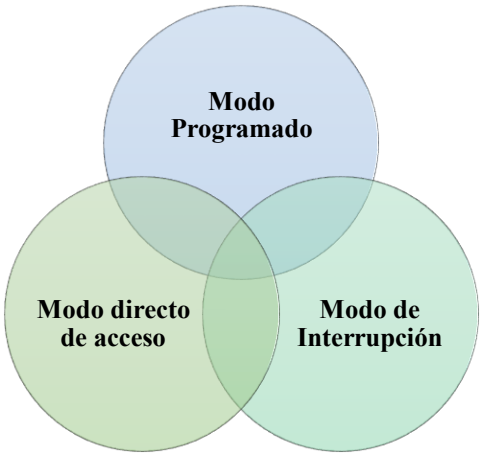


Figura 5. Modo de Operaciones del Sistema E/S

Fuente: (Roa, 2017)

- ***Modo programado de E/S:*** es lento e involucra al CPU en una operación de E/S, por consiguiente, solo puede ejecutarse una operación de E/S.

- ***Modo de interrupción:*** es también lento, de modo que se ejecuta una trasferencia de datos byte por byte; de cualquier forma, aquél libera la CPU de las trasferencias de bytes.
- ***Modo directo de acceso*** a la memoria (DMA) puede trasferir un bloque de datos entre la memoria y un dispositivo de E/S sin involucrar al CPU.

La interrupción y los medios (DMA) permiten la ejecución simultánea de varias operaciones (Roa, 2017).

g. Seguridad y Protección

Según (Cristiá, 2021), la seguridad informática se entiende como la combinación de tres atributos de calidad: confidencialidad, integridad y disponibilidad (p.3). Preservar la confidencialidad de los datos significa que solo los usuarios autorizados puedan ver dichos datos. Preservar la integridad de los datos y programas significa que solo el usuario autorizado y solo por los mecanismos autorizados pueda modificar dichos datos y programas. Preservar la disponibilidad de datos y programas significa que los usuarios autorizados puedan usar los datos y programas cada vez que los necesiten (Cristiá, 2021).

Si un sistema informático tiene múltiples usuarios y permite la ejecución concurrente de múltiples procesos, entonces el acceso a los datos debe estar regulado. Para ese propósito, los mecanismos aseguran que los archivos, segmentos de memoria, CPU y otros recursos puedan ser operados por solo aquellos procesos que hayan obtenido la autorización adecuada del sistema operativo. Por ejemplo, el hardware de direccionamiento de memoria asegura que un proceso se pueda ejecutar solo dentro de su propio espacio de direcciones.

El temporizador asegura que ningún proceso pueda obtener el control de la CPU sin renunciar finalmente al control. Los usuarios no pueden acceder a los registros de control de dispositivos, por lo que la integridad

de los distintos dispositivos periféricos está protegida. La protección, entonces, es cualquier mecanismo para controlar el acceso a los procesos.

En concordancia con esto, (Avenía, 2017) indica que la seguridad de la información es la protección de la información y de los sistemas de información del acceso, uso, divulgación y destrucción no autorizada a través de estándares, procesos, procedimientos, estrategias, recursos informáticos, recursos educativos y recursos humanos (p.12).

La protección, entonces, es cualquier mecanismo para controlar el acceso de procesos o usuarios a los recursos definidos por un sistema informático. Este mecanismo debe proporcionar medios para especificar los controles que se impondrán y hacer cumplir los controles. La protección puede mejorar la confiabilidad al detectar errores latentes en las interfaces entre los subsistemas de componentes. La detección temprana de errores de interfaz a menudo puede evitar la contaminación de un subsistema en buen estado por otro subsistema que no funciona correctamente. Además, un recurso desprotegido no puede defenderse del uso (o mal uso) por parte de un usuario no autorizado o incompetente. Un sistema orientado a la protección proporciona un medio para distinguir entre el uso autorizado y no autorizado.

Un sistema puede tener la protección adecuada, pero aun así ser propenso a fallar y permitir un acceso inadecuado. Considere un usuario cuya información de autenticación (su medio de identificarse en el sistema) es robada. Sus datos se pueden copiar o eliminar, aunque la protección de archivos y memoria esté funcionando. El trabajo de la seguridad es defender un sistema de ataques externos e internos. Dichos ataques se extienden por una amplia gama e incluyen virus y gusanos, ataques de denegación de servicio (que usan todos los recursos del sistema y, por lo tanto, mantienen a los usuarios legítimos fuera del sistema), robo de identidad y robo de servicio (uso no autorizado de un sistema).

La prevención de algunos de estos ataques se considera una función del sistema operativo en algunos sistemas, mientras que otros sistemas lo dejan en manos de políticas o software adicional. Debido al alarmante aumento de los incidentes de seguridad, las características de seguridad del sistema operativo son un área de investigación e implementación de rápido crecimiento.

La protección y la seguridad requieren que el sistema pueda distinguir entre todos sus usuarios. La mayoría de los sistemas operativos mantienen una lista de nombres de usuario y un identificador de usuario asociado (ID de usuario). En lenguaje de Windows, este es un ID de seguridad (SID). Estos ID numéricos son únicos, uno por usuario. Cuando un usuario inicia sesión en el sistema, la etapa de autenticación determina el ID de usuario apropiado para el usuario. Ese ID de usuario está asociado con todos los procesos e hilos del usuario. Cuando un usuario necesita leer una identificación, se vuelve a traducir al nombre de usuario a través de la lista de nombres de usuario.

En algunas circunstancias, deseamos distinguir entre conjuntos de usuarios en lugar de usuarios individuales. Por ejemplo, al propietario de un archivo en un sistema UNIX se le puede permitir ejecutar todas las operaciones en ese archivo, mientras que a un grupo seleccionado de usuarios solo se le puede permitir leer el archivo. Para lograr esto, necesitamos definir un nombre de grupo y el conjunto de usuarios que pertenecen a ese grupo. La funcionalidad de grupo se puede implementar como una lista de todo el sistema de nombres de grupo e identificador de grupo. Un usuario puede estar en uno o más grupos, según las decisiones de diseño del sistema operativo. Los ID de grupo del usuario también se incluyen en cada proceso e hilo asociado.

En el curso del uso normal del sistema, el ID de usuario y el ID de grupo de un usuario son suficientes. Sin embargo, un usuario a veces necesita escalar privilegios para obtener permisos adicionales para una actividad.

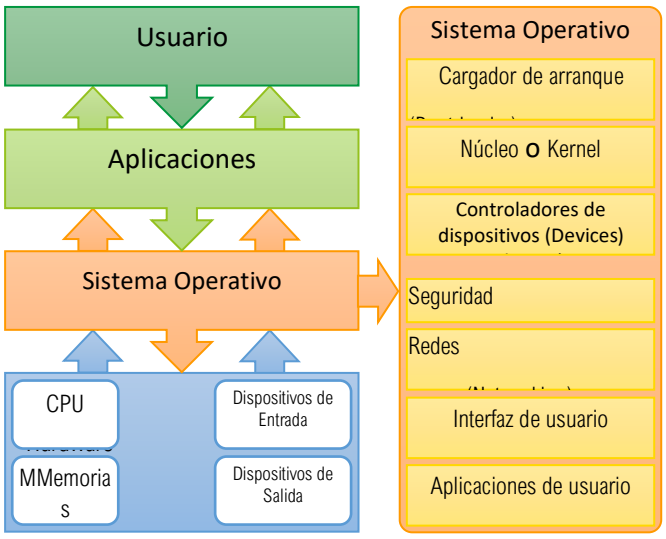
El usuario puede necesitar acceso a un dispositivo que está restringido, por ejemplo. Los sistemas operativos proporcionan varios métodos para permitir la escalada de privilegios. En UNIX, por ejemplo, el atributo setuid en un programa hace que ese programa se ejecute con el ID de usuario del propietario del archivo, en lugar del ID del usuario actual. El proceso se ejecuta con este UID efectivo hasta que desactiva los privilegios adicionales o termina.

1.7.- Componentes del Sistema Operativo (SO)

En consideraciones de (BUA, 2015), la parte más importante de un sistema operativo es el kernel o núcleo, que se encarga de facilitar a las distintas aplicaciones acceso seguro al hardware del sistema informático (p.4). En este marco, (Villarreal, 2017), también expone el sistema operativo es la pieza fundamental del software y se ejecuta en modo kernel (también conocido como modo supervisor). En este modo, el sistema operativo tiene acceso completo a todo el hardware y puede ejecutar cualquier instrucción que la máquina sea capaz de ejecutar. El resto del software se ejecuta en modo usuario, en el cual sólo un subconjunto de las instrucciones de máquina es permitido (p.6).

Núcleo

Es un software que constituye una parte fundamental del sistema operativo, y se define como la parte que se ejecuta en modo privilegiado (conocido también como modo núcleo). Es el principal responsable de facilitar a los distintos programas acceso seguro al hardware de la computadora o en forma



básica, es el encargado de gestionar recursos, a través de servicios de llamada al sistema. De esta forma, (BUA, 2015), subraya que los núcleos tienen como funciones básicas:

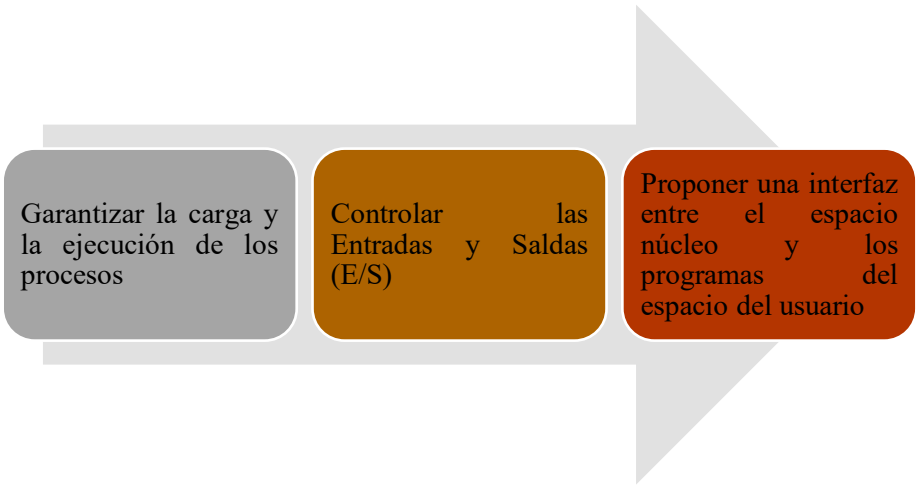


Figura 6. Funciones básicas del kernel o núcleo en un Sistema Operativo
Fuente: (BUA, 2015)

Como hay muchos programas y el acceso al hardware es limitado, también se encarga de decidir qué programa podrá usar un dispositivo de hardware y durante cuánto tiempo, lo que se conoce como multiprogramación. Acceder al hardware directamente puede ser realmente complejo, por lo que los núcleos suelen implementar una serie de abstracciones del hardware. Esto permite esconder la complejidad, y proporcionar una interfaz limpia y uniforme al hardware subyacente, lo que facilita su uso al programador.

En algunos sistemas operativos, no existe un núcleo como tal (algo común en sistemas embebidos), debido a que en ciertas arquitecturas no hay distintos modos de ejecución.

Cargador de arranque

Un cargador de arranque carga un sistema operativo en la memoria, realiza la inicialización y comienza la ejecución del sistema. Al respecto, (Villarreal, 2017) señala "la secuencia de eventos que ocurren entre el tiempo que se enciende la computadora y el tiempo que se pone listo para

aceptar comando, recibe el nombre de proceso de arranque" (p.17). Este permite seleccionar la partición raíz la cual contiene el núcleo del sistema operativo y a veces otros archivos del sistema.

Controladores de dispositivos

Normalmente, los sistemas operativos tienen un software controlador de dispositivo (device driver) para cada controlador electrónico de dispositivo (device controller). Este software controlador de dispositivo comprende el controlador electrónico de dispositivo y proporciona al resto del sistema operativo una interfaz uniforme para el dispositivo. La CPU y los controladores electrónicos de dispositivos pueden ejecutarse en paralelo, compitiendo por ciclos de memoria. Para garantizar el acceso ordenado a la memoria compartida, un controlador electrónico de memoria sincroniza el acceso a la memoria.

Los softwares controladores de dispositivos permiten habilitar las operaciones de E/S de un sistema, porque un programa en ejecución puede requerir E/S, lo que puede implicar un archivo o un dispositivo de E/S. Para dispositivos específicos, se pueden desear funciones especiales (como leer desde una interfaz de red o escribir en un sistema de archivos). Por razones de eficiencia y protección, los usuarios generalmente no pueden controlar los dispositivos de E/S directamente. Por lo tanto, el sistema operativo a través de estos softwares controladores de dispositivos puede proporcionar un medio para realizar las actividades de E/S.

Seguridad

La importancia de la protección y seguridad hace que los propietarios de la información almacenada en un sistema informático multiusuario o en red pueden querer controlar el uso de esa información. Cuando varios procesos separados se ejecutan al mismo tiempo, no debería ser posible que un proceso interfiera con los demás o con el sistema operativo en sí. En relación con esto, (Avenía, 2017) asevera que el objetivo de la seguridad

informática es proteger los valiosos recursos informáticos de la organización, tales como la información, hardware o software (p.17).

La protección implica garantizar que se controle todo el acceso a los recursos del sistema. La seguridad del sistema frente a personas externas también es importante. Dicha seguridad comienza requiriendo que cada usuario se autentique en el sistema, generalmente mediante una contraseña, para obtener acceso a los recursos del sistema. Se extiende a la defensa de los dispositivos de E/S externos, incluidos los adaptadores de red, de los intentos de acceso no válidos y al registro de todas esas conexiones para la detección de robos. Para proteger y asegurar un sistema, se deben tomar precauciones a través de este.

Redes

Una red, en los términos más simples, es una ruta de comunicación entre dos o más sistemas. Los sistemas distribuidos dependen de las redes para su funcionalidad. Las redes varían según los protocolos utilizados, las distancias entre los nodos y los medios de transporte. TCP/IP es el protocolo de red más común y proporciona la arquitectura fundamental de Internet. La mayoría de los sistemas operativos admiten TCP/IP, incluidos todos los de uso general. Algunos sistemas admiten protocolos propietarios para adaptarse a sus necesidades. Para un sistema operativo, solo es necesario que un protocolo de red tenga un dispositivo de interfaz (un adaptador de red, por ejemplo) con un controlador de dispositivo para administrarlo, así como un software para manejar datos.

Interfaz de usuario

Casi todos los sistemas operativos tienen una interfaz de usuario (UI). Según (Villarreal, 2017), la principal función de la interfaz de usuario del sistema operativo es permitir al usuario acceder y manipular sus objetos; es lo que el usuario ve en pantalla y hace posible la interacción con hardware, permitiéndole dar órdenes o ejecutar programas (p.18).

Esta interfaz puede adoptar varias formas. Por lo general, se utiliza una interfaz gráfica de usuario (GUI – Graphic User Interface). Aquí, la interfaz es un sistema de ventana con un raton que sirve como un dispositivo señalador para dirigir E/S, elegir entre menús y hacer selecciones y un teclado para ingresar texto. Los sistemas móviles, como teléfonos y tabletas, proporcionan una interfaz de pantalla táctil que permite a los usuarios deslizar los dedos por la pantalla o presionar botones en la pantalla para seleccionar opciones. Otra opción es una interfaz de línea de comandos (CLI – Command Line Interface), que usa comandos de texto y un método para ingresarlos (por ejemplo, un teclado para ingresar comandos en un formato específico con opciones específicas). Algunos sistemas proporcionan dos o las tres variaciones.

Aplicaciones de usuario

La ejecución de programa o aplicaciones de usuarios es una parte fundamental de los sistemas operativos, porque el sistema debe poder cargar un programa en la memoria y ejecutar ese programa. El programa debe poder finalizar su ejecución, ya sea normal o anormalmente (indicando error).

De acuerdo con (Villarreal, 2017), en la interfaz gráfica de usuario el usuario interacciona utilizando in dispositivo apuntador (por ejemplo, ratón o pantalla táctil) sobre ventanas, iconos y menús. Es tipo de interfaz usada por los sistemas operativos con gestión de ventanas como Linux y Windows (p.17).

Niveles de ejecución y secuencia de arranque de los sistemas operativos

De acuerdo con lo expresado por (Villarreal, 2017) unos de los componentes de más importancia en una computadora es la memoria RAM, pero esta es volátil, es decir, no almacena ninguna información cuando la computadora es apagada, por tal razón la computadora no puede usar el contenido de la memoria RAM para recordar muchas de sus

funciones básicas, de cómo comunicarse con sus puertos de entrada y de salida con el exterior (p.17).

Continua exponiendo el citado autor, una computadora necesita de alguna manera dar a la memoria RAM los archivos del Sistema Operativo para que esta pueda iniciar. Este es uno de los principales objetivos del proceso de arranque (Villarreal, 2017).

1.8.- Principales Sistemas Operativos para Computadoras u Ordenadores

En la actualidad en el mercado global existen diversos sistemas operativos para ordenadores, como ejemplos de estos sistemas operativos (Jaramillo, 2015), destaca los siguientes

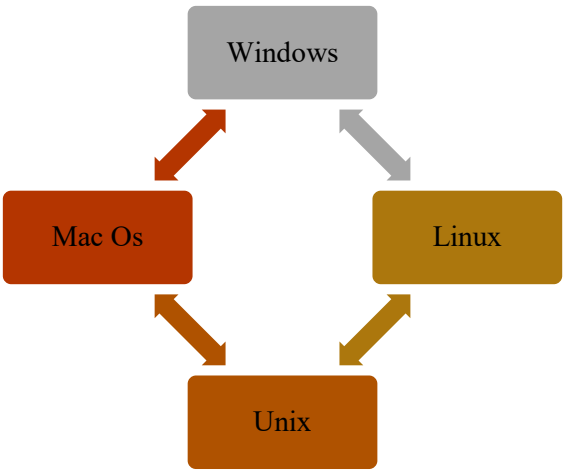


Ilustración 7. Principales Sistemas operativos para ordenadores
Fuente: (Jaramillo, 2015)

Asimismo, destaca (Jaramillo, 2015), con el avance de la tecnología han surgido nuevos sistemas operativos para los teléfonos móviles que ya no se usan solo para realizar llamadas y recibirlas, y que al igual que con los ordenadores, son los intermediarios entre el teléfono móvil y el usuario (p.18). Dentro de los sistemas operativos para los teléfonos móviles, entre otros, se tiene:

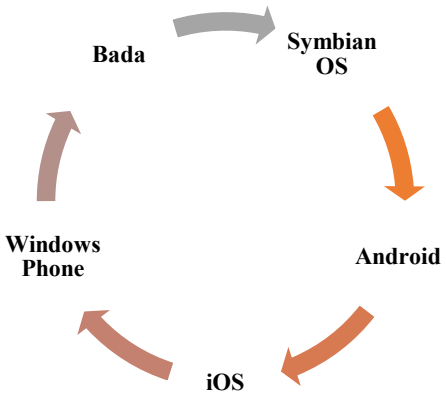


Figura 8. Sistemas operativos para teléfonos móviles

Fuente: (Jaramillo, 2015),

Sistema Operativo Linux

El proceso de arranque de un sistema operativo Linux se inicializa de la siguiente manera:

Cuando usted enciende su servidor o su computadora personal, esta hace que el BIOS de su equipo inicie las operaciones relacionadas con el arranque. El BIOS (Basic Input Output System) es un pequeño programa escrito en lenguaje ensamblador cuya función es cargar el sistema operativo en la memoria RAM (Random Access Memory), una vez que el BIOS carga el sistema operativo en RAM este inicia un proceso llamado POST (Power On Self Test) el cual es un proceso de diagnóstico y verificación de los componentes de entrada y salida de un servidor o computadora y se encarga de configurar y diagnosticar el estado del hardware, una vez verificado el hardware se inicia la fase de arranque del sistema (bootstrapping) el cual cede el control al GRUB (Grand Unified Bootloader), el GRUB es un gestor de arranque que hace uso de un menú gráfico que permite elegir el Sistema Operativo que se desea arrancar; Así mismo, el GRUB realiza las siguientes tareas:

- Cargar el kernel en memoria.

- Cargar el sistema de ficheros virtual initrd el cual es usado típicamente para hacer los arreglos necesarios antes de que el sistema de ficheros raíz pueda ser montado.
- Pasarle los argumentos runlevel e init al kernel
- Comenzar la ejecución del kernel

*Al terminar de ejecutar todas las tareas anteriores el GRUB le cede el control total del arranque al kernel y este a su vez se encarga de realizar la llamada a la función **starup** la cual tiene como función detectar el tipo de CPU con el que el equipo cuenta así como de lo principal del sistema operativo, como el manejo de memoria, planificador de tareas, entradas y salidas, comunicación interprocesos, y demás sistemas de control, a partir de este momento se ejecuta el proceso **INIT**.*

El Proceso INIT

INIT es el primer proceso en ejecutarse después de la carga del kernel de Linux e implementa dos modelos bajo los cuales puede trabajar, estos son

- SystemV
- BSD

Estos modelos son arrancados por un programa (script) de arranque que establece como deben inicializarse los diferentes servicios, programas o registros que sean necesarios para que el sistema funcione como el administrador lo requiere.

Explicaremos brevemente como es que trabajan estos modelos

SystemV

Es un modelo usado para controlar el inicio y apagado del sistema y fue originalmente desarrollado por la compañía estadounidense de telecomunicaciones AT&T.

SystemV fue una de las versiones del sistema operativo Unix que se encargaba de controlar el arranque de los programas en el instante de inicio del equipo. Este modelo es considerado por muchos como fácil, potente y flexible en comparación con el sistema de inicio BSD.

Existen cuatro versiones reléase de SystemV (**SVR**), las cuales son:

SVR1. *Primera versión de SystemV lanzada en 1984, incluía el editor de textos Vi*

SVR2*. Incluye mejoras con respecto al núcleo el cual esta implementado como memoria virtual paginada, el sistema operativo Apple está basado en este modelo.*

SVR3. *Incluye mejoras en el sistema de ficheros, así como una nueva API de red, el sistema operativo AIX de IBM hace uso de este modelo*

Los niveles de ejecución en SystemV describen ciertos estados del equipo los cuales se caracterizan por ejecutar ciertos procesos. En general existen 8 niveles de ejecución los cuales van del 0 al 6 y S o s, que son alias del mismo nivel de ejecución, de estos ochos niveles, tres son considerados reservados, estos son:

1.Halt

2.Single user mode

3.Reboot

Aparte de los niveles de ejecución 0,1 y 6 todos los sistemas operativos Linux tratan a los niveles de ejecución un poco diferente. El denominador común de todas las distribuciones Linux es el fichero /etc./inittab

0 indica halt o apagado de la máquina.

1 indica monousuario.

2 indica modo multiusuario sin soporte de red.

3 indica modo multiusuario completo con soporte de red.

4 no usado, con esta opción el administrador puede personalizar el inicio para cargar algún servicio.

5 indica multiusuario completo con inicio gráfico (X11)

6 indica shutdown y reboot: Se apaga inmediatamente la máquina para reinicio.

Un administrador (root) puede editar el archivo /etc./inittab como mejor convenga al usuario, sin embargo, también tiene el poder de establecerlo en 0 o en 6.

A continuación, un ejemplo de cuantos niveles de ejecución tiene cada una de las distribuciones más importantes de Linux, así como del sistema operativo Solaris y AIX, como muestra la Tabla 3.

Tabla 3. Distribuciones más importantes de Linux

Sistema Operativo	Niveles de ejecución por default
AIX	2
debian	2
gentoo linux™	3
Mandriva	5
redhat	3 o 5
fedora	3 o 5

	3
	5
	2
	2

Fuente: Los autores (2022)

1 Bibliografía

Aguilar, J., & Sumoza, R. (2007). Cache memory coherence protocol for distributed systems . *Revista Técnica de la Facultad de Ingeniería Universidad del Zulia. Vol.30. Nro.2. Versión impresa ISSN 0254-0770. http://ve.scielo.org/scielo.php?script=sci_arttext&pid=S0254-07702007000200008* , pp.1-20.

Avenía, C. (2017). Fundamentos de Seguridad Informática. *Fundación Universitaria del Área Andina. Bogotá D.C. https://digitk.areandina.edu.co/bitstream/handle/areandina/1367/Fundamentos%20de%20seguridad%20inform%C3%A1tica.pdf?sequence=1&isAllowed=y*, pp.98.

BUA. (2015). Sistemas Operativos. *Biblioteca de la Universidad de Alicante (BUA). Material Formativo*, pp.1-24.

Candela, S., García, C., Quesada, A., Santana, F., & Santos, J. (2007). *Fundamentos de los Sistemas Operativos.* Madrid, España: Thomsom.

Carmona, J. (2018). Evolución de Windows desde sus inicios hasta Windows 10 April 2018 Update. *https://www.xatakawindows.com/actualidad-en-redmond/como-pasa-el-tiempo-esta-es-la-evolucion-de-windows-desde-sus-inicios-hasta-windows-10-april-2018-update*.

Catalán, G. (1998). DE COMO LAS CONCEPCIONES DEL PROFESORADO INCIDE EN LA SALUD. En *sALUD EDUCACIÓN Y CALIDAD DE VIDA* (pág. 165). Santa Fe.Bogota: Cooperativa Editorial Magisterio.

Chacón, J. (2008). Sistemas informáticos: Estructura y funciones. Elementos de "hardware". Elementos de "software. *Procesos Comerciales. Sistemas Informáticos. Tema 51. Madrid, España.*

https://www.preparadores.eu/temamuestra/PTecnicos/PComerciales.pdf, pp.1-22.

Cristiá, M. (2021). Seguridad Informática. *Universidad Nacional de Rosario. Argentina. Apunte de clase. https://www.fceia.unr.edu.ar/~mcristia/apunte-si.pdf*, pp.87.

Fernández Fernández, G. (2015). Elementos de sistemas operativos, de representación de la información y de procesadores hardware y software. *Universidad Politécnica de Madrid. https://oa.upm.es/36552/1/SORYP.pdf*, pp.366.

Glenn Brookshear, J. (2007). *Computer Science, an Overview.* Pearson/Addison-Wesley, 9ª ed.

IONOS. (2017). Interfaz de Usuario (UI): qué define a una buena interfaz gráfica de usuario. *IONOS. Digital guide. https://www.ionos.es/digitalguide/paginas-web/diseno-web/ui-que-es-una-interfaz-de-usuario/*, pp.1-20.

Jaramillo, C. (2015). Compilación Unidad Temática: Sistemas Operativos. *Universidad de la Amazonia. Florencia, Caquetá. Colombia. https://www.uniamazonia.edu.co/documentos/docs/Programas%20Academicos/Tecnologia%20en%20Informatica%20y%20Sistemas/Compilados/Compilado%20Sistemas%20Operativos.pdf*, pp.36.

La Red Martínez, D. (2008). Sistemas operativos. *http://exa.unne.edu.ar/depar/areas/informatica/SistemasOperativos/SOF.htmhttp://sistemasoperativos.notlong.com/*.

O´Brien, J. (2006). *Sistema Operativo. Sistemas de Información Gerencial.* México DF.

Ocampo, A., & Sánchez, E. (2015). Hardware y Software. *Universidad Autónoma del Estado de Hidalgo (UAEH). https://repository.uaeh.edu.mx/bitstream/bitstream/handle/1234*

56789/16657/PE_T1_U1_HardwareSoftware.pdf?sequence=1, pp.1-21.

Redhat. (26 de 04 de 2021). *RedHat*. Obtenido de https://www.redhat.com/es/topics/virtualization/what-is-a-virtual-machine

Roa, K. (2017). Sistemas Operativos. *Fundación Universitaria del Área Andina. Bogotá D.C.Fondo editorial Areandino. Primera edición. https://digitk.areandina.edu.co/bitstream/handle/areandina/1313/Sistemas%20operativos.pdf?sequence=1&isAllowed=y*, pp.97.

Serna, M., & Allende, S. (2020). Sistemas operativos: Linux. En J. Sarmiento (Ed.). Universitas.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating Systems Concepts* (10th ed.). Hoboken: Wiley.

Stallings, W. (2018). *Operating Systems, Internals and Design Princiles* (Ninth ed.). Malaysia: Pearson.

Villarreal, V. (2017). Sistemas Operativos. *Universidad Tecnológica de Panamá (UTP). Curso. https://ridda2.utp.ac.pa/bitstream/handle/123456789/5074/folleto_sistemas_operativos.pdf?sequence=3&isAllowed=y*, pp.99.

vmware. (2021). *vmware*. Recuperado el 26 de 04 de 2021, de https://www.vmware.com/es/topics/glossary/content/virtual-desktops.html

Wikipedia. (15 de 11 de 2020). *BootX (Apple)*. Obtenido de BootX (Apple): https://en.wikipedia.org/wiki/BootX_(Apple)

Wikipedia. (15 de 11 de 2020). *GNU GRUB*. Obtenido de GNU GRUB: https://en.wikipedia.org/wiki/GNU_GRUB

Wikipedia. (15 de 11 de 2020). *iBoot*. Obtenido de iBoot: https://en.wikipedia.org/wiki/IBoot

Wikipedia. (15 de 11 de 2020). *NTLDR*. Obtenido de NTLDR: https://en.wikipedia.org/wiki/NTLDR

Wikipedia. (15 de 11 de 2020). *Windows NT 6 startup process*. Obtenido de Windows NT 6 startup process: https://en.wikipedia.org/wiki/Windows_NT_6_startup_process

Wolf, G., Ruiz, E., Bergero, F., & Meza, E. (2015). Fundamentos de Sistemas Operativos. *Universidad Nacional Autónoma de México (UNAM). México D.F. Primera edición. https://ru.iiec.unam.mx/2718/1/sistemas_operativos.pdf*, pp.367.

CAPÍTULO II

PROCESOS E HILOS

RESULTADO DE APRENDIZAJE DE LA ASIGNATURA

Conocer los conceptos fundamentales, partiendo del enfoque clásico de los sistemas operativos como administrador eficiente de recursos, componentes, estructuras y funciones de los sistemas operativos, con la finalidad que entiendan el funcionamiento y puedan realizar software de sistemas.

SISTEMAS OPERATIVOS

UNIDAD 2: PROCESOS E HILOS

Resultado de aprendizaje de la unidad:

Describir la importancia de la planificación, la comunicación y la sincronización de procesos e hilos en sistemas de multiprocesamiento.

Tema 1: Procesos

En un sistema multiprogramado o de tiempo compartido (que aparecieron por los 70 con el surgimiento del microprocesador), **un proceso es la imagen en memoria de un programa**, junto con la información relacionada con el estado de su ejecución, o simplemente una secuencia de instrucciones ejecutándose en ese instante de tiempo.

Un programa es una entidad pasiva, una lista de instrucciones; un proceso es una entidad activa, que define la actuación que tendrá el sistema.

En contraste con **proceso**, en un sistema por **lotes** se habla de **tareas**. Una tarea requiere mucha menos estructura, típicamente basta con guardar la información relacionada con la contabilidad de los recursos empleados.

Por lo tanto, un proceso puede caracterizarse de manera única, y esencialmente consta de dos elementos, el código del programa (*program code*) y un conjunto de datos (*set of data*) asociados con ese código, los cuales se almacenan en memoria durante la ejecución de dicho proceso, ahora para conocer el estado actual del proceso, la CPU se apoya en el contador de programa (***program counter, PC***), cuyo valor almacena la dirección de memoria del código en ejecución, que está almacenado en esa dirección de memoria, además de los valores almacenados en ese instante en los registros del procesador. El diseño en memoria de un proceso generalmente se divide en varias secciones como se muestra en la Figura 2.1.

Estas secciones incluyen:

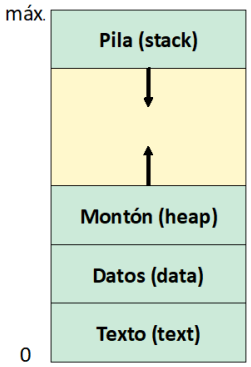


Figura 2.1: Distribución de un proceso en memoria.

- **Texto (Text)** – Esta sección almacena el código ejecutable
- **Datos (Data)** – Esta sección almacena las variables globales
- **Amontonar o montón (Heap)** – Esta sección, representa el almacenamiento dinámico de memoria, el cual se va amontonando o asignando a medida que el programa se está ejecutando, durante el tiempo que dure su ejecución.
- **Pila (Stack)** – Esta sección es un almacenamiento temporal de datos (parámetros, variables locales, direcciones de retorno, etc), lo cuales

van *apilándose* y *desapilándose* a medida que una función es llamada o autollamada (*función recursiva*).

Es importante tomar en cuenta que las secciones de *texto* y *datos* tienen un tamaño fijo, es decir no cambian durante su ejecución, pero los tamaños en memoria de las secciones de *pila* y *montón* se van a ajustando dinámicamente según el programa en ejecución lo requiera. Se debe tomar en cuenta como trabaja la pila, cada vez que una función es llamada, al igual que sus parámetros, variables locales y dirección de retorno los cuales son almacenados en un **registro de activación**, y este es insertado (*pushed*) en la pila, proceso que seguirá, mientras la función sea llamada una y otra vez, apilando un registro de activación sobre otro, hasta que la función deje de ser solicitada por el proceso, luego de esto, es devuelto (*returned*) el control desde la función y los registro de activación que fueron apilados, son tomados (*popped*) de la pila, hasta dejarla vacía. Similar sucede con el montón, porque este crecerá a medida que la memoria sea asignada dinámicamente y se reducirá cuando la memoria sea regresada al sistema. Aunque las secciones de pila y el montón crezcan dinámicamente la una hacia (***toward***) la otra, el sistema operativo debe asegurar de que no haya sobre posición (***overlap***) entre ellas (Serna, 2020, p.109).

Recuerda que:
Se debe tener cuidado porque los tamaños en memoria de las secciones de *pila* y *montón* jamás puede superar la capacidad máxima de la memoria física.

Es importante detallar que un programa por sí mismo no es un proceso, porque mientras no se esté ejecutando es una entidad *pasiva*, el cual puede ser un archivo con un conjunto de instrucción que está almacenado en algún lugar, este puede ser cualquier archivo que tenga características

ejecutable e interpretables por el sistema y que está a la espera de ser ejecutado o no, por su lado un proceso es una entidad activa, que está en ejecución, y mientras se está ejecutando, el contador de programa está a la expectativa de cuál debe ser la siguiente instrucción a ejecutar, además de otros recursos asociados. Ahora, un programa se convierte en un proceso cuando este archivo ejecutable se carga en la memoria. Es más, un mismo programa ejecutado varias veces, está asociado a varios procesos independientes, porque, aunque es el mismo programa, cada secuencia de ejecución es independiente. Un ejemplo común son las pestanas de un navegador web, cada una es un proceso independiente (Figura 2.2). Si volvemos a ver la Figura 2.1, cada proceso independientemente tendrá la sección de texto con el mismo contenido, pero los contenidos de las otras secciones (datos, montón y pila) van a ser distintos.

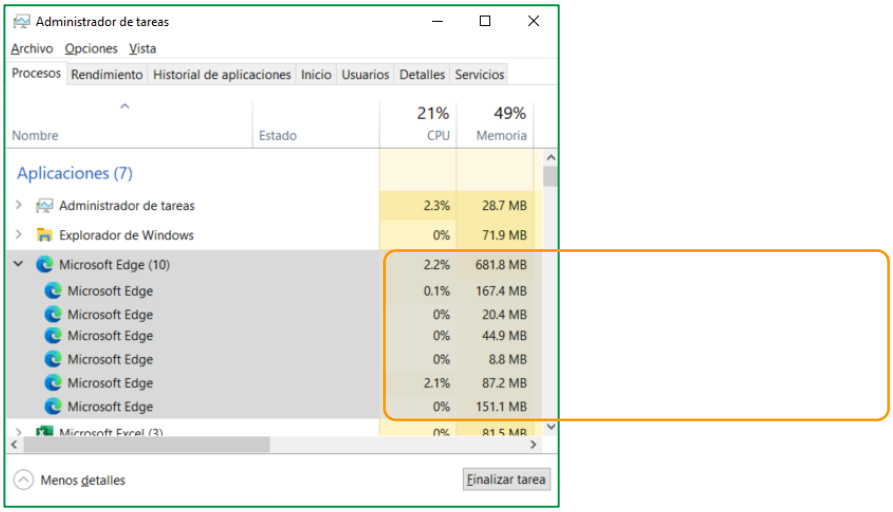


Figura 2.2: Ejemplo del navegador web Microsoft Edge, ejecutándose en varios procesos

Un proceso en sí mismo, puede ser un entorno de ejecución para otro código, por ejemplo: Los programas escritos en Java para poder ejecutarse, necesitan que previamente esté ejecutándose en memoria la máquina virtual de Java (*Java Virtual Machine, JVM*). La JVM se ejecuta como un proceso que interpreta el código Java cargado y realiza acciones (a través de las instrucciones de la máquina nativa) en nombre de ese código.

Ahora si ¿qué es el proceso y como se lo trata Linux?

Un proceso es un programa en ejecución, coloquialmente un proceso es una unidad de actividad que sigue unas secuencias de instrucciones y usa recursos del sistema como ***CPU, MEMORIA*** entre otros, y tiene un estado actual, que tiene que ver esto con Linux. ***Linux es un sistema multiprocesos,*** esto quiere decir que cada proceso puede actuar al mismo tiempo que otro, sin que unos interfieran con otros, para aquellos que tienen conocimientos más avanzado y experimentados sabrán qué esto lo hace por turnos y calendarización los mismo que son pasados por el kernel de uno en uno, para que el kernel pueda tener este tipo de manejo lo que se suele usar es un **PID,** una identificación por sus siglas en inglés **(Processing Identification) identificación de procesos** permitiendo que el kernel pueda hacer su trabajo apropiadamente, procedemos en este apartado a revisar los comando para la ejecución y administración de procesos.

Recuerda que:
Un proceso es una unidad de actividad que sigue una secuencias de instrucciones y usa recursos del sistemas como CPU, MEMORIA entre otros, y tiene un estado actual.

Práctica 1. Estados de la CPU

Lo primero que debemos realizar es arrancar nuestro sistema operativo, para este ejemplo se va utilizar los sistemas operativos Linux CentOS 7.9, a pesar que los comando se pueden utilizar en debían, Ubuntu, Redhat, etc.

Una vez ya dentro del sistema operativo procedemos a abrir una terminal.

Figura 2.3: Escritorio de Linux CentOS 7.9.

Luego en cualquier parte del escritorio damos clic derecho del mouse y escogemos la opción Abrir terminal como lo indica la figura 2.4.

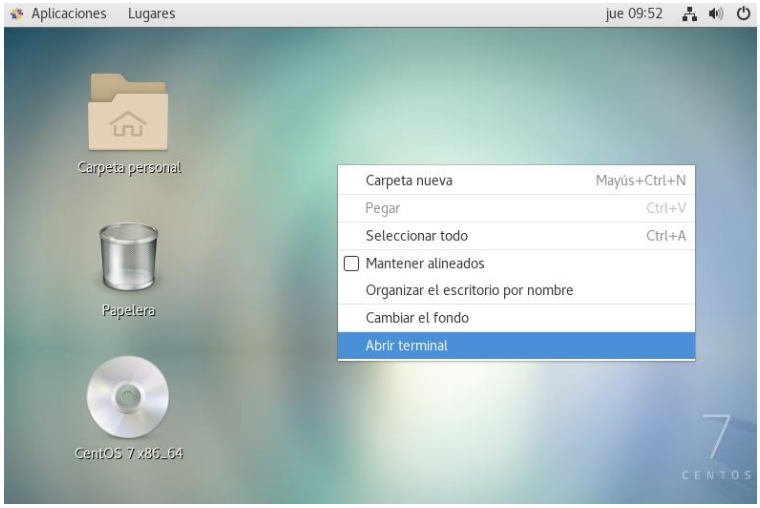


Figura 2.4: abrir terminal en Linux CentOS 7.9.

Luego de haber ejecutado la terminal procedemos a ingresar el comando **top**, el mismo que nos va muestra una interfaz en modo texto que se va a ir actualizando cada 3 segundos. Muestra un resumen del estado de nuestro sistema y la lista de procesos que se están ejecutando.

```
[root@localhost ~]# top
```

La ejecución del comando anterior nos traerá la siguiente ventana con los resultados.

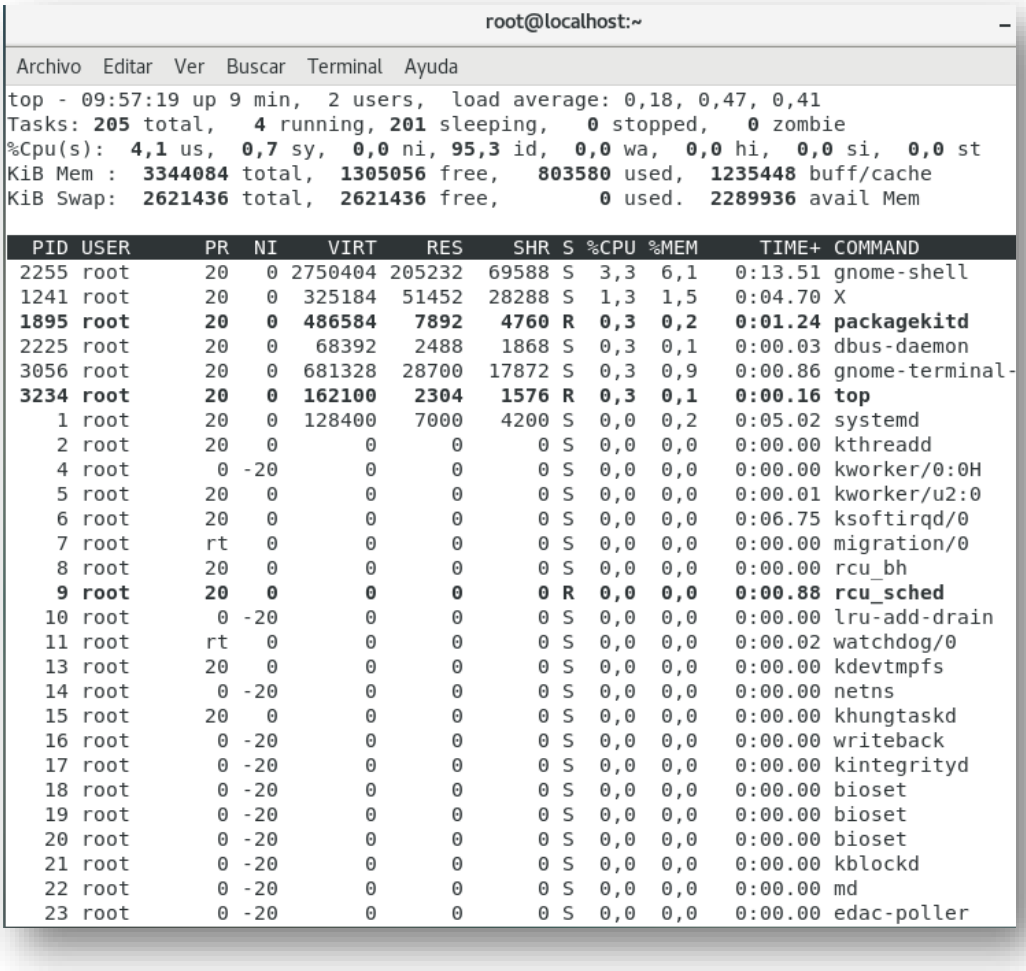
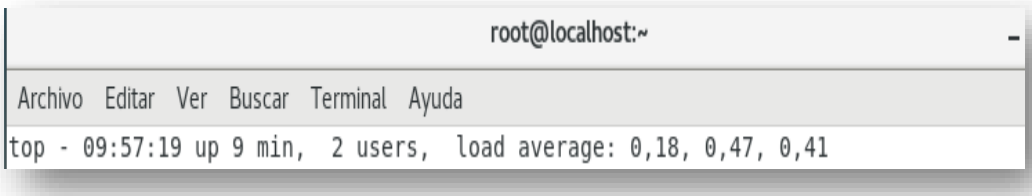


Figura 2.5: Resultado de la ejecución del top, lista de procesos que se están ejecutando.

Interpretación de las diferentes funciones que muestra la lista de procesos que se están ejecutando en el sistema.

Tiempo de actividad y carga media del sistema



En la primera línea nos muestra:

- ✓ Hora actual.
- ✓ Tiempo que ha estado el sistema encendido.
- ✓ Número de usuarios.
- ✓ Carga media en intervalos de 5, 10 y 15 minutos respectivamente.

Tareas

```
Tasks: 205 total,   4 running, 201 sleeping,   0 stopped,   0 zombie
```

La segunda línea muestra el total de tareas y procesos, los cuales pueden estar en diferentes estados.

- ✓ **Running (ejecutando):** procesos ejecutándose actualmente o preparados para ejecutarse.
- ✓ **Sleeping (hibernando):** procesos dormidos esperando que ocurra algo (depende del proceso) para ejecutarse.
- ✓ **Stopped (detenido):** ejecución de proceso detenido.
- ✓ **Zombie:** el proceso no está siendo ejecutado. Estos procesos se quedan en este estado cuando el proceso que los ha iniciado muere (padre).

Estados de la CPU

```
%Cpu(s):  4,1 us,  0,7 sy,  0,0 ni, 95,3 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
```

Esta línea nos muestra los porcentajes de uso del procesador diferenciado por el uso que se le dé.

- ✓ **us** (usuario): tiempo de CPU de usuario.
- ✓ **sy** (sistema): tiempo de CPU del kernel.
- ✓ **id** (inactivo): tiempo de CPU en procesos inactivos.
- ✓ **wa** (en espera): tiempo de CPU en procesos en espera.

- ✓ **hi** (interrupciones de hardware): interrupciones de hardware.
- ✓ **si** (interrupciones de software): tiempo de CPU en interrupciones de software.

Memoria física

```
KiB Mem :  3344084 total,  1305056 free,   803580 used,  1235448 buff/cache
```

- ✓ Memoria total: **(3344084).**
- ✓ Memoria utilizada: **(1305056).**
- ✓ Memoria libre: **(803580).**
- ✓ Memoria utilizada por buffer: **(1235448).**

Memoria virtual

```
KiB Swap:  2621436 total,  2621436 free,        0 used.  2289936 avail Mem
```

- ✓ Memoria total: **(2621436).**
- ✓ Memoria libre: **(2621436).**
- ✓ Memoria usada: **(0).**
- ✓ Memoria disponible: **(2289936).**

Columnas

```
  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 2255 root      20   0 2750404 205232  69588 S  3,3  6,1   0:13.51 gnome-shell
 1241 root      20   0  325184  51452  28288 S  1,3  1,5   0:04.70 X
 1895 root      20   0  486584   7892   4760 R  0,3  0,2   0:01.24 packagekitd
 2225 root      20   0   68392   2488   1868 S  0,3  0,1   0:00.03 dbus-daemon
 3056 root      20   0  681328  28700  17872 S  0,3  0,9   0:00.86 gnome-terminal-
```

PID: es el identificador de proceso. Cada proceso tiene un identificador único.

USER (USUARIO): usuario propietario del proceso.

PR: prioridad del proceso. Si pone RT es que se está ejecutando en tiempo real.

NI: asigna la prioridad. Si tiene un valor bajo (hasta -20) quiere decir que tiene más prioridad que otro con valor alto (hasta 19).

VIRT: cantidad de memoria virtual utilizada por el proceso.

RES: cantidad de memoria RAM física que utiliza el proceso.

SHR: memoria compartida.

S (ESTADO): estado del proceso.

%CPU: porcentaje de CPU utilizado desde la última actualización.

%MEM: porcentaje de memoria física utilizada por el proceso desde la última actualización.

TIME+ (HORA+): tiempo total de CPU que ha usado el proceso desde su inicio.

COMMAND: comando utilizado para iniciar el proceso.

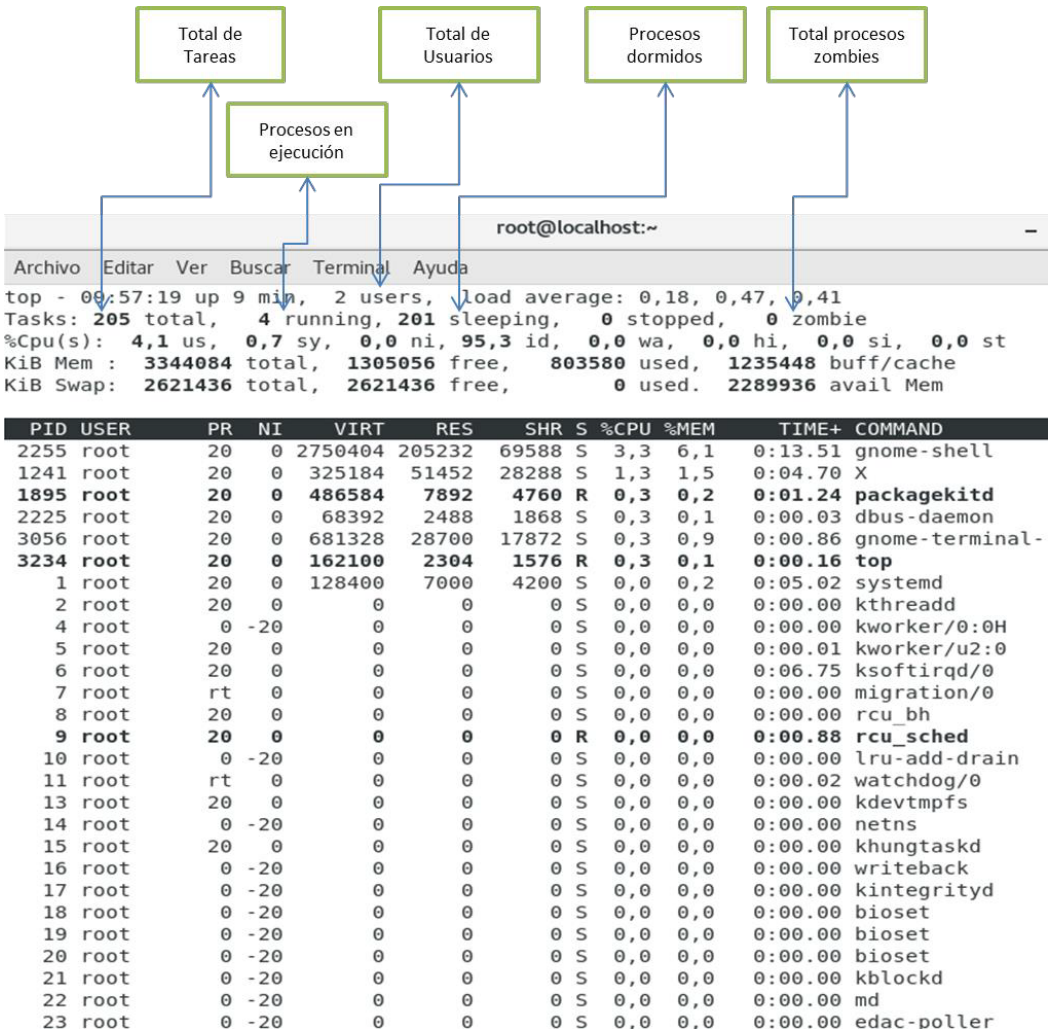


Figura 2.6: Identificación de procesos que se están ejecutando.

De la misma manera se puede identificar el ID del proceso, tenemos información quien está corriendo el proceso y la cantidad de memoria que está usando el proceso, así tambien el valor de CPU usado y el nombre del comando, se encuentra un NI que identifica la prioridad que tiene el proceso, más adelante vamos a ver un ejemplo de cómo cambiar la prioridad del proceso, para salir de esta vista o terminal solo bastara con presionar la tecla **CTRL + C** y recuperamos la terminal.

Procedemos a limpiar la pantalla mediante el comando:

```
[root@localhost ~]# clear
```

Estado de los procesos

Cuando un proceso se está ejecutando, este cambia su estado, el estado de un proceso se define en parte por actividad que esté realizando actualmente ese proceso. Si nos ponemos en el lugar del procesador (CPU), cada proceso ejecuta instrucciones tomadas del conjunto de instrucciones que sólo el CPU puede ejecutar, estas instrucciones son supervisadas por el contador de programa, donde este siempre apunta a la próxima instrucción en ejecutarse, sea esta de un mismo proceso o de otro.

El comportamiento de un proceso se puede enumerar con la siguiente secuencia de estados, según la secuencia de las instrucciones que se están ejecutando, ver diagrama de estados en la

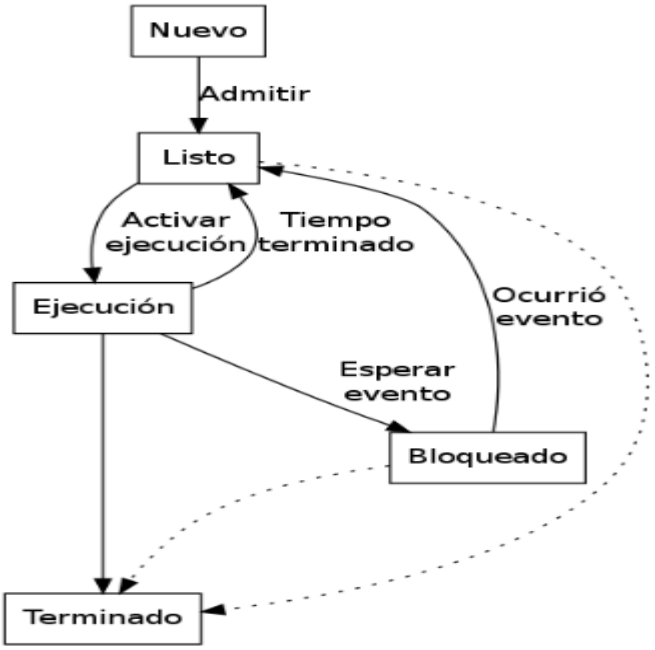


Figura 2.7: Diagrama de estados de los procesos

- **Nuevo** - Se solicitó al sistema operativo la creación de un proceso, y sus recursos y estructuras están siendo creadas.
- **Listo** - Está listo para iniciar o continuar su ejecución pero el sistema no le ha asignado un procesador.
- **En ejecución** - El proceso está siendo ejecutado en este momento. Sus instrucciones están siendo procesadas en algún procesador.
  - Bloqueado - En espera de algún evento para poder continuar su ejecución (aun si hubiera un procesador disponible, no podría avanzar)
  - **Zombie** - El proceso ha finalizado su ejecución, pero el sistema operativo debe realizar ciertas operaciones de limpieza para poder eliminarlo de la lista
- **Terminado** - El proceso terminó de ejecutarse; sus estructuras están a la espera de ser limpiadas por el sistema operativo.

Recuerda que:
Solo un proceso puede ejecutarse en el núcleo de procesador, no importa cuántos núcleos tenga este. Sin embargo, muchos procesos pueden estar listos y en espera.

Práctica 2. Control de tareas en la CPU

En una terminal procedemos a ingresar comandos más robustos como es **ps** para ver los procesos que se están ejecutando en la Shell

```
[root@localhost ~]# ps
```

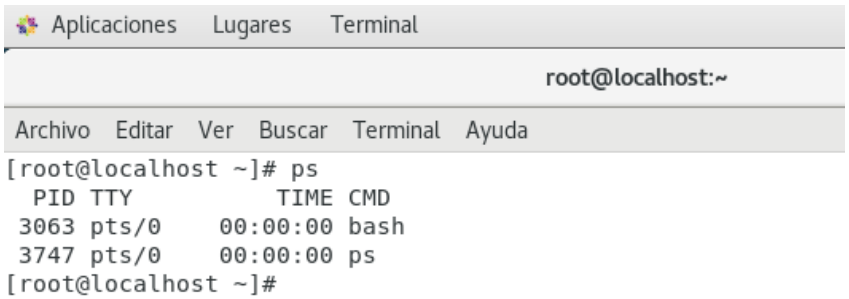


Figura 2.7: Identificación de procesos que se están ejecutando en consola o Shell.

Como se puede observar en la imagen 2.7 nos muestra el ID del proceso y el nombre del proceso

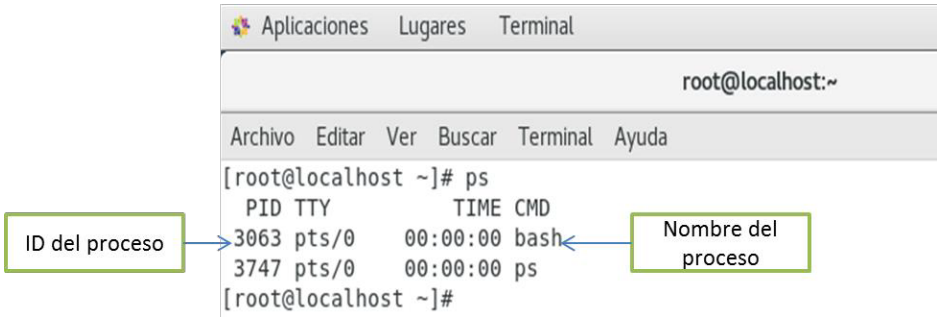


Figura 2.8: Identificación de procesos que se están ejecutando en consola o Shell.

Como podemos apreciar en la figura 2.8, en este caso específico, el proceso **bash** es padre del proceso **ps**, entonces podemos darnos cuenta que solo corre **ps,** que es lo que estamos realizando, pero si deseamos ver una forma más arbolada para identificar quien es padre de quien y quien es hijo de quien se puede utilizar el comando:

```
ps axjf [root@localhost ~]# ps axjf
```

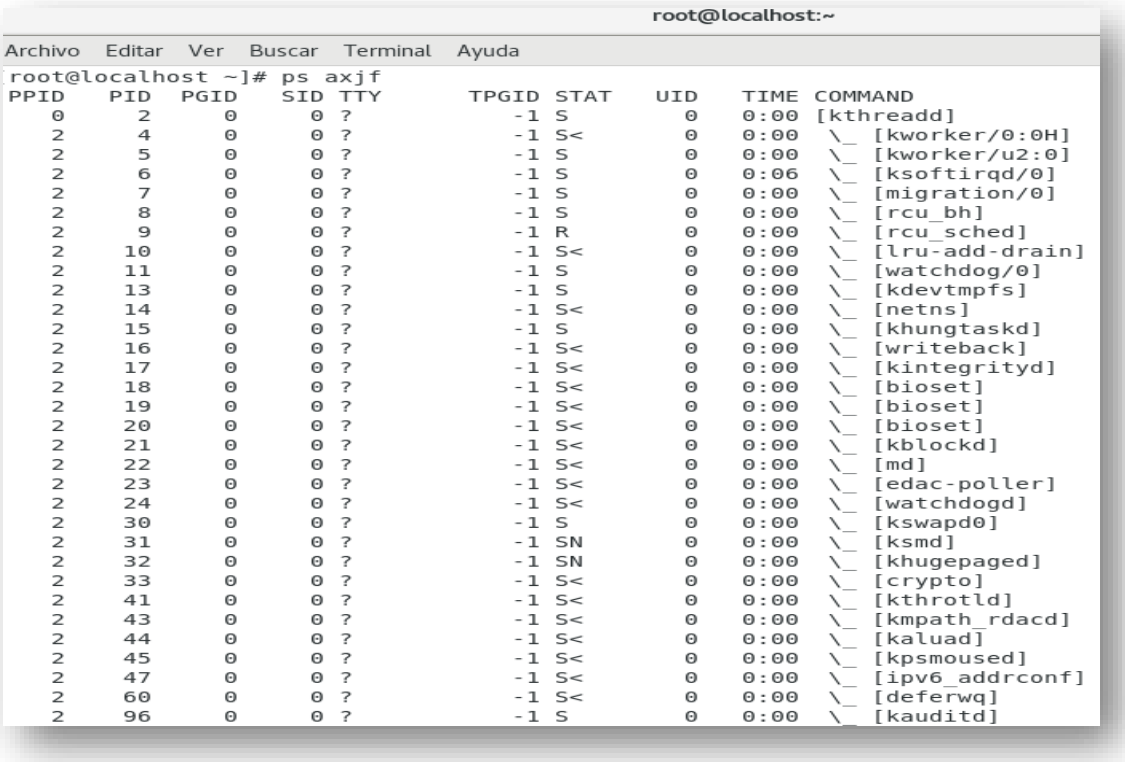


Figura 2.9: Identificación de procesos padre y proceso hijo de manera arbolada.

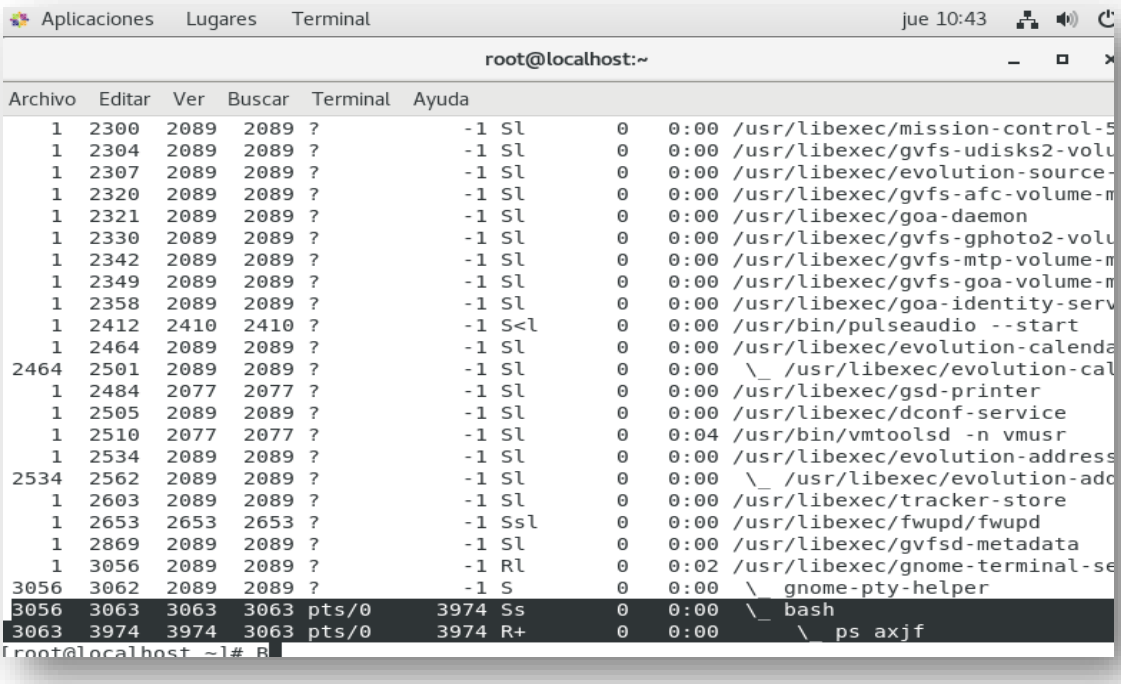


Figura 2.10: Identificación de procesos padre y proceso hijo de manera arbolada.

Como se aprecia en la figura 2.10, el ***proceso ps axjf es hijo del proceso bash*** (ver el área sombreada)

Para tener una mejor apreciación de la estructura de los procesos podemos utilizar un comando **pstree** que nos dará mejor vista de cómo están definidos los procesos para este ejemplo.

Utilizamos el comando:

```
[root@localhost ~]# pstree
```

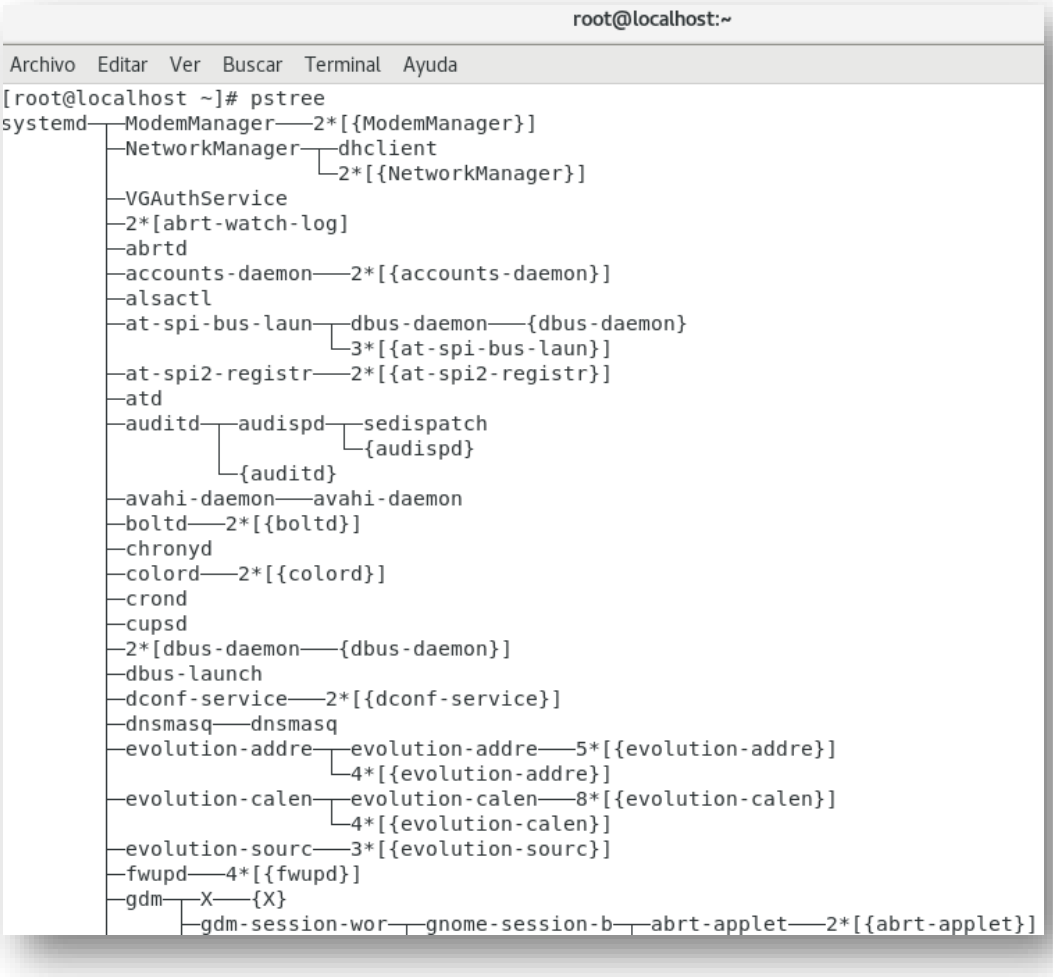


Figura 2.11: Identificación de procesos padre y proceso hijo de manera arbolada mediante el comando pstree.

Podemos observar que la información de los procesos obtenidos mediante el comando **pstree** nos muestra una escena más arboleada de los procesos que se encuentran en ejecución. Esto es una forma de ver las visualizaciones de los procesos.

Cabe indicar que cuando se ejecuta un proceso en la consola este puede estar dentro de dos estados **attack y diattack** lo que significa que un

proceso puede estar atado a la consola o desatado de la consola, para comando **ps** es imposible ver el resultado, pero si ejecutamos comando que tardan en ejecutarse podremos tener una mejor apreciación, ejemplo ejecutamos el comando **find / > /dev/null** lo que va suceder que el proceso se va quedar bloqueado, va bloquear la **consola**

```
[root@localhost ~]# find / > /dev/null
```

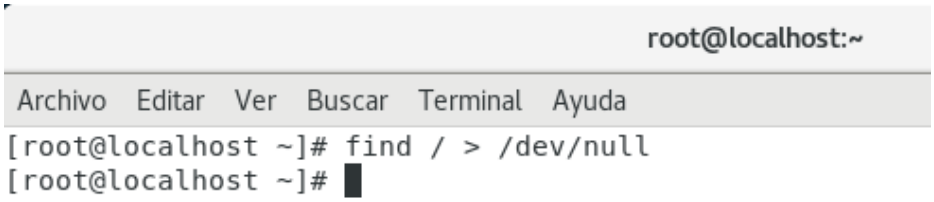


Figura 2.12: Ejecución del comando find.

Este es un proceso apegado a la consola y hasta que termine no va devolver el **promt**, como solución para evitar el bloqueo de la consola es utilizar el símbolo & al final del comando, esto lo que va hacer es ejecutar el comando en background (**segundo plano**):

```
[root@localhost ~]# find / > /dev/null &
```

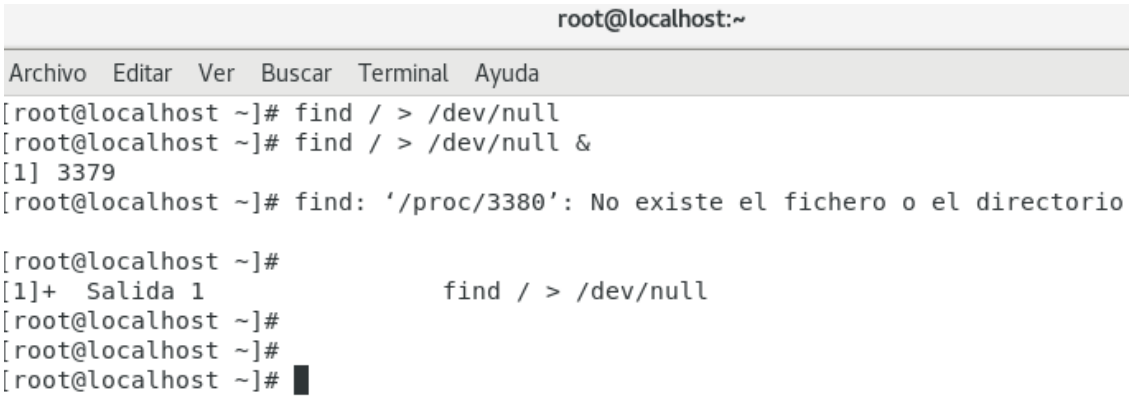


Figura 2.13: Ejecución del comando find pero en segundo plano

De esta manera podemos seguir trabajando en la consola, pero eventualmente el proceso termino.

Esto es fundamental cuando se quiere hacer una combinación de ver un archivo de log mediante el comando **tail** ejemplo: `tail –0f /var/log/apache2/access.log &`, `root@ubuntu:~# tail 0f`

`/var/log/httpd/access.log &` lo que nos va mostrar que el proceso corrió con el número de ID y cuando alguien se conecte al **puerto 80** del servidor apache va aparecer una línea de **log**, para esto vamos a forzarlo desde otra consola ejecutando el comando `telnet localhost 80`

Antes de realizar este tipo de actividad debemos de instalar el servidor web apache y el servidor de telnet

```
[root@localhost ~]# yum install httpd
```

```
[root@localhost ~]# yum install telnet
```

Luego de realizar la instalación de cada uno de los demonios procedemos a iniciar el servidor web apache mediante el comando `systemctl start httpd.service`

Luego procedemos a darle los permisos necesarios para leer en consola los logs mediante el comando:

```
[root@localhost httpd]# chmod -R go+r /var/log/httpd/
```

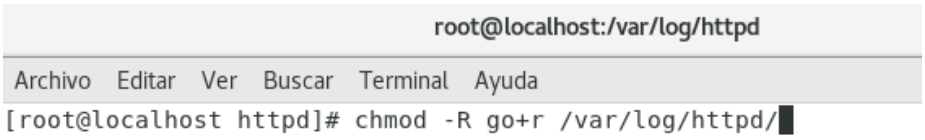


Figura 2.13: configuración de permisos de lectura al archivo logs.

Luego ejecutamos procedemos a ingresar el comando `ls –la` el mismo que permite lista los ficheros que tiene httpd.

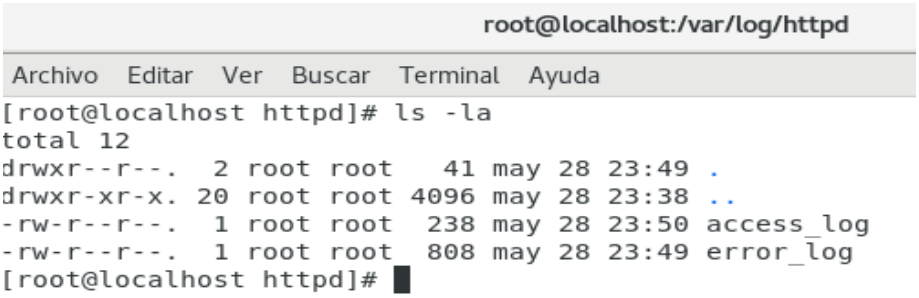


Figura 2.14: Listar los directorios mediante ls -ls.

Podemos observar que el fichero que nos interesa en este caso en **access_log** y procedemos a ejecutar el comando

```
[root@localhost httpd]# tail -0f /var/log/httpd/access_log &
```

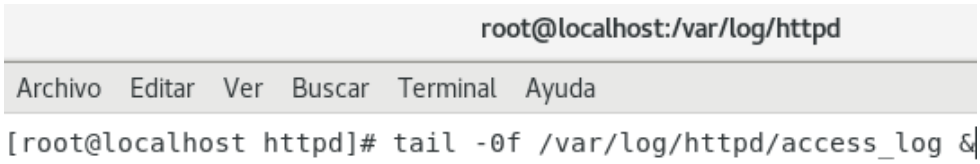


Figura 2.15: Ejecución del comando tail en segundo plano.

Ya en este apartado debemos de abrir dos terminales para realizar la práctica correspondiente

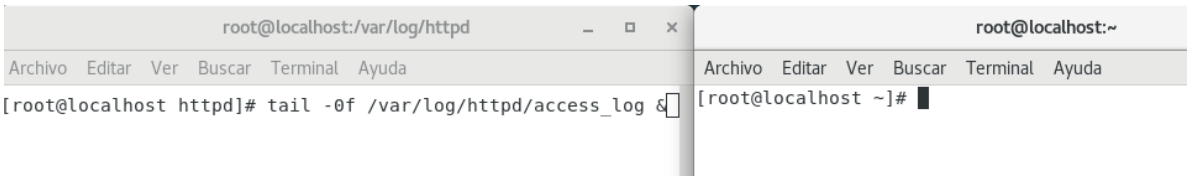
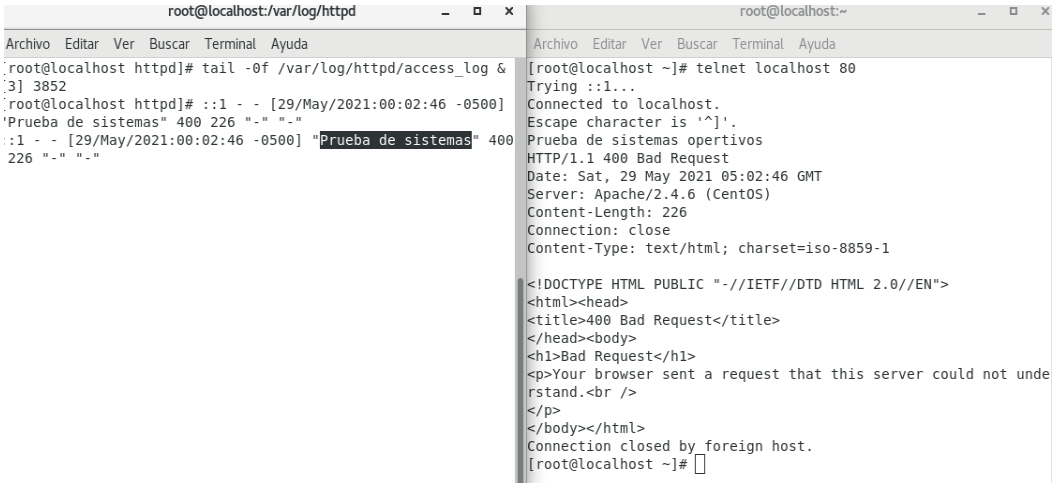


Figura 2.16: Ejecución de dos terminales para la práctica.

Procedemos a ingresar en la consola 2 el comando *telnet localhost puerto 80*, después enviamos el mensaje Prueba de sistemas como muestra



en consola 1

Figura 2.17: Ejecución de dos terminales para la práctica.

Podemos observar que el proceso sigue corriendo en background debido a que puedo seguir utilizando la consola, y, ¿cómo puedo saber que procesos están corriendo en background dentro de la consola?, para esto debemos de utilizar el comando **jobs** (trabajos) el mismo que me dice cuántos trabajos están corriendo

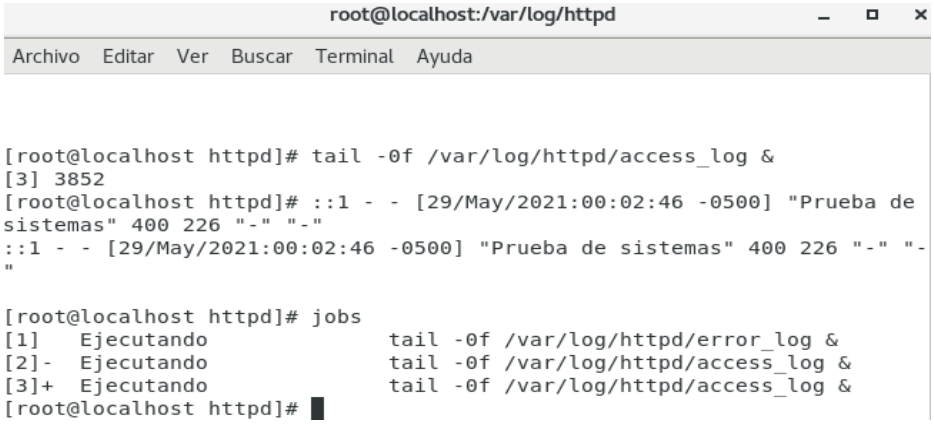


Figura 2.18: Estado de la Ejecución los procesos.

Si deseamos traer el proceso al frente debemos ejecutar el comando **fg foreground (primer plano)** y nos aparecerá en la consola para comprobar que se ha ejecutado correctamente debería bloquearnos la consola.

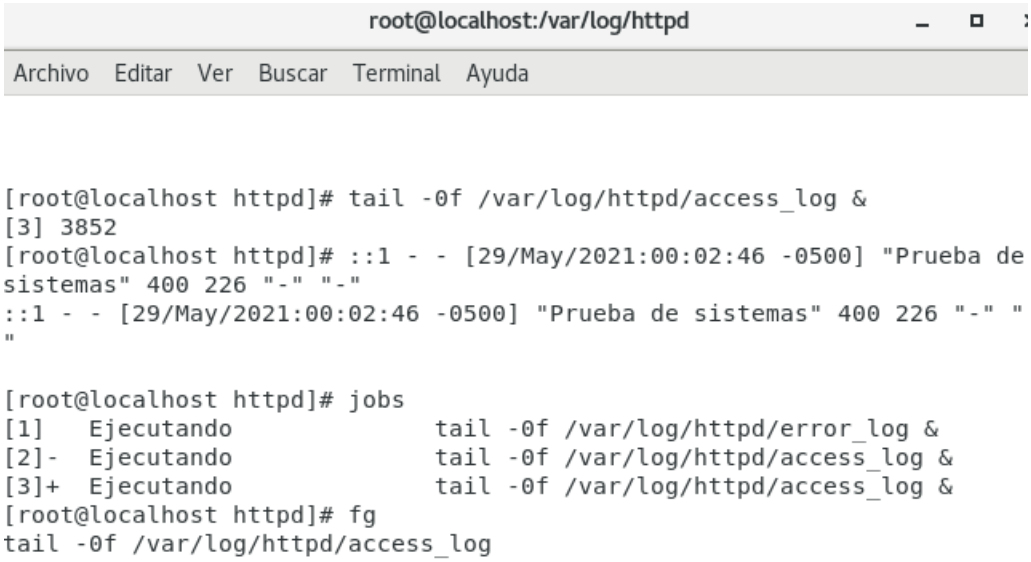


Figura 2.19: Estado de la Ejecución los procesos en fg.

Si desde el otro terminal procesos a ejecutar el comando telnet estos son los resultados

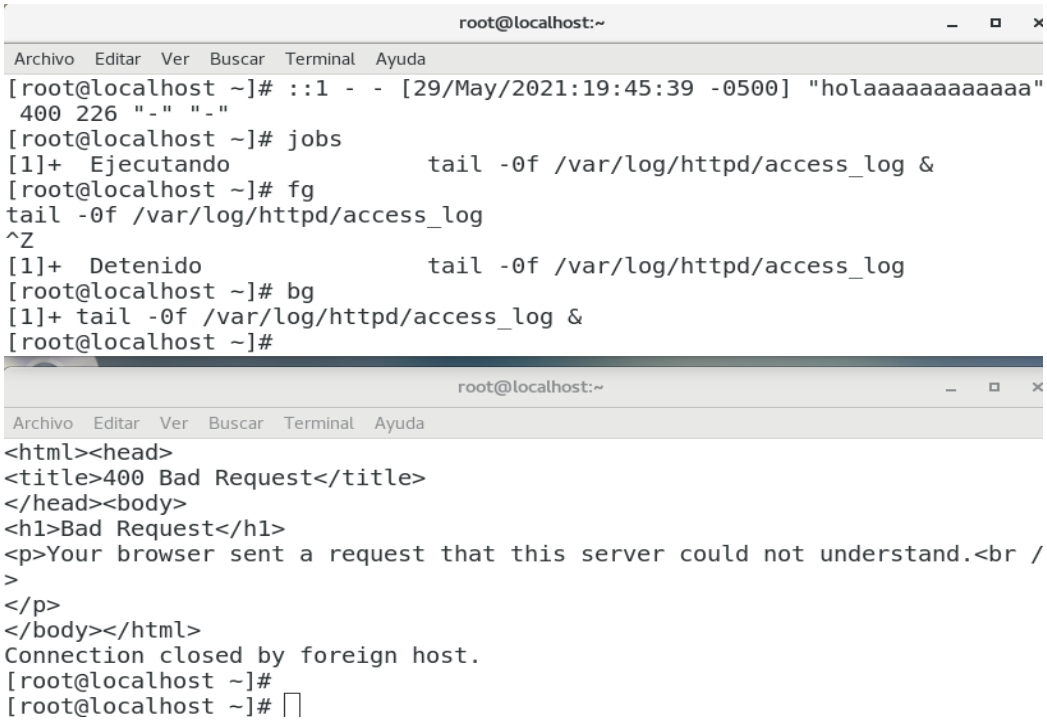


Figura 2.20: resultados de la ejecución del comando telnet en fg.

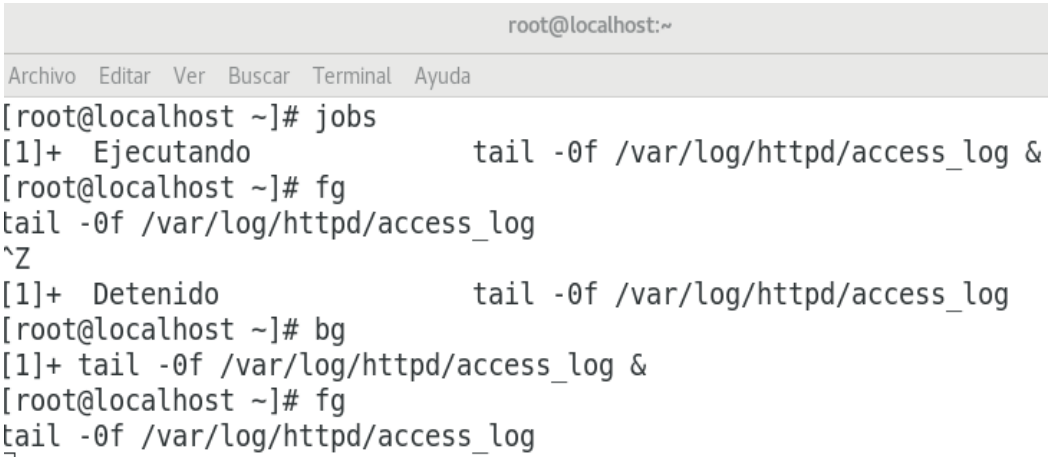


Figura 2.21: Estado de la Ejecución los procesos en fg.

Ejemplo especifico de cómo enviar un proceso a **bg background** y traerlo de vuela a **fg foreground**

Nosotros en este nivel podemos utilizar y comprender como funciona la combinación de teclas **Ctrl + Z y Ctrl + C**

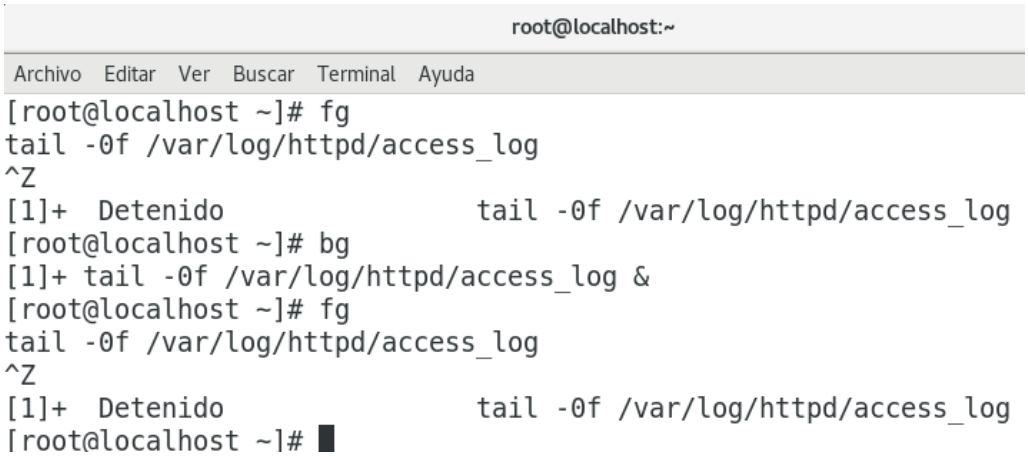


Figura 2.22: Estado de la Ejecución los procesos en fg.

La figura 2.22 muestra el resultado de ejecutar la combinación de teclas `Ctrl + Z`, lo que nos da a entender que este comando está corriendo en el sistema, de echo si escribimos en la consola el comando `ps axf | grep tail`

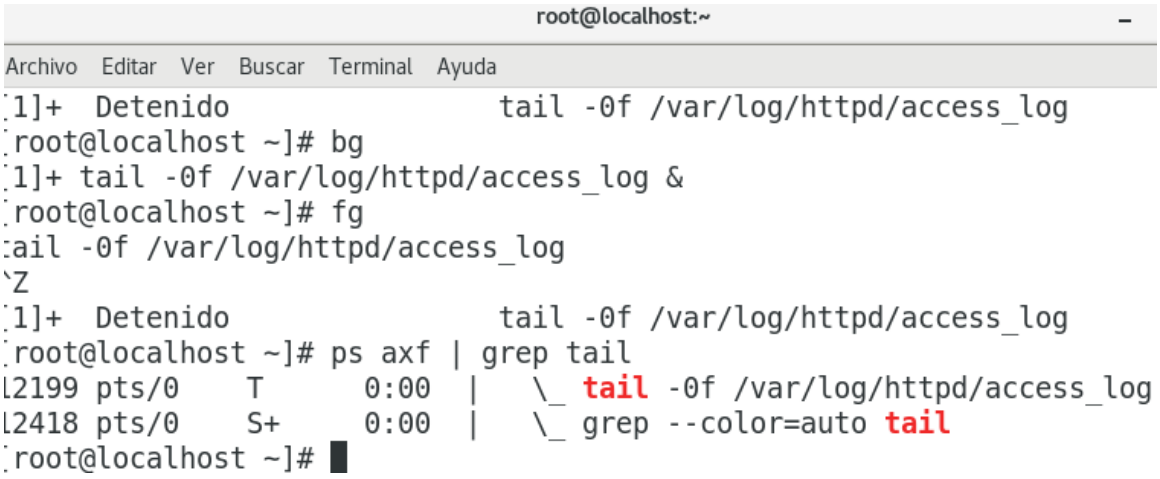


Figura 2.23: Estado de la Ejecución de los procesos en fg.

Podemos observar en la figura 2.23 que el comando **tail** es visible, podemos observar que está pero se encuentra detenido, si ahora vamos a la otra consola 2 y ejecutamos el comando **telnet localhost 80** podemos ver que no se muestra nada en la consola debido a que el procesador no le está otorgando **ciclos de CPU** a ese comando, entonces para enviarlo a **background ejecutamos el comando bg**

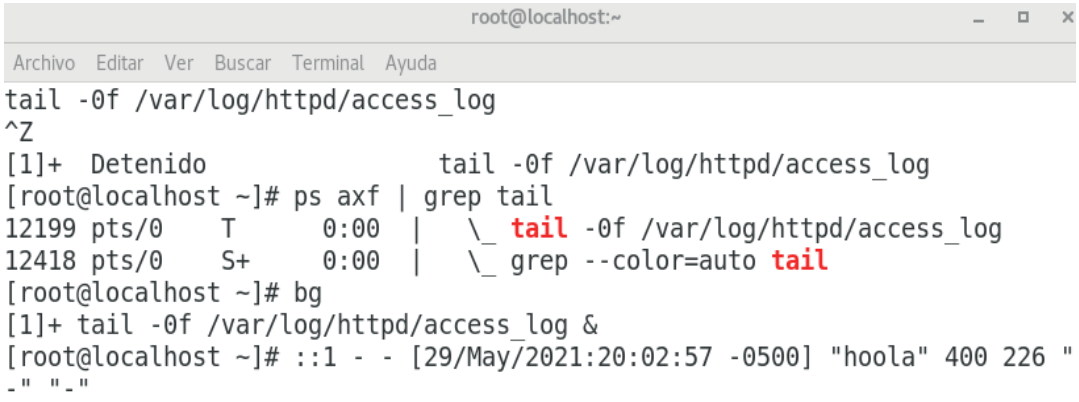


Figura 2.24: Estado de la Ejecución los procesos en ciclos de CPU en bg.

Como observamos el proceso se envió a **background** mediante el comando **bg** y de manera automática podemos ver que muestra la línea que se encolo luego de haber realizado o enviado el comando **telnet**, si ahora mediante la línea de telnet volvemos a ejecutar el mismo comando telnet podremos observar en consola los resultados.

Mucha atención, con el comando **foreground fg** lo traigo hacia adelante el proceso, con **Ctrl + Z** lo detenemos y con el comando **background bg** lo enviamos devuelta al fondo, con el comando jobs podemos ver en qué estado se encuentra

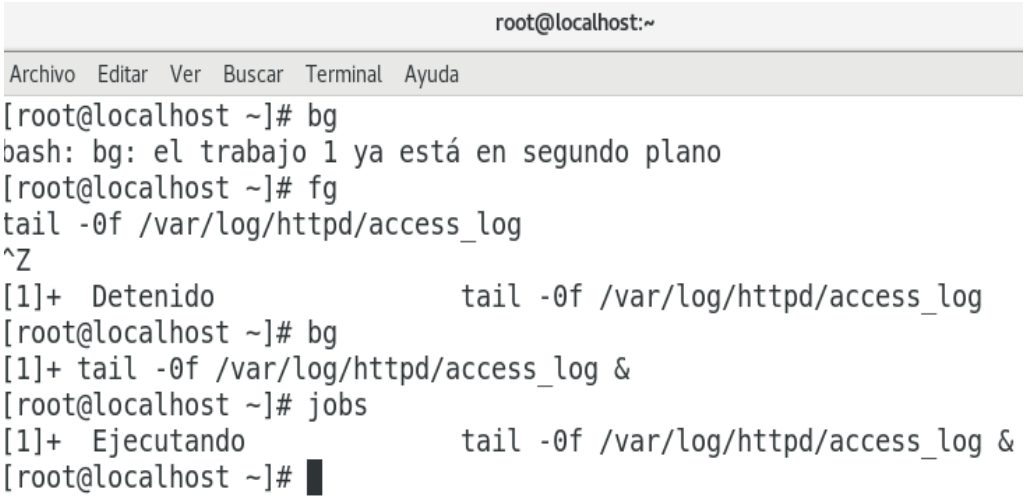


Figura 2.25: Estado de la Ejecución los procesos en ciclos de CPU en bg.

Existe un comando que podemos ejecutar es **Ctrl + r**, ahora podemos ver con un ejemplo cómo funciona, necesitamos cargar un segundo proceso,

para esto ejecutamos el comando `[root@localhost ~]# tail -0f /var/log/httpd/access_log |grep -i access_log` este comando se ejecutara en el **foreground** ahora debemos presionar la combinación de teclas **Ctrl + z y** luego escribir el comando **jobs** para ver los estados de los trabajos

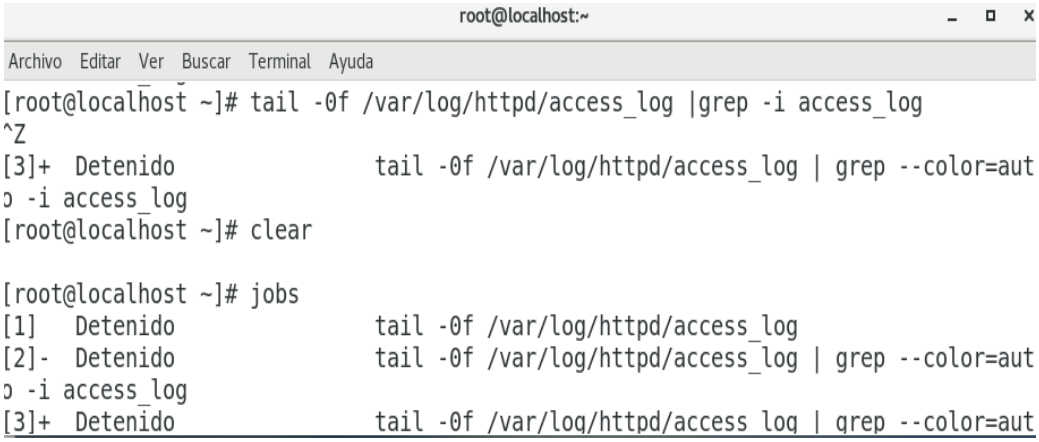


Figura 2.26: Ejecución de procesos anidados.

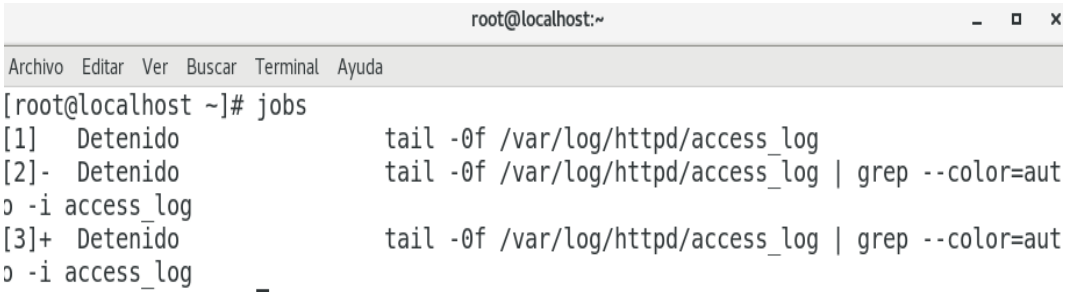


Figura 2.27: Revisión de los trabajos en ejecución fg.

Ahora nos apareen que hay 3 programas detenidos y aparece uno con el signo + ese que aparece con el signo + será el que se seleccionara por defecto al escribir el comando **foreground fg**

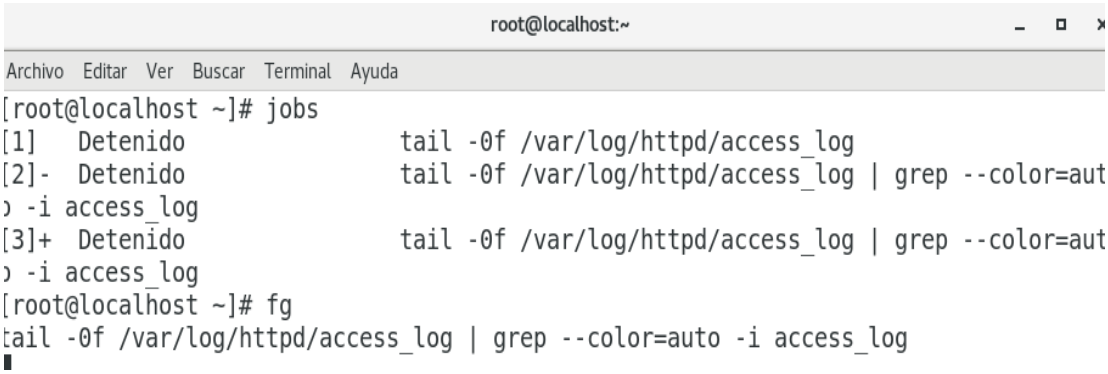


Figura 2.28: Revisión de los trabajos procesos detenidos bg.

Y al presionar **CTRL + Z** y escribir el comando **bg** y luego **jobs** veremos los resultados que ahora se encuentra en estado de ejecutándose

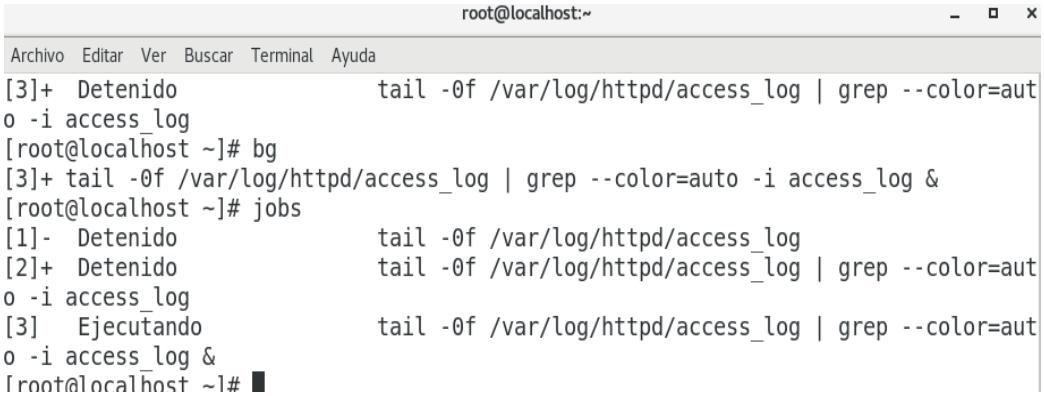


Figura 2.29: Revisión de los trabajos procesos ejecutándose bg.

Si ahora escribimos el comando fg nos traerá el proceso [2]+ que se puede ver que está en estado detenido.

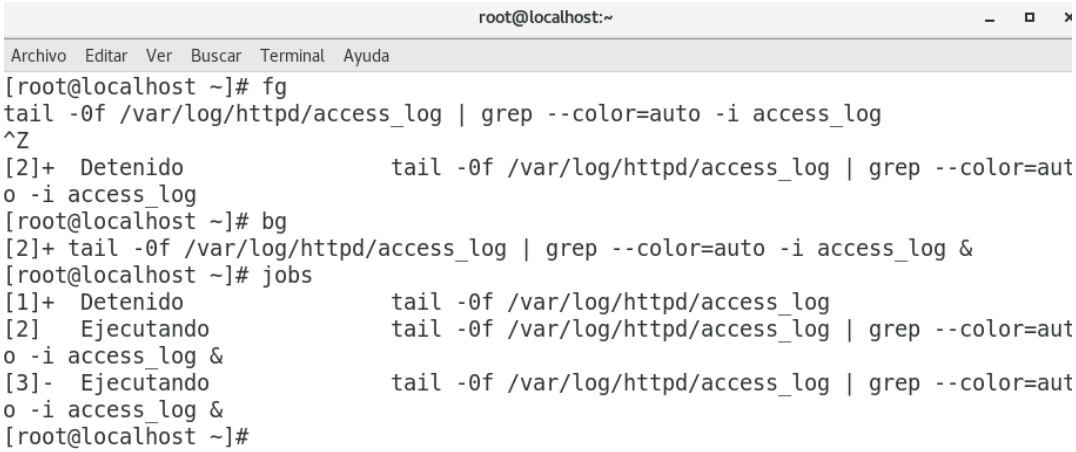


Figura 2.30: Recuperación de procesos ejecutándose fg

Cabe indicar que se puede especificar el número de ID para definir que procesos se pueden traer al frente mediante el comando **fg** y que procesos podemos enviar al fondo **fg**

```
[root@localhost ~]# fg 1
tail -0f /var/log/httpd/access_log
```

Figura 2.31: Recuperación de procesos ejecutándose en bg a fg

```
root@localhost:~
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
o -i access_log &
[root@localhost ~]# fg 1
tail -0f /var/log/httpd/access_log
^Z
[1]+  Detenido                tail -0f /var/log/httpd/access_log
[root@localhost ~]# bg 1
[1]+ tail -0f /var/log/httpd/access_log &
[root@localhost ~]# jobs
[1]   Ejecutando              tail -0f /var/log/httpd/access_log &
[2]-  Ejecutando              tail -0f /var/log/httpd/access_log | grep --color=aut
o -i access_log &
[3]+  Ejecutando              tail -0f /var/log/httpd/access_log | grep --color=aut
o -i access_log &
[root@localhost ~]#
```

Figura 2.32: Procesos ejecutándose en bg.

Como podemos observar en la figura tenemos los tres procesos corriendo en el **background**, ahora podemos matar cualquier de estos procesos mediante el comando **kill %** número de ID del proceso ejemplo **kill %1**

```
[root@localhost ~]# kill %1
```

```
[root@localhost ~]# kill %1
[1]   Terminado               tail -0f /var/log/httpd/access_log
[root@localhost ~]# jobs
[2]-  Ejecutando              tail -0f /var/log/httpd/access_log | grep --color=aut
o -i access_log &
[3]+  Ejecutando              tail -0f /var/log/httpd/access_log | grep --color=aut
o -i access_log &
[root@localhost ~]#
```

Figura 2.33: Matar Procesos mediante el comando kill %1.

Podemos observar que el proceso [1] se encuentra en estado de Terminado

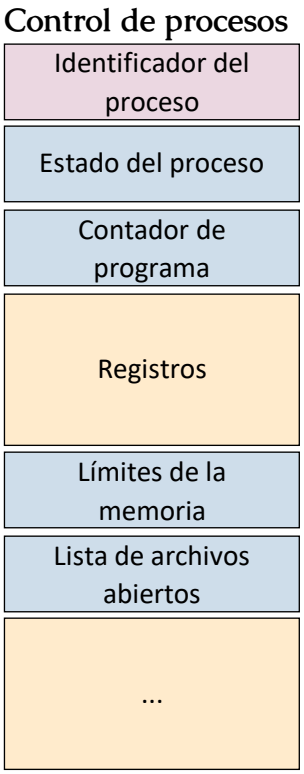


Figura 2.34: PCB - Bloque de Control de Procesos

Cada proceso es representado en el sistema operativo por un **bloque de control de proceso** (*Process Control Block, PCB*), el cual es un registro especial, en donde un sistema operativo agrupa toda la información asociada con un proceso específico durante todo su tiempo de ejecución, por lo tanto, el sistema operativo crea una nueva PCB cada vez que se inicia un nuevo proceso, la información asociada al proceso durante su ejecución en la PCB, son:

- **Identificador del proceso** (*Process Identificador, PID*) – El Process ID, es un número entero que asigna el kernel de algunos sistemas operativos para identificar un proceso de forma única.
- **Estado del proceso (Process state)** – El estado puede ser nuevo, listo, en ejecución, en espera, detenido, etc.
- **Contador de programa (Program counter)** – El contador indica la dirección de la siguiente instrucción que se ejecutará para este proceso.

- **Registros de CPU (CPU registers)** – Los registros varían en número y tipo, dependiendo de la arquitectura de la computadora. Incluyen acumuladores, registros de índice, punteros de pila y registros de uso general, además de cualquier información de código de condición. Junto con el contador del programa, esta información de estado debe guardarse cuando se produce una interrupción, para permitir que el proceso continúe correctamente después cuando se reprograme su ejecución.
- **Planificación de la CPU (CPU scheduling)** – Esta información incluye una prioridad de proceso, punteros a las colas de planificación y cualquier otro parámetro de planificación.
- **Gestión de memoria (Memory management)** – Esta información puede incluir elementos tales como el valor de los registros base y límite y las tablas de páginas, o las tablas de segmentos, dependiendo del sistema de memoria utilizado por el sistema operativo.
- **Estadísticas del proceso (Accounting information)** – Esta información incluye la cantidad de CPU y tiempo real utilizado, límites de tiempo, números de trabajo o proceso, etc.
- **Información de estado de E/S (I/O status information)** – Esta información incluye la lista de dispositivos de E/S asignados al proceso, una lista de archivos abiertos, etc.

En resumen, la PCB simplemente sirve como depósito de todos los datos necesarios para iniciar o reiniciar un proceso, junto con algunos datos de control numérico.

Práctica 3. Prioridades de los procesos.

Para esto debemos de ingresar el comando top `[root@localhost ~]# top` e identificar el parámetro **NI** (En él se muestran una lista de valores como: 0, 0, 0 -20) estos valores son la prioridad que tienen los procesos,

esto se lo puede agregar de manera manual, como hacemos esto, mediante la variable o comando que se le conoce como **[root@localhost ~]# nice**

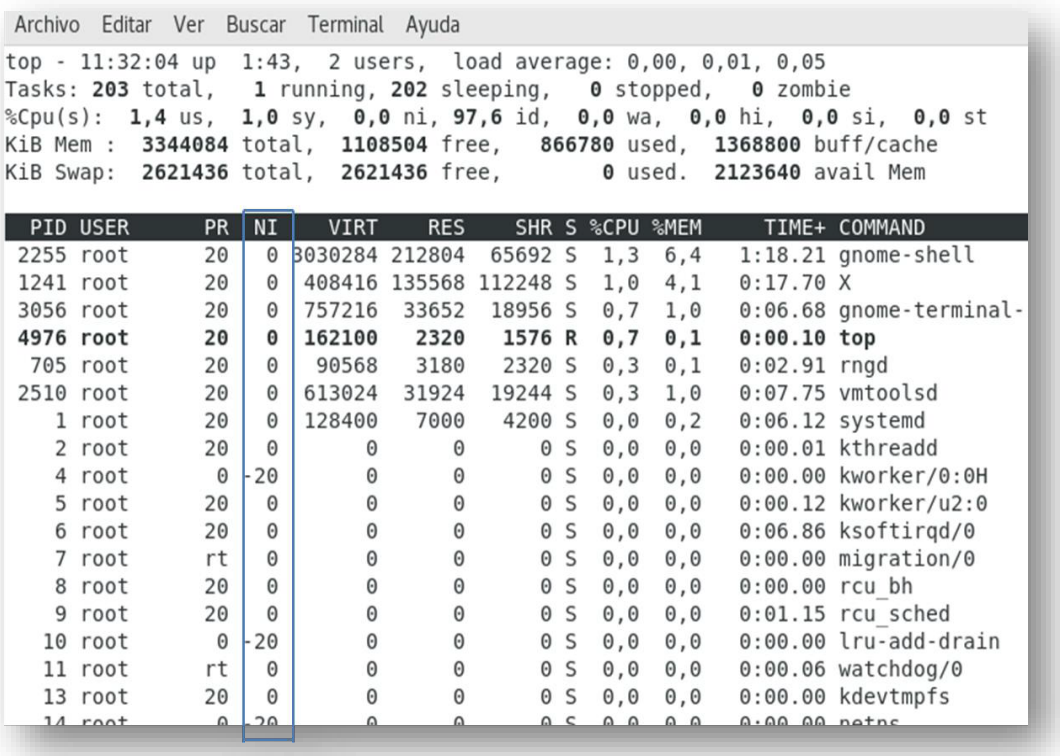


Figura 2.36: Identificación de prioridad de procesos comando nice.

Para proceder a cambiar la prioridad de los procesos de **-20** manualmente utilizando el comando **NICE,** cabe indicar que los valores varían de **-20 a 19** siendo **-20** el de mayor prioridad y 19 el de menor prioridad en este caso a el **proceso gedit** le vamos a colocar el valor de **10**, ejemplo **nice -10 gedit**

[root@localhost ~]# nice -10 gedit

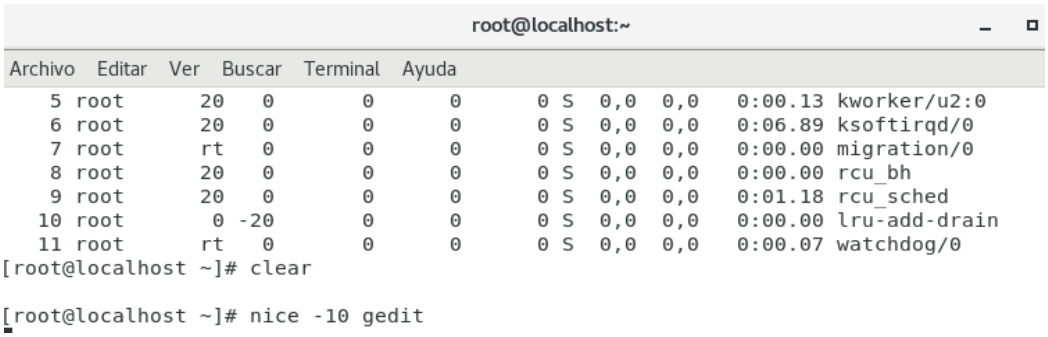


Figura 2.37: Prioridad de procesos comando nice.

Para ver los cambios realizado ejecutamos el comando ps –fl –C ”gedit”

```
[root@localhost ~]# ps -fl -C "gedit"
```

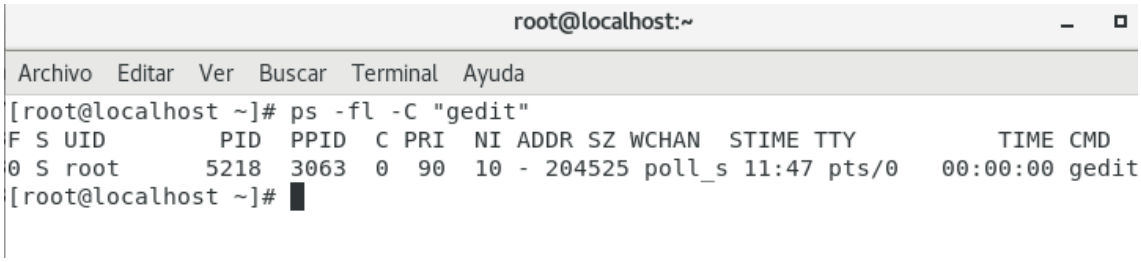


Figura 2.38: Prioridad de procesos establecida comando nice.

Como podemos apreciar la ejecución de la instrucción anterior nos trae el procedo **gedit**, con el identificador **PID 5218** y con **NICE** de **10** que fue lo que se realizó.

Para cambiar la prioridad de un proceso en ejecución se utiliza el siguiente comando

Debe tener permiso de administrador, ejecutamos el comando **sudo renice –n 6 –p 5218**, donde **renice** realiza un cambio en caliente del proceso, **-n identifica la nueva prioridad número de la nueva prioridad en este caso 6 y –p** especifica el **identificador del proceso en este caso 5218**

```
[root@localhost ~]# renice -n 6 -p 5218
```

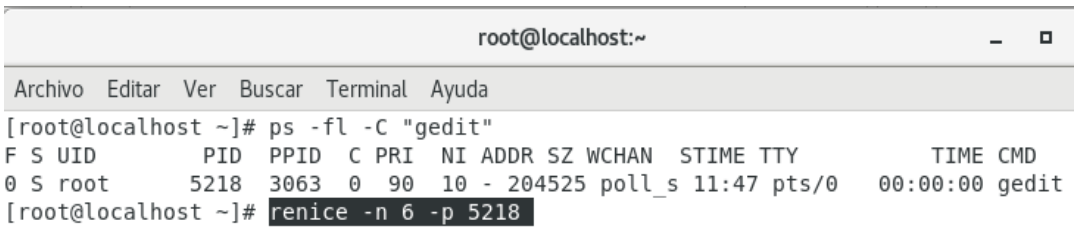


Figura 2.39: Cambio de prioridad de proceso en caliente comando renice.

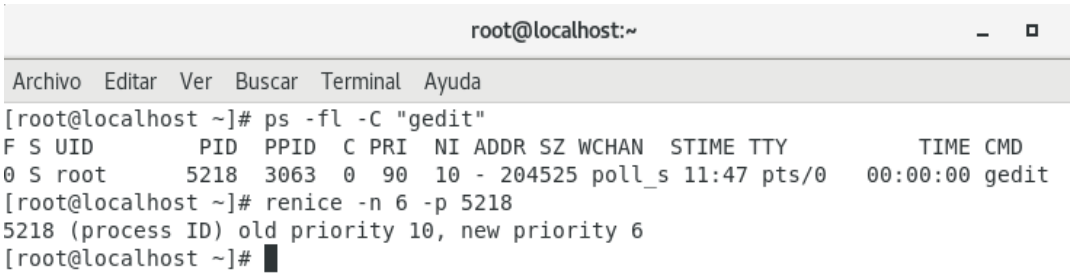


Figura 2.40: Resultado de la ejecución del cambio de prioridad del proceso comando renice.

Para ver la nueva prioridad del proceso solo basta con ejecutar el comando `ps –fl –C “gedit”`

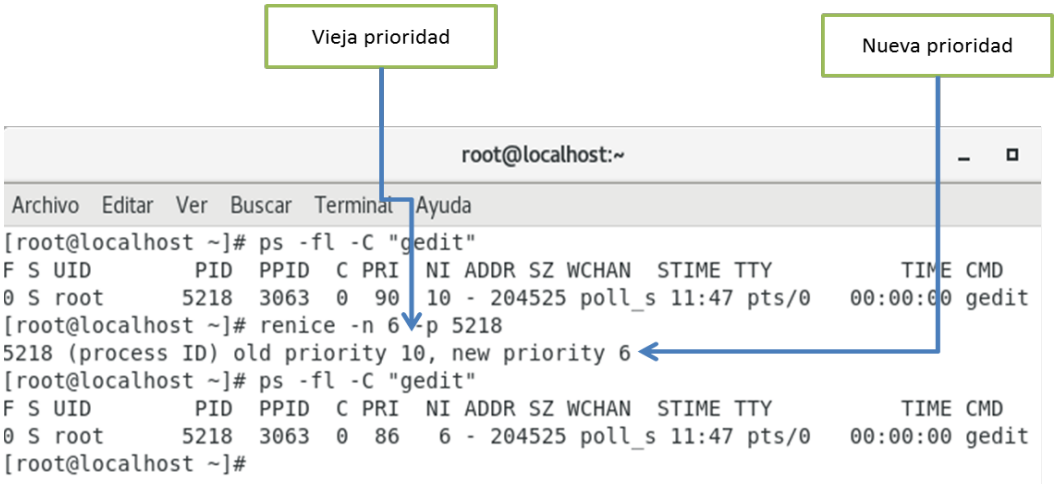


Figura 2.41: Identificación del proceso con el Nuevo NI.

Descripción de los resultados:

Como se aprecia en la imagen nos dice que la vieja **prioridad era 10** y que ahora la nueva **prioridad es 6.**

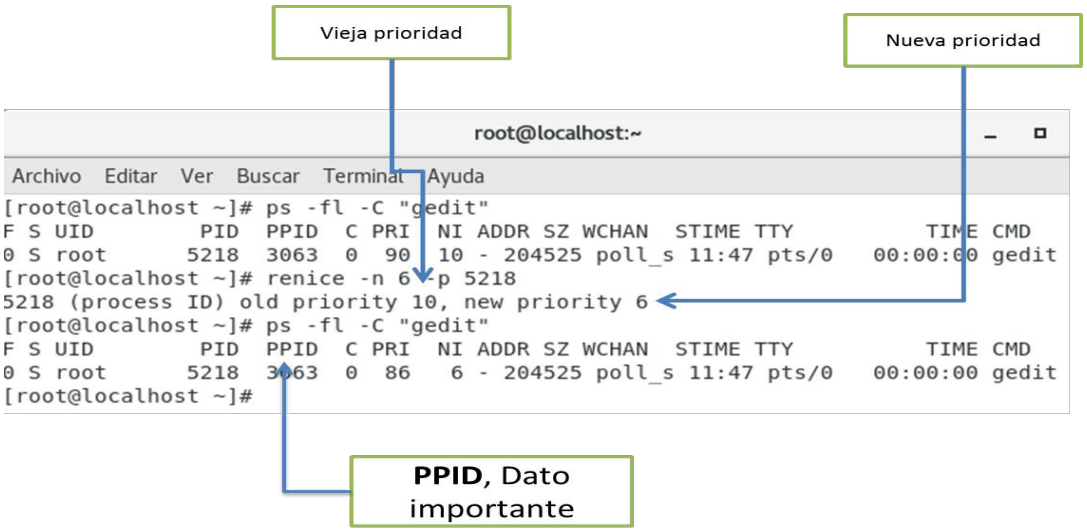


Figura 2.42: Identificación del proceso con la nueva prioridad 6 y la vieja prioridad 10.

En la imagen podemos identificad una variable llamada PPID. El PPID es conocido como el Parent Process ID.

PID (Process ID) y PPID (Parent Process ID)

A cada proceso le corresponderá un número **PID** que le identifica totalmente. **Es decir en un mismo momento es imposible que existan dos procesos con el mismo PID**. Lo mismo que todos los procesos tienen un atributo PID que es el número de proceso que lo identifica en el sistema también existe un atributo llamado **PPID. Este número se corresponde con el número PID del proceso padre.** Todos los procesos deben de tener un proceso que figure como padre pero entonces que ocurre si un padre muere antes que alguno de sus hijos. En estos casos el proceso 'init' del cual hablaremos en seguida adoptará a estos procesos para que no queden huérfanos.

Práctica 4. Identificación de los procesos

Como liberar la terminal de los procesos

Para realizar este procedimiento debemos de ejecutar el siguiente comando en la terminal `[root@localhost ~]# gedit &` con el signo **ampersand** al final del comando lo que estamos diciendo al proceso que lo envié a segundo plano o **background** con el objeto de liberar la terminal y continuar trabajando sobre ella.

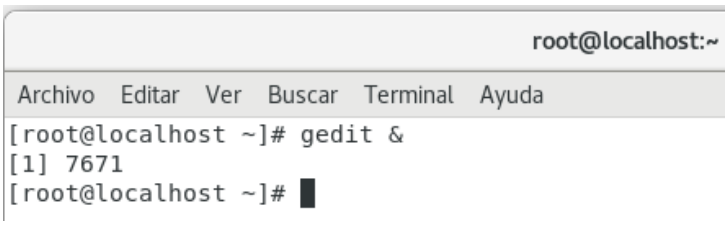


Figura 2.43: Envió de un proceso a background.

El procedimiento anterior muestra la imagen de ejecución del **proceso gedit &,** el mismo que muestra el **PID** y el número de trabajos **Jobs,** esto se da porque el proceso o trabajo está corriendo en **bg background**, entonces toma el nombre de **Jobs.**

Para poder revisar todos los **trabajos o jobs** simplemente debemos de ejecutar el siguiente comando sobre la terminal jobs `[root@localhost ~]# jobs`

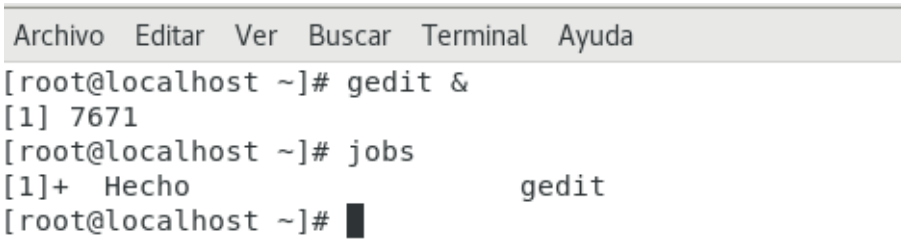


Figura 2.44: Resultado de la ejecución el comando Jobs y los procesos a background.

Si deseamos conocer el PID del proceso debemos de ejecutar el comando **jobs –l** `[root@localhost ~]# jobs –l`

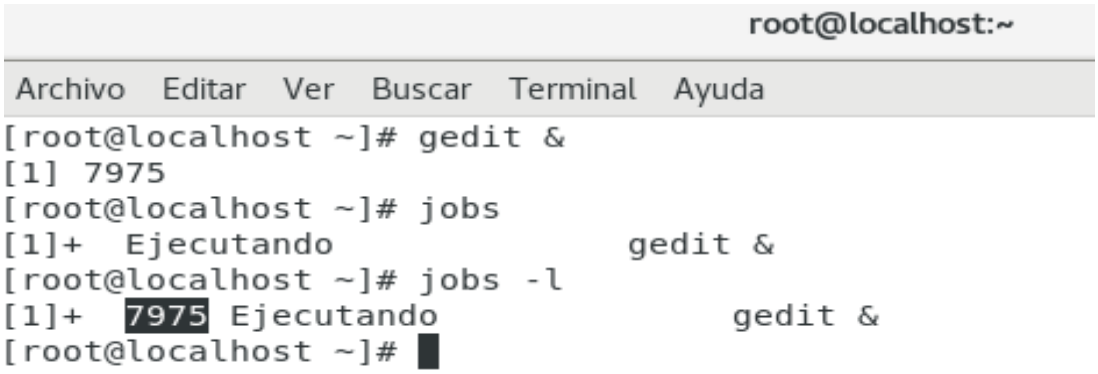


Figura 2.45: Obtención del PID del trabajo Jobs.

Si solamente se desea conocer el PID del trabajo podemos ejecutar simplemente el comando jobs –p

`[root@localhost ~]# jobs –p`

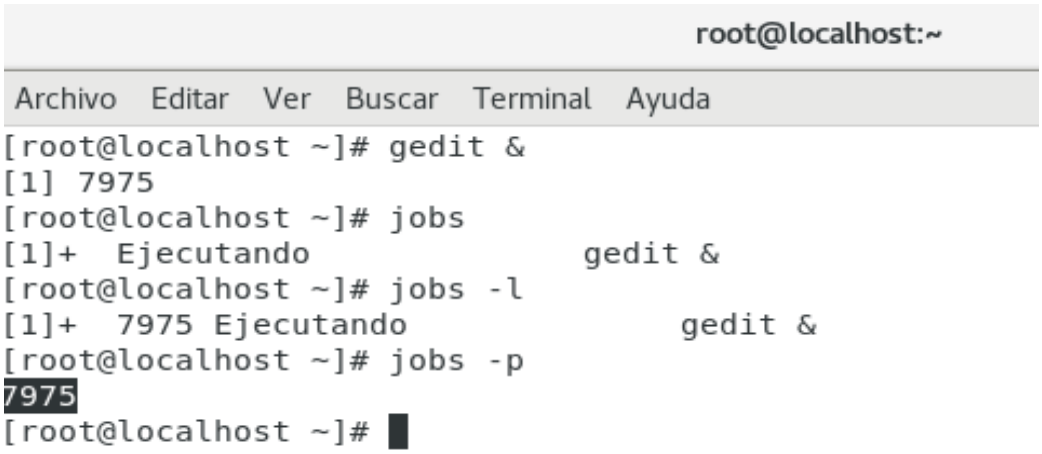


Figura 2.46: Obtención del PID del trabajo Jobs.

Práctica 5. Como regresar los procesos de background al frente

Para traer los procesos que se están ejecutando en **background** al frente **foreground** debemos de ejecutar el comando `[root@localhost ~]# fg 1` o en vez de **fg 1** pueden utilizar **%1**

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
root@localhost ~]# gedit &
1] 7975
root@localhost ~]# jobs
1]+  Ejecutando                 gedit &
root@localhost ~]# jobs -l
1]+  7975 Ejecutando                 gedit &
root@localhost ~]# jobs -p
7975
root@localhost ~]# fg 1
gedit
```

Figura 2.47: Mecanismos para reanudar la ejecución de un proceso en primer plano, proceso a foreground.

Con los comandos **fg foreground** y **bg background** es posible manipular procesos que estén suspendidos temporalmente, ya sea porque se les envió una señal de suspensión como **STOP (20)** o porque al estarlos ejecutando se **presionó CTRL-Z.** Entonces para reanudar su ejecución en primer plano usaríamos **fg.**

CRTL + C Termina el proceso

CRTL +Z Envía un proceso a segundo plano

```
                                                root@localhost:~
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
[root@localhost ~]# ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
root      5277  3056  0 11:48 pts/1    00:00:00 bash
root      8638  5277  0 15:59 pts/1    00:00:00 ps -f
[root@localhost ~]#
```

Figura 2.48: Identificación de procesos en consola.

Con el comando `[root@localhost ~]# ps –f` se puede obtener información muy importante como **UID** nombre de quien ejecuto el proceso, **PID** identificador del proceso hijo, **PPID** identificador del proceso padre, **C** que representa el uso del **CPU, TIME** que identifica cuando se inició el proceso.

Es posible obtener más información de los procesos y el consumo de recursos, esto mediante el comando ps –u ejecutado dentro de una consola `[root@localhost ~]# ps –u`

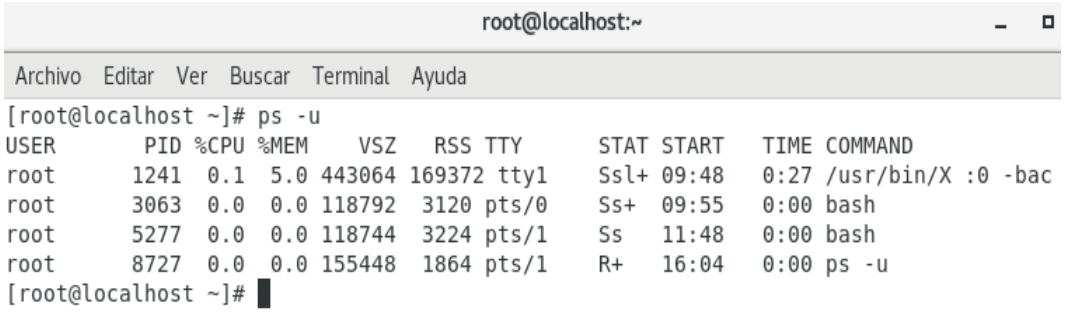


Figura 2.49: Identificación de procesos y consumo de recursos en consola.

Como se puede observar en la imagen el comando **ps –u** muestra el porcentaje de **CPU** y porcentaje de memoria RAM y los estados del proceso, **S** cuando está esperando y **R** cuando está ejecutando dentro del procesador, **D** cuando se encuentra esperando pero ocasionado por una operación de entrada o de salida, **T** si esta pausado y **Z** que es un proceso **zombi** es decir que ya está terminado pero sigue en la tabla de procesos del sistema, como se aprecia en la imagen en la columna **STAT** se identifica **Ssl+** esto se da cuando el proceso está compuesto por hilos del procesador y **R+** identifica que el proceso se ejecuta en primer plano y **S** si es un proceso padre.

Multiprogramación y conmutación de tareas.

Hoy en día una característica muy importante de los sistemas operativos es el poder ejecutar varios programas a la vez, porque un solo programa no puede tener a la CPU o a los dispositivos de E/S ocupados todo el tiempo, sin contar que hoy en día los usuarios están ejecutando varias aplicaciones simultáneamente. Por lo tanto, la multiprogramación (*multiprogramming*) incrementa el uso del CPU evitando que este se encuentre desocupado, y aumenta la satisfacción del usuario, además de organizar los programas para que la CPU siempre tenga al menos uno en ejecución.

Con esta técnica, el sistema operativo mantiene varios procesos (Figura 2.50) residiendo en memoria simultáneamente (***RECUERDE:*** *todo programa en ejecución reside en memoria*) Por lo tanto el sistema operativo selecciona y ejecuta uno de estos procesos, donde este puede quedar en espera mientras alguna otra actividad del CPU se está cumpliendo, ej.: una operación de E/S.

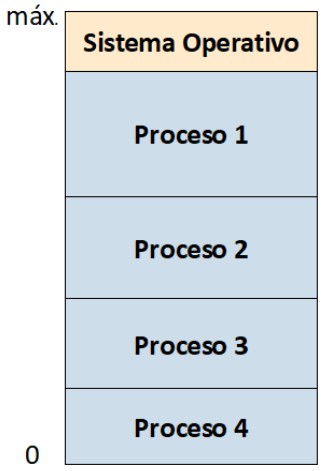


Figura 2.50: Distribución de la memoria en un sistema multiprogramado

Por esto, en un sistema no multiprogramado, estos solo ejecutan un proceso a la vez, permitiendo a la CPU se pueda quedar inactiva por algún tiempo, hasta que este proceso sea reanudado, Ahora, la ventaja de un sistema multiprogramado con respecto a un no multiprogramado es que el sistema operativo simplemente cambia a otro proceso y lo ejecuta. De

este modo, cuando un proceso necesita esperar, el CPU, no se queda ocioso, sino que conmuta a otro proceso, y así sucesivamente. Eventualmente, el primer proceso termina de esperar y recupera la CPU, completando su ejecución, Por lo tanto, mientras haya al menos un proceso que necesite ejecutarse o ejecutándose, la CPU nunca estará inactiva. Por lo cual la multiprogramación trae consigo, la multitarea (*multitasking*).

Los sistemas multitarea, ejecutan varios procesos en la CPU cambiando entre ellos, esta actividad produce cambios con tal frecuencia, que proporcionan al usuario un tiempo de respuesta rápida de tal forma que los usuarios pueden interactuar con cada programa, mientras este está en ejecución. Normalmente el tiempo de ejecución de un proceso, desde que inicia y hasta que termina) es corto o simplemente dura hasta que el usuario necesite una operación de E/S (Cura, 2020, P.229).

Sabías que:

Las operaciones de entrada y salida (E/S) son interactivas, es decir que el usuario está interactuando con el sistema enviando datos hacia este a través de periféricos de entrada como el teclado, el ratón, etc., de igual forma que el sistema operativo responde a este mostrando información al usuario a través de dispositivos de salida como el monitor, parlantes, impresora, etc., procesos que se ejecutan a la "velocidad de un humano", por lo cual puede demorar en ejecutar y completarse. Porque a pesar de ser rápido para un ser humano, es MUY lento para el CPU.

Por lo tanto, mientras un usuario completa un proceso de E/S, el sistema operativo en vez de dejar al CPU quedarse inactivo, rápidamente lo pondrá a realizar otro proceso.

Para poder tener varios procesos en memoria al mismo tiempo, es necesario que ser realice una gestión en la memoria, tema que veremos más adelante en la última unidad de esta asignatura, adicional a esto,

recordando lo que hemos visto anteriormente, varios procesos podrían estar listos (*ready, según el diagrama de estado de la*

Figura), por lo cual es sistema operativo debe elegir qué proceso se ejecutará en ese instante, para esto el sistema operativo realiza una toma de decisiones a través de una planificación de la CPU (*CPU scheduling*). Por lo cual, en un entorno multitareas, el sistema operativo debe garantizar un tiempo de respuesta razonable, y a veces un proceso puede necesitar más memoria de la que existe físicamente, lo cual podría causar un desbordamiento de esta, entonces un método que utiliza el sistema operativo para evitar esto, es utilizar la **memoria virtual** (tema que veremos en la última unidad).

Práctica 6. Procesos anidados

El comando para ingresar a una sesión dentro de la misma sesión es:

```
[root@localhost ~]# su -
Último inicio de sesión:sáb may 29 21:13:10 -05 2021en pts/0
[root@localhost ~]# ps
  PID TTY          TIME CMD
12099 pts/0    00:00:00 bash
13716 pts/0    00:00:00 su
13719 pts/0    00:00:00 bash
13761 pts/0    00:00:00 su
13764 pts/0    00:00:00 bash
13803 pts/0    00:00:00 ps
[root@localhost ~]#
```

Figura 2.51: Ejecución de procesos anidados en consola.

Mediante el comando **ps f** podemos observar la sesión creada de una manera arborizada:

```
[root@localhost ~]# ps f
```

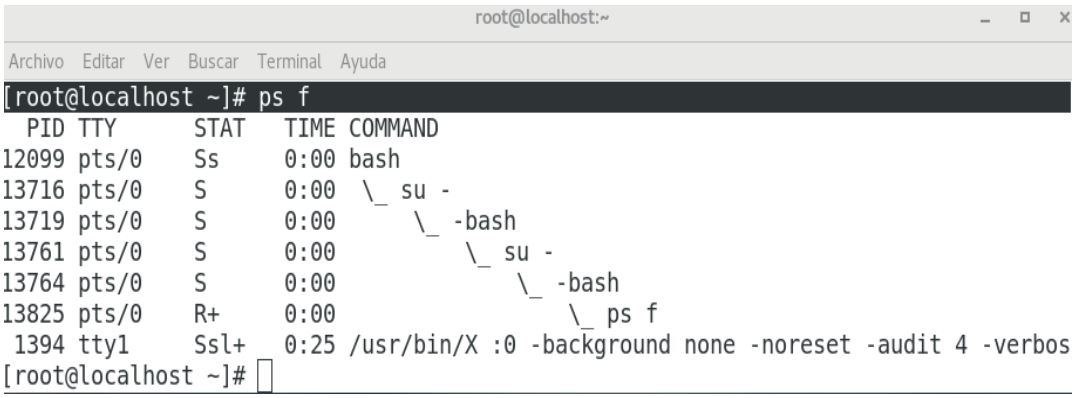


Figura 2.52: Revisión de procesos anidados en consola.

Si ahora hacemos la búsqueda reversa y ejecutamos el comando

```
[root@localhost ~]# tail -0f /var/log/httpd/access_log &
```

podremos observar que se encuentra en **background** ese proceso

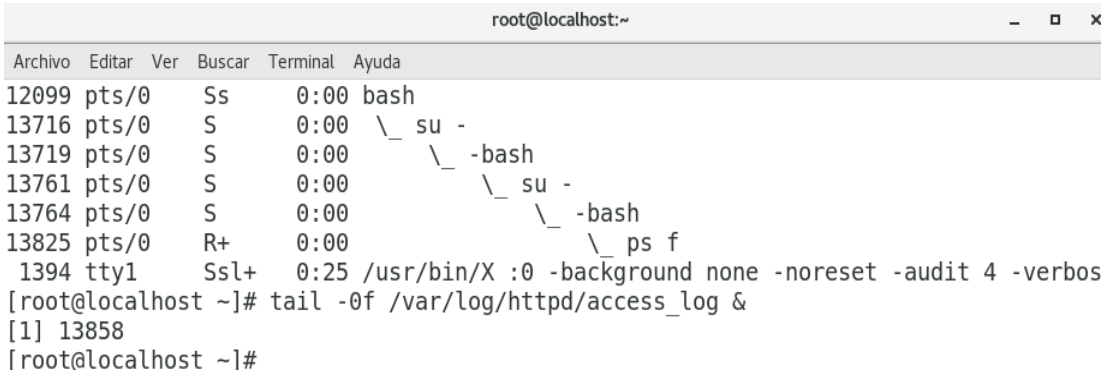


Figura 2.53: Revisión de procesos de manera reversa en consola

Ahora si ejecutamos el comando `[root@localhost ~]# ps axf | grep -i tail` podremos ver el proceso

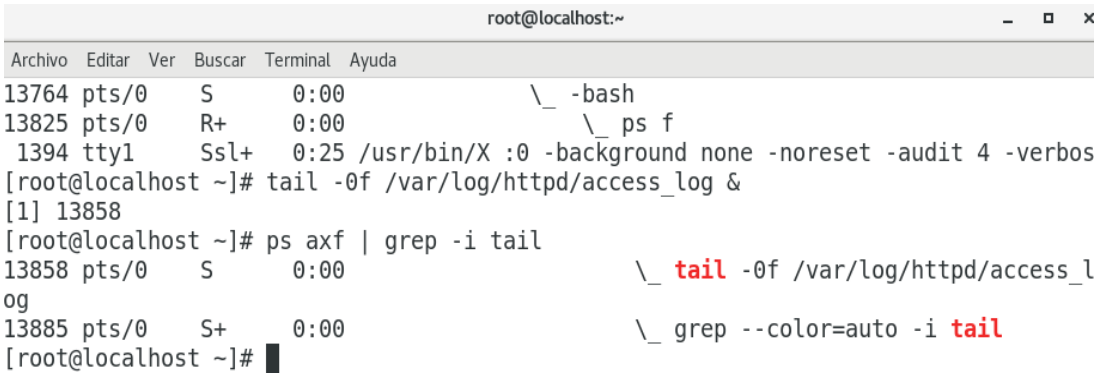


Figura 2.54: Revisión de procesos de manera reversa en consola

Si ahora si salimos de la consola mediante el comando **exit** o **Ctrl + d** buscamos el mismo proceso y debería de haber desaparecido.

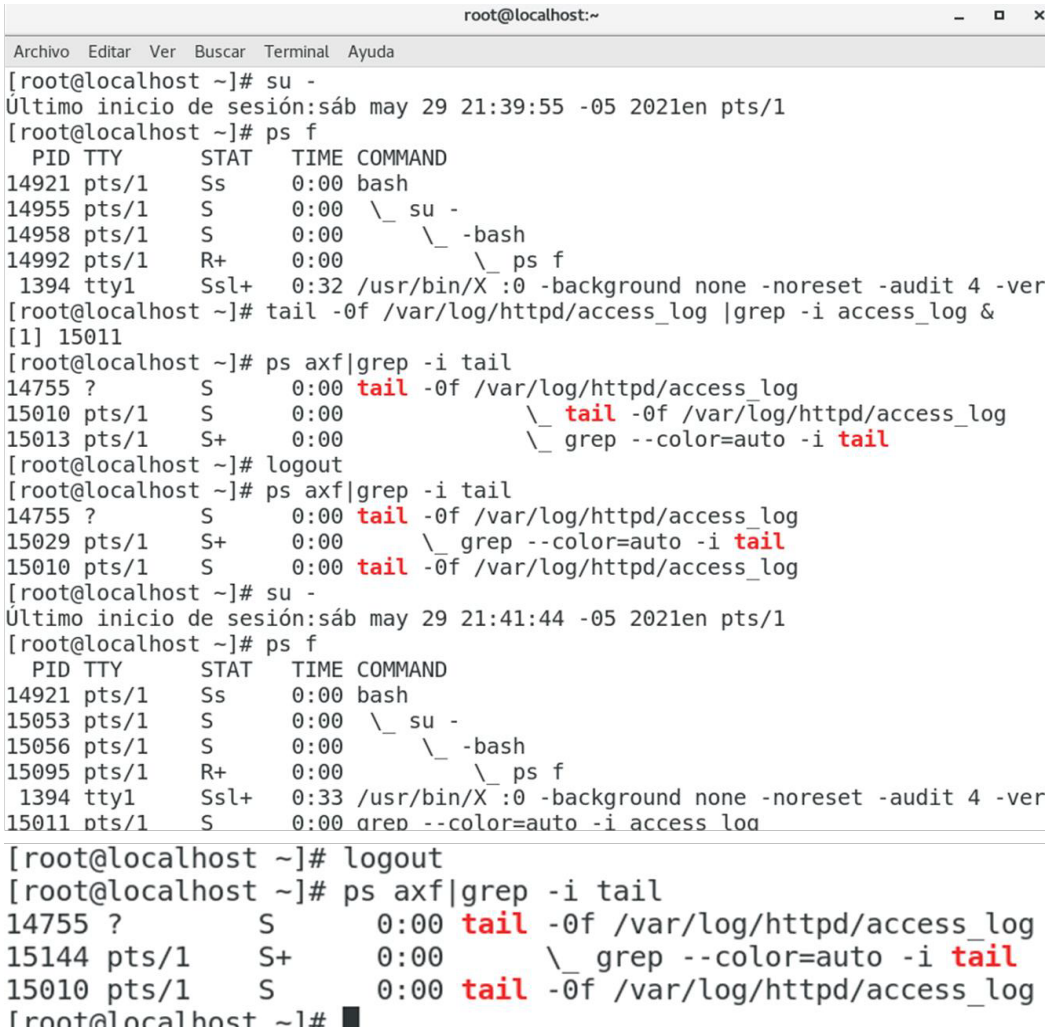


Figura 2.54: Ejecución de bash anidado en consola

Como se aprecia en la imagen hemos generado un bash anidado invocamos un proceso y como vemos el proceso a desaparecido, ¿por que sucede esto?, sencillo esto sucede porque se murio el proceso padre de esos procesos hijos. Al momento que se realizó un exit todos los procesos que dependen del proceso bash mueren.

Práctica 7. Desacoplar procesos

Para evitar que los procesos mueran al momento que el proceso padre muere se debe utilizar el comando **disown** que tiene un significado como de desheredar

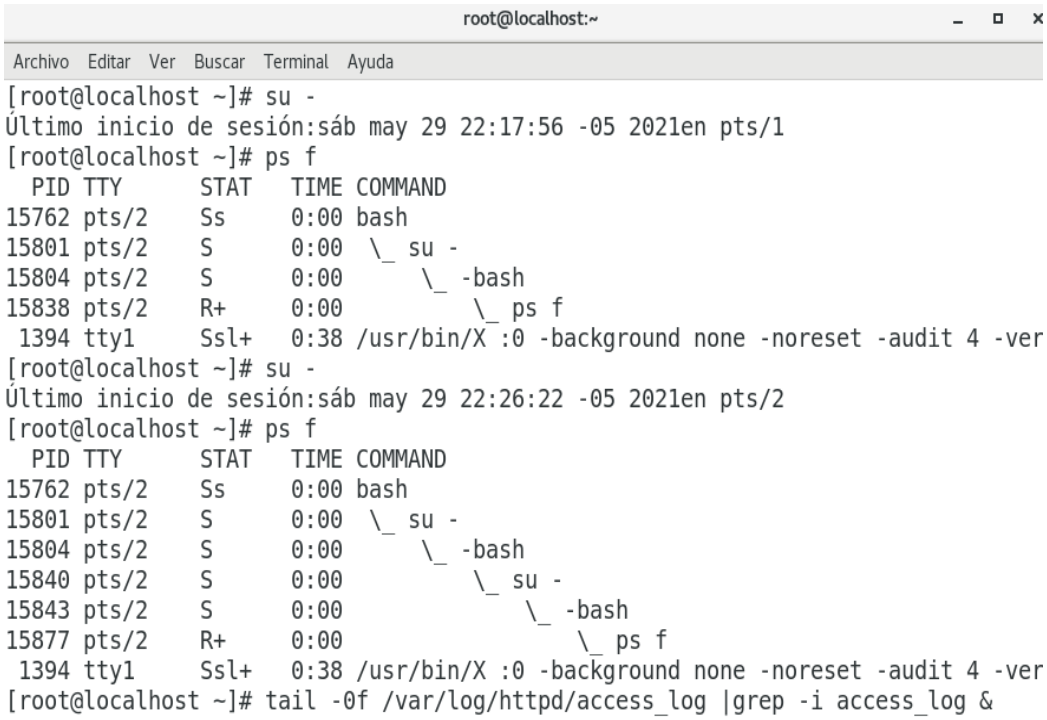


Figura 2.55: Ejecución del comando disown en consola.

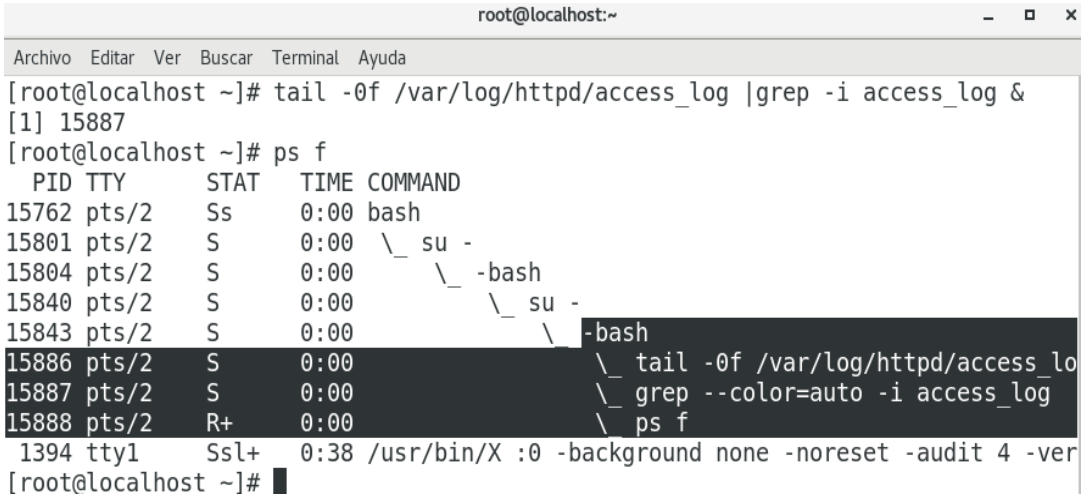


Figura 2.56: Revisión del estado del trabajo en consola

Revisamos el trabajo mediante el comando: `[root@localhost ~]# jobs`

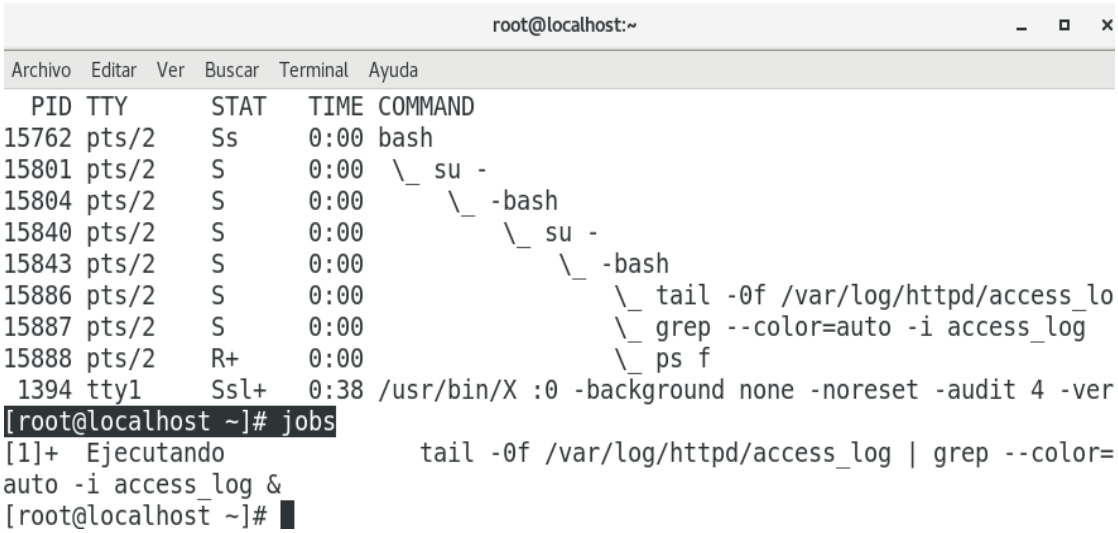


Figura 2.57: Revisión del estado del trabajo en consola

Luego de haber realizado los pasos anteriores procedemos a ejecutar el comando **disown** el mismo que me permitirá mantener el proceso hijo aun habiendo muerto el proceso padre `[root@localhost ~]# disown`

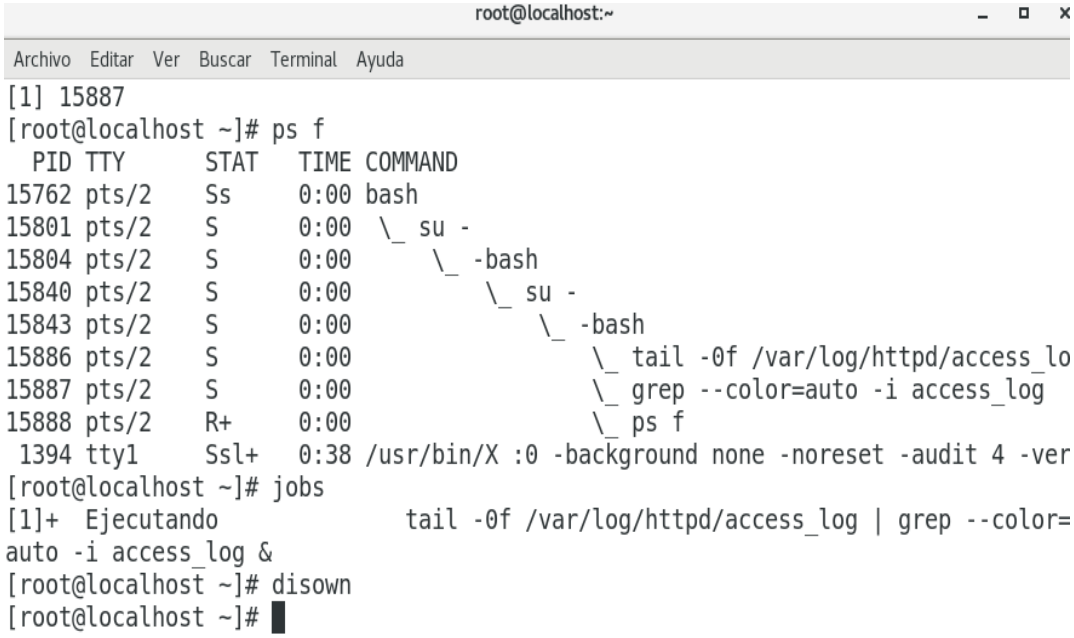


Figura 2.58: Ejecución del comando disown en consola

Como podemos apreciar la ejecución del comando `disown` no muestra ninguna salida, no toma **estándar input** no emite **estándar output** puede llegar a emitir **estándar error** pero con que nos haya devuelto la consola con el cursos es suficiente, para comprobar lo acontecido escribimos sobre la consola **Ctrl + d** o **exit**

```
[root@localhost ~]# disown
[root@localhost ~]# exit
logout
```

Figura 2.59: Resultado del comando disown en consola

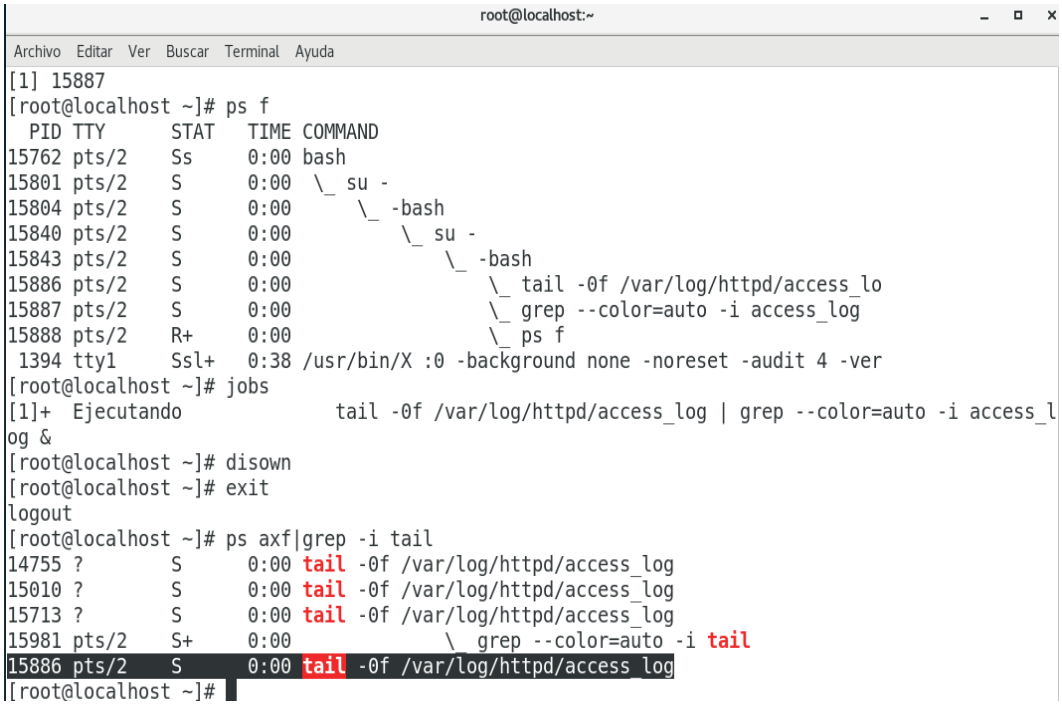


Figura 2.60: Resultado del comando ps axf | grep –i tail en consola

Como podemos apreciar en la figura 2.60 el proceso está corriendo en **background,** esto es fundamental para la administración de servidores y dejar corriendo ciertos servicios que son importantes sin que este asociado a la consola y que cuando se desee salir de la consola no solamente se termine el proceso de la consola sino que también se terminen todos los procesos hijos, el problema de este comando es que cualquier string de datos emitidos hacia **estándar output estándar error,** no se lo va capturar excepto que se lo especifique, lo que se puede hacer es una redirección para que no lo envié a null, como realizamos eso de la siguiente manera.

Pasos necesarios para realizar el ejercicio

1. [root@localhost ~]# su –
2. [root@localhost ~]# tail -0f /var/log/httpd/access_log |grep -i access_log &
3. [root@localhost ~]# tail -0f /var/log/httpd/access_log > ./ApacheReal &
4. [root@localhost ~]# disown
5. [root@localhost ~]# ps fax |grep -i access_log
6. [root@localhost ~]# ls ApacheReal

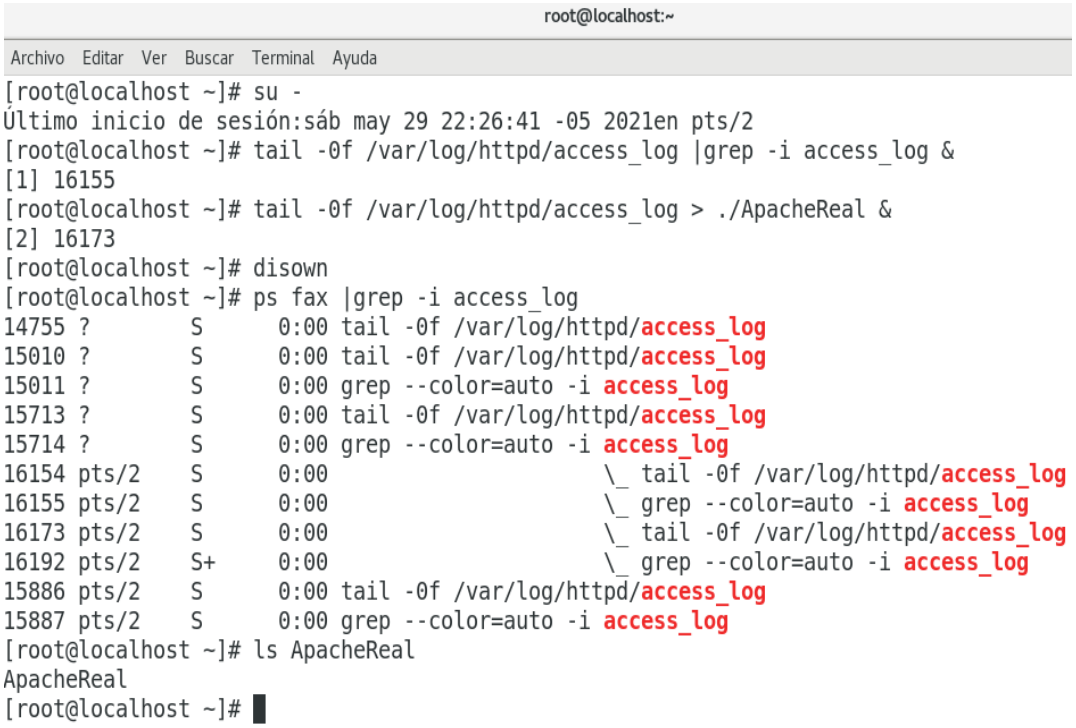


Figura 2.61: Resultado de la ejecución de los comandos anteriores pasó a paso 1, 2, 3, 4, 5, 6 en consola

Práctica 8. Método de prevención de ejecución de proceso mediante el comando disown

Ahora para comprobar que el proceso sigue corriendo debemos de salir de la session mediante el comando exit e invocar nuevamente el proceso mediante el comando `[root@localhost ~]# ps fax |grep -i access_log`

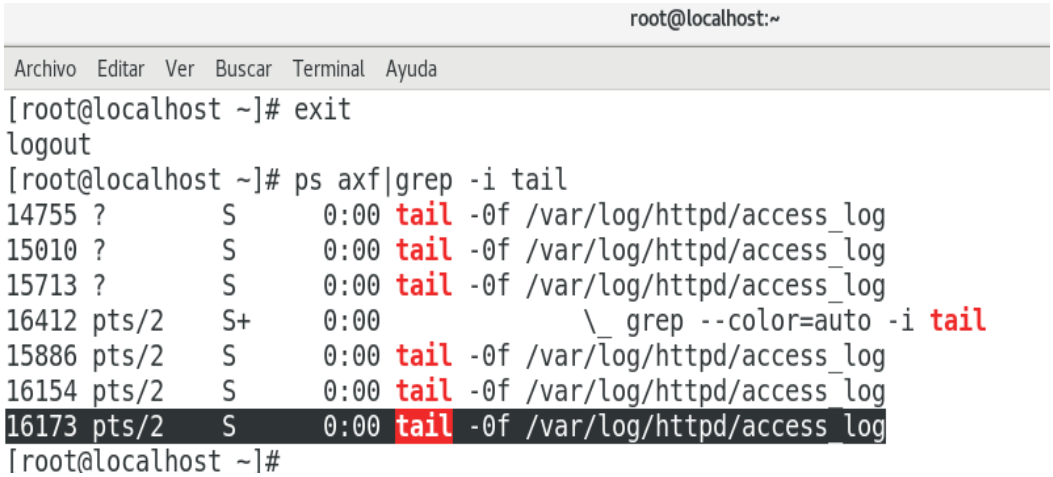


Figura 2.62: Resultado de la ejecución del comando ps fax |grep -i access_log en consola

Podemos listar el fichero mediante el comando:

```
[root@localhost ~]# ls -l ApacheReal
```

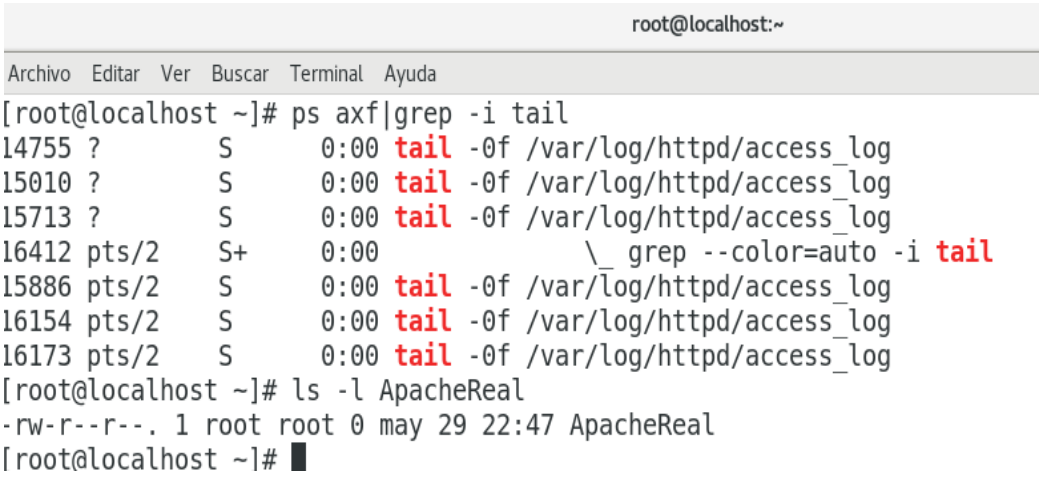


Figura 2.63: Resultado de listar el fichero ApacheReal consola

Como podemos comprobar se ha guardado el log en el fichero **ApacheReal**, hora especifica que se realizó el proceso.

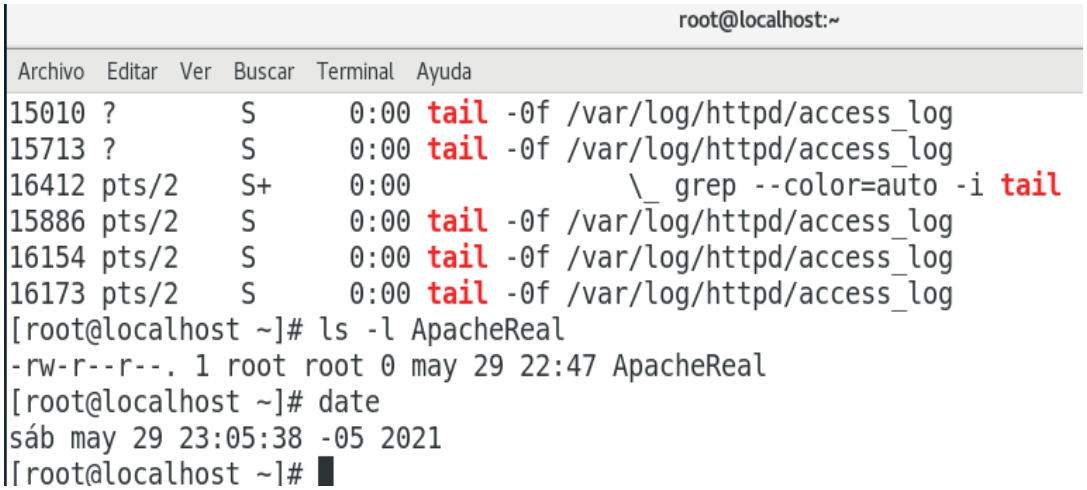


Figura 2.64: Resultado de listar el fichero ApacheReal consola

Para comprobar ejecutamos el comando telnet en consola y revisamos nuevamente la fecha de modificación

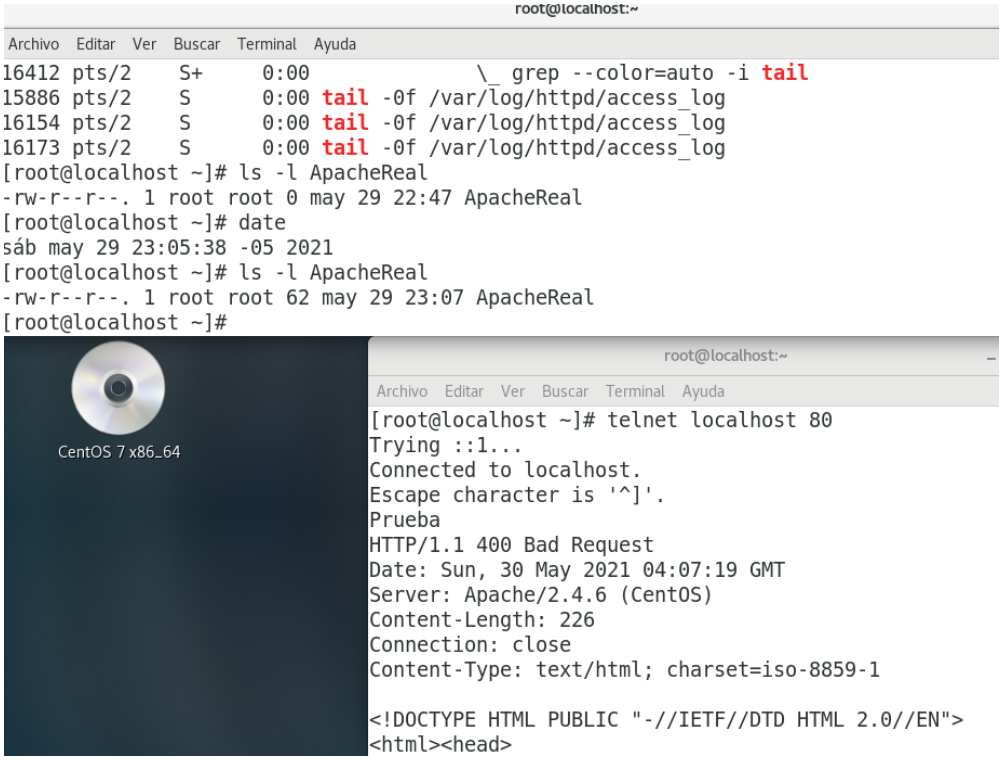


Figura 2.65: Resultado general de la ejecución del comando telnet y revisión del fichero ApacheReal consola

Procedemos a revisar la actualización mediante el comando `cat ApacheReal` y de echo podemos observar que si existe una modificación (mirar la fecha de modificación may 29 23:03)

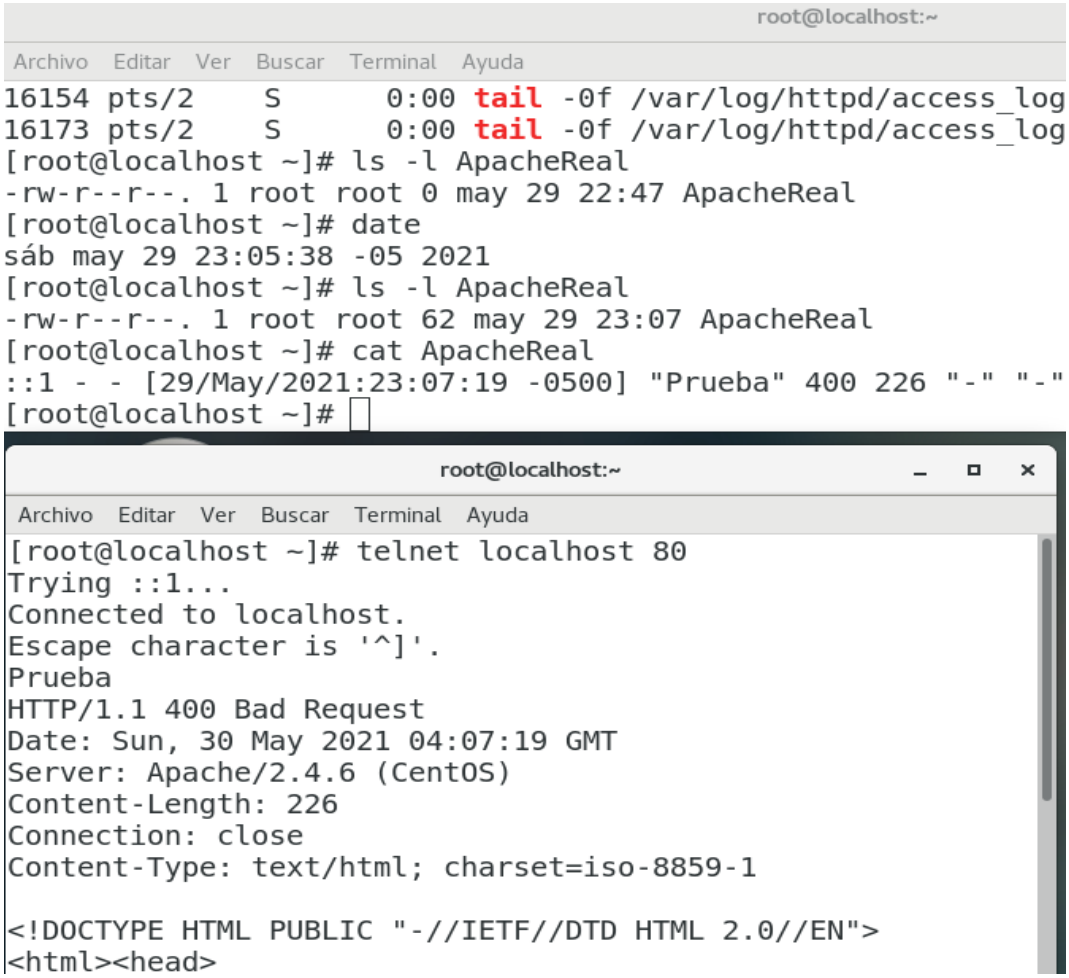


Figura 2.66: Resultado general de la ejecución del comando telnet y revisión del fichero ApacheReal consola

Si deseamos comprobar la misma instrucción del evento, ejecutar el comando **tail -f ApacheReal**

```
[root@localhost ~]# tail -f ApacheReal
```

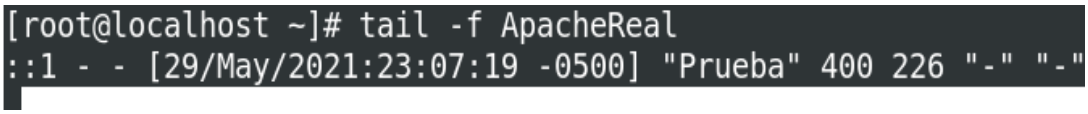


Figura 2.67: Resultado general de la revisión tail -f ApacheReal.

Procedemos a matar el proceso mediante los comandos **kill ID** del proceso o **killall** nombre del proceso

```
[root@localhost ~]# kill 15886
```

```
[root@localhost ~]# killall tail
```

La figura 2.68 muestra cómo utilizar el comando kill y killall para matar los procesos colgados

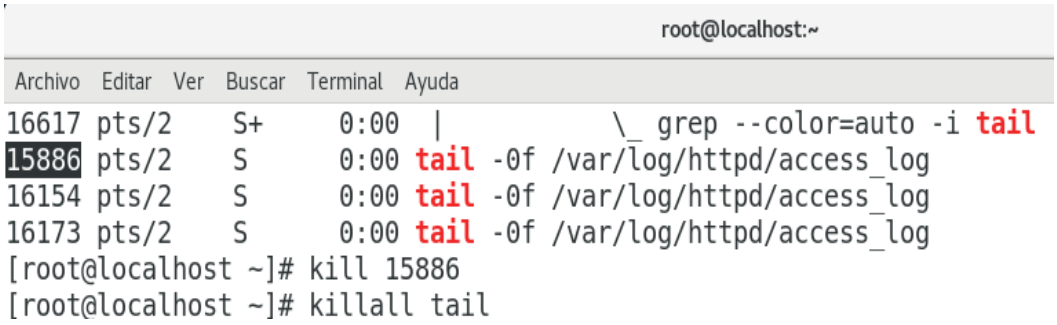


Figura 2.68: Resultado de matar el proceso con el comando kill

Planificación de procesos

Recordemos que la multiprogramación busca no tener ocioso al CPU, y siempre tener un proceso en ejecución para maximizar su utilización, así como la canalización por tiempo compartido, busca que cada uno de los procesos que están en la memoria, se ejecuten en un núcleo diferente del CPU, y que este una vez que termine de ejecutar el proceso busque y comience otro proceso de los que están listos para ejecutarse, intercambiando entre procesos de tal forma que un usuario puede interactuar con cada programa mientras estos se están ejecutando, y para alcanzar este objetivo, el **planificador de procesos** es el encargado de escogerlo de una lista de procesos con estado disponible (*recuerde que el proceso debe estar en el estado listo*) para ser ejecutado por un núcleo del CPU que esté desocupado.

Recuerda que:
Un núcleo del CPU, solo puede ejecutar un proceso a la vez.

La planificación de procesos se refiere a cómo determina el sistema operativo al orden en que irá cediendo el uso del procesador a los procesos

que lo vayan solicitando, y a las políticas que empleará para que el uso que den a dicho tiempo no sea excesivo respecto al uso esperado del sistema.

Hay tres tipos principales de planificación:

A largo plazo. - Decide qué procesos serán los siguientes en ser iniciados. Este tipo de planificación era el más frecuente en los sistemas de lotes (principalmente aquellos con spool) y multiprogramados en lotes; las decisiones eran tomadas considerando los requisitos pre-declarados de los procesos y los que el sistema tenía libres al terminar algún otro proceso. La planificación a largo plazo puede llevarse a cabo con periodicidad de una vez cada varios segundos, minutos e inclusive horas.

En los sistemas de uso interactivo, casi la totalidad de los que se usan hoy en día, este tipo de planificación no se efectúa, dado que es típicamente el usuario quien indica expresamente qué procesos iniciar.

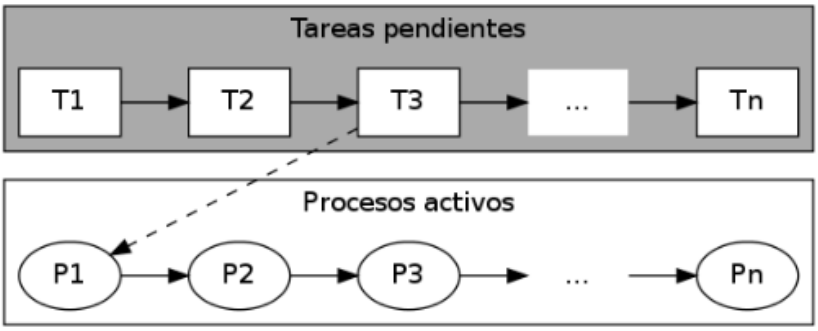


Figura 2.68: Planificador a largo plazo

A mediano plazo.- Decide cuáles procesos es conveniente bloquear en determinado momento, sea por escasez/saturación de algún recurso (como la memoria primaria) o porque están realizando alguna solicitud que no puede satisfacerse momentáneamente; se encarga de tomar decisiones respecto a los procesos conforme entran y salen del estado de bloqueado (esto es, típicamente, están a la espera de algún evento externo o de la finalización de transferencia de datos con algún dispositivo).

En algunos textos, al planificador a mediano plazo se le llama agendador (scheduler).

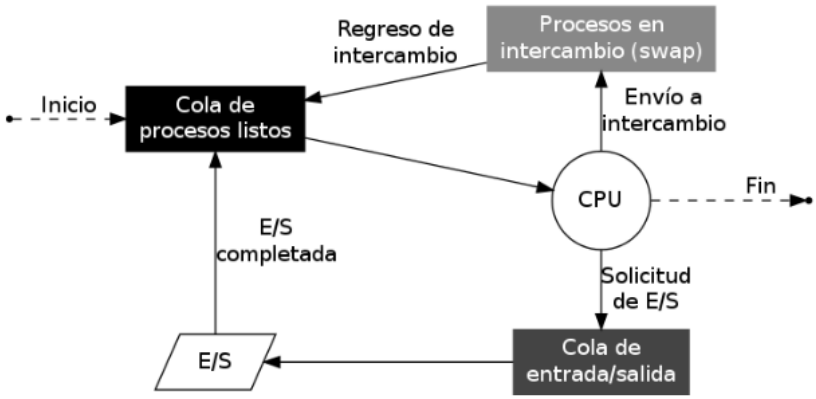


Figura 2.68: Planificador a mediano plazo, o agendador

A corto plazo Decide cómo compartir momento a momento al equipo entre todos los procesos que requieren de sus recursos, especialmente el procesador. La planificación a corto plazo se lleva a cabo decenas de veces por segundo (razón por la cual debe ser código muy simple, eficiente y rápido); es el encargado de planificar los procesos que están listos para ejecución.

El planificador a corto plazo es también frecuentemente denominado despachador (dispatcher).

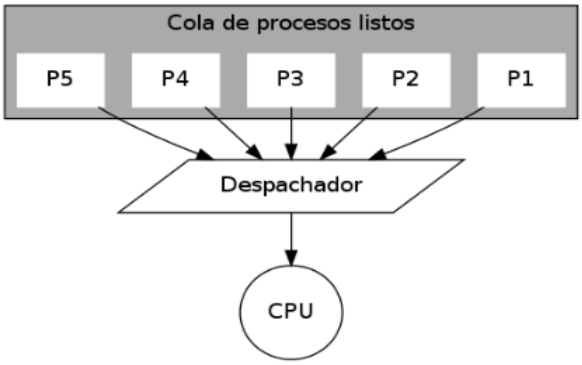


Figura 2.68: Planificador a corto plazo, o despachador

interruptor de contexto

Los sistemas operativos de uso general (Microsoft Windows, mac OS X, distribuciones de Linux, etc.) gestionan un conjunto de interrupciones, las cuales al ser atendidas provocan que un proceso que se está ejecutando en un núcleo de la CPU, sea suspendido, cambiando su estado, e iniciando

uno nuevo, pero al momento de atender la interrupción el sistema operativo hace un respaldo de toda la ejecución del proceso vigente (datos y posición del PC), al que llamaremos **contexto actual**, para poder restaurarlo y reanudarlo una vez que la interrupción finalizó y fue atendida. Por lo tanto, en la PCB es almacenado el contexto actual que incluye valores en los registros y la memoria, haciendo un guardado de los estados actual de la CPU y restauración, para que pueda reanudar el proceso.

Cambiar el núcleo de la CPU de un proceso a otro proceso requiere realizar un guardado de estado del proceso actual y una restauración de estado de un proceso diferente, a esta tarea se la conoce como **cambio de contexto** (*context switch*) y se ilustra en la Figura . Cuando se produce un cambio de contexto, el núcleo guarda el contexto del proceso anterior en su PCB y carga el contexto guardado del nuevo proceso planificado para ejecutarse. El tiempo del intercambio de contexto es la sobrecarga pura, porque el sistema no hace ningún trabajo útil mientras se cambia. La velocidad de conmutación varía de una máquina a otra, dependiendo de la velocidad de memoria, el número de registros que se deben copiar y la existencia de instrucciones especiales (como una sola instrucción para cargar o almacenar todos los registros). Una velocidad típica es de varios microsegundos

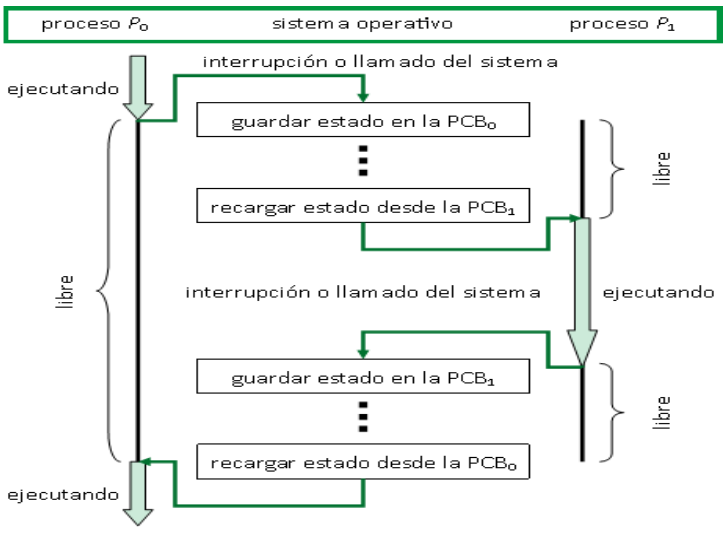


Figura 2.70: Diagrama mostrando el intercambio de contexto desde un proceso a otro

Los tiempos de cambio de contexto dependen en gran medida del soporte de hardware. Por ejemplo, algunos procesadores proporcionan múltiples conjuntos de registros. Un cambio de contexto aquí simplemente requiere cambiar el puntero al conjunto de registros actual. Por supuesto, si hay procesos más activos que conjuntos de registros, el sistema recurre a la copia de datos de registro a la memoria, como antes. Además, cuanto más complejo es el sistema operativo, mayor es la cantidad de trabajo que debe realizarse durante un cambio de contexto.

Práctica 9. Uso del comando nohup

El comando **nohup** es la señal que envía el kernel a un proceso cuando tiene que terminar ejemplo.

Debemos de ejecutar una session dentro de la session de consola mediante el comando **su –** después de haber ejecutado la consola anidad proceso a ejecutar el comando:

```
[root@localhost ~]# nohup tail -0f /var/log/httpd/access_log
&
```

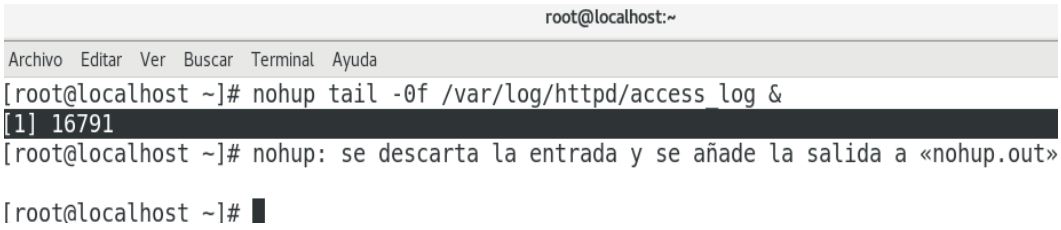


Figura2.71: Señales que envía el kernel a un proceso nohup.

Vemos que con la ejecución del comando obtuvimos el identificador del proceso PID, el mismo que esta resaltado en color negro.

Si procedemos a ejecutar el comando ps axf|grep –i numero de proceso podemos ver que aparece el proceso ejemplo: `[root@localhost ~]# ps axf|grep -i 16791`

```
[root@localhost ~]# ps axf|grep -i 16791
16791 pts/2    S      0:00  |                  \_ tail -0f /var/log/httpd/access_log
16821 pts/2    S+     0:00  |                  \_ grep --color=auto -i 16791
[root@localhost ~]#
```

Figura 2.72: Señales que envía el kernel a un proceso nohup

Curiosamente podemos identificar que el proceso no aparece con el **nohup**, lo que quiere decir que no es un proceso apegado a la consola, por lo que si se realiza un **exit** y luego se busca el proceso mediante el comando:

`[root@localhost ~]# ps axf|grep -i 16791`

En la figura 2.73 vemos que esta el proceso y no tuvo que ejecutar el comando **disown**, además se crea un fichero **nohup.out** dentro del directorio donde se ejecutó el comando **nohup**.

```
[root@localhost ~]# ps axf|grep -i 16791
16855 pts/2    S+     0:00  |           \_ grep --color=auto -i 16791
16791 pts/2    S      0:00 tail -0f /var/log/httpd/access_log
[root@localhost ~]#
```

Figura 2.73: Señales que envía el kernel a un proceso nohup

Además, debemos de conocer que si revisamos los ficheros mediante el comando `[root@localhost ~]# ls –ltr` podemos observar que se creó un fichero dentro del directorio que se ejecutó **nohup** con el nombre de **nohup.out**

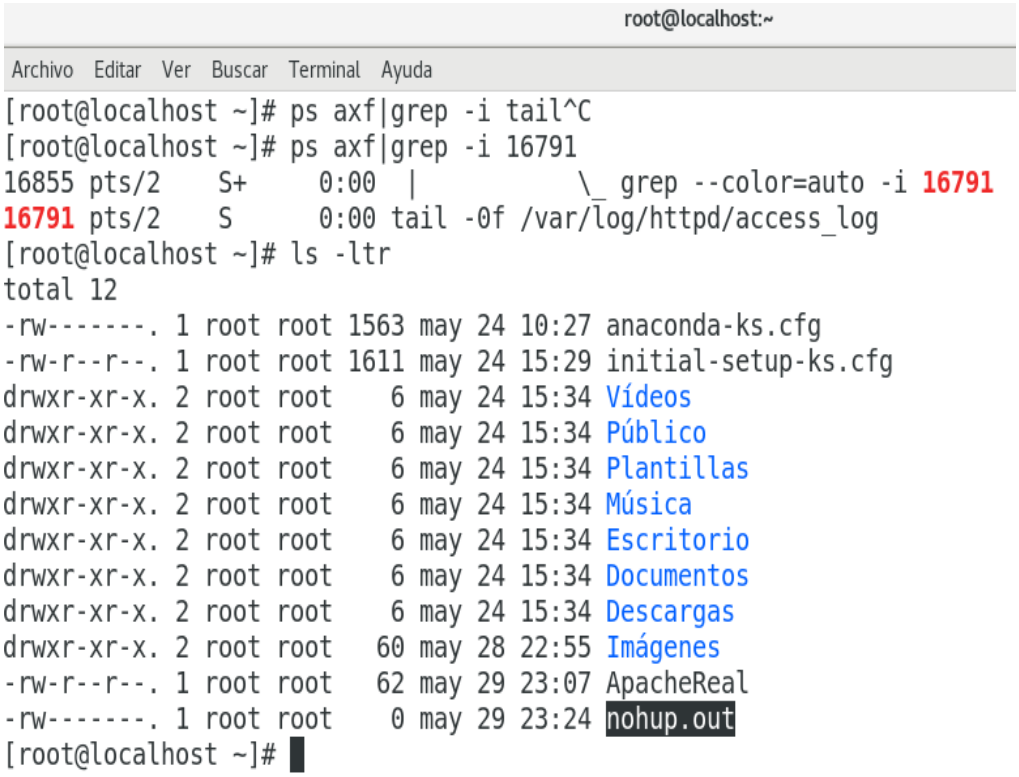


Figura 2.74: Revisión de la creación del fichero nohup.out

Para realizar la revisión de la creación del fichero **nohup.out,** debemos de ejecutar el comando **tail -0f nohup.out** y abrimos otra terminal y sobre ella ejecutamos el comando **telnet localhost 80** estos serían los resultados.

```
[root@localhost ~]# tail -0f nohup.out
```

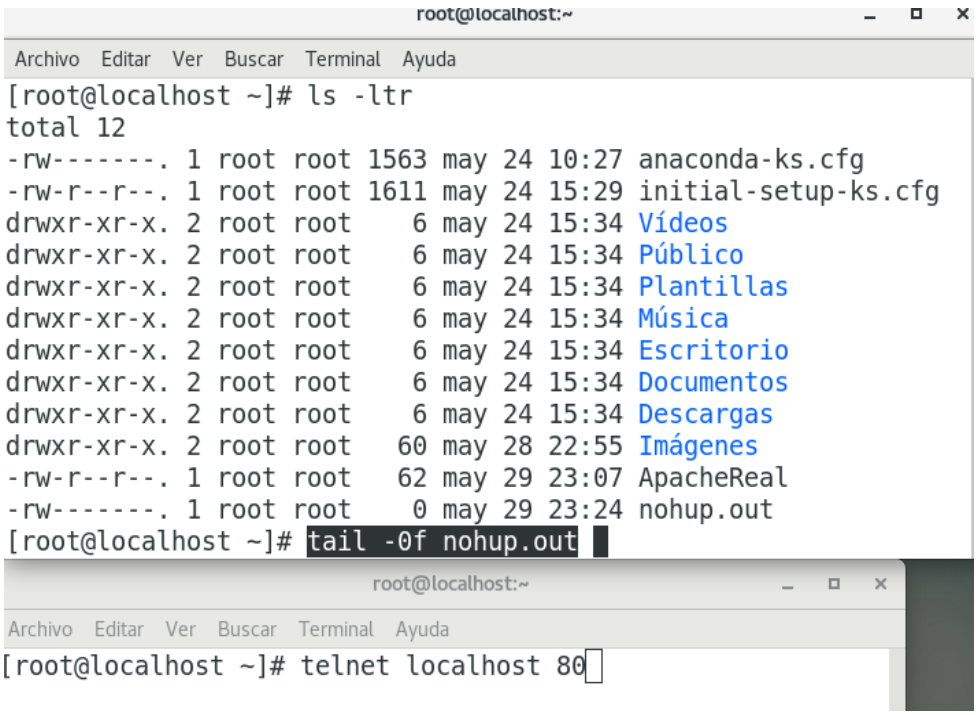


Figura2.75: Revisión de los resultados del fichero nohup.out

La figura 2.76 muestra cómo queda el registro el evento

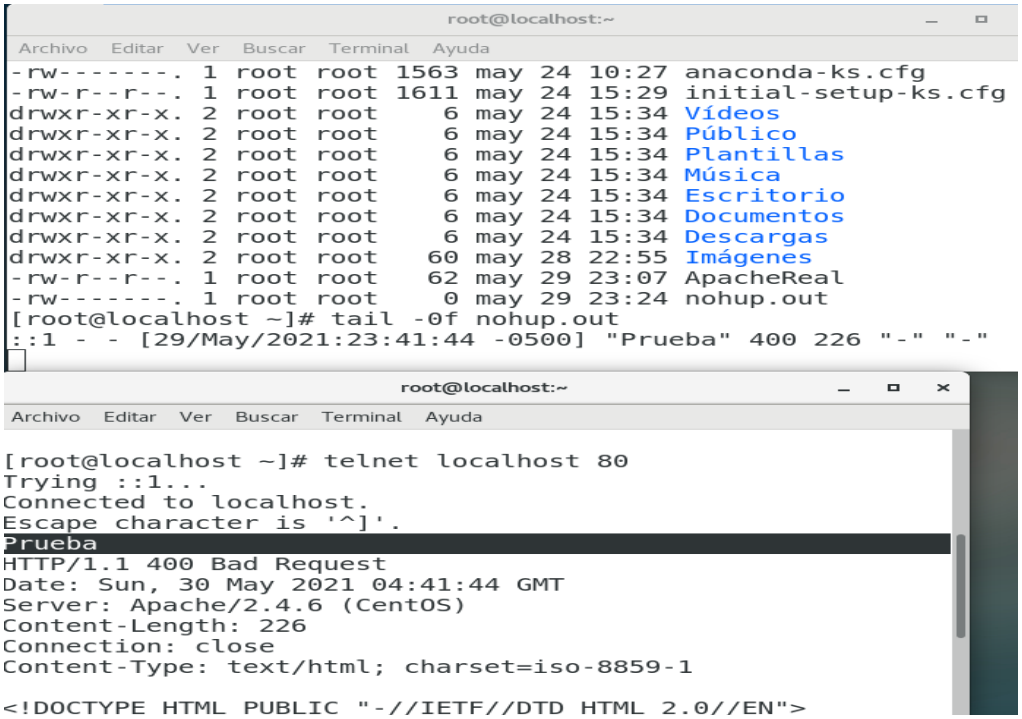


Figura 2.76: Revisión de los resultados del fichero nohup.out

A continuación, se indica otra forma de ejecutar un comando que no esté apegado a la consola, por lo cual al momento de salir de esa consola el proceso no va terminar y se va convertir en un proceso desheredado.

En los sistemas operativos modernos el **huponexit** viene desactivado, en sistemas operativos más antiguos esta opción venia activada para poder comprobar la misma es fundamental ejecutar el comando **shopt.**

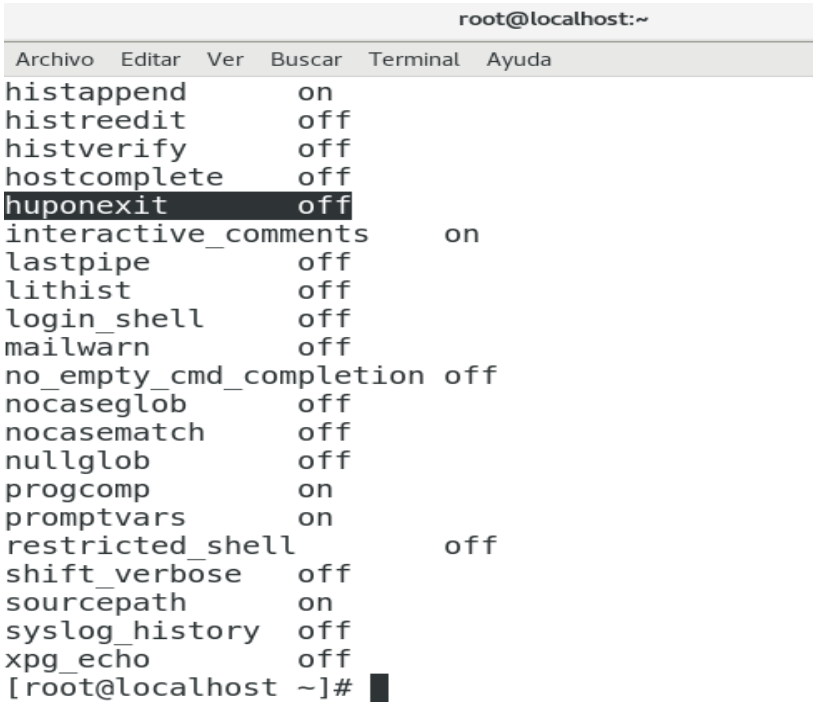


Figura 2.76: Revisión de los resultados del comando shopt

Para conocer las diferentes señales que envía el kernel a los procesos debemos de ejecutar el comando **trap –l** y nos van a aparecer todas las señales que existen disponibles con el nombre y el código ejemplo.

```
[root@localhost ~]# trap –l
```

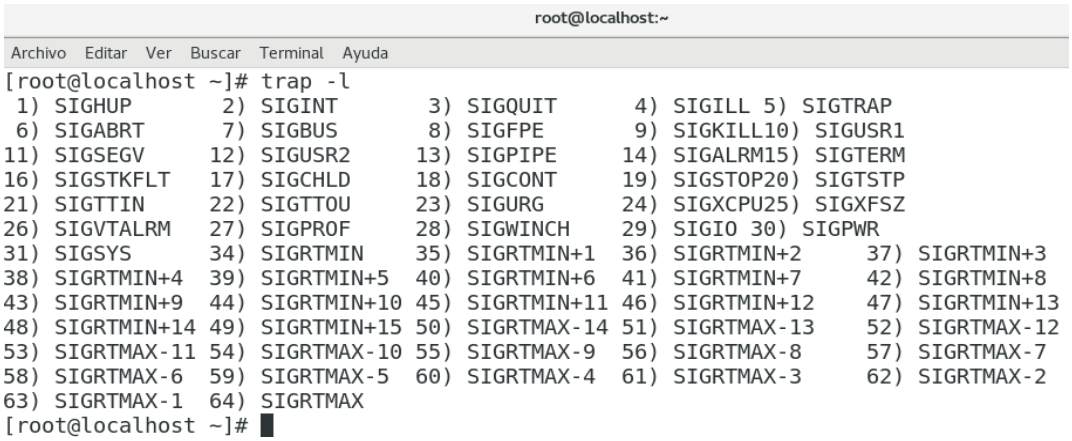


Figura 2.77: Señales que envía el kernel a los procesos.

Para entender mejor las diferentes señales que envía el kernel a los procesos es fundamental conocer todas estas señales ejemplo:

Si ejecutamos el comando `[root@localhost ~]# tail -0f /var/log/httpd/access_log` y luego en la segunda consola buscamos el proceso mediante el comando `[root@localhost ~]# ps axf |grep -`

`i tail` identificamos el proceso ahora procedemos a enviar distintas señales ejemplo:

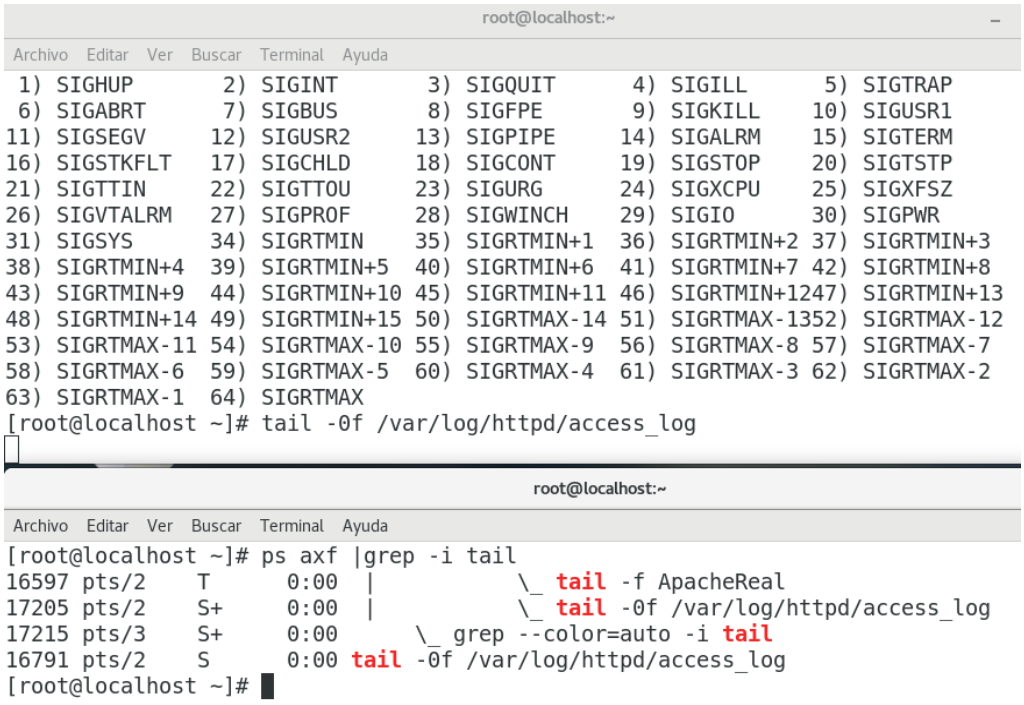


Figura 2.78: Señales que envía el kernel a los procesos.

Señales de kernel a los procesos

Procedemos a enviar `kill -1`

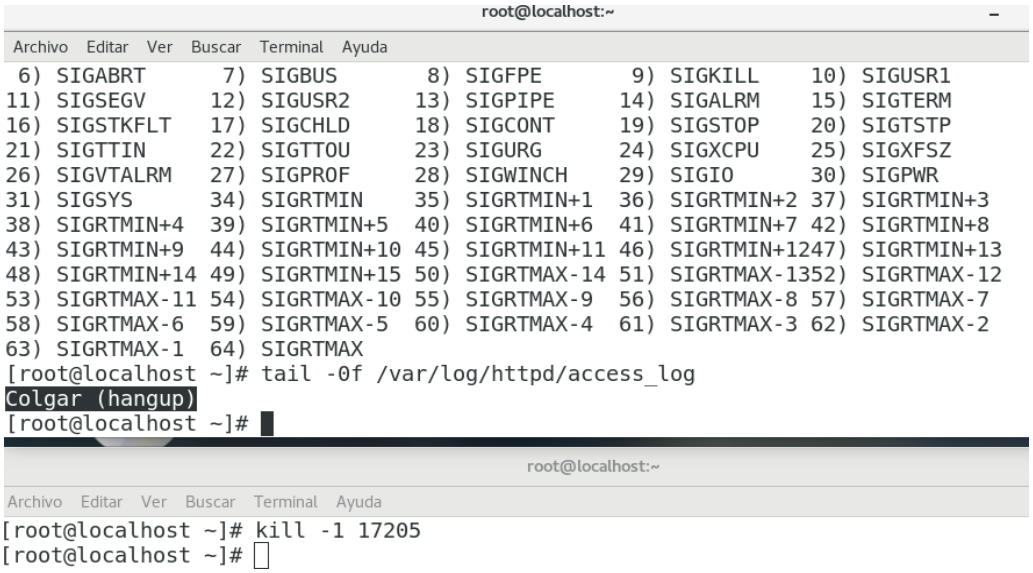


Figura 2.79: Señales que envía el kernel a los procesos kill -1.

Podemos observar en la imagen el resultado que muestra la consola 1, la señal que recibió el proceso es colgar **hangup**, esta es la señal que recibe el proceso cuando se cierra la consola.

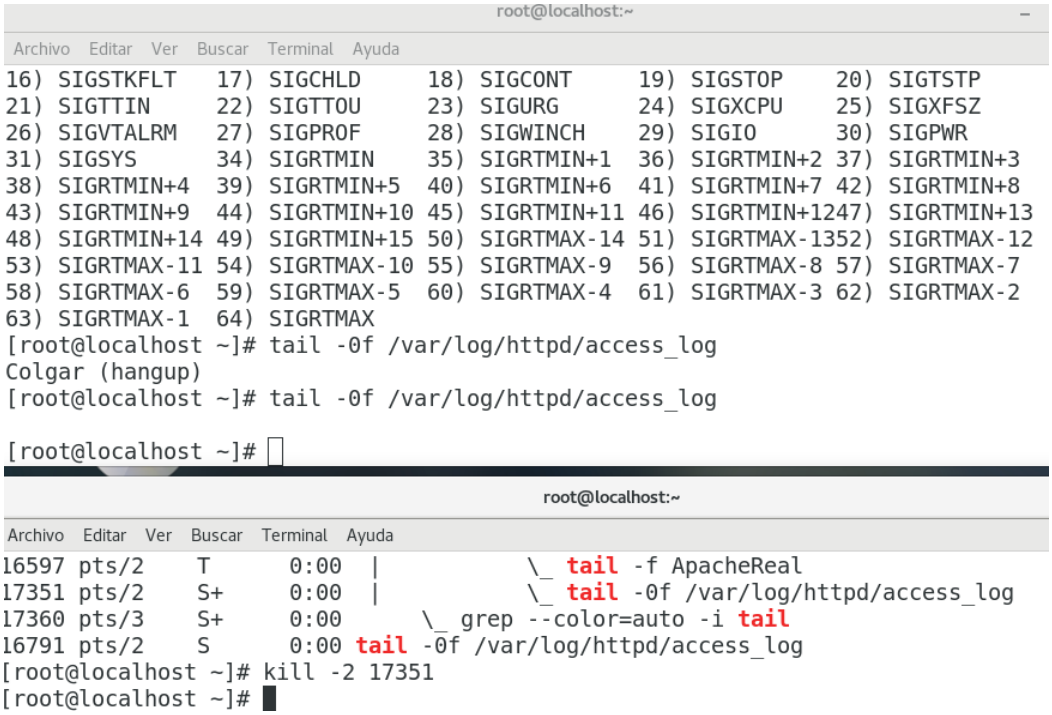


Figura 2.80: Señales que envía el kernel a los procesos kill -2 ID.

Como se aprecia en la imagen la señal que envió el kernel ha el proceso fue de interrupción

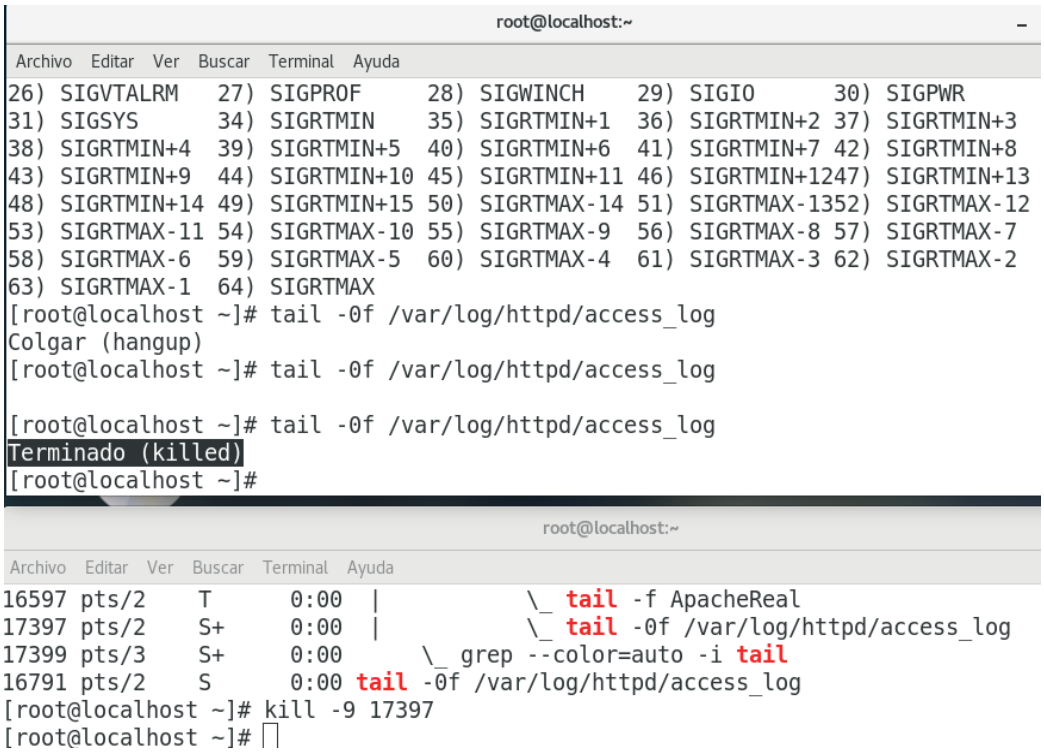


Figura 2.80: Señales que envía el kernel a los procesos kill -2 ID (killed).

Resultado de haber enviado el comando `[root@localhost ~]# kill -9 17397` señal que envía el kernel a un proceso:

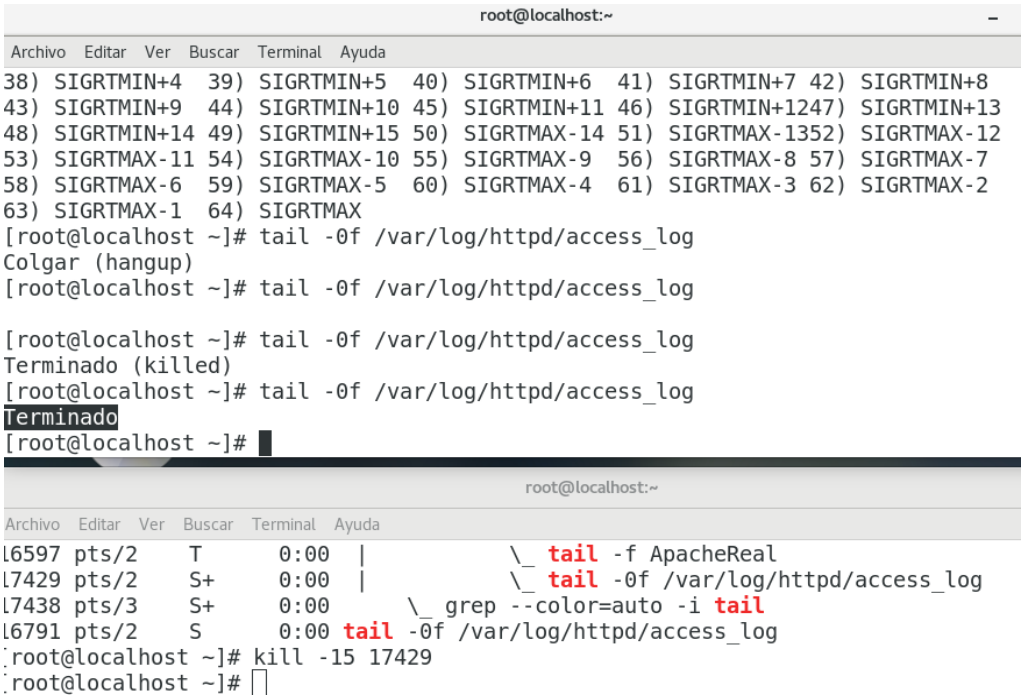


Figura 2.81: Señales que envía el kernel a los proceso kill -2 ID (killed).

Resultado de haber enviado la interrupción **15) SIGTERM** que es la señal para terminar un proceso correctamente.

Para profundizar:

Para buscar información en el sistemas operativo lo puedes realizar mediante el comando `apropos signal`, y para acceder a una página del manual puedes ejecutar el comando `man 7 signal`.

Tema 2: Hilos (Threads)

Conceptos de hilos

Hasta este punto, hemos definido al proceso, y se dijo que es un programa que se ejecuta en un solo núcleo a la vez, pero, en la actualidad los CPUs

tienen varios núcleos y estos a su vez varios hilos. También definimos que, con la multiprogramación y la multitarea, se pueden tener varios procesos listos en memoria, pero ¿qué tiene que ver todo esto…?

Pues bien, la mayoría de los sistemas operativos modernos han ampliado el concepto de proceso para permitir que un proceso tenga varios subprocesos a través de los hilos y sean ejecutados simultáneamente, y por lo tanto, realice más de una tarea a la vez.

Primero, vamos a definir qué es un hilo (*thread*). Un hilo, también llamado subproceso, es la unidad más pequeña de la CPU que puede ejecutar un proceso, esta comprende un identificador (*Thread ID*), un contador de programa (*PC*), un conjunto de registros, y una pila, además comparte con los otros hilos, que pertenecen a un mismo proceso padre, la sección de código, datos y otros recursos del sistema operativo, como archivos abiertos y señales. Un proceso tradicional tiene un único subproceso (hilo) de control. Si un proceso tiene varios subprocesos (hilos) de control, puede realizar más de una tarea a la vez, ver la Figura2.90 donde se ilustra la diferencia entre un proceso tradicional de un **solo subproceso (hilo)** (*single-threaded*) y un **proceso multi subprocesos (hilos)** (*multithreaded*).

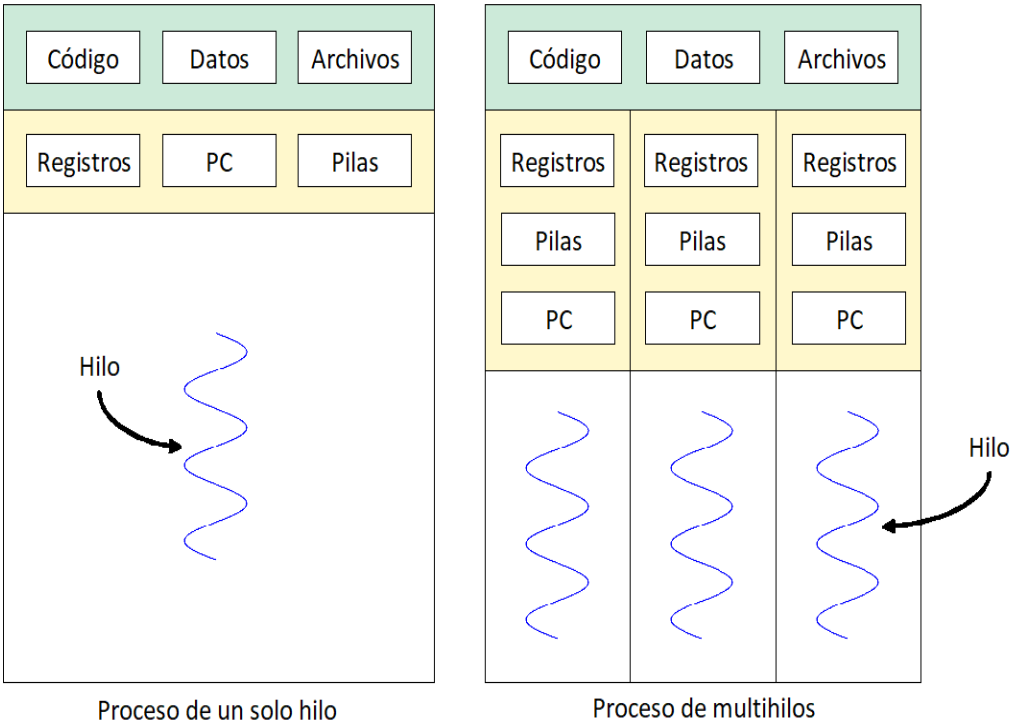


Figura2.90: Procesos con un hilo y múltiples hilo

Algoritmos de planificación.

Tener una adecuada planificación de la CPU, alivia el problema de decidir qué núcleo de la CPU, se hará cargo de los procesos que se encuentran en la cola de procesos en estado preparado y en la actualidad hay muchos algoritmos de planificación (*Scheduling Algoritms*) de dónde escoger. Actualmente los CPU modernos cuentan con arquitecturas que traen varios núcleos de procesamiento y cada núcleo varios hilos, pese a esto, para realizar un estudio más objetivo de cómo funcionan los algoritmos de planificación nos centraremos en el contexto de los CPU que sólo tienen un núcleo de procesamiento. Más adelante estudiaremos la planificación del CPU en el contexto de sistemas con multiprocesador.

Como resumen, un algoritmo de planificación se utiliza para calcular los recursos que consume otro algoritmo o conjunto de algoritmos (programa) al realizar una determinada tarea. Ejemplo: Tiempo de finalización y porcentaje de utilización de la CPU

En los algoritmos existen ciertas medidas que se utilizan para evaluarlos:

- **Porcentaje de utilización de la CPU por procesos de usuario**, como la CPU es el recurso más caro que tenemos, este necesita ser más utilizado y tratar de salir de los valores reales que suelen estar entre el 40 y 90 porciento.
- **Rendimiento**, también llamado *throughput*, por su nombre en inglés es el número de ráfagas que se ejecutan por unidad de tiempo. Una ráfaga es definida como el período de tiempo en que un proceso necesita de la CPU durante su vida, también se lo suele definir como el número de trabajos por unidad de tiempo.
- **Tiempo de espera (E)**, es el tiempo que una ráfaga ha permanecido en la cola de procesos con estado de preparado o listo.
- **Tiempo de finalización (F)**, es el tiempo transcurrido desde que una ráfaga comienza a existir hasta que finaliza. Su fórmula de cálculo es: $F = E + t$, donde t es el tiempo de la ráfaga en el CPU.
- **Penalización (P)**, es una medida sin dimensiones que se puede aplicar homogéneamente a las ráfagas independientemente de su longitud. Su forma de calcular es: $P = (E + t) / t = F / t$

En general, hay que maximizar los dos primeros parámetros y minimizar los tres últimos. Sin embargo, estos objetivos son contradictorios, el dedicar más tiempo de CPU a los usuarios se hace a costa de llamar menos al algoritmo de planificación (menos cambios de proceso), y de simplificarlo. Esto provoca que la CPU se reparta menos equitativamente entre los procesos, en detrimento de los últimos tres parámetros.

Dependiendo de los objetivos se elegirá cierto algoritmo. En los sistemas por lotes suele primar el rendimiento del sistema, mientras que en los sistemas interactivos es preferible minimizar, por ejemplo, el tiempo de espera.

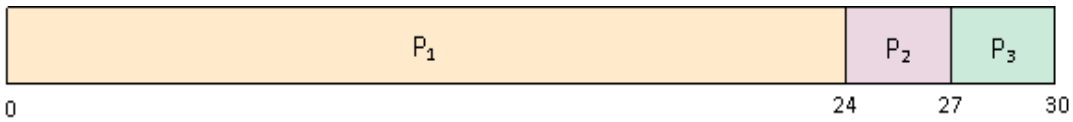
Planificación por orden de llegada

También llamada planificación de primero en llegar, primero en servirse (***FCFS** – First-Come, First-Served Scheduling*) por mucho es el algoritmo de planificación de la CPU más simple. Con este esquema, la CPU será asignada al proceso que primero la solicite. La implementación de la directiva FCFS se gestiona fácilmente con una cola FIFO. Cuando un proceso entra en la cola de procesos en estado preparado, su PCB se vincula al final de la cola. Entonces, cuando el CPU es libre, se asigna al proceso que está en el principio de la cola, luego de esto, el proceso en ejecución es eliminado de la cola. El código del algoritmo de planificación FCFS es fácil de escribir y entender.

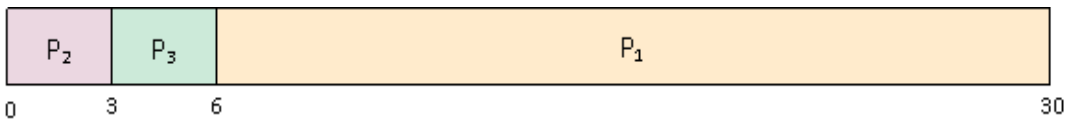
En el lado negativo, el tiempo medio de espera con el algoritmo FCFS es a menudo bastante largo. Considere el siguiente conjunto de procesos que llegan en el momento 0, con la longitud de la ráfaga de CPU dada en milisegundos:

Proceso	Tiempo de ráfaga
P_1	24
P_2	3
P_3	3

Si los procesos llegan en el orden P_1, P_2, P_3 y se sirven en el orden FCFS, obtenemos el resultado que se muestra en el siguiente gráfico de Gantt.



También los procesos podrían llegar en el orden P_2, P_3, P_1, según el siguiente diagrama de Gantt.



Además, considere el rendimiento de la planificación FCFS en una situación dinámica. Suponiendo que tenemos un proceso enlazado a la CPU y muchos procesos enlazados a E/S. A medida que los procesos

fluyen alrededor del sistema, puede producirse el siguiente escenario. El proceso enlazado a la CPU obtendrá y sostendrá la CPU. Durante este tiempo, todos los demás procesos terminarán su E/S y se moverán a la cola lista, esperando la CPU. Mientras los procesos esperan en la cola de procesos en estado preparado, los dispositivos de E/S están inactivos. Eventualmente, el proceso enlazado a la CPU finaliza su ráfaga de CPU y se mueve a un dispositivo de E/S. Todos los procesos enlazados a E/S, que tienen ráfagas de CPU cortas, se ejecutan rápidamente y vuelven a las colas de E/S. En este momento, la CPU se encuentra inactiva. A continuación, el proceso enlazado a la CPU volverá a la cola de procesos en estado preparado y se le asignará la CPU. Una vez más, todos los procesos de E/S terminan esperando en la cola de procesos en estado preparado hasta que se realiza el proceso enlazado a la CPU. Hay un efecto de convoy, ya que todos los otros procesos esperan a que el proceso grande se salga de la CPU. Este efecto da lugar a una utilización menor de la CPU y de los dispositivos, de lo que podría ser posible si se permitiera que los procesos más cortos se ejecuten primero.

Tenga en cuenta también que el algoritmo de planificación FCFS no es preventivo. Una vez que a la CPU se le ha asignado un proceso, ese proceso se mantiene la CPU hasta que la libera, ya sea terminando su ejecución o porque se realiza una solicitud de E/S. Por lo tanto, el algoritmo FCFS es particularmente problemático para los sistemas interactivos, donde es importante que cada proceso obtenga una parte de la CPU a intervalos regulares. Sería desastroso permitir que un proceso mantenga la CPU durante un período prolongado.

Uso de simuladores para algoritmos FCFS

Simulador 1

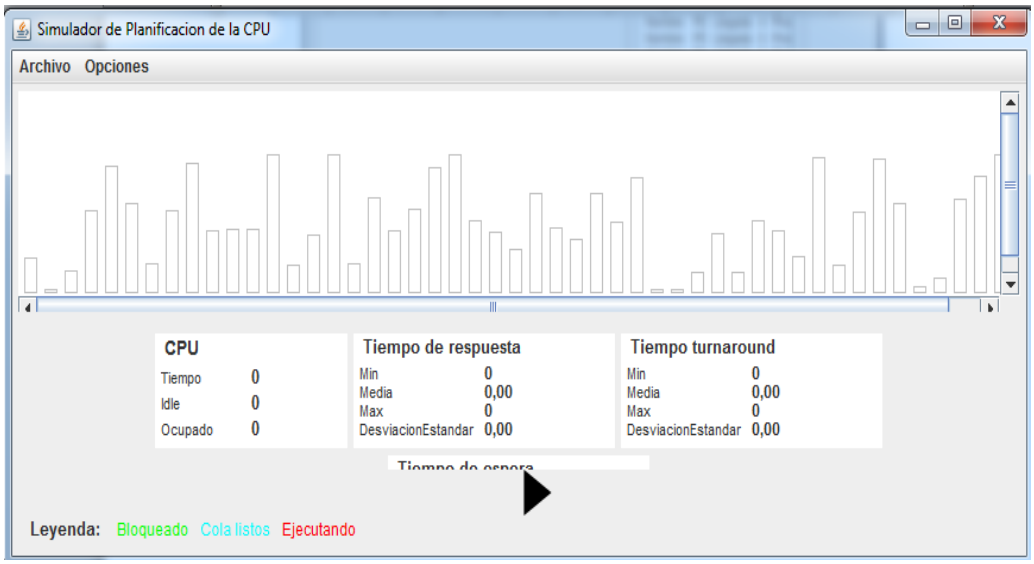


Figura 0.91: Simulador de planificador de procesos con un hilo y múltiples hilo algoritmo FCFS

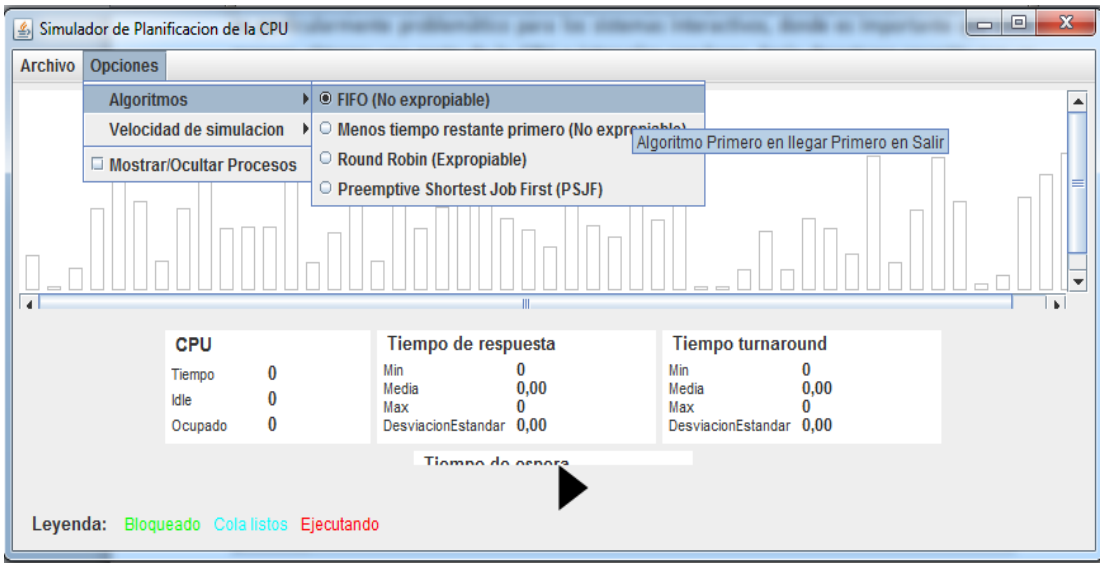


Figura 0.92: Selección del tipo de algoritmo FCFS

Figura 0.93: Ejecución de los procesos algoritmo FCFS con cola FIFO

Figura 0.94: Ejecución de los procesos y procesos en cola.

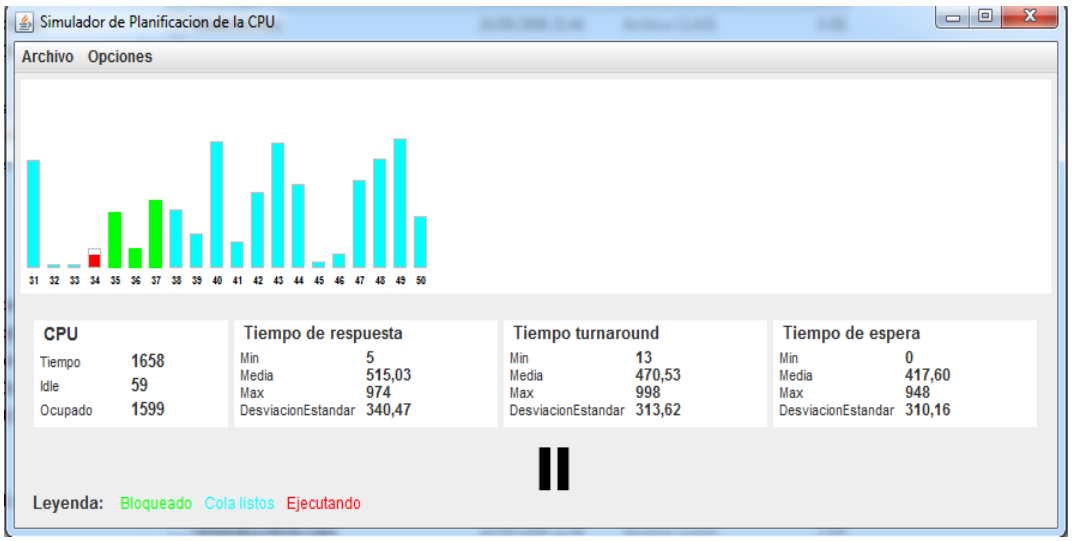


Figura 0.95: Ejecución de los procesos y procesos bloqueado, en cola listos, ejecutando.

Simulador 2

Algoritmo de planificación por orden de llegada FCFS

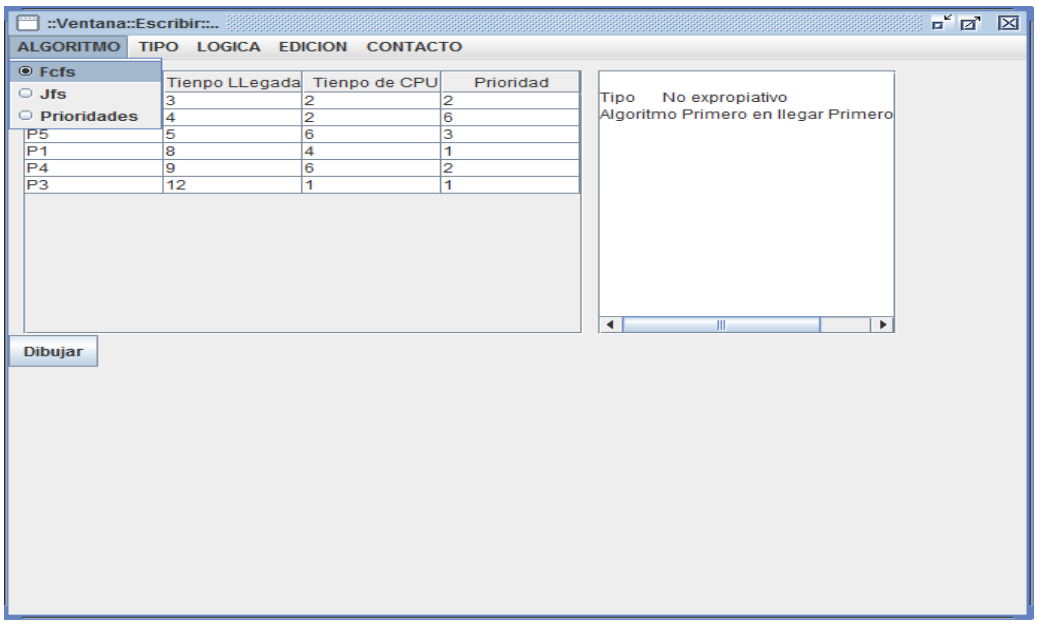


Figura 0.96: Selección del tipo de algoritmo Fcfs.

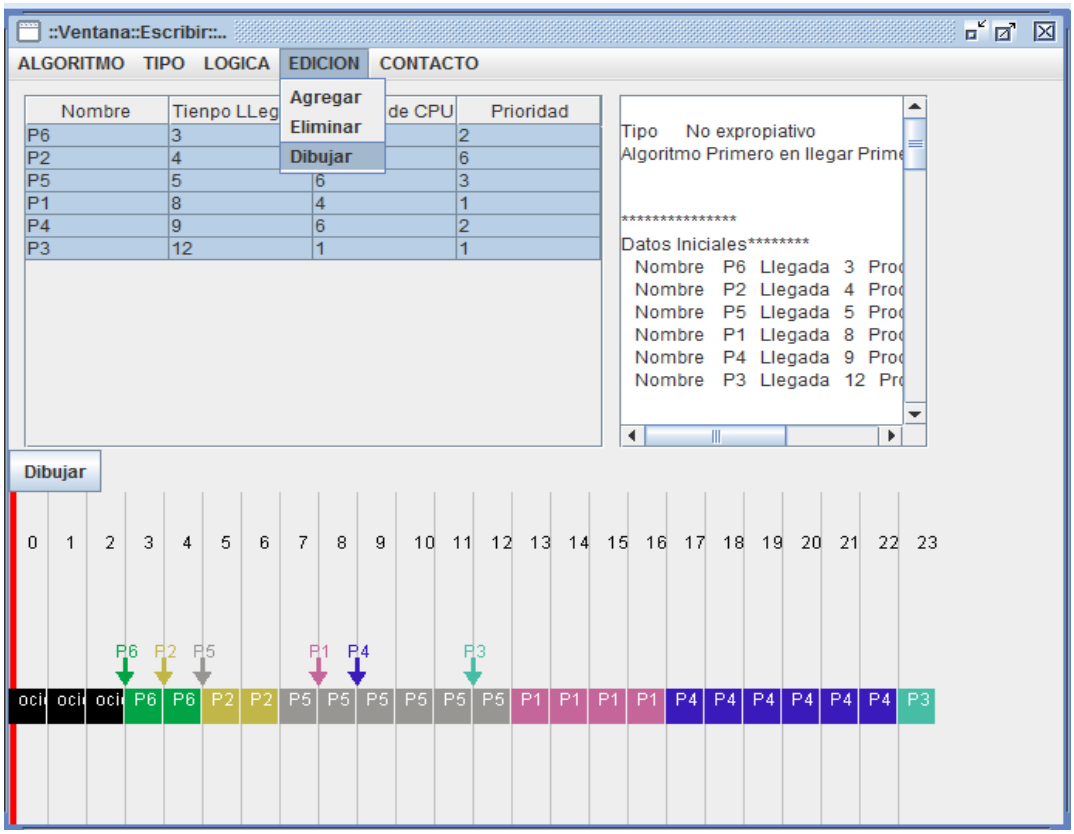


Figura 0.97: Ejecución y modelado del proceso y los hilos a utilizar tipo de algoritmo Fcfs.

Simulador 3

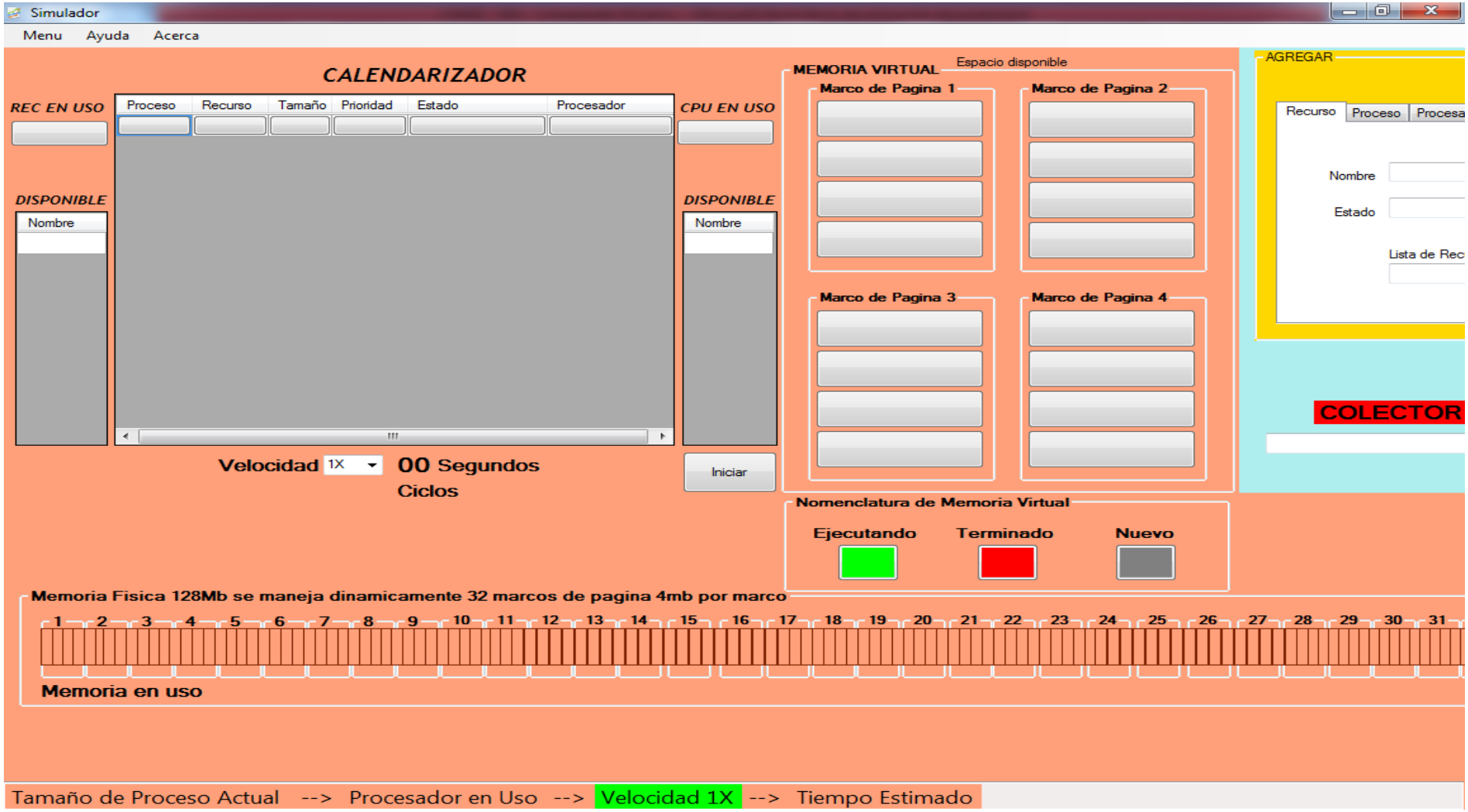


Figura 0.98: Simulación masiva de procesos con múltiples hilos y calendarización.

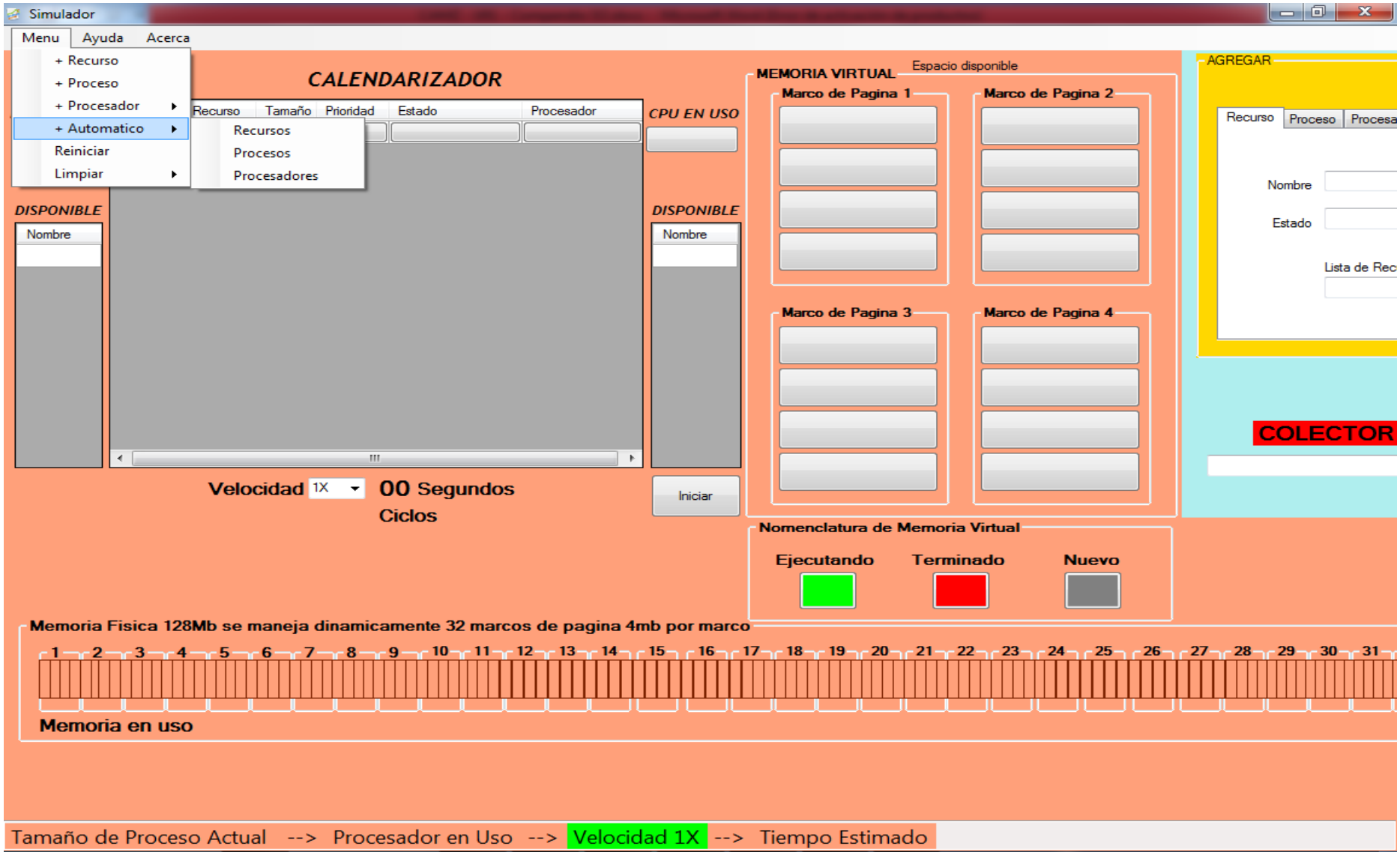


Figura 0.99: Simulación masiva de procesos con múltiples hilos y calendarización.

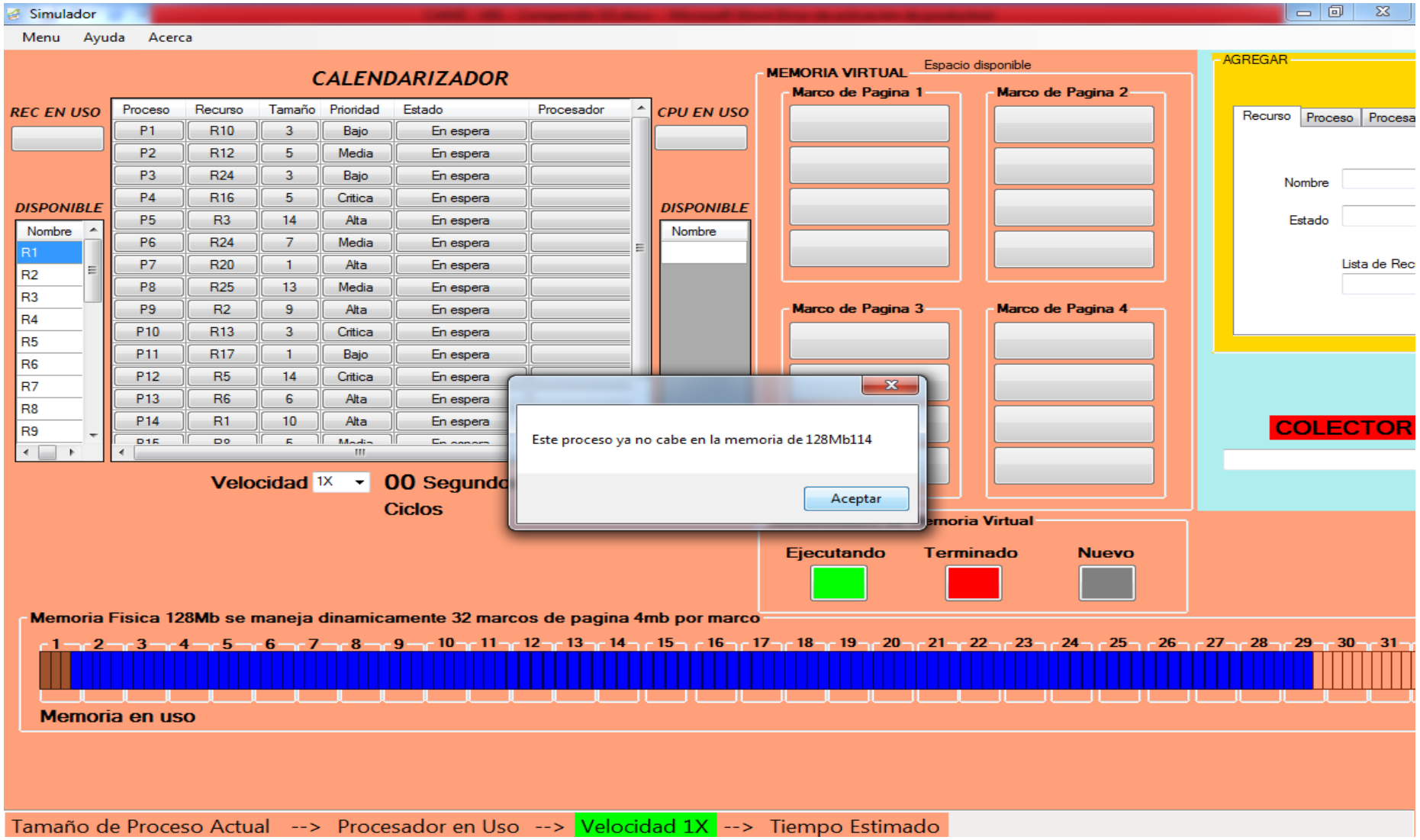


Figura 0.100: Espacio total de los procesos en memoria disponible 128Mb a utilizar 114.

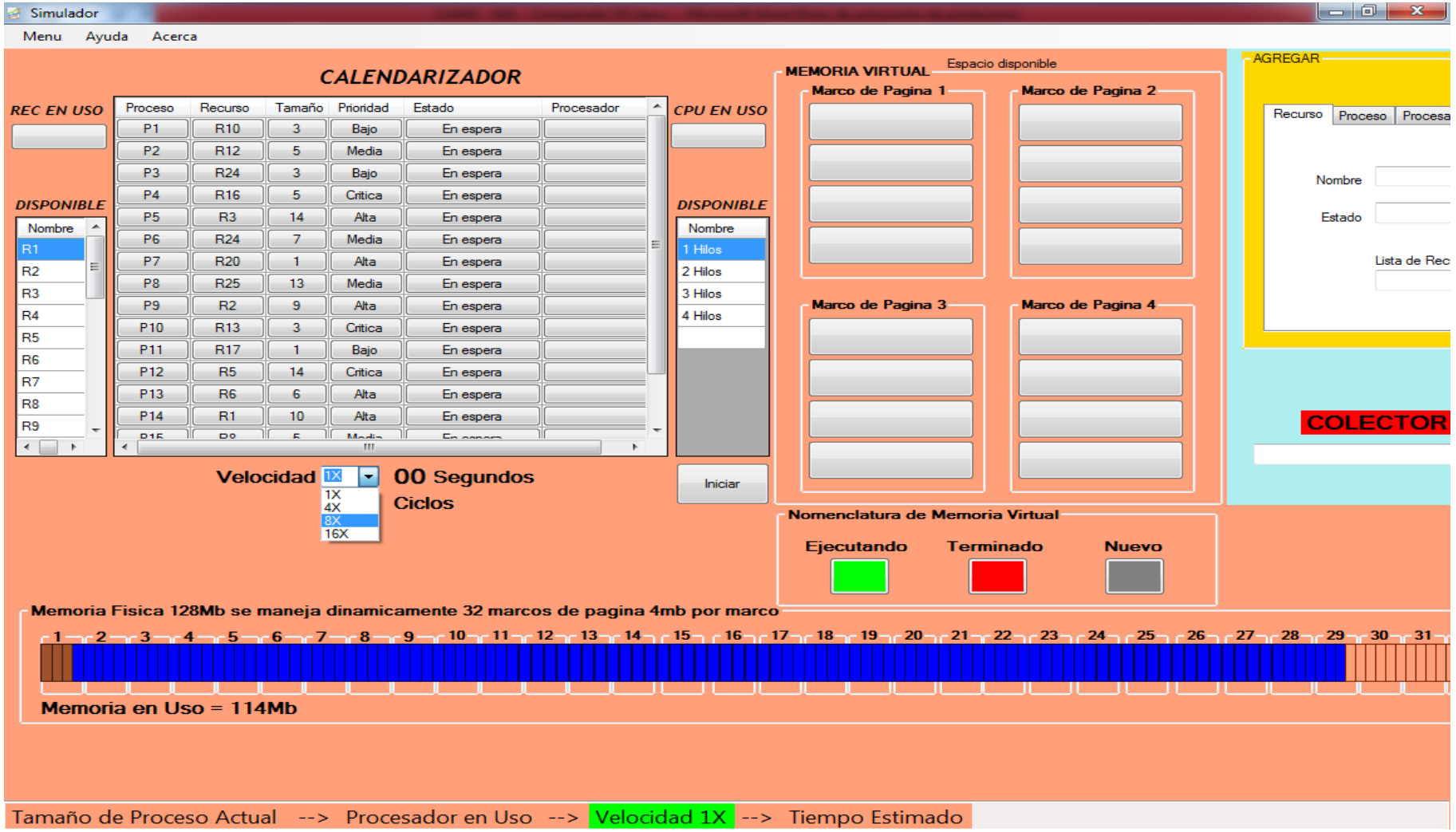


Figura 0.100: Espacio total de los procesos en memoria disponible 128Mb a utilizar 114.

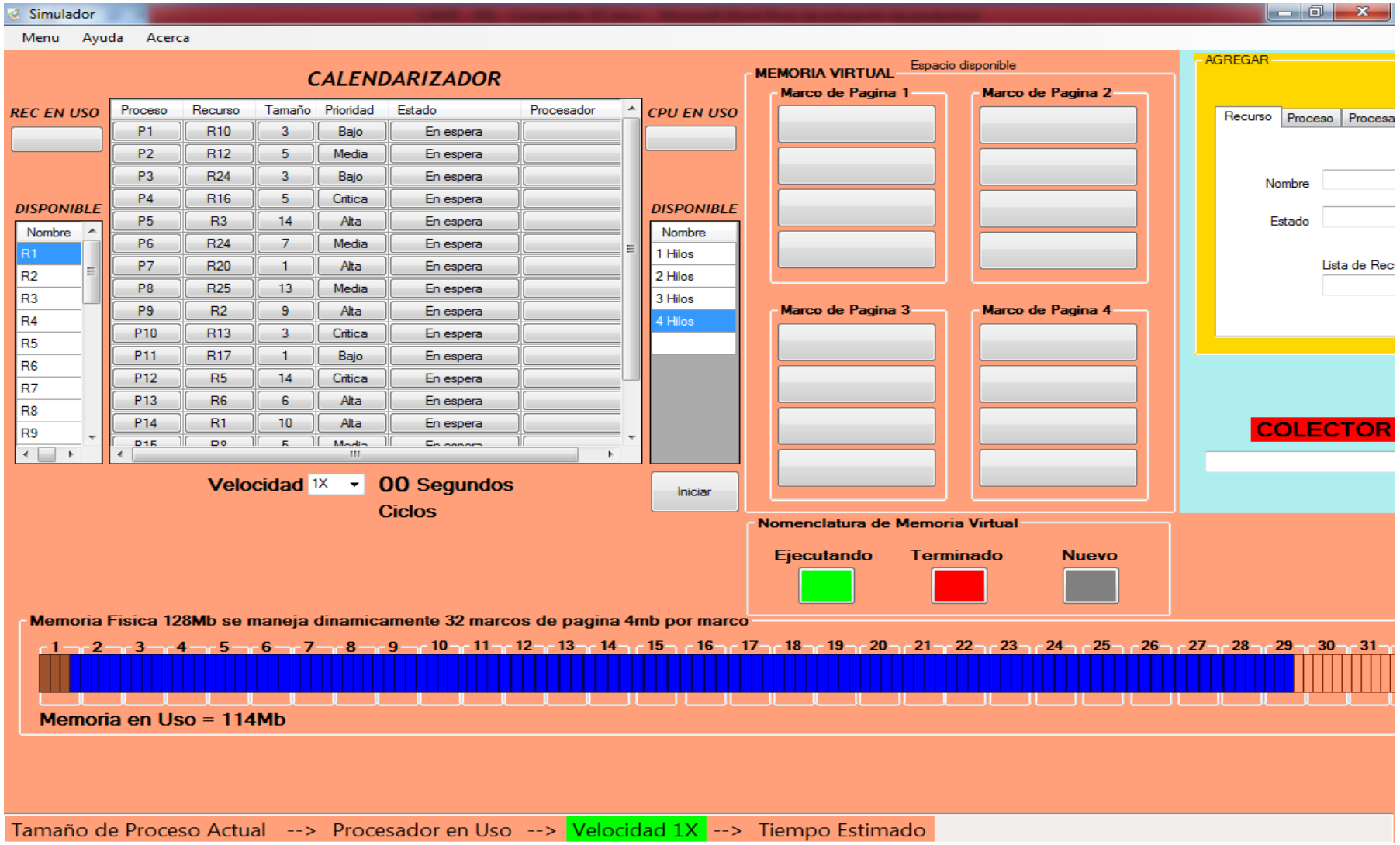


Figura 0.101: Procesos cargador, ejecución con múltiples hilos.

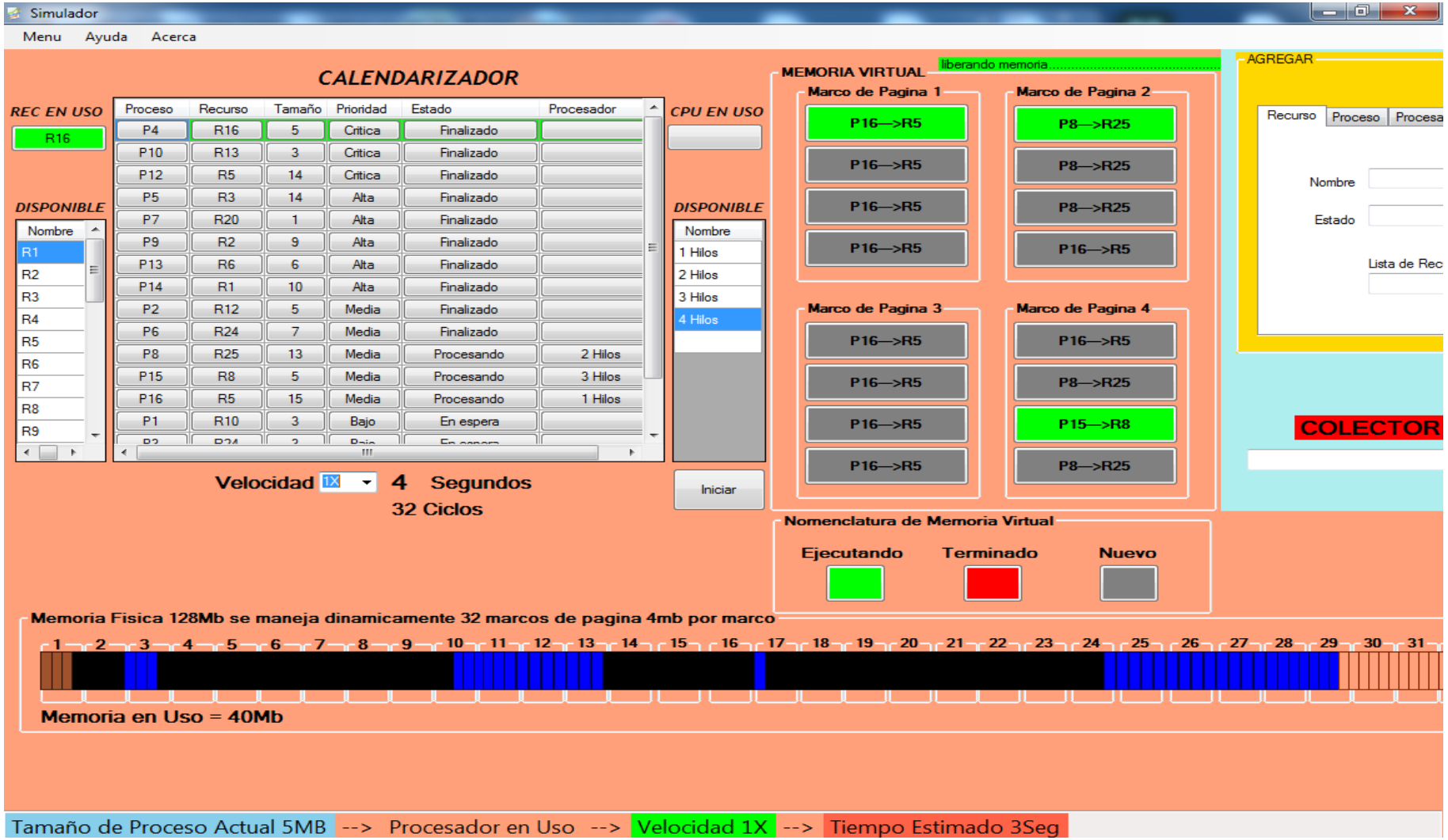


Figura 0.102: Procesos cargador, ejecución con múltiples hilos.

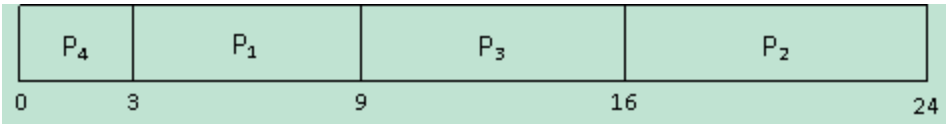
Planificación con selección del trabajo más corto

Un enfoque diferente para la planificación de la CPU es el algoritmo de programación con selección del trabajo más corto (***SJF*** – ***S****hortest-****J****ob-****F****irst*). Este algoritmo asocia con cada proceso la longitud de la siguiente ráfaga de CPU del proceso. Cuando la CPU está disponible, se asigna al proceso que tiene la siguiente ráfaga de CPU más pequeña. Si las ráfagas de CPU siguientes de dos procesos son las mismas, la planificación FCFS es utilizada para romper el empate. Tenga en cuenta que un término más adecuado para este método de programación sería el algoritmo de la siguiente ráfaga de CPU más corta, porque la planificación depende de la longitud de la siguiente ráfaga de CPU de un proceso, en lugar de su longitud total. Usamos el término SJF porque la mayoría de las personas y libros de texto utilizan este término para referirse a este tipo de planificación.

Como ejemplo de planificación SJF, considere el siguiente conjunto de procesos, con la longitud de la ráfaga de CPU dada en milisegundos

Proceso	Tiempo de ráfaga
P_1	6
P_2	8
P_3	7
P_4	3

A través de un diagrama de Gantt, mostraremos como trabaja una planificación SJF.



El algoritmo de planificación SJF es probablemente el óptimo, porque mover un proceso corto antes de uno largo disminuye el tiempo de espera del proceso corto en mayor medida de lo que aumenta el tiempo de espera del proceso largo. En consecuencia, la media del tiempo de espera disminuye.

Aunque el algoritmo SJF es óptimo, no se puede implementar en el nivel de programación de CPU, ya que no hay manera de conocer la longitud de la ráfaga de CPU siguiente. La

planificación SJF a veces se denomina planificación del proceso con tiempo restante más corto.

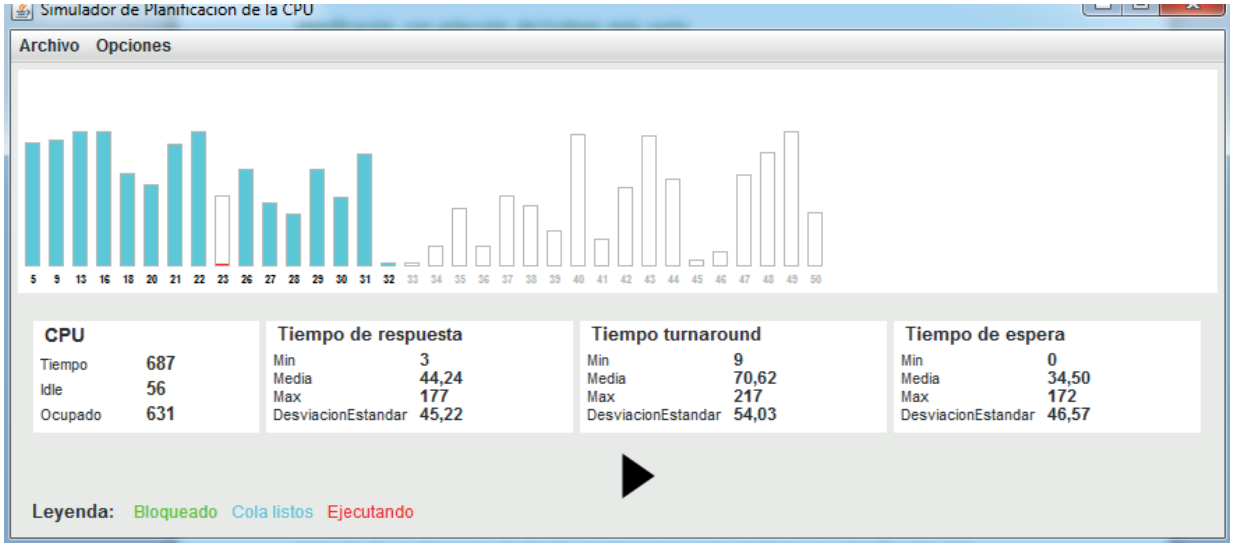


Figura 0.103: Algoritmo Planificación con selección del trabajo más corto y velocidad de 1000 fps.

Planificación por turno (Round-Robin)

El algoritmo de planificación por turnos (*RR*) está diseñado para los sistemas de tiempo compartido, es similar a la planificación FCFS, pero se añade la preferencia para permitir que el sistema cambie entre procesos. En estos sistemas se define una pequeña unidad de tiempo, llamada cantidad de tiempo o sector de tiempo. Se define una pequeña unidad de tiempo, llamada intervalo de tiempo o de tiempo. Un cuanto de tiempo es generalmente de 10 a 100 milisegundos de longitud. La cola de procesos en estado preparado se trata como una cola circular. El planificador de la CPU va alrededor de la cola de procesos en estado preparado, asignando la CPU a cada proceso para un intervalo de tiempo de hasta un cuanto de tiempo.

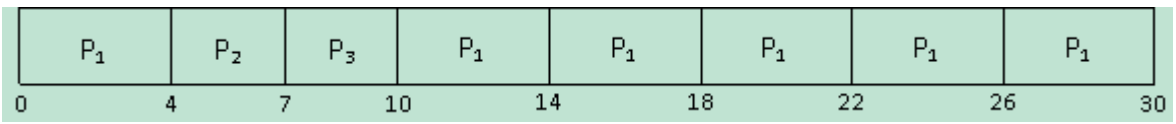
Para implementar la planificación de RR, de nuevo tratamos la cola de procesos en estado preparado como una cola FIFO de procesos. Los nuevos procesos se agregan al final de la cola de procesos en estado preparado. El planificador de la CPU selecciona el primer proceso de la cola de procesos en estado preparado, establece un temporizador para interrumpir después de un cuanto de tiempo, y despacha el proceso.

Durante este proceso, una de dos cosas sucederá. El proceso puede tener una ráfaga de CPU de menos de un cuanto de tiempo. En este caso, el propio proceso liberará la CPU voluntariamente. El planificador pasará al siguiente proceso en la cola de procesos en estado preparado. Ahora, si la ráfaga de CPU del proceso actualmente en ejecución es más de un cuanto de tiempo, el temporizador se apagará y causará una interrupción al sistema operativo. Se ejecutará un cambio de contexto, y el proceso se pondrá al final de la cola de procesos en estado preparado. El planificador de CPU seleccionará el siguiente proceso en la cola de procesos en estado preparado.

El tiempo promedio de espera bajo la política RR es a menudo largo. Considere el siguiente conjunto de procesos que llegan en el momento 0, con la longitud de la ráfaga de CPU dado en milisegundos:

Proceso	Tiempo de ráfaga
P_1	24
P_2	3
P_3	3

Si usamos un cuanto de tiempo de 4 milisegundos, entonces el proceso P_1 obtiene los primeros 4 milisegundos. Puesto que requiere otros 20 milisegundos, se adelanta después del cuántico por primera vez, y la CPU se da al siguiente proceso en la cola, proceso P_2. Ahora el proceso P_2 no necesita 4 milisegundos, por lo que se cierra antes de que expire su cuanto de tiempo. Entonces la CPU se da al siguiente proceso, el proceso P_3. Una vez que cada proceso ha recibido un cuanto de tiempo, la CPU se devuelve para procesar P_1 durante un tiempo adicional. La programación RR resultante es la siguiente:



En el algoritmo de planificación RR, no se asigna ningún proceso a la CPU durante más de un cuanto de tiempo en cada turno (a menos que sea el único proceso ejecutable). Si la ráfaga de CPU de un proceso excede 1 cuanto de tiempo, ese proceso se adelanta y se

vuelve a poner en la cola de procesos en estado preparado. El algoritmo de planificación RR es, por lo tanto, un mecanismo de desalojo.

El rendimiento del algoritmo RR depende en gran medida del tamaño del cuanto de tiempo. En un extremo, si el cuanto de tiempo es extremadamente grande, la política RR es la misma que la política FCFS. Por el contrario, si el cuanto de tiempo es extremadamente pequeño (digamos, 1 milisegundo), el enfoque RR puede dar lugar a un gran número de cambios de contexto. Supongamos, por ejemplo, que sólo tenemos un proceso de 10 unidades de tiempo. Si el cuanto es de 12 unidades de tiempo, el proceso termina en menos de 1 tiempo cuántico, sin sobrecarga. Sin embargo, si el cuanto es de 6 unidades de tiempo, el proceso requiere 2 quantos, lo que resulta en un cambio de contexto. Si el cuanto de tiempo es 1 unidad de tiempo, entonces se producirán nueve cambios de contexto, ralentizando la ejecución del proceso en consecuencia (Figura 0.3).

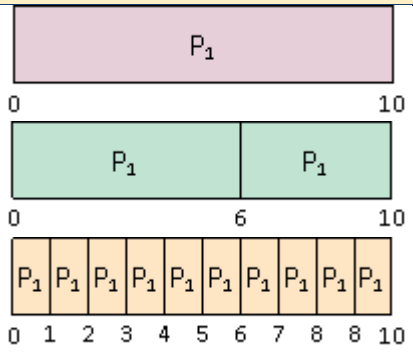
Proceso con un tiempo de 10	Cuanto de tiempo	Intercambios de contexto
	12 6 1	0 1 9

Figura 0.3: La relación entre el cuanto de tiempo y el intercambio de contextos

Por lo tanto, queremos que el cuándo de tiempo sea más grande con respecto al tiempo de cambio de contexto. Si el tiempo de cambio de contexto es aproximadamente el 10 por ciento del cuanto de tiempo, entonces alrededor del 10 por ciento del tiempo de CPU se gastará en el cambio de contexto. En la práctica, la mayoría de los sistemas modernos tienen cantidad de tiempo que oscila entre 10 y 100 milisegundos. El tiempo necesario para un modificador de contexto suele ser inferior a 10 microsegundos; por lo tanto, el tiempo de cambio de contexto es una pequeña fracción del cuanto de tiempo. El tiempo de respuesta también depende del tamaño del cuántico de tiempo.

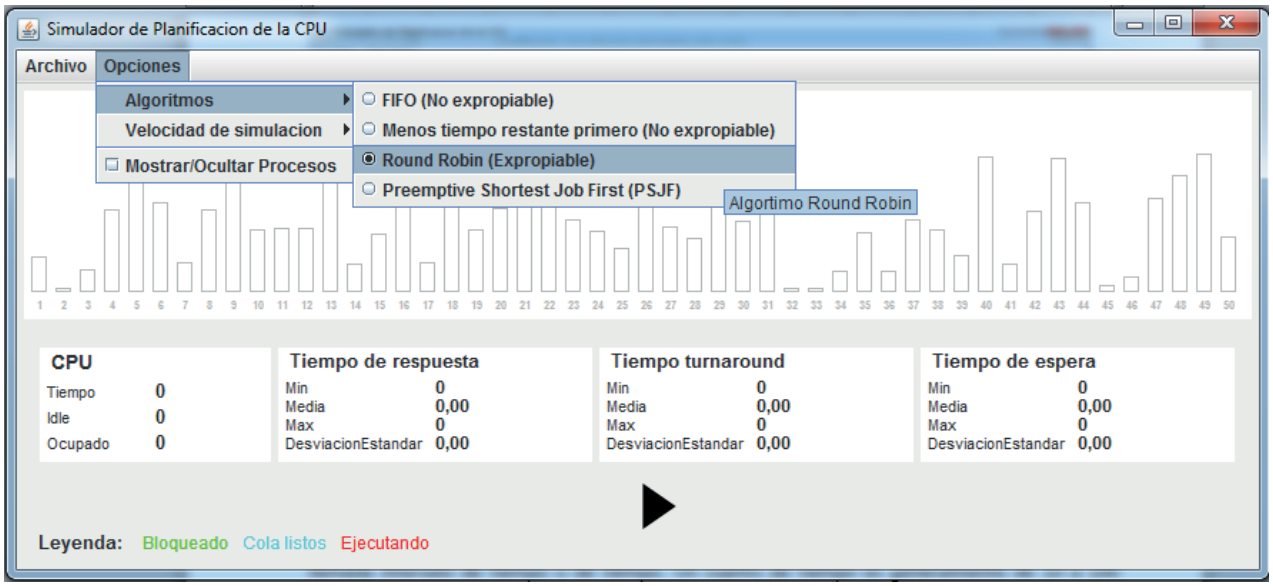


Figura 0.104: Algoritmo Planificación por turno (Round-Robin) velocidad de 1000 fps

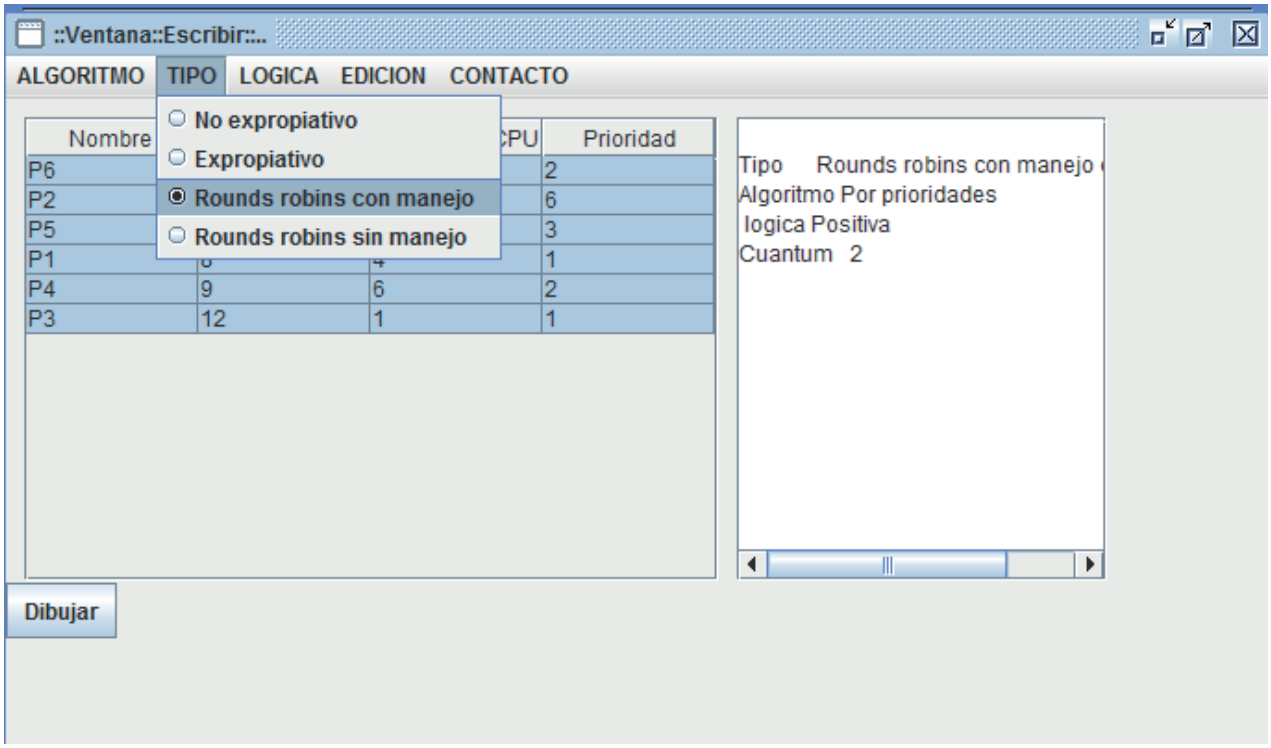


Figura 0.105: Algoritmo Planificación por turno (Round-Robin).

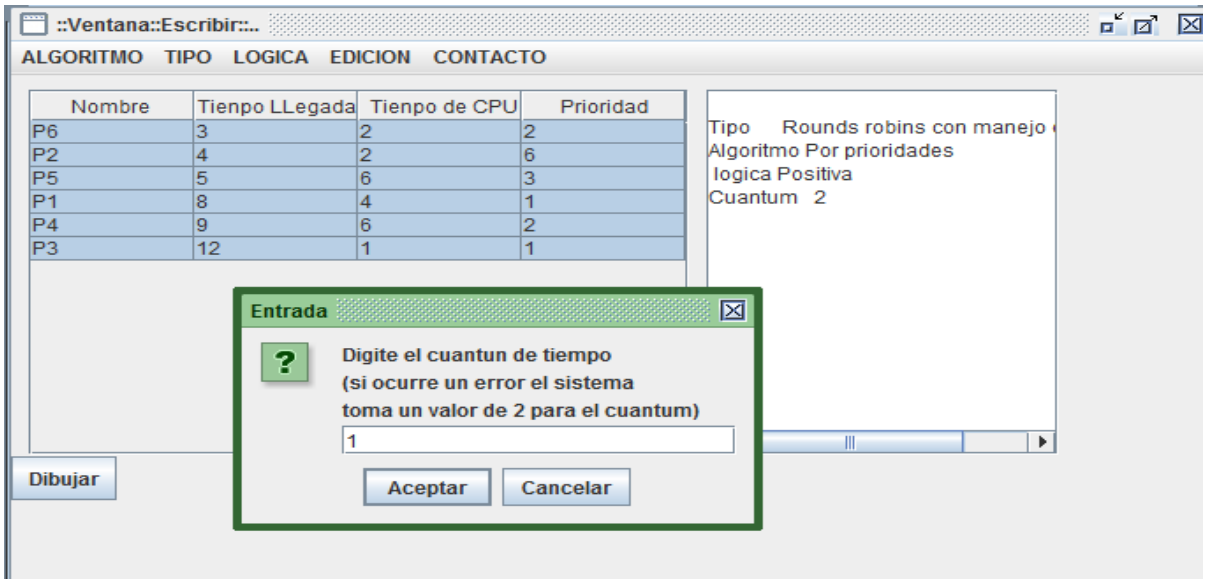


Figura 0.106: Definición de valor del cuantum para el algoritmo planificación por turno (Round-Robin).

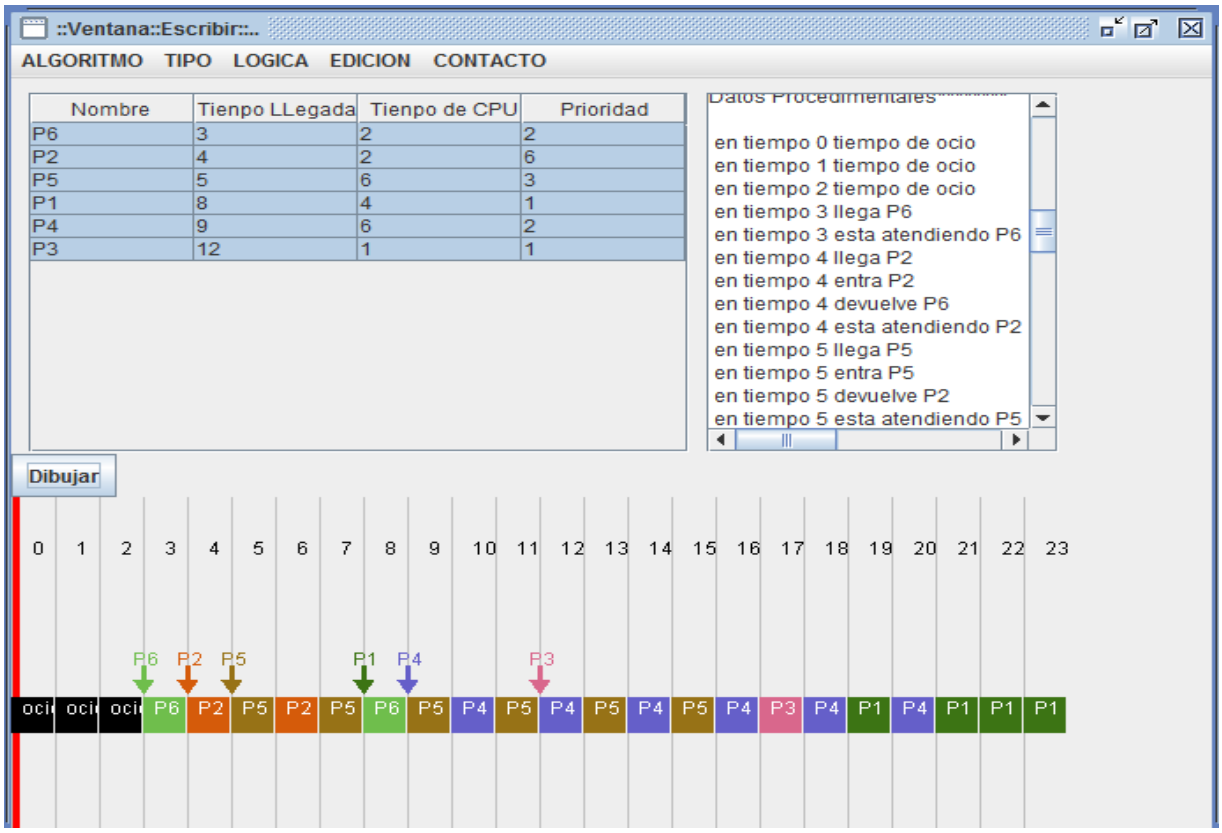


Figura 0.107: Resultado de los procesos y prioridad de llegada planificación por turno (Round-Robin).

Planificación por prioridades

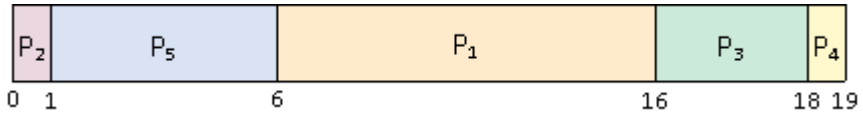
El algoritmo SJF es un caso especial del algoritmo de planificación por prioridades general. Una prioridad es asociada a cada proceso, y la CPU se asigna al proceso con la prioridad más alta. Los procesos de igual prioridad se planifican en orden FCFS. Un algoritmo SJF es simplemente un algoritmo de prioridad donde la prioridad (p) es la inversa de la ráfaga de CPU siguiente (predicha). Cuanto mayor sea la ráfaga de CPU, menor será la prioridad y viceversa.

Tenga en cuenta que discutimos la planificación en términos de alta prioridad y baja prioridad. Las prioridades se indican generalmente mediante un rango fijo de números, como 0 a 7 o 0 a 4.095. Sin embargo, no hay un acuerdo general sobre si 0 es la prioridad más alta o más baja. Algunos sistemas utilizan números bajos para representar una prioridad baja; otros usan números bajos para la prioridad alta. Esta diferencia puede llevar a confusión. En este texto, suponemos que los números bajos representan una prioridad alta.

Por ejemplo, considere el siguiente conjunto de procesos, que se supone han llegado en el momento 0 en el orden P_1, P_2, …, P_5, con la longitud de la ráfaga de CPU dada en milisegundos:

Proceso	Tiempo de ráfaga	Prioridad
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

A través de un diagrama de Gantt, mostraremos como trabaja una planificación por prioridades.



Las prioridades se pueden definir interna o externamente. Las prioridades definidas internamente utilizan alguna cantidad medible para calcular la prioridad de un proceso. Por ejemplo, los límites de tiempo, los requisitos de memoria, el número de

archivos abiertos y la relación entre la ráfaga de E/S media y la ráfaga de CPU media. Las prioridades externas se establecen por criterios fuera del sistema operativo, como la importancia del proceso, el tipo y la cantidad de fondos que se pagan por el uso de computadoras, el departamento que patrocina el trabajo y otros factores, a menudo políticos.

La planificación por prioridades puede ser preventiva o no preventiva. Cuando un proceso llega a la cola de procesos en estado preparados, su prioridad se compara con la prioridad del proceso actualmente en ejecución. El algoritmo de planificación de prioridad anterior adelantará la CPU si la prioridad del proceso recién llegado es mayor que la prioridad del proceso que se está ejecutando actualmente. Un algoritmo de programación de prioridad no anterior simplemente pondrá el nuevo proceso en la cabeza de la cola de procesos en estado preparado.

Un problema importante con los algoritmos de planificación por prioridades es el bloqueo indefinido o la muerte por inanición. Un proceso que está listo para ejecutarse pero que está esperando para tener acceso a una CPU puede considerarse bloqueado. Un algoritmo de planificación de prioridades puede dejar algunos procesos de baja prioridad esperando indefinidamente. En un sistema informático muy cargado, un flujo constante de procesos de mayor prioridad puede evitar que un proceso de baja prioridad pueda obtener acceso a la CPU. Generalmente, una de dos cosas sucederá. O bien el proceso eventualmente se ejecutará (a las 2 a.m. del domingo, cuando el sistema finalmente se carga ligeramente), o el sistema informático finalmente se bloqueará y perderá todos los procesos de baja prioridad inacabados. (Se rumorea que cuando cerraron el IBM 7094 en el MIT en 1973, encontraron un proceso de baja prioridad que había sido presentado en 1967 y aún no se había ejecutado.)

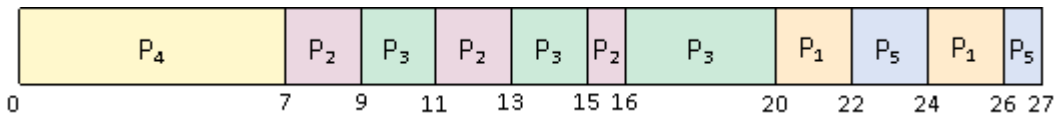
Una solución al problema del bloqueo indefinido de los procesos de baja prioridad es el envejecimiento. El envejecimiento implica aumentar gradualmente la prioridad de los procesos que esperan en el sistema durante mucho tiempo. Por ejemplo, si las prioridades oscilan entre 127 (baja) y 0 (alta), podríamos aumentar periódicamente (digamos, cada segundo) la prioridad de un proceso de espera en 1. Eventualmente, incluso un proceso

con una prioridad inicial de 127 tendría la prioridad más alta en el sistema y se ejecutaría. De hecho, un proceso de prioridad 127 tardaría un poco más de 2 minutos en envejecer hasta un proceso de prioridad 0.

Otra opción es combinar la planificación por turnos (RR) y la de prioridades de tal manera que el sistema ejecute el proceso de mayor prioridad y ejecute procesos con la misma prioridad mediante la planificación por turnos (RR). Vamos a ilustrar con un ejemplo utilizando el siguiente conjunto de procesos, con el tiempo de ráfaga en milisegundos:

Proceso	Tiempo de ráfaga	Prioridad
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

Usando la planificación por prioridades con la planificación por turnos (RR) para procesos con la misma prioridad, planificaríamos estos procesos de acuerdo con el siguiente diagrama de Gantt usando un período de tiempo de 2 milisegundos:



En este ejemplo, el proceso P_4 tiene la prioridad más alta, por lo que se ejecutará hasta su finalización. Los procesos P_2 y P_3 tienen la siguiente prioridad más alta, y se ejecutarán según la planificación por turnos (RR). Observe que cuando el proceso P_2 finaliza en el momento 16, el proceso P_3 es el proceso de mayor prioridad, por lo que se ejecutará hasta que complete la ejecución. Ahora, sólo quedan los procesos P_1 y P_5, y como tienen la misma prioridad, se ejecutarán en según la planificación por turnos (RR) hasta que se completen.

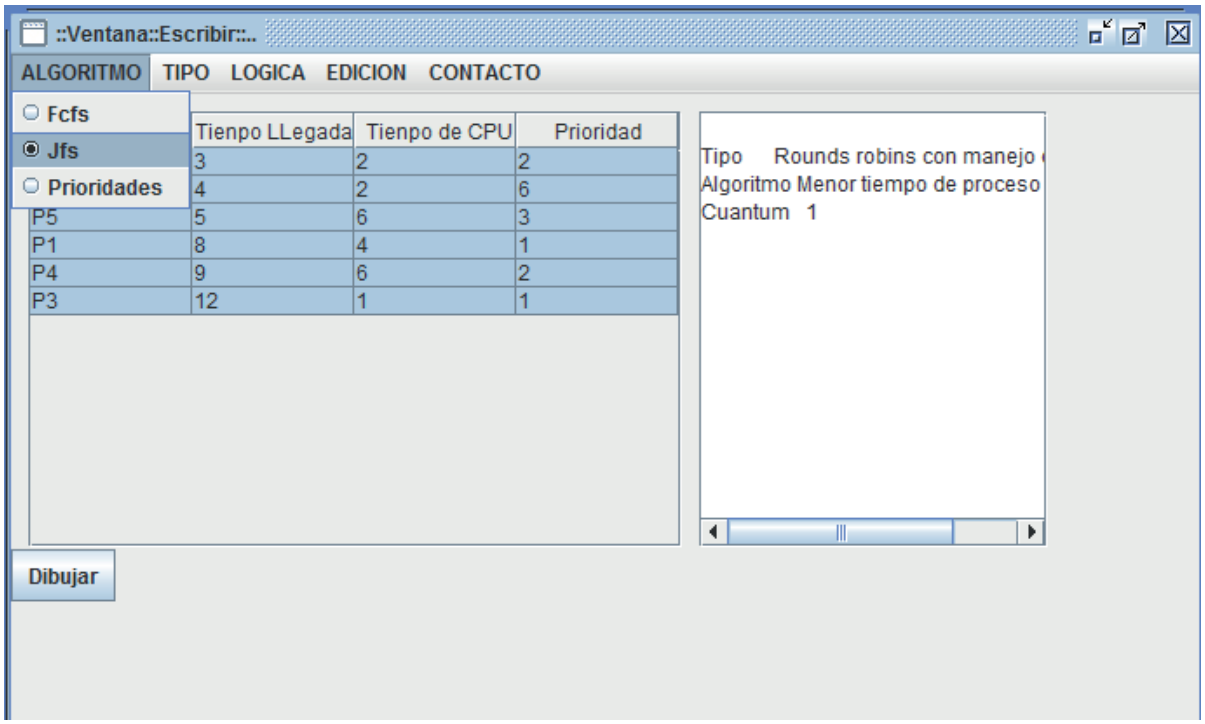


Figura 0.108: Algoritmo de Planificación por prioridades Jfs.

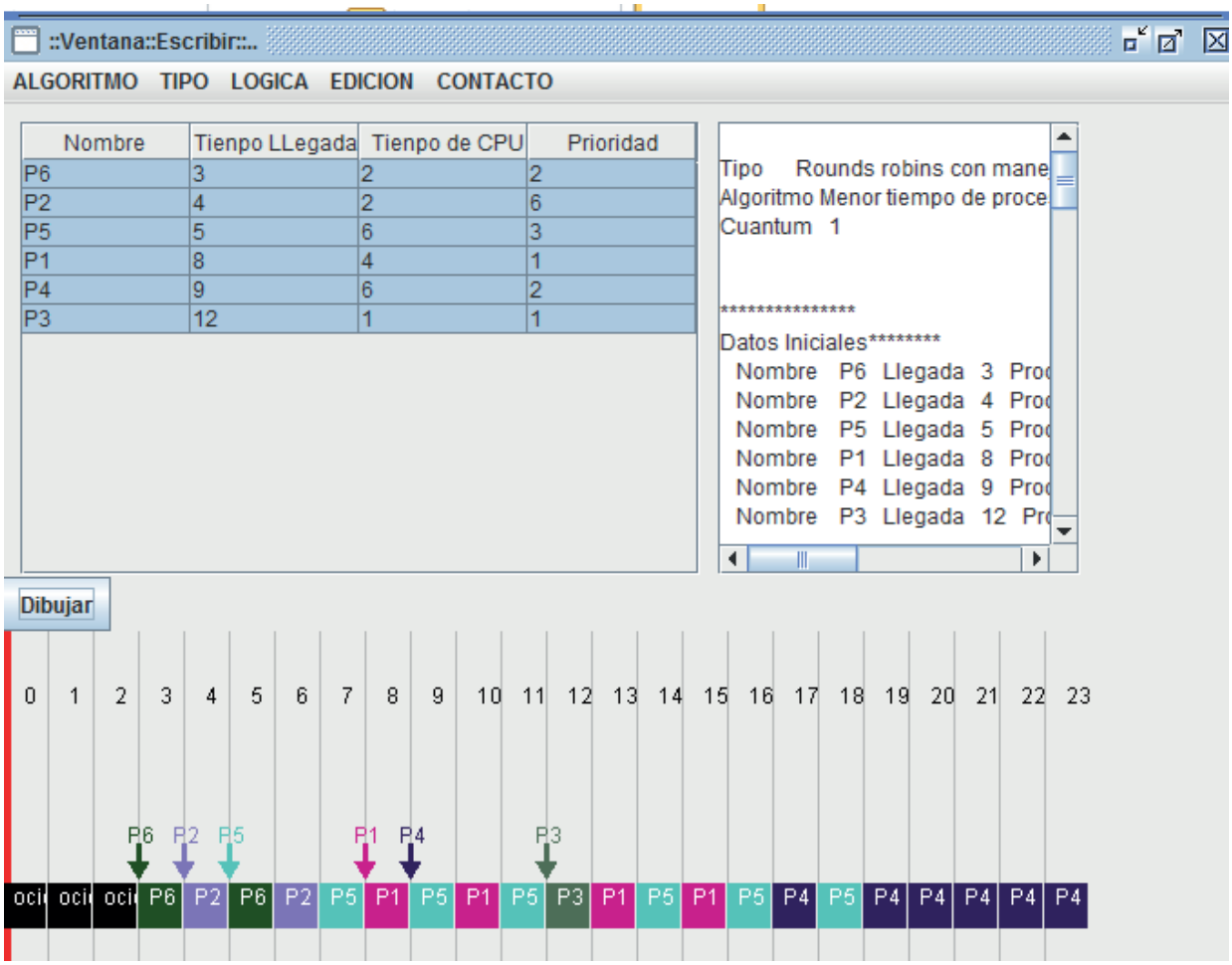


Figura 0.109: Tiempo de llegada de los procesos con su prioridad algoritmo Jfs.

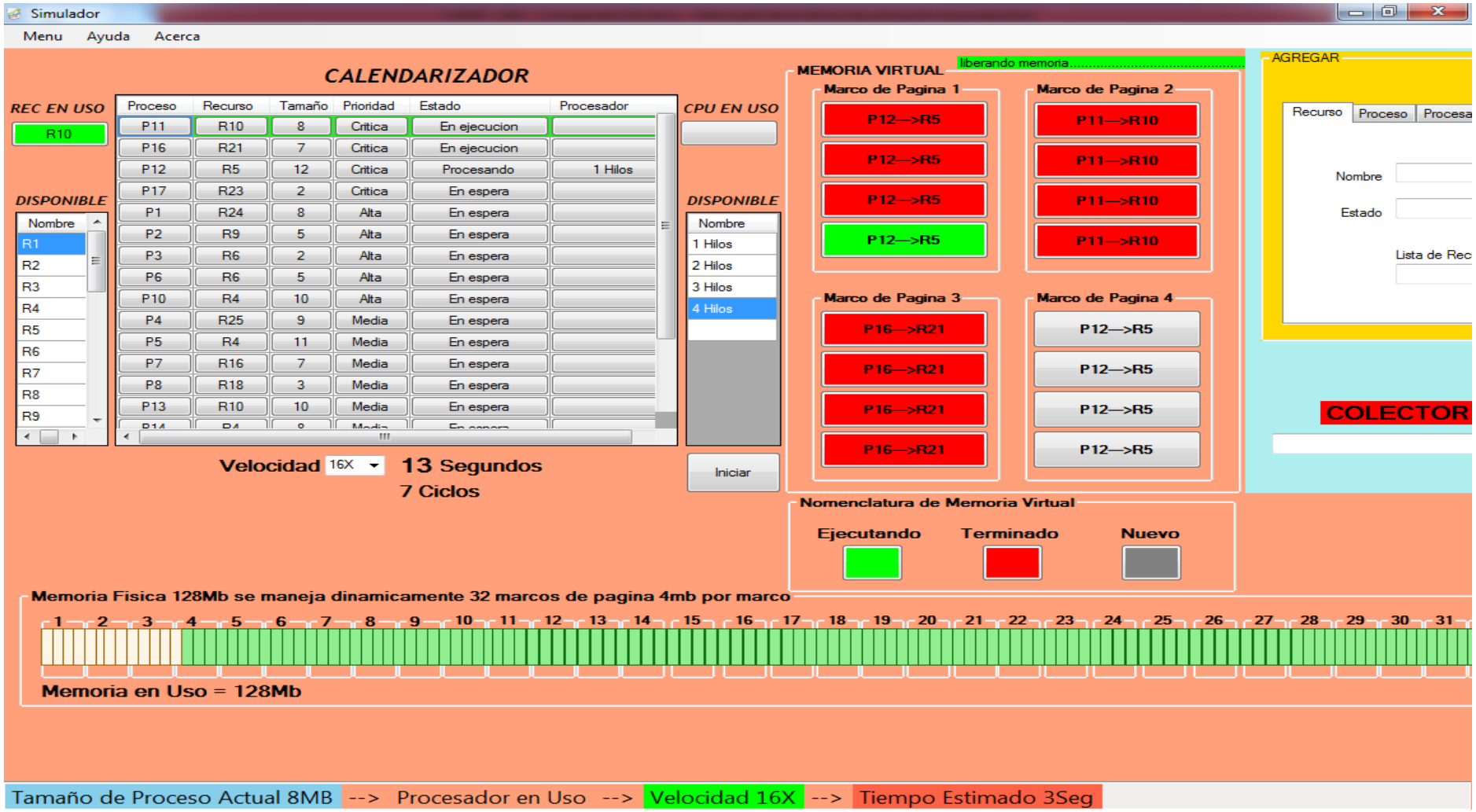


Figura 0.110: Tiempo de llegada de los procesos con su prioridad algoritmo Jfs

Planificación mediante colas multinivel

Con las planificaciones por prioridades y por turnos (RR), todos los procesos se pueden colocar en una sola cola y, a continuación, el planificador selecciona el proceso con la prioridad más alta para ejecutarse. Dependiendo de cómo se administren las colas, puede ser necesaria una búsqueda O(n) para determinar el proceso de mayor prioridad. En la práctica, a menudo es más fácil tener colas independientes para cada prioridad distinta, y la planificación por prioridades simplemente planifica el proceso en la cola de mayor prioridad. Esto se ilustra en la Figura 0.. Este enfoque, es conocido como colas multinivel, además funciona muy bien cuando la planificación por prioridades es combinada con la planificación por turnos (RR): si hay varios procesos en la cola de mayor prioridad, se ejecutan según el orden de la planificación por turnos (RR). En la forma más generalizada de este enfoque, se asigna una prioridad estáticamente a cada proceso y un proceso permanece en la misma cola durante el tiempo de ejecución.

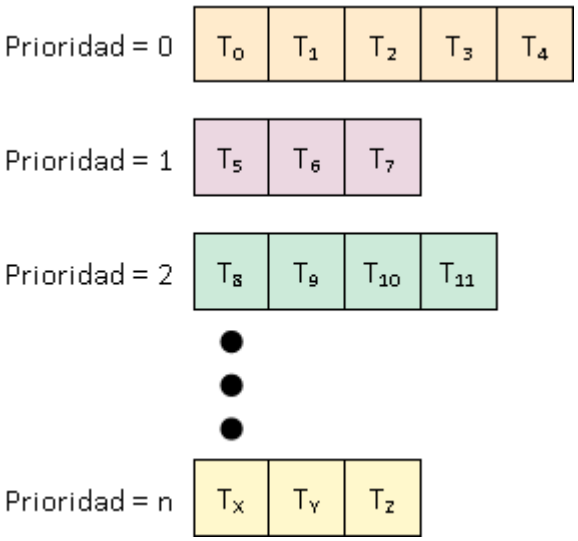


Figura 0.11: Colas separadas para cada prioridad

Un algoritmo de planificación mediante colas multinivel también se puede utilizar para particionar procesos en varias colas independientes basadas en el tipo de proceso (Figura 0.). Por ejemplo, se realiza una división común entre los procesos de primer plano (interactivos) y los procesos en segundo plano (procesamiento por lote). Estos dos tipos de procesos tienen diferentes requisitos de tiempo de respuesta, por lo que pueden tener diferentes necesidades de planificación. Además, los procesos en primer

plano pueden tener prioridad (definida externamente) sobre los procesos en segundo plano. Las colas independientes se pueden utilizar para los procesos en primer plano y en segundo plano, y cada cola puede tener su propio algoritmo de planificación. La cola en primer plano puede ser planificada por un algoritmo de planificación por turnos (RR), mientras que la cola en segundo plano está planificada por un algoritmo por orden de llegada (FCFS).

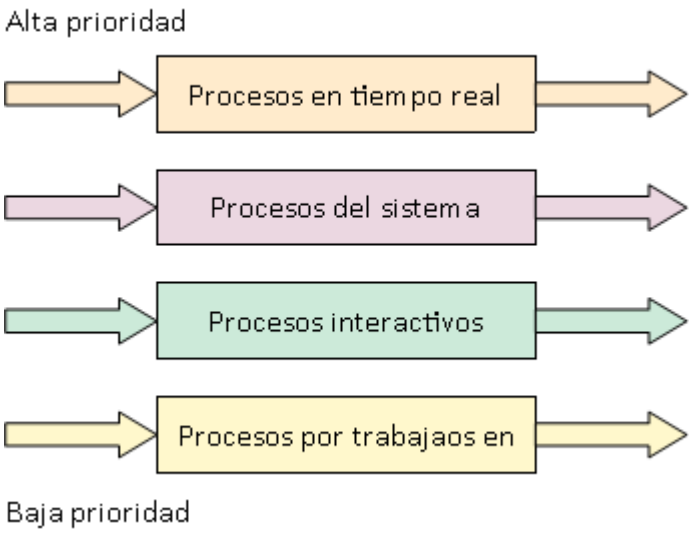


Figura 0.12: Planificación mediante colas multinivel

Además, debe haber una planificación entre las colas, que comúnmente se implementa como planificación preventiva de prioridad fija. Por ejemplo, la cola en tiempo real puede tener prioridad absoluta sobre la cola interactiva.

Echemos un vistazo a un ejemplo de un algoritmo de programación de colas multinivel con cuatro colas, que se enumeran a continuación en orden de prioridad:

1. Procesos en tiempo real
2. Procesos del sistema
3. Procesos interactivos

4. Procesos por lotes

Cada cola tiene prioridad absoluta sobre las colas de menor prioridad. Ningún proceso en la cola por lotes, por ejemplo, podría ejecutarse a menos que las colas para procesos en tiempo real, procesos del sistema y procesos interactivos estuvieran vacías. Si un proceso interactivo entró en la cola de procesos en estado preparado mientras se ejecutaba un proceso por lotes, el proceso por lotes se apretaría.

Otra posibilidad es cortar el tiempo entre las colas. Aquí, cada cola obtiene una cierta porción del tiempo de la CPU, que entonces puede planificar entre sus diversos procesos. Por ejemplo, en el caso de la cola con procesos en primer plano y segundo plano, la cola en primer plano se puede disponer del 80 por ciento del tiempo de la CPU para la planificación por turnos (RR) entre sus procesos, mientras que la cola en segundo plano recibe el 20 por ciento de la CPU para gestionar sus procesos mediante una planificada por orden de llegada (FCFS).

Planificación mediante colas multinivel retroalimentadas

Normalmente, cuando se utiliza el algoritmo de planificación mediante colas multinivel, los procesos se asignan permanentemente a una cola cuando entran en el sistema. Si hay colas independientes para los procesos en primer plano y en segundo plano, por ejemplo, los procesos no se mueven de una cola a la otra, ya que los procesos no cambian su naturaleza en primer plano o en segundo plano. Esta configuración tiene la ventaja de una baja carga de trabajo de planificación, pero es inflexible.

Por su parte el algoritmo de planificación mediante colas multinivel retroalimentadas permite que un proceso se mueva entre colas. La idea es separar los procesos de acuerdo con las características de sus ráfagas de CPU. Si un proceso utiliza demasiado tiempo de CPU, se moverá a una

cola de menor prioridad. Este esquema deja los procesos interactivos y enlazados a E/S, que normalmente se caracterizan por ráfagas cortas de CPU, en las colas de mayor prioridad. Además, un proceso que espera demasiado tiempo en una cola de menor prioridad se puede mover a una cola de mayor prioridad. Esta forma de envejecimiento previene el bloqueo indefinido.

Por ejemplo, considere una planificación mediante colas multinivel retroalimentadas con tres colas, numeradas de 0 a 2 (Figura 0.413). El planificador ejecuta primero todos los procesos en la cola 0. Sólo cuando la cola 0 está vacía ejecutará los procesos en la cola 1. Del mismo modo, los procesos de la cola 2 se ejecutarán solo si las colas 0 y 1 están vacías. Un proceso que llega para la cola 1 adelantará un proceso en la cola 2. Un proceso en la cola 1 será a su vez anticipado por un proceso que llega para la cola 0.

Un proceso de entrada se coloca en la cola 0. Un proceso en la cola 0 recibe un cuanto de tiempo de 8 milisegundos. Si no termina dentro de este tiempo, se mueve a la cola de la cola 1. Si la cola 0 está vacía, el proceso en la cabecera de la cola 1 recibe un cuanto de 16 milisegundos. Si no completa, se adelanta y se pone en la cola 2. Los procesos en la cola 2 se ejecutan sobre una planificación por orden de llegada (FCFS), pero se ejecutan solamente cuando las colas 0 y 1 están vacías. Para evitar la inanición, un proceso que espera demasiado tiempo en una cola de menor prioridad se puede mover gradualmente a una cola de mayor prioridad.

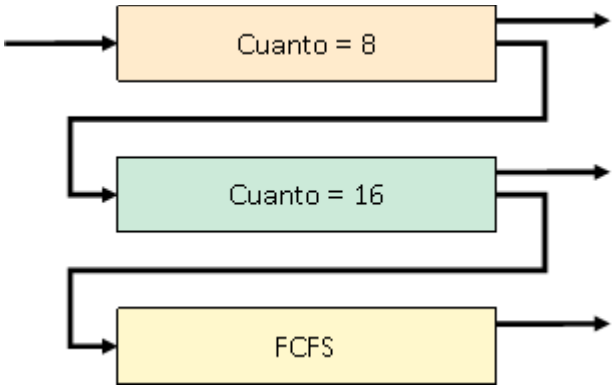


Figura 0.4: Colas multinivel retroalimentadas

Este algoritmo de planificación da la prioridad más alta a cualquier proceso con una ráfaga de CPU de 8 milisegundos o menos. Tal proceso conseguirá rápidamente a la CPU y terminaran su ráfaga de CPU y se irá a su siguiente ráfaga de E/S. Los procesos que necesitan más de 8 pero menos de 24 milisegundos también se sirven rápidamente, aunque con menor prioridad que los procesos más cortos. Los procesos largos se van automáticamente a la cola 2 y se sirven según la planificación por orden de llegada (FCFS) con los ciclos de CPU sobrantes de las colas 0 y 1.

En general, un planificador de colas multinivel retroalimentadas se define mediante los siguientes parámetros:

- El número de colas
- El algoritmo de planificación para cada cola
- El método utilizado para determinar cuándo actualizar un proceso a una cola de prioridad más alta
- El método utilizado para determinar cuándo a distanciar un proceso a una cola de prioridad más baja
- El método utilizado para determinar en qué cola entrará un proceso cuando ese proceso necesite servicio

La definición de un planificador de colas multinivel retroalimentadas lo convierte en el algoritmo de planificación de la CPU más general. Se puede

configurar para que coincida con un sistema específico que se desee diseñar. Desafortunadamente, también es el algoritmo más complejo, ya que la definición del mejor planificador requiere algunos medios para seleccionar valores para todos los parámetros.

Gestión de procesos

Todos los hilos comparten el mismo espacio de direcciones, y así comparten también las mismas variables globales. Mientras que los procesos pueden ser de distintos usuarios y competir por recursos entre sí, los hilos de un mismo proceso de usuario cooperan entre si, sin luchar entre sí.

Al igual que los procesos tradicionales, los hilos pueden estar en alguno de los siguientes estados:

En ejecución (Running): cuando el hilo posee la CPU y se encuentra activo.

Bloqueado (Suspend): cuando el hilo se encuentra esperando algún evento o a la espera de que otro libere el bloqueo por el que se encuentra detenido.

Listo o preparado (Ready): cuando el hilo está preparado para su ejecución, y se encuentra a la espera de ser elegido por el planificador.

Terminado (Finished): cuando el hilo ha finalizado, pero todavía no ha sido recogido por el hilo padre, aunque no puede ser planificado nunca más.

Un proceso puede tener uno o más hilos. Los hilos son un mecanismo que permite mejorar el rendimiento de los sistemas operativos tratando de reducir la sobrecarga producida por el cambio de contexto entre procesos. Los hilos de un mismo proceso comparten los recursos (memoria, archivos, etc.), y son la unidad de planificación. Así, un proceso será un objeto

estático que posee un conjunto de recursos para una serie de hilos, que son los objetos dinámicos panificables.

Los hilos en un entorno multihilo tienen las siguientes características que pueden hacerles deseables en muchas aplicaciones que requieren multitarea:

- ✓ **Necesitan poca memoria.**
- ✓ **Tienen un bajo coste de creación.**
- ✓ **Tienen un bajo coste de sincronización.**
- ✓ **Comparten el mismo espacio de direcciones.**
- ✓ **Pueden progresar independientemente unos de otros.**

Planificación del procesador o CPU

La función del **planificador del procesador** o **planificador de CPU** (*CPU scheduler*) es la de seleccionar entre los procesos que están listos en la cola de espera, y luego verificar si algún CPU se desocupó y asignan ese núcleo de CPU a uno de procesos de la cola. El planificador de CPU debe seleccionar un nuevo proceso para la CPU frecuentemente, esto se debe a que un proceso enlazado a E/S puede ejecutarse solo durante unos pocos milisegundos antes de esperar una solicitud de E/S, mientras que un proceso enlazado a la CPU requerirá un núcleo de CPU para duraciones más largas, esto hace que sea poco probable que el planificador conceda el núcleo a un proceso durante un período prolongado.

En su lugar, es probable que esté diseñado para quitar por la fuerza la CPU de un proceso y planificar otro proceso para ejecutarse. Por lo tanto, el planificador de CPU se ejecuta al menos una vez cada 100 milisegundos, aunque normalmente con mucha más frecuencia. Existen algunos sistemas operativos que utilizan otra técnica de planificación llamada **intercambio** (*swapping*), la cual busca eliminar un proceso de la memoria

y la CPU copiándola al disco, para poder asignar ese recurso a otro proceso buscando reducir de esta manera el grado de multiprogramación, luego este proceso puede ser reintroducido y continuar su ejecución donde se quedó, copiando el proceso de nuevo a la memoria y eliminándola del disco, es decir, haciendo un intercambio entre el disco y la memoria, este proceso es recomendable cuando la memoria física del computador se ha sobre comprometido y debe liberarse.

Herramientas de entornos digitales

Herramientas SO	Link de descarga
Ubuntu 20.04	http://releases.ubuntu.com/focal/
Redhat	https://access.redhat.com/downloads
Simulador de máquinas virtuales L-W	http://copy.sh/v86/

Bibliografía

Redhat. (2021, 04 26). *RedHat*. Retrieved from https://www.redhat.com/es/topics/virtualization/what-is-a-virtual-machine

Serna, M., & Allende, S. (2020). Sistemas operativos: Linux. In J. Sarmiento (Ed.). Universitas.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating Systems Concepts* (10th ed.). Hoboken: Wiley.

Stallings, W. (2018). *Operating Systems, Internals and Design Princiles* (Ninth ed.). Malaysia: Pearson.

vmware. (2021). *vmware*. Retrieved 04 26, 2021, from https://www.vmware.com/es/topics/glossary/content/virtual-desktops.html

Wikipedia. (2020, 11 15). *BootX (Apple)*. Retrieved from BootX (Apple): https://en.wikipedia.org/wiki/BootX_(Apple)

Wikipedia. (2020, 11 15). *GNU GRUB*. Retrieved from GNU GRUB: https://en.wikipedia.org/wiki/GNU_GRUB

Wikipedia. (2020, 11 15). *iBoot*. Retrieved from iBoot: https://en.wikipedia.org/wiki/IBoot

Wikipedia. (2020, 11 15). *NTLDR*. Retrieved from NTLDR: https://en.wikipedia.org/wiki/NTLDR

Wikipedia. (2020, 11 15). *Windows NT 6 startup process*. Retrieved from Windows NT 6 startup process: https://en.wikipedia.org/wiki/Windows_NT_6_startup_process

CAPÍTULO III

Concurrencia, exclusión mutua y sincronización

3 Resultado de aprendizaje de la asignatura

Conocer los conceptos fundamentales, partiendo del enfoque clásico de los sistemas operativos como administrador eficiente de recursos, componentes, estructuras y funciones de los sistemas operativos, con la

finalidad que entiendan el funcionamiento y puedan realizar software de sistemas.

Unidad 3: Concurrencia, exclusión mutua y sincronización

Resultado de aprendizaje de la unidad:

Describir conceptos relacionados con la concurrencia, exclusión mutua y sincronización como base para su aplicación en el entorno empresarial.

Tema 1: Concurrencia

Los procesos concurrentes esperan que los usuarios puedan ejecutar varios procesos a la vez y no verse obligado a ejecutar los programas de forma secuencial. Los sistemas operativos multitarea, como Windows y Linux, se encargan de que varios programas se puedan ejecutar al mismo tiempo de manera concurrente incluso disponiendo de una sola CPU.

Se debe comprender que la programación concurrente es el conjunto de técnicas y notaciones que sirven para expresar el paralelismo potencial en los programas, así como resolver problemas de comunicación y sincronización. Cuando se trabaja en entornos distribuidos o con máquinas con múltiples procesadores, se suele hablar de programación distribuida o paralela, respectivamente. El diseño de los sistemas operativos está centrado en la gestión de procesos e hilos, donde sus temas

centrales son: multiprogramación, multiprocesamiento y procesamiento distribuido (Berzal, 2019).

Pero la concurrencia, también llamada simultaneidad es el tema central y fundamental en todas estas áreas, e incluso en el diseño de los sistemas operativos, además abarca varios aspectos entre los cuales se destacan la comunicación entre procesos, el uso compartido y la competencia de recursos (como memoria, archivos y acceso de E/S), la sincronización de las actividades de varios procesos y la asignación de tiempo de procesador a los procesos.

Veremos que estos problemas surgen no sólo en entornos de multiprocesamiento y procesamiento distribuido, sino también en sistemas multiprogramados de un solo procesador.

La concurrencia aparece en tres contextos diferentes: múltiples aplicaciones, aplicaciones estructuradas y estructura del sistema operativo

Principios de concurrencia

En un sistema multiprogramado de un solo procesador, los procesos se entrelazan a tiempo para producir la apariencia de la ejecución simultánea (ver Figura.1). Aunque no se logra el procesamiento paralelo real, y aunque hay una cierta cantidad de sobrecarga implicada en el cambio de ida y vuelta entre procesos, la ejecución entrelazada proporciona importantes beneficios en la eficiencia del procesamiento y en la estructuración del programa. En un sistema multiprocesador, no sólo es posible intercalar la ejecución de varios procesos, sino también superponerlos.

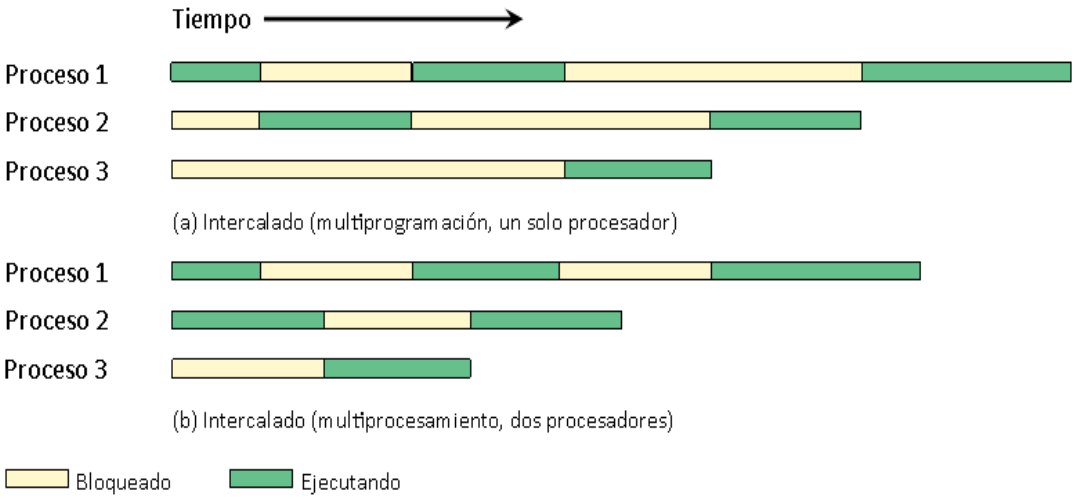


Ilustración 9 Multiprogramación y multiprocesamiento

A primera vista, puede parecer que el entrelazado y la superposición representan modos de ejecución fundamentalmente diferentes y que presentan diferentes problemas. De hecho, ambas técnicas se pueden ver como ejemplos de procesamiento concurrente (simultáneo) y ambas presentan los mismos problemas. En el caso de un monoprocesador, los problemas se derivan de una característica básica de los sistemas multiprogramados: no se puede predecir la velocidad relativa de ejecución de los procesos. Esta depende de las actividades de otros procesos, de la forma en que el sistema operativo maneja las interrupciones y las directivas o políticas de programación del sistema operativo. Surgen las siguientes dificultades:

1 La compartición de recursos globales está cargada de peligros. Por ejemplo, si dos procesos utilizan ambos la misma variable global y ambos realizan lecturas y escrituras sobre esa variable, entonces el orden en que se ejecuten las lecturas y escrituras es crítico. En la siguiente subsección se muestra un ejemplo de este problema.

2 Para el sistema operativo es complicado gestionar la asignación de recursos de manera óptima. Por ejemplo, el proceso A puede solicitar

el uso de un canal concreto de E/S, y serle concedido el control, y luego ser suspendido justo antes de utilizar ese canal. Puede no ser deseable que el sistema operativo simplemente bloquee el canal e impida su utilización por otros procesos; de hecho, esto puede conducir a una condición de interbloqueo.

3 Llega a ser muy complicado localizar errores de programación porque los resultados son típicamente no deterministas (incierto) y no reproducibles.

Todas las dificultades precedentes se presentan también en un sistema multiprocesador, porque aquí tampoco es predecible la velocidad relativa de ejecución de los procesos. Un sistema multiprocesador debe bregar también con problemas derivados de la ejecución simultánea de múltiples procesos. Sin embargo, fundamentalmente los problemas son los mismos que aquéllos de un sistema monoprocesador.

Problemas de Concurrencia

En los sistemas de tiempo compartido con varios usuarios, procesos, tareas, trabajos que comparten el uso de CPU entre todos, lo que ocasiona que se presentan varios problemas debido a que los procesos compiten por los recursos del sistema. Imagine que un proceso está escribiendo en la unidad de cinta y se le termina su turno de ejecución e inmediatamente después el proceso elegido para ejecutarse comienza a escribir sobre la misma cinta. El resultado es una cinta cuyo contenido es un desastre de datos mezclados. Así como la cinta, existen una multitud de recursos cuyo acceso debe der controlado para evitar los problemas de la concurrencia.

Por lo tanto, el sistema operativo debe ofrecer mecanismos para sincronizar la ejecución de procesos: semáforos, envío de mensajes, pipes, etc. Los semáforos son rutinas de software (que en su nivel más interno se auxilian del hardware) para lograr exclusión mutua en el uso de recursos.

Para entender este y otros mecanismos es importante entender los problemas generales de concurrencia, los cuales se describen enseguida.

Condiciones de Carrera o Competencia

La condición de carrera (race condition) ocurre cuando dos o más procesos acceden un recurso compartido sin control, de manera que el resultado combinado de este acceso depende del orden de llegada. Suponga, por ejemplo, que dos clientes de un banco realizan cada uno una operación en cajeros diferentes al mismo tiempo. El usuario A quiere hacer un depósito. El B un retiro. El usuario A comienza la transacción y lee su saldo que es 1000. En ese momento pierde su turno de ejecución (y su saldo queda como 1000) y el usuario B inicia el retiro: lee el saldo que es 1000, retira 200 y almacena el nuevo saldo que es 800 y termina. El turno de ejecución regresa al usuario A el cual hace su depósito de 100, quedando saldo = saldo + 100 = 1000 + 100 = 1100. Como se ve, el retiro se perdió y eso le encanta al usuario A y B, pero al banquero no le convino esta transacción. El error pudo ser al revés, quedando el saldo final en 800 (Google, 2021).

Postergación o Aplazamiento Indefinido

Consiste en el hecho de que uno o varios procesos nunca reciban el suficiente tiempo de ejecución para terminar su tarea. Por ejemplo, que un proceso ocupe un recurso y lo marque como ocupado y que termine sin marcarlo como desocupado. Si algún otro proceso pide ese recurso, lo verá ocupado y esperará indefinidamente a que se desocupe.

Condición de Espera Circular

Esto ocurre cuando dos o más procesos forman una cadena de espera que los involucra a todos. Por ejemplo, suponga que el proceso A tiene asignado el recurso cinta y el proceso B tiene asignado el recurso disco. En ese momento al proceso A se le ocurre pedir el recurso disco y al proceso B el recurso cinta. Ahí se forma una espera circular entre esos dos procesos

que se puede evitar quitándole a la fuerza un recurso a cualquiera de los dos procesos.

Un ejemplo sencillo

Considere el siguiente procedimiento:

```
void eco (){
cent = getchar();
csal = cent;
putchar(csal);
}
```

Este procedimiento muestra los elementos esenciales de un programa que proporcionará un procedimiento de eco de un carácter; la entrada se obtiene del teclado, una tecla cada vez. Cada carácter introducido se almacena en la variable cent. Luego se transfiere a la variable csal y se envía a la pantalla. Cualquier programa puede llamar repetidamente a este procedimiento para aceptar la entrada del usuario y mostrarla por su pantalla.

Considere ahora que se tiene un sistema multiprogramado de un único procesador y para un único usuario. El usuario puede saltar de una aplicación a otra, y cada aplicación utiliza el mismo teclado para la entrada y la misma pantalla para la salida. Dado que cada aplicación necesita usar el procedimiento eco, tiene sentido que éste sea un procedimiento compartido que esté cargado en una porción de memoria global para todas las aplicaciones. De este modo, sólo se utiliza una única copia del procedimiento eco, economizándose espacio.

La compartición de memoria principal entre procesos es útil para permitir una interacción eficiente y próxima entre los procesos. No obstante, esta interacción puede acarrear problemas. Considere la siguiente secuencia:

1 El proceso P1 invoca el procedimiento eco y es interrumpido inmediatamente después de que getchar devuelva su valor y sea almacenado en cent. En este punto, el carácter introducido más recientemente, *x*, está almacenado en cent.
2 El proceso P2 se activa e invoca al procedimiento eco, que ejecuta hasta concluir, habiendo leído y mostrado en pantalla un único carácter, *y*.
3 Se retoma el proceso P1. En este instante, el valor *x* ha sido sobrescrito en cent y por tanto se ha perdido. En su lugar, cent contiene *y*, que es transferido a csal y mostrado.

Así, el primer carácter se pierde y el segundo se muestra dos veces. La esencia de este problema es la variable compartida global, cent. Múltiples procesos tienen acceso a esta variable. Si un proceso actualiza la variable global y justo entonces es interrumpido, otro proceso puede alterar la variable antes de que el primer proceso pueda utilizar su valor. Suponga ahora, que decidimos que sólo un proceso al tiempo pueda estar en dicho procedimiento. Entonces la secuencia anterior resultaría como sigue:

4 El proceso P1 invoca el procedimiento eco y es interrumpido inmediatamente después de que concluya la función de entrada. En este punto, el carácter introducido más recientemente, *x*, está almacenado en cent.
5 El proceso P2 se activa e invoca al procedimiento eco. Sin embargo, dado que P1 está todavía dentro del procedimiento eco, aunque actualmente suspendido, a P2 se le impide entrar en el procedimiento. Por tanto, P2 se suspende esperando la disponibilidad del procedimiento eco.
6 En algún momento posterior, el proceso P1 se retoma y completa la ejecución de **eco**. Se muestra el carácter correcto, *x*.

7 Cuando P1 sale de eco, esto elimina el bloqueo de P2. Cuando P2 sea más tarde retomado, invocará satisfactoriamente el procedimiento eco.

La lección por aprender de este ejemplo es que es necesario proteger las variables globales compartidas (así como otros recursos globales compartidos) y que la única manera de hacerlo es controlar el código que accede a la variable. Si imponemos la disciplina de que sólo un proceso al tiempo pueda entrar en eco y que una vez en eco el procedimiento debe ejecutar hasta completarse antes de estar disponible para otro proceso, entonces el tipo de error que se acaba de describir no ocurrirá. Cómo se puede imponer esa disciplina es uno de los temas capitales de esta unidad.

Este problema fue enunciado con la suposición de que se trataba de un sistema operativo multiprogramado para un monoprocesador. El ejemplo demuestra que los problemas de la concurrencia suceden incluso cuando hay un único procesador. En un sistema multiprocesador, aparecen los mismos problemas de recursos compartidos protegidos, y funcionan las mismas soluciones. Primero, supóngase que no hay mecanismo para controlar los accesos a la variable global compartida:

1 Los procesos P1 y P2 están ambos ejecutando, cada cual en un procesador distinto. Ambos procesos invocan el procedimiento eco.
2 Ocurren los siguientes eventos; los eventos en la misma línea suceden en paralelo.

Tabla 4 Procesos invocan el procedimiento eco

Proceso P1		Proceso P2
• cent = getchar(); • csal = cent; putchar(csal);		• cent = getchar (); • csal = cent; • putchar(csal);

El resultado es que el carácter introducido a P1 se pierde antes de ser mostrado, y el carácter introducido a P2 es mostrado por ambos P1 y P2. De nuevo, añádase la capacidad de cumplir la disciplina de que sólo un proceso al tiempo pueda estar en eco. Entonces, sucede la siguiente secuencia:

1 Los procesos P1 y P2 están ambos ejecutando, cada cual en un procesador distinto. Ambos procesos invocan al procedimiento eco.
2 Mientras P1 está dentro del procedimiento eco, P2 invoca a eco. Dado que P1 está todavía dentro del procedimiento eco (ya esté P1 suspendido o ejecutando), a P2 se le bloqueará la entrada al procedimiento. Por tanto, P2 se suspende en espera de la disponibilidad del procedimiento eco.
3 En algún momento posterior, el proceso P1 completa la ejecución de eco, sale del procedimiento y continúa ejecutando. Inmediatamente después de que P1 salga de eco, se retoma P2 que comienza la ejecución de eco.

En el caso de un sistema monoprocesador, el motivo por el que se tiene un problema es que una interrupción puede parar la ejecución de instrucciones en cualquier punto de un proceso. En el caso de un sistema multiprocesador, se tiene el mismo motivo y, además, puede suceder porque dos procesos pueden estar ejecutando simultáneamente y ambos intentando acceder a la misma variable global. Sin embargo, la solución a ambos tipos de problema es la misma: controlar los accesos a los recursos compartidos.

Condición de carrera

Una condición de carrera sucede cuando múltiples procesos o hilos leen y escriben datos de manera que el resultado final depende del orden de ejecución de las instrucciones en los múltiples procesos. Consideremos dos casos sencillos.

Como primer ejemplo, suponga que dos procesos, P1 y P2, comparten la variable global a. En algún punto de su ejecución, P1 actualiza a al valor 1 y, en el mismo punto en su ejecución, actualiza a al valor 2. Así, las dos tareas compiten en una carrera por escribir la variable a. En este ejemplo el «perdedor» de la carrera (el proceso que actualiza el último) determina el valor de a.

Para nuestro segundo ejemplo, considere dos procesos, P3 y P4, que comparten las variables globales b y c, con valores iniciales b = 1 y c = 2. En algún punto de su ejecución, P3 ejecuta la asignación b = b + c y, en algún punto de su ejecución, P4 ejecuta la asignación c = b + c. Note que los dos procesos actualizan diferentes variables. Sin embargo, los valores finales de las dos variables dependen del orden en que los dos procesos ejecuten estas dos asignaciones. Si P3 ejecuta su sentencia de asignación primero, entonces los valores finales serán b = 3 y c = 5. Si P4 ejecuta su sentencia de asignación primero, entonces los valores finales serán b = 4 y c = 3.

Preocupaciones del sistema operativo

¿Qué aspectos de diseño y gestión surgen por la existencia de la concurrencia? Pueden enumerarse las siguientes necesidades:

1 El sistema operativo debe ser capaz de seguir la pista de varios procesos. Esto se consigue con el uso de bloques de control de proceso y fue descrito en el compendio anterior.

2 El sistema operativo debe ubicar y desubicar varios recursos para cada proceso activo. Estos recursos incluyen:
   - **Tiempo de procesador**. Esta es la misión de la planificación.
   - **Memoria**. La mayoría de los sistemas operativos usan un esquema de memoria virtual.
   - **Ficheros**.
   - **Dispositivos de E/S**.
3 El sistema operativo debe proteger los datos y recursos físicos de cada proceso frente a interferencias involuntarias de otros procesos. Esto involucra técnicas que relacionan memoria, ficheros y dispositivos de E/S.
4 El funcionamiento de un proceso y el resultado que produzca debe ser independiente de la velocidad a la que suceda su ejecución en relación con la velocidad de otros procesos concurrentes.

Para entender cómo puede abordarse la cuestión de la independencia de la velocidad, necesitamos ver las formas en que los procesos pueden interaccionar

Interacción de procesos

Podemos clasificar las formas en que los procesos interaccionan en base al grado en que perciben la existencia de cada uno de los otros. La Tabla 4.1 enumera tres posibles grados de percepción más las consecuencias de cada uno:

Procesos que no se perciben entre sí.

Son procesos independientes que no se pretende que trabajen juntos. El mejor ejemplo de esta situación es la multiprogramación de múltiples procesos independientes. Estos bien pueden ser trabajos por lotes o bien sesiones interactivas o una mezcla. Aunque los procesos no

estén trabajando juntos, el sistema operativo necesita preocuparse de la **competencia** por recursos. Por ejemplo, dos aplicaciones independientes pueden querer ambas acceder al mismo disco, fichero o impresora. El sistema operativo debe regular estos accesos.

Procesos que se perciben indirectamente entre sí

. Son procesos que no están necesariamente al tanto de la presencia de los demás mediante sus respectivos ID de proceso, pero que comparten accesos a algún objeto, como un *buffer* de E/S. Tales procesos exhiben **cooperación** en la compartición del objeto común.

Procesos que se perciben directamente entre sí.

Son procesos capaces de comunicarse entre sí vía el ID del proceso y que son diseñados para trabajar conjuntamente en cierta actividad. De nuevo, tales procesos exhiben **cooperación**.

Las condiciones no serán siempre tan claras como se sugiere en la Tabla 4.1. Mejor dicho, algunos procesos pueden exhibir aspectos tanto de competición como de cooperación. No obstante, es constructivo examinar cada uno de los tres casos de la lista precedente y determinar sus implicaciones para el sistema operativo.

Tabla 5 Interacción de procesos

Grado de percepción	Relación	Influencia que un proceso tiene sobre el otro	Potenciales problemas de control
Procesos que no se	Competencia	• Los resultados de un proceso son independientes de la acción de los otros.	• Exclusión mutua

Grado de percepción	Relación	Influencia que un proceso tiene sobre el otro	Potenciales problemas de control
perciben entre sí		La temporización del proceso puede verse afectada	• Interbloqueo (recurso renovable) • Inanición
Procesos que se perciben indirectamente entre sí (por ejemplo, objeto compartido)	Cooperación por compartición	• Los resultados de un proceso pueden depender de la información obtenida de otros	• Exclusión mutua • Interbloqueo (recurso renovable) • Inanición • Coherencia de datos
Procesos que se perciben directamente entre sí (tienen primitivas de comunicación a su	Cooperación por comunicación	• Los resultados de un proceso pueden depender de la información obtenida de otros • La temporización del proceso puede verse afectada	• Interbloqueo (recurso consumible) • Inanición

Grado de percepción	Relación	Influencia que un proceso tiene sobre el otro	Potenciales problemas de control
disposició n)			

Competencia entre procesos por recursos

Los procesos concurrentes entran en conflicto entre ellos cuando compiten por el uso del mismo recurso. En su forma pura, puede describirse la situación como sigue. Dos o más procesos necesitan acceso a un recurso durante el curso de su ejecución. Ningún proceso se percata de la existencia de los otros procesos y ninguno debe verse afectado por la ejecución de los otros procesos. Esto conlleva que cada proceso debe dejar inalterado el estado de cada recurso que utilice. Ejemplos de recursos son los dispositivos de E/S, la memoria, el tiempo de procesador y el reloj.

No hay intercambio de información entre los procesos en competencia. No obstante, la ejecución de un proceso puede afectar al comportamiento de los procesos en competencia. En concreto, si dos procesos desean ambos acceder al mismo recurso único, entonces, el sistema operativo reservará el recurso para uno de ellos, y el otro tendrá que esperar. Por tanto, el proceso al que se le deniega el acceso será ralentizado. En un caso extremo, el proceso bloqueado puede no conseguir nunca el recurso y por tanto no terminar nunca satisfactoriamente.

En el caso de procesos en competencia, deben afrontarse tres problemas de control. Primero está la necesidad de **exclusión mutua**. Supóngase que dos o más procesos requieren acceso a un recurso único no compartible, como una impresora. Durante el curso de la ejecución, cada proceso estará enviando mandatos al dispositivo de E/S, recibiendo información de estado, enviando datos o recibiendo datos. Nos referiremos a tal recurso como un **recurso crítico**, y a la porción del programa que lo utiliza como la **sección crítica** del programa.

Exclusión mutua
Requisito de que cuando un proceso esté en una sección crítica que accede a recursos compartidos, ningún otro proceso pueda estar en una sección crítica que acceda a ninguno de esos recursos compartidos.

Es importante que sólo se permita un programa al tiempo en su sección crítica. No podemos simplemente delegar en el sistema operativo para que entienda y aplique esta restricción porque los detalles de los requisitos pueden no ser obvios. En el caso de una impresora, por ejemplo, queremos que cualquier proceso individual tenga el control de la impresora mientras imprime el fichero completo. De otro modo, las líneas de los procesos en competencia se intercalarían.

La aplicación de la exclusión mutua crea dos problemas de control adicionales. Uno es el del **interbloqueo**. Por ejemplo, considere dos procesos, P1 y P2, y dos recursos, R1 y R2. Suponga que cada proceso necesita acceder a ambos recursos para realizar parte de su función. Entonces es posible encontrarse la siguiente situación: el sistema operativo asigna R1 a P2, y R2 a P1. Cada proceso está esperando por uno de los dos recursos. Ninguno liberará el recurso que ya posee hasta haber conseguido

el otro recurso y realizado la función que requiere ambos recursos. Los dos procesos están interbloqueados.

Un último problema de control es la **inanición**. Suponga que tres procesos (P1, P2, P3) requieren todos accesos periódicos al recurso R. Considere la situación en la cual P1 está en posesión del recurso y P2 y P3 están ambos retenidos, esperando por ese recurso. Cuando P1 termine su sección crítica, debería permitírsele acceso a R a P2 o P3. Asúmase que el sistema operativo le concede acceso a P3 y que P1 solicita acceso otra vez antes de completar su sección crítica. Si el sistema operativo le concede acceso a P1 después de que P3 haya terminado, y posteriormente concede alternativamente acceso a P1 y a P3, entonces a P2 puede denegársele indefinidamente el acceso al recurso, aunque no suceda un interbloqueo.

El control de la competencia involucra inevitablemente al sistema operativo ya que es quien ubica los recursos. Además, los procesos necesitarán ser capaces por sí mismos de expresar de alguna manera el requisito de exclusión mutua, como bloqueando el recurso antes de usarlo. Cualquier solución involucrará algún apoyo del sistema operativo, como proporcionar un servicio de bloqueo. La Figura 4.2 ilustra el mecanismo de exclusión mutua en términos abstractos. Hay *n* procesos para ser ejecutados concurrentemente. Cada proceso incluye (1) una sección crítica que opera sobre algún recurso Ra, y (2) código adicional que precede y sucede a la sección crítica y que no involucra acceso a Ra. Dado que todos los procesos acceden al mismo recurso Ra, se desea que sólo un proceso esté en su sección crítica al mismo tiempo. Para aplicar exclusión mutua se proporcionan dos funciones: entrarcritica y salircritica. Cada función toma como un argumento el nombre del recurso sujeto de la competencia. A cualquier proceso que intente entrar en su sección crítica mientras otro proceso está en su sección crítica, por el mismo recurso, se le hace esperar.

Condición de Exclusión Mutua

Cuando un proceso usa un recurso del sistema realiza una serie de operaciones sobre el recurso y después lo deja de usar. A la sección de código que usa ese recurso se le llama región crítica. La condición de exclusión mutua establece que solamente se permite a un proceso estar dentro de la misma región crítica. Esto es, que en cualquier momento solamente un proceso puede usar un recurso a la vez. Para lograr la exclusión mutua se ideo también el concepto de región crítica. Para logar la exclusión mutua generalmente se usan algunas técnicas para lograr entrar a la región crítica: semáforos, monitores, el algoritmo de Dekker y Peterson, los candados (Google, 2021).

Algoritmo de Dekker

Dijkstra presentó un algoritmo de exclusión mutua para dos procesos que diseñó el matemático holandés Dekker. Según Dijkstra, la solución se desarrolla por etapas. Este método tiene la ventaja de ilustrar la mayoría de los errores habituales que se producen en la construcción de programas concurrentes. A medida que se construya el algoritmo, se emplearán algunas ilustraciones pintorescas tomadas de Ben-Ari para escenificar la acción.

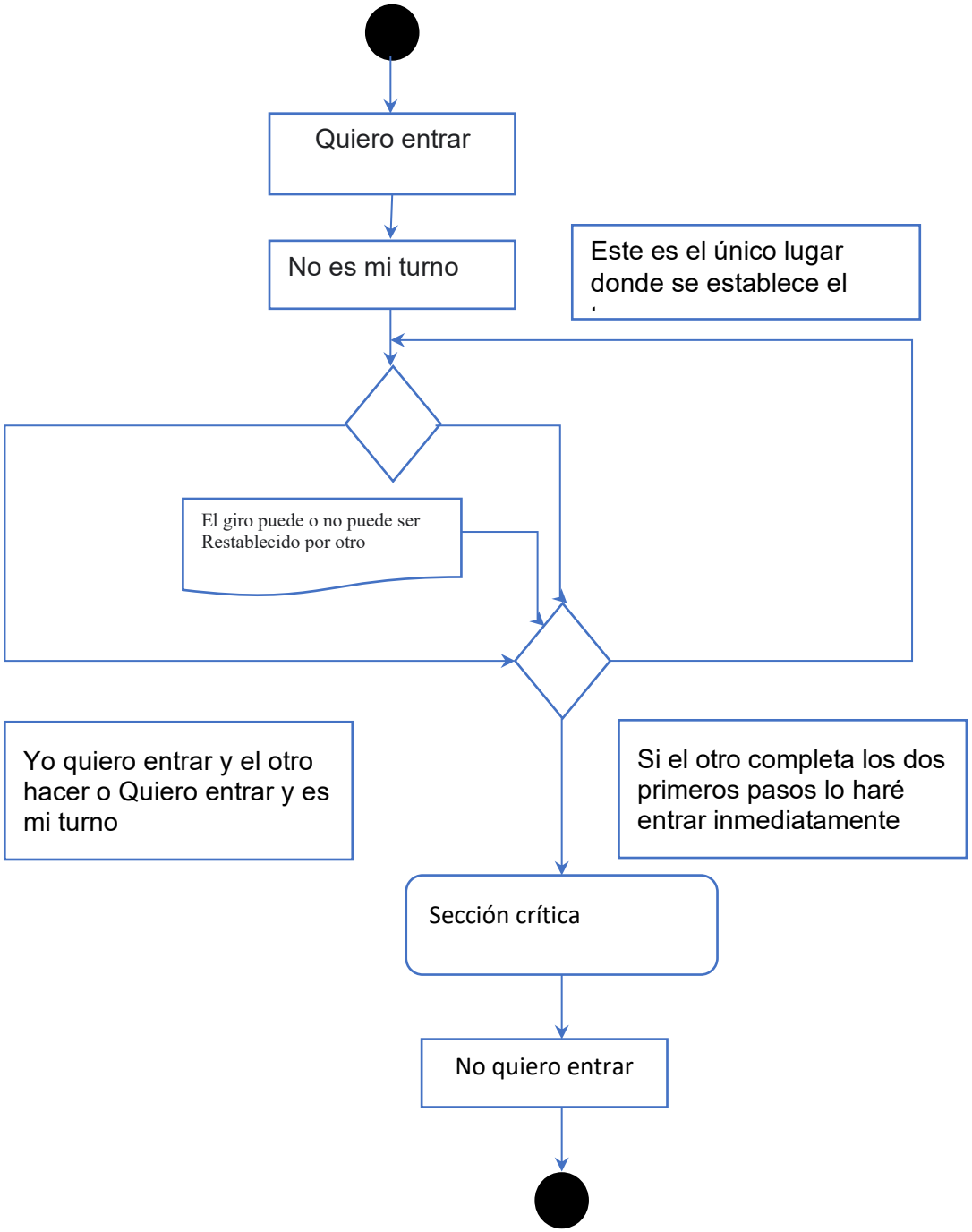


Figura.2: Diagrama de flujo del algoritmo de Dekker

El algoritmo de Dekker y de Peterson son algoritmos de exclusión mutua, la concurrencia parte del concepto del paralelismo, esto se da cuando dos procesos inician al mismo tiempo, pero con la concurrencia es que no

necesariamente deben iniciar al mismo tiempo si no que pueden iniciar en diferentes tiempos, pero no ocurren al mismo tiempo.

Condición de carreras, para prevenir ese problema se inventaron los algoritmos de exclusión mutua el mismo que evitan la condición de carrera ahora

Programa desarrollado en C++

Algoritmo de Dekker

```
#include <iostream>
#include <pthread.h>
#include <windows.h>
int turno=1;
bool peticion1=false;
bool peticion2=true;
void hilo1(){
        peticion1=true; //si quiero entrar
                if (peticion2==true){
                if (turno==1){
                        peticion1=false;
                        printf("Hilo 1 cede el turno %s\n");
                        if (turno==1){
                        printf("Turno del hilo 2 %s\n");
                        printf("Esperando %s\n");
                        printf("Finalizado turno del hilo 2 %\n");
                        //turno=0;
                        }
                peticion1=true;
        }
```

```
}
printf("Hilo 1 entra en la seccion critica %s\n");
turno=1;
peticion1=false;
}
int main(){
        hilo1();
        return 0;
}
```

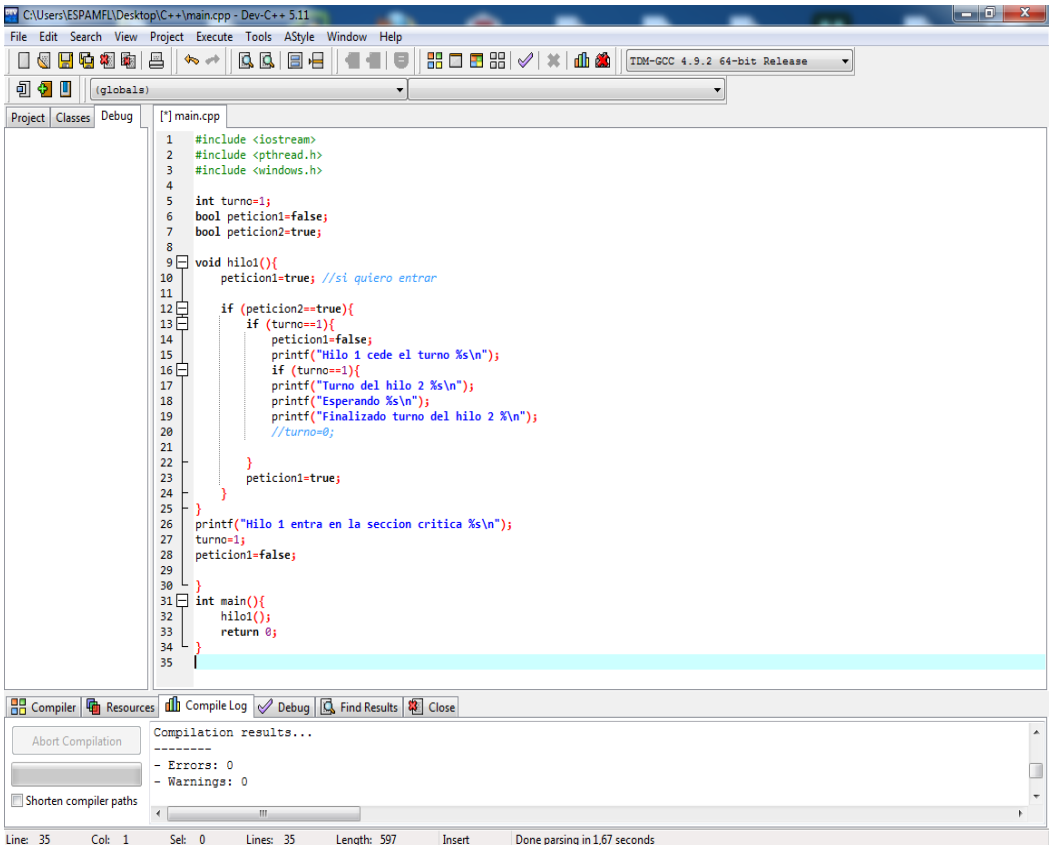


Ilustración 10 Aplicación C++ Algoritmo de Dekker

Algoritmo de Peterson

El algoritmo de Dekker resuelve el problema de la exclusión mutua pero con un programa complejo, difícil de seguir y cuya corrección es difícil de demostrar. Peterson desarrollado una solución simple y elegante. Como

antes, la variable global señal indica La posición de cada proceso con respecto a la exclusión mutua y la variable global turno resuelve los conflictos de simultaneidad (Google, 2021).

```
#include <stdio.h>
#include <pthread.h>
//#include"mythreads.h"

int flag [2];
int turn;
const int MAX = 1e9;
int ans = 0;

void lock_init()
{
   // Initialize lock by reseting the desire of
   // both the threads to acquire the locks.
   // And, giving turn to one of them.
   flag[0] = flag[1] = 0;
   turn = 0;
}

// Executed before entering critical section
void lock(int self)
{
   // Set flag[self] = 1 saying you want to acquire lock
   flag[self] = 1;

   // But, first give the other thread the chance to
```

```
    // acquire lock
    turn = 1-self;

    // Wait until the other thread looses the desire
    // to acquire lock or it is your turn to get the lock.
    while (flag[1-self] ==1 && turn==1-self);
}

// Executed after leaving critical section
void unlock (int self)
{
    // You do not desire to acquire lock in future.
    // This will allow the other thread to acquire
    // the lock.
    flag[self] = 0;
}

// A Sample function run by two threads created
// in main()
void* func(void *s)
{
    int i = 0;
    int self = *((int *)s);
    printf("Thread Entered: %d\n", self);

    lock(self);

    // Critical section (Only one thread
    // can enter here at a time)
```

```
    for (i=0; i<MAX; i++)
        ans++;

    unlock(self);
}

// Driver code
int main()
{
    // Initialized the lock then fork 2 threads
    pthread_t p1, p2;
    lock_init();

    // Create two threads (both run func)
    pthread_create(&p1, NULL, func, (void*)0);
    pthread_create(&p2, NULL, func, (void*)1);

    // Wait for the threads to end.
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    printf("Actual Count: %d | Expected Count: %d\n",
                                ans, MAX*2);

    return 0;
 }
```

Fundamento del Algoritmo de Peterson

Cuando dos o más procesos secuenciales en cooperación ejecutan todos ellos asíncronamente y comparten datos comunes se produce el

problema de la sección crítica. Un proceso productor genera información que es utilizada por un proceso consumidor. Para que los procesos productores y consumidores ejecuten concurrentemente, tenemos que creer un conjunto de buffers que puedan ser alimentados por el productor y vaciados por el consumidor.

El consumidor tiene que esperar si todos los buffers se encuentran vacíos, y el productor tiene que esperar si todos los buffers están llenos.

Una solución simple al problema de la sección crítica nos la proporciona Peterson. Esta solución es básicamente una combinación del Algoritmo que asocia cada proceso con su variable de cerradura y de una pequeña modificación del Algoritmo de Alternancia Estricta

Condición de Ocupar y Esperar un Recurso

Consiste en que un proceso pide un recurso y se le asigna. Antes de soltarlo, pide otro recurso que otro proceso ya tiene asignado. Los problemas descritos son todos importantes para el sistema operativo, ya que debe ser capaz de prevenir o corregirlos. Tal vez el problema más serio que se puede presentar en un ambiente de concurrencia es el abrazo mortal, también llamado trabazón y en inglés deadlock. El deadlock es una condición que ningún sistema o conjunto de procesos quisiera exhibir, ya que consiste en que se presentan al mismo tiempo cuatro condiciones necesarias: La condición de no apropiación, la condición de espera circular, la condición de exclusión mutua y la condición de ocupar y esperar un recurso. Ante esto, si el deadlock involucra a todos los procesos del sistema, el sistema ya no podrá hacer algo productivo. Si el deadlock involucra algunos procesos, éstos quedarán congelados para siempre (Google, 2021).

Asignar recursos en orden lineal

Esto significa que todos los recursos están etiquetados con un valor diferente y los procesos solo pueden hacer peticiones de recursos hacia adelante. Esto es, que si un proceso tiene el recurso con etiqueta 5 no puede pedir recursos cuya etiqueta sea menor que 5. Con esto se evita la condición de ocupar y esperar un recurso.

Asignar todo o nada

Este mecanismo consiste en que el proceso pida todos los recursos que va a necesitar de una vez y el sistema se los da solamente si puede dárselos todos, si no, no le da nada y lo bloquea.

Algoritmo del banquero

Este algoritmo usa una tabla de recursos para saber cuántos recursos tiene de todo tipo. También requiere que los procesos informen del máximo de recursos que va a usar de cada tipo. Cuando un proceso pide un recurso, el algoritmo verifica si asignándole ese recurso todavía le quedan otros del mismo tipo para que alguno de los procesos en el sistema todavía se le pueda dar hasta su máximo. Si la respuesta es afirmativa, el sistema se dice que está en estado seguro y se otorga el recurso. Si la respuesta es negativa, se dice que el sistema está en estado inseguro y se hace esperar a ese proceso (Google, 2021).

Tabla 6 Ilustración de la exclusión mutua

/* PROCESO 1 */
void P1{
while (true){
/* código anterior */;
	/* PROCESO 2 */
	void P2{
	while (true) {
		• • •	/* PROCESO n */
			void Pn{
			while (true){
			/* código anterior */;

entrarcritica (Ra); /* sección crítica */; salircritica (Ra); /* código posterior */; } }	/* código anterior */; entrarcritica (Ra); /* sección crítica */; salircritica (Ra); /* código posterior */; } }		entrarcritica (Ra); /* sección crítica */; salircritica (Ra); /* código posterior */; } }

Falta examinar mecanismos específicos para proporcionar las funciones entrarcritica y salircritica. Por el momento, postergamos este asunto mientras consideramos los otros casos de interacción de procesos.

Cooperación entre procesos vía compartición.

El caso de cooperación vía compartición cubre procesos que interaccionan con otros procesos sin tener conocimiento explícito de ellos. Por ejemplo, múltiples procesos pueden tener acceso a variables compartidas o a ficheros o bases de datos compartidas. Los procesos pueden usar y actualizar los datos compartidos sin referenciar otros procesos, pero saben que otros procesos pueden tener acceso a los mismos datos. Así, los procesos deben cooperar para asegurar que los datos que comparten son manipulados adecuadamente. Los mecanismos de control deben asegurar la integridad de los datos compartidos.

Dado que los datos están contenidos en recursos (dispositivos, memoria), los problemas de control de exclusión mutua, interbloqueo e inanición están presentes de nuevo. La única diferencia es que los datos individuales pueden ser accedidos de dos maneras diferentes, lectura y escritura, y sólo las operaciones de escritura deben ser mutuamente exclusivas.

Sin embargo, por encima de estos problemas, surge un nuevo requisito: el de la coherencia de datos. Como ejemplo sencillo, considérese una aplicación de contabilidad en la que pueden ser actualizados varios datos individuales. Supóngase que dos datos individuales *a* y *b* han de ser mantenidos en la relación *a* = *b*. Esto es, cualquier programa que actualice un valor, debe también actualizar el otro para mantener la relación. Considérense ahora los siguientes dos procesos:

P1:

$a = a + 1;$

$b = b + 1;$

P2:

$b = 2 * b;$

$a = 2 * a;$

Si el estado es inicialmente consistente, cada proceso tomado por separado dejará los datos compartidos en un estado consistente. Ahora considere la siguiente ejecución concurrente en la cual los dos procesos respetan la exclusión mutua sobre cada dato individual (*a* y *b*).

$a = a + 1;$

$b = 2 * b;$

$b = b + 1;$

$a = 2 * a;$

Al final de la ejecución de esta secuencia, la condición a = b ya no se mantiene. Por ejemplo, si se comienza con $a = b = 1$, al final de la ejecución de esta secuencia, tendremos $a = 4$ y $b = 3$. El problema puede ser evitado declarando en cada proceso la secuencia completa como una sección crítica.

Así se ve que el concepto de sección crítica es importante en el caso de cooperación por compartición. Las mismas funciones abstractas entrarcritica y salircritica tratadas anteriormente (Figura 4.2) pueden usarse aquí. En este caso, el argumento de las funciones podría ser una variable, un fichero o cualquier otro objeto compartido. Es más, si las secciones críticas se utilizan para conseguir integridad de datos, entonces puede no haber recurso o variable concreta que pueda ser identificada como argumento. En este caso, puede pensarse en el argumento como un identificador compartido entre los procesos concurrentes que identifica las secciones críticas que deben ser mutuamente exclusivas.

Cooperación entre procesos vía comunicación.

En los dos primeros casos que se han tratado, cada proceso tiene su propio entorno aislado que no incluye a los otros procesos. Las interacciones entre procesos son indirectas. En ambos casos hay cierta compartición. En el caso de la competencia, hay recursos compartidos sin ser conscientes de los otros procesos. En el segundo caso, hay compartición de valores y aunque cada proceso no es explícitamente consciente de los demás procesos, es consciente de la necesidad de mantener integridad de datos. Cuando los procesos cooperan vía comunicación, en cambio, los diversos procesos involucrados participan en un esfuerzo común que los vincula a todos ellos. La comunicación proporciona una manera de sincronizar o coordinar actividades varias.

Típicamente, la comunicación se fundamenta en mensajes de algún tipo. Las primitivas de envío y recepción de mensajes deben ser proporcionadas como parte del lenguaje de programación o por el núcleo del sistema operativo.

Dado que en el acto de pasar mensajes los procesos no comparten nada, la exclusión mutua no es un requisito de control en este tipo de cooperación. Sin embargo, los problemas de interbloqueo e inanición están presentes. Como ejemplo de interbloqueo, dos procesos pueden estar bloqueados, cada uno esperando por una comunicación del otro. Como ejemplo de inanición, considérense tres procesos, P1, P2 y P3, que muestran el siguiente comportamiento. P1 está intentando repetidamente comunicar con P2 o con P3, y P2 y P3 están ambos intentando comunicar con P1. Podría suceder una secuencia en la cual P1 y P2 intercambiasen información repetidamente, mientras P3 está bloqueado esperando comunicación de P1. No hay interbloqueo porque P1 permanece activo, pero P3 pasa hambre.

Requisitos para la exclusión mutua

Cualquier mecanismo o técnica que vaya a proporcionar exclusión mutua debería cumplimentar los siguientes requisitos:

1 La exclusión mutua debe hacerse cumplir: sólo se permite un proceso al tiempo dentro de su sección crítica, de entre todos los procesos que tienen secciones críticas para el mismo recurso u objeto compartido.
2 Un proceso que se pare en su sección no crítica debe hacerlo sin interferir con otros procesos.
3 No debe ser posible que un proceso que solicite acceso a una sección crítica sea postergado indefinidamente: ni interbloqueo ni inanición.

4 Cuando ningún proceso esté en una sección crítica, a cualquier proceso que solicite entrar en su sección crítica debe permitírsele entrar sin demora.
5 No se hacen suposiciones sobre las velocidades relativas de los procesos ni sobre el número de procesadores.
6 Un proceso permanece dentro de su sección crítica sólo por un tiempo finito.

Hay varias maneras de satisfacer los requisitos para la exclusión mutua. Una manera es delegar la responsabilidad en los procesos que desean ejecutar concurrentemente. Esos procesos, ya sean programas del sistema o programas de aplicación, estarían obligados a coordinarse entre sí para

Cumplir la exclusión mutua, sin apoyo del lenguaje de programación ni del sistema operativo. Podemos referirnos a esto como soluciones software. Aunque este enfoque es propenso a una alta sobrecarga de procesamiento y a errores, sin duda es útil examinar estas propuestas para obtener una mejor comprensión de la complejidad de la programación concurrente.

Tema 2: Sincronización

La sincronización se encuentra relacionada con la exclusión mutua: es la posibilidad de que múltiples procesos coordinen sus actividades para intercambiar información. Es una de las funciones típicas de un núcleo de sistema operativo, donde la sincronización de procesos y soporte para comunicación entre procesos es parte de la "*Gestión de Procesos*", tema tratado en el compendio anterior

Fundamentos de sincronización

Actualmente, una de las funciones comunes es la **Sincronización de hilos**, donde todos los hilos de un proceso comparten el mismo espacio de

direcciones y otros recursos, como por ejemplo, los archivos abiertos. Cualquier alteración de un recurso por cualquiera de los hilos, afecta al entorno del resto de los hilos del mismo proceso. Por tanto, es necesario sincronizar las actividades de los hilos para que no interfieran entre ellos o corrompan estructuras de datos. Por ejemplo, si dos hilos de modo simultáneo intentan añadir un elemento a una lista doblemente enlazada, se podría perder un elemento o la lista podría acabar malformada.

Los asuntos que surgen y las técnicas que se utilizan en la sincronización de los hilos son, en general, los mismos que en la **sincronización de procesos**.

Entonces, con múltiples procesos activos, que pueden acceder a espacios de direcciones compartidas o recursos compartidos de E/S, se debe tener cuidado en proporcionar una sincronización eficaz. La sincronización es un servicio que fuerza la exclusión mutua y el orden de los eventos. Un mecanismo común de sincronización que se utiliza en los sistemas operativos multiprocesador son los cerrojos.

Hardware de sincronización

Se han desarrollado cierto número de algoritmos software para conseguir exclusión mutua, de los cuales el más conocido es el algoritmo de Dekker. La solución software es fácil que tenga una alta sobrecarga de procesamiento y es significativo el riesgo de errores lógicos. No obstante, el estudio de estos algoritmos ilustra muchos de los conceptos básicos y de los potenciales problemas del desarrollo de programas concurrentes. Para el lector interesado, el Apéndice A incluye un análisis de las soluciones software. En esta sección se consideran varias interesantes soluciones hardware a la exclusión mutua.

Deshabilitar interrupciones

En una máquina monoprocesador, los procesos concurrentes no pueden solaparse, sólo pueden entrelazarse. Es más, un proceso continuará ejecutando hasta que invoque un servicio del sistema operativo o hasta que sea interrumpido. Por tanto, para garantizar la exclusión mutua, basta con impedir que un proceso sea interrumpido. Esta técnica puede proporcionarse en forma de primitivas definidas por el núcleo del sistema para deshabilitar y habilitar las interrupciones. Un proceso puede cumplir la exclusión mutua del siguiente modo (compárese con la Figura 4.2):

```
while (true) {
/* deshabilitar interrupciones */;
/* sección crítica */;
/* habilitar interrupciones */;
/* resto */;
```

Dado que la sección crítica no puede ser interrumpida, se garantiza la exclusión mutua. El precio de esta solución, no obstante, es alto. La eficiencia de ejecución podría degradarse notablemente porque se limita la capacidad del procesador de entrelazar programas. Un segundo problema es que esta solución no funcionará sobre una arquitectura multiprocesador. Cuando el sistema de cómputo incluye más de un procesador, es posible (y típico) que se estén ejecutando al tiempo más de un proceso. En este caso, deshabilitar interrupciones no garantiza exclusión mutua.

Instrucciones máquina especiales

En una configuración multiprocesador, varios procesadores comparten acceso a una memoria principal común. En este caso no hay una relación maestro/esclavo; en cambio los procesadores se comportan independientemente en una relación de igualdad. No hay mecanismo de

interrupción entre procesadores en el que pueda basarse la exclusión mutua.

A un nivel hardware, como se mencionó, el acceso a una posición de memoria excluye cualquier otro acceso a la misma posición. Con este fundamento, los diseñadores de procesadores han propuesto varias instrucciones máquina que llevan a cabo dos acciones atómicamente2, como leer y escribir o leer y comprobar, sobre una única posición de memoria con un único ciclo de búsqueda de instrucción. Durante la ejecución de la instrucción, el acceso a la posición de memoria se le bloquea a toda otra instrucción que referencie esa posición. Típicamente, estas acciones se realizan en un único ciclo de instrucción.

En esta sección, se consideran dos de las instrucciones implementadas más comúnmente.

- **Instrucción *Test and Set***. La instrucción *test and set* (comprueba y establece) puede definirse como sigue:

```
boolean testset (int i) {
if (i == 0) {
i = 1;
return true;
}
else {
return false;
```

La instrucción comprueba el valor de su argumento i. Si el valor es 0, entonces la instrucción reemplaza el valor por 1 y devuelve cierto. En caso contrario, el valor no se cambia y devuelve falso. La función testset completa se realiza atómicamente; esto es, no está sujeta a interrupción.

La Figura 4.3a muestra un protocolo de exclusión mutua basado en el uso de esta instrucción. La construcción paralelos (P1, P2, …, Pn) significa lo siguiente: suspender la ejecución del programa principal; iniciar la ejecución concurrente de los procedimientos P1, P2, …, Pn; cuando todos los P1, P2, …, Pn hayan terminado, retomar al programa principal. Una variable compartida cerrojo se inicializa a 0. El único proceso que puede entrar en su sección crítica es aquél que encuentra la variable cerrojo igual a 0. Todos los otros procesos que intenten entrar en su sección crítica caen en un modo de espera activa.

El término **espera activa** (*busy waiting*), o **espera cíclica** (*spin waiting*) se refiere a una técnica en la cual un proceso no puede hacer nada hasta obtener permiso para entrar en su sección crítica, pero continúa ejecutando una instrucción o conjunto de instrucciones que comprueban la variable apropiada para conseguir entrar. Cuando un proceso abandona su sección crítica, restablece cerrojo a 0; en este punto, a uno y sólo a uno de los procesos en espera se le concederá acceso a su sección crítica. La elección del proceso depende de cuál de los procesos es el siguiente que ejecuta la instrucción testset.

Tabla 7 Soporte hardware para la exclusión mutua

/* **programa** exclusión mutua */
const int n = /* número de procesos */;
int cerrojo;
void P(**int** i){
  while (true){
  while (!testset (cerrojo))
  /* no hacer nada */;
  /* sección crítica */;
	/* **programa** exclusión mutua */
	const int n = /* número de procesos */;
	int cerrojo;
	void P(**int** i){
	  int llavei = 1;
	  while (true){
	  do exchange (llavei, cerrojo)

```
    cerrojo = 0;
    /* resto */
  }
}

void main(){
  cerrojo = 0;
  paralelos (P(1), P(2), . . . ,P(n));
}
```

(a) Instrucción ***test and set***

```
    while (llavei != 0);
    /* sección crítica */;
    exchange (llavei, cerrojo);
    /* resto */
  }
}

void main(){
  cerrojo = 0;
  paralelos (P (1), P(2), . . . ,P(n));
}
```

(b) Instrucción ***exchange***

- **Instrucción Exchange.** La instrucción exchange (intercambio) puede definirse como sigue:

```
void exchange (int registro, int memoria) {
int temp;
temp = memoria;
memoria = registro;
registro = temp;
```

La instrucción intercambia los contenidos de un registro con los de una posición de memoria. Los procesadores actuales contienen una instrucción para esto, en el caso de los procesadores Intel la instrucción es *XCHG*.

La Figura 4.3b muestra un protocolo de exclusión mutua basado en el uso de una instrucción *exchange*. Una variable compartida cerrojo se inicializa a 0. Cada proceso utiliza una variable local llavei que se inicializa a 1. El único proceso que puede entrar en su sección crítica

es aquél que encuentra cerrojo igual a 0, y al cambiar cerrojo a 1 se excluye a todos los otros procesos de la sección crítica. Cuando el proceso abandona su sección crítica, se restaura cerrojo al valor 0, permitiéndose que otro proceso gane acceso a su sección crítica.

Nótese que la siguiente expresión siempre se cumple dado el modo en que las variables son inicializadas y dada la naturaleza del algoritmo exchange:

$$cerrojo + \sum_{i} \quad llave_i = n$$

Si cerrojo = 0, entonces ningún proceso está en su sección crítica. Si cerrojo = 1, entonces exactamente un proceso está en su sección crítica, aquél cuya variable llavei es igual a 0.

Propiedades de la solución instrucción máquina.

El uso de una instrucción máquina especial para conseguir exclusión mutua tiene ciertas ventajas:

- Es aplicable a cualquier número de procesos sobre un procesador único o multiprocesador de memoria principal compartida.
- Es simple y, por tanto, fácil de verificar.
- Puede ser utilizado para dar soporte a múltiples secciones críticas: cada sección crítica puede ser definida por su propia variable.

Hay algunas desventajas serias:

- **Se emplea espera activa**. Así, mientras un proceso está esperando para acceder a una sección crítica, continúa consumiendo tiempo de procesador.
- **Es posible la inanición**. Cuando un proceso abandona su sección crítica y hay más de un proceso esperando, la selección del proceso en

espera es arbitraria. Así, a algún proceso podría denegársele indefinidamente el acceso.

- **Es posible el interbloqueo.** Considérese el siguiente escenario en un sistema de procesador único. El proceso P1 ejecuta la instrucción especial (por ejemplo, *testset*, *exchange*) y entra en su sección crítica. Entonces P1 es interrumpido para darle el procesador a P2, que tiene más alta prioridad. Si P2 intenta ahora utilizar el mismo recurso que P1, se le denegará el acceso, dado el mecanismo de exclusión mutua. Así caerá en un bucle de espera activa. Sin embargo, P1 nunca será escogido para ejecutar por ser de menor prioridad que otro proceso listo, P2.

Dados los inconvenientes de ambas soluciones software y hardware que se acaban de esbozar, es necesario buscar otros mecanismos.

Semáforos

Pasamos ahora a mecanismos del sistema operativo y del lenguaje de programación que se utilizan para proporcionar concurrencia. Comenzando, en esta sección, con los semáforos.

El primer avance fundamental en el tratamiento de los problemas de programación concurrente ocurre en 1965 con el tratado de Dijkstra. Dijkstra estaba involucrado en el diseño de un sistema operativo como una colección de procesos secuenciales cooperantes y con el desarrollo de mecanismos eficientes y fiables para dar soporte a la cooperación. Estos mecanismos podrían ser usados fácilmente por los procesos de usuario si el procesador y el sistema operativo colaborasen en hacerlos disponibles.

El principio fundamental es éste: dos o más procesos pueden cooperar por medio de simples señales, tales que un proceso pueda ser obligado a parar en un lugar específico hasta que haya recibido una señal específica.

Cualquier requisito complejo de coordinación puede ser satisfecho con la estructura de señales apropiada. Para la señalización, se utilizan unas variables especiales llamadas semáforos. Para transmitir una señal vía el semáforo s, el proceso ejecutará la primitiva semSignal(s). Para recibir una señal vía el semáforo s, el proceso ejecutará la primitiva semWait(s); si la correspondiente señal no se ha transmitido todavía, el proceso se suspenderá hasta que la transmisión tenga lugar.

Para conseguir el efecto deseado, el semáforo puede ser visto como una variable que contiene un valor entero sobre el cual sólo están definidas tres operaciones:

1 Un semáforo puede ser inicializado a un valor no negativo.
2 La operación semWait decrementa el valor del semáforo. Si el valor pasa a ser negativo, entonces el proceso que está ejecutando semWait se bloquea. En otro caso, el proceso continúa su ejecución.
3 La operación semSignal incrementa el valor del semáforo. Si el valor es menor o igual que cero, entonces se desbloquea uno de los procesos bloqueados en la operación semWait.

Aparte de estas tres operaciones no hay manera de inspeccionar o manipular un semáforo.

La Figura 4.4 sugiere una definición más formal de las primitivas del semáforo. Las primitivas semWait y semSignal se asumen atómicas. Una versión más restringida, conocida como **semáforo binario** o **mutex**, se define en la Figura 4.5. Un semáforo binario sólo puede tomar los valores 0 y 1 y se puede definir por las siguientes tres operaciones:

1 Un semáforo binario puede ser inicializado a 0 o 1.
2 La operación semWaitB comprueba el valor del semáforo. Si el valor es cero, entonces el proceso que está ejecutando semWaitB se bloquea.

Si el valor es uno, entonces se cambia el valor a cero y el proceso continúa su ejecución.

3 La operación semSignalB comprueba si hay algún proceso bloqueado en el semáforo. Si lo hay, entonces se desbloquea uno de los procesos bloqueados en la operación semWaitB. Si no hay procesos bloqueados, entonces el valor del semáforo se pone a uno.

En principio debería ser más fácil implementar un semáforo binario, y puede demostrarse que tiene la misma potencia expresiva que un semáforo general. Para contrastar los dos tipos de semáforos, el semáforo no-binario es a menudo referido como **semáforo con contador** o **semáforo general**.

Para ambos, semáforos con contador y semáforos binarios, se utiliza una cola para mantener los procesos esperando por el semáforo. Surge la cuestión sobre el orden en que los procesos deben ser extraídos de tal cola. La política más favorable es FIFO (primero-en-entrar-primero-en-salir): el proceso que lleve más tiempo bloqueado es el primero en ser extraído de la cola; un semáforo cuya definición incluye esta política se denomina **semáforo fuerte**. Un semáforo que no especifica el orden en bque los procesos son extraídos de la cola es un **semáforo débil**.

La Figura 4.6, es un ejemplo de la operación de un semáforo fuerte. Aquí los procesos A, B y C dependen de un resultado del proceso D. Inicialmente (1), A está ejecutando; B, C y D están listos; y el contador del semáforo es 1, indicando que uno de los resultados de D está disponible. Cuando A realiza una instrucción semWait sobre el semáforo s, el semáforo se decrementa a 0 y A puede continuar ejecutando; posteriormente se adjunta a la lista de listos. Entonces B ejecuta (2), finalmente realiza una instrucción semWait y es bloqueado, permitiendo que D ejecute (3). Cuando D completa un nuevo resultado, realiza una instrucción

semSignal, que permite a B moverse a la lista de listos (4). D se adjunta a la lista de listos y C comienza a ejecutar (5) pero se bloquea cuando realiza una instrucción semWait. De manera similar, A y B ejecutan y se bloquean en el semáforo, permitiendo a D retomar la ejecución (6). Cuando D tiene un resultado realiza un semSignal, que transfiere a C a la lista de listos. Posteriores ciclos de D liberarán A y B del estado Bloqueado.

Para el algoritmo de exclusión mutua tratado ilustrado en la Figura 4.7, los semáforos fuertes garantizan estar libres de inanición mientras que los semáforos débiles no.

Se asumirán semáforos fuertes dado que son más convenientes y porque ésta es la forma típica del semáforo proporcionado por los sistemas operativos.

```
struct semaphore {
  int cuenta;
  queueType cola;
};
void semWait(semaphore s){
  s.cuenta—;
  if (s.cuenta < 0){
    poner este proceso en s.cola;
    bloquear este proceso;
  }
}
void semSignal(semaphore s){
  s.cuenta++;
  if (s.cuenta <= 0) {
    extraer un proceso P de s.cola;
```

```
struct binary_semaphore {
  enum {cero, uno} valor;
  queueType cola;
};
void semWaitB(binary_semaphore
s){
  if (s.valor == 1)
    s.valor = 0;
  else{
    poner este proceso en s.cola;
    bloquear este proceso;
  }
}
void
semSignalB(binary_semaphore s){
```

```
    poner el proceso P en la lista de
listos;
  }
}
```

Figura 4.4: Una definición de las primitivas del semáforo

```
  if (esta_vacia(s.cola))
    s.valor = 1;
  else{
    extraer un proceso P de s.cola;
    poner el proceso P en la lista de
listos;
  }
}
```

Figura 4.5: Una definición de las primitivas del semáforo binario

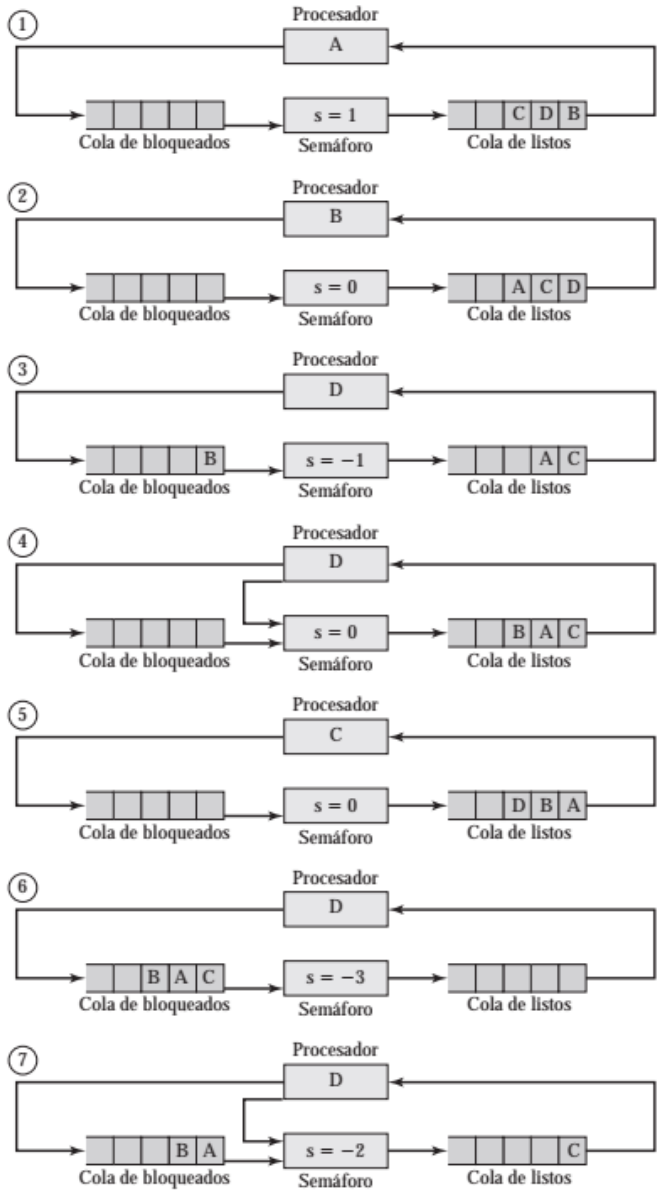


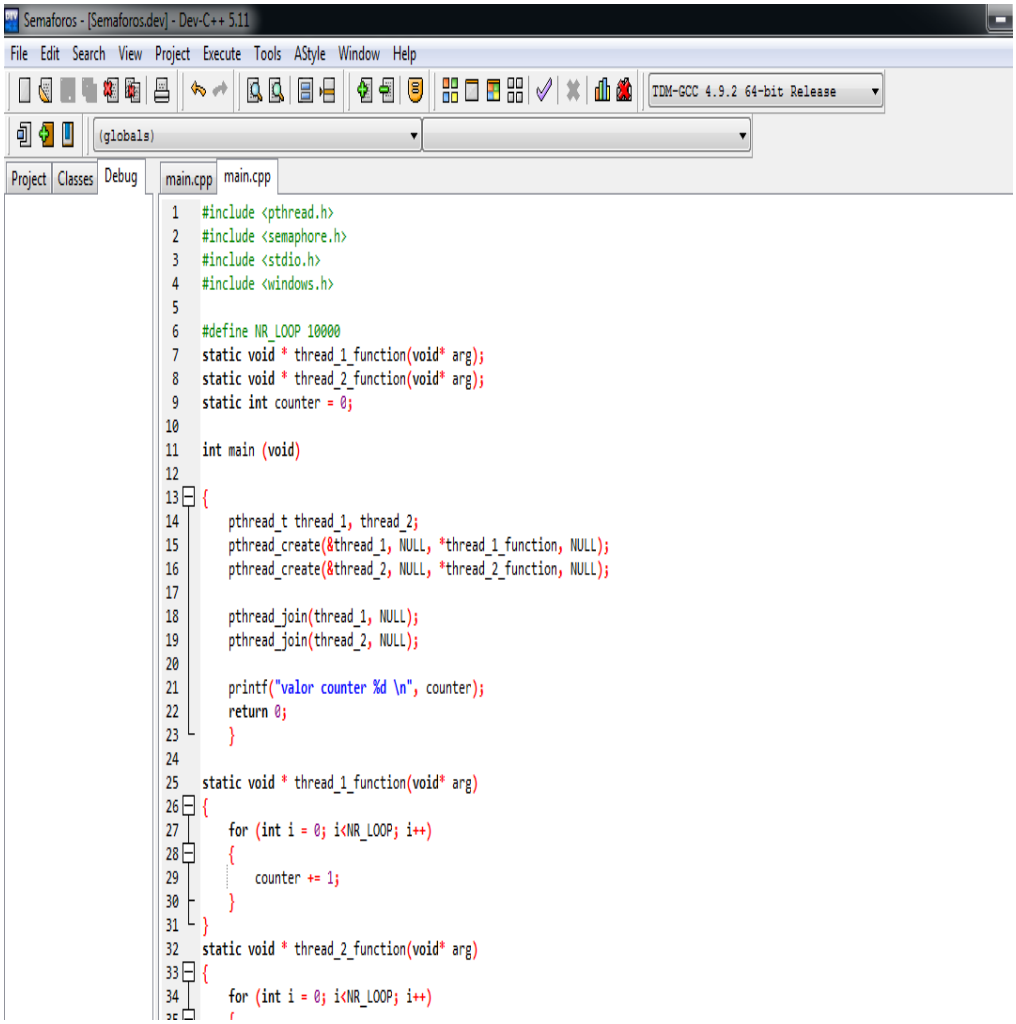
Figura 4.6: Ejemplo de mecanismo semáforo

```
/* programa exclusión mutua */

const int n = /* número de procesos */;

semafore s = 1;

void P(int i){

   while (true){
```

```
    semWait(s);
    /* sección crítica */;
    semSignal(s);
    /* resto */
  }
}
void main(){
  paralelos (P(1), P(2), . . . ,P(n));
}
```

Figura 4.7: Exclusión mutua usando semáforos

Programación en C de semáforos



Ejemplo en el que dos hilos (threads) acceden a un "recurso compartido" y se producen errores, por lo que es necesario implementar semáforos para proteger la "sección crítica":

Codito fuente de programación en C++ de semáforos.

#include <pthread.h> //libreria necesaria para trabajar con hilos

#include <semaphore.h>

#include <stdio.h>

#include <windows.h>

```
#define NR_LOOP 10000 //definicion de la constante
static void * thread_1_function(void* arg); //definicion de los hilos
static void * thread_2_function(void* arg); //definicion de los hilos
static int counter = 0; //definicion de la variable conuter

int main (void)

        pthread_t thread_1, thread_2;
        pthread_create(&thread_1, NULL, *thread_1_function, NULL);
//creacion de las funciones de hilos 1
        pthread_create(&thread_2, NULL, *thread_2_function, NULL);
//creaciojn de la funcion hilo 2

        pthread_join(thread_1, NULL); //funcion que permite termina el
hilos
        pthread_join(thread_2, NULL);

        printf("valor counter %d \n", counter);
        return 0;
        }

static void * thread_1_function(void* arg)
{ for (int i = 0; i<NR_LOOP; i++) //incrementa en 10.000 el valor
{ counter += 1;
static void * thread_2_function(void* arg) //decrementa en 10.000 el valor
{ for (int i = 0; i<NR_LOOP; i++)
{ counter -= 1; }
```

Codito fuente de programación en C++ de semáforos 2.

```
#include <pthread.h> //libreria necesaria para trabajar con hilos
#include <semaphore.h>
#include <stdio.h>
#include <windows.h>

#define NR_LOOP 10000 //definicion de la constante
static void * thread_1_function(void* arg); //definicion de los hilos
static void * thread_2_function(void* arg); //definicion de los hilos
static int counter = 0; //definicion de la variable conuter

sem_t sem1; //Declaracion del semaforo

int main (void)

{ pthread_t thread_1, thread_2;

sem_init(&sem1, 0, 1); //Inicializacion de la funcion del semaforo en un hilo

pthread_create(&thread_1, NULL, *thread_1_function, NULL); //creacion de las funciones de hilos 1
        pthread_create(&thread_2, NULL, *thread_2_function, NULL); //creaciojn de la funcion hilo 2

        pthread_join(thread_1, NULL); //funcion que permite termina el hilo
        pthread_join(thread_2, NULL);
```

```
        printf("valor counter %d \n", counter);
        return 0;
        }

static void * thread_1_function (void* arg)
{ for (int i = 0; i<NR_LOOP; i++) //incrementa en 10.000 el valor
{ sem_wait(&sem1); // Funcion que decrementa el valor
counter += 1;
sem_post(&sem1); //incrementa una unidad del semaforo}
static void * thread_2_function(void* arg) //decrementa en 10.000 el valor
{ for (int i = 0; i<NR_LOOP; i++)
{ sem_wait(&sem1); // Funcion que incrementa el valor
counter -= 1;
sem_post(&sem1); //incrementa una unidad del semaforo }
```

Monitores

Los semáforos proporcionan una herramienta potente y flexible para conseguir la exclusión mutua y para la coordinación de procesos. Sin embargo, como la Figura 4.8 sugiere, puede ser difícil producir un programa correcto utilizando semáforos. La dificultad es que las operaciones semWait y semSignal pueden estar dispersas a través de un programa y no resulta fácil ver el efecto global de estas operaciones sobre los semáforos a los que afectan.

El monitor es una construcción del lenguaje de programación que proporciona una funcionalidad equivalente a la de los semáforos, pero es más fácil de controlar. El concepto se definió formalmentepor primera vez en “Monitors: An Operating System Structuring Concept” de Hoare en 1974. La construcción monitor ha sido implementada en cierto número de

lenguajes de programación, incluyendo Pascal Concurrente, Pascal-Plus, Modula-2, Modula-3 y Java. También ha sido implementada como una biblioteca de programa. Esto permite a los programadores poner cerrojos monitor sobre cualquier objeto. En concreto, para algo como una lista encadenada, puede quererse tener un único cerrojo para todas las listas, por cada lista o por cada elemento de cada lista.

```
/* programa productor consumidor */
int n;
binary_semaphore s = 1;
binary_semaphore retardo = 0;
void productor(){
  while (true){
    producir();
    semWaitB(s);
    anyadir();
    n++;
    if (n==1)
      semSignalB(retardo);
    semSignalB(s);
  }
}
void consumidor(){
  semWaitB(retardo);
  while (true){
    semWaitB(s);
    extraer();
    n--;
    semSignalB(s);
```

```
      consumir();
      if (n==0)
         semWaitB(retardo);
   }
}
void main(){
   n = 0;
   paralelos (productor, consumidor);
}
```

Figura 4.8: Una solución incorrecta al problema productor/consumidor con buffer infinito

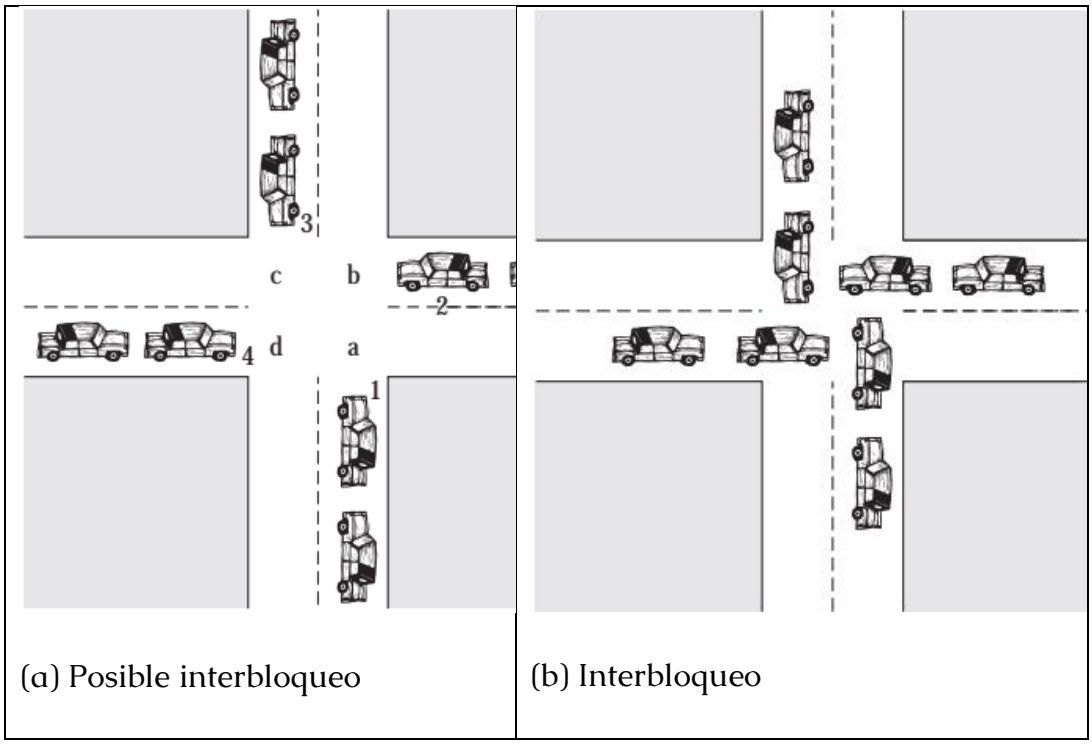
Fundamentos de Interbloqueo

Se puede definir el interbloqueo como el bloqueo bien compite por recursos del sistema o se comunican entre sí. Un conjunto de procesos está interbloqueado cuando cada proceso del conjunto está bloqueado esperando un evento (normal- permanente de un conjunto de procesos que o mente la liberación de algún recurso requerido) que sólo puede generar otro proceso bloqueado del conjunto. El interbloqueo es permanente porque no puede producirse ninguno de los eventos. A diferencia de otros problemas que aparecen en la gestión de procesos concurrentes, no hay una solución eficiente para el caso general.

Todos los interbloqueos involucran necesidades conflictivas que afectan a los recursos de dos o más procesos. Un ejemplo habitual es el interbloqueo del tráfico. La Figura 4.9a muestra una situación en la que cuatro coches han llegado aproximadamente al mismo tiempo a una intersección donde confluyen cuatro caminos. Los cuatro cuadrantes de la intersección son los recursos que hay que controlar. En particular, si los cuatro coches desean cruzar la intersección, los requisitos de recursos son los siguientes:

- El coche 1, que viaja hacia el norte, necesita los cuadrantes a y b.
- El coche 2 necesita los cuadrantes b y c.
- El coche 3 necesita los cuadrantes c y d.
- El coche 4 necesita los cuadrantes d y a.

La norma de circulación habitual es que un coche en un cruce de cuatro caminos debería dar preferencia a otro coche que está justo a su derecha. Esta regla funciona si hay sólo dos o tres coches en la intersección. Por ejemplo, si sólo llegan a la intersección los coches que vienen del norte y del oeste, el del norte esperará y el del oeste proseguirá. Sin embargo, si todos los coches llegan aproximadamente al mismo tiempo, cada uno se abstendrá de cruzar la intersección, produciéndose un interbloqueo. Si los cuatro coches olvidan las normas y entran (cuidadosamente) en la intersección, cada uno posee un recurso (un cuadrante) pero no pueden continuar porque el segundo recurso requerido ya se lo ha apoderado otro coche. De nuevo, se ha producido un interbloqueo. Nótese también que debido a que cada coche tiene justo detrás otro coche, no es posible dar marcha atrás para eliminar el interbloqueo.



(a) Posible interbloqueo (b) Interbloqueo

Figura 4.9: Ilustración del interbloqueo

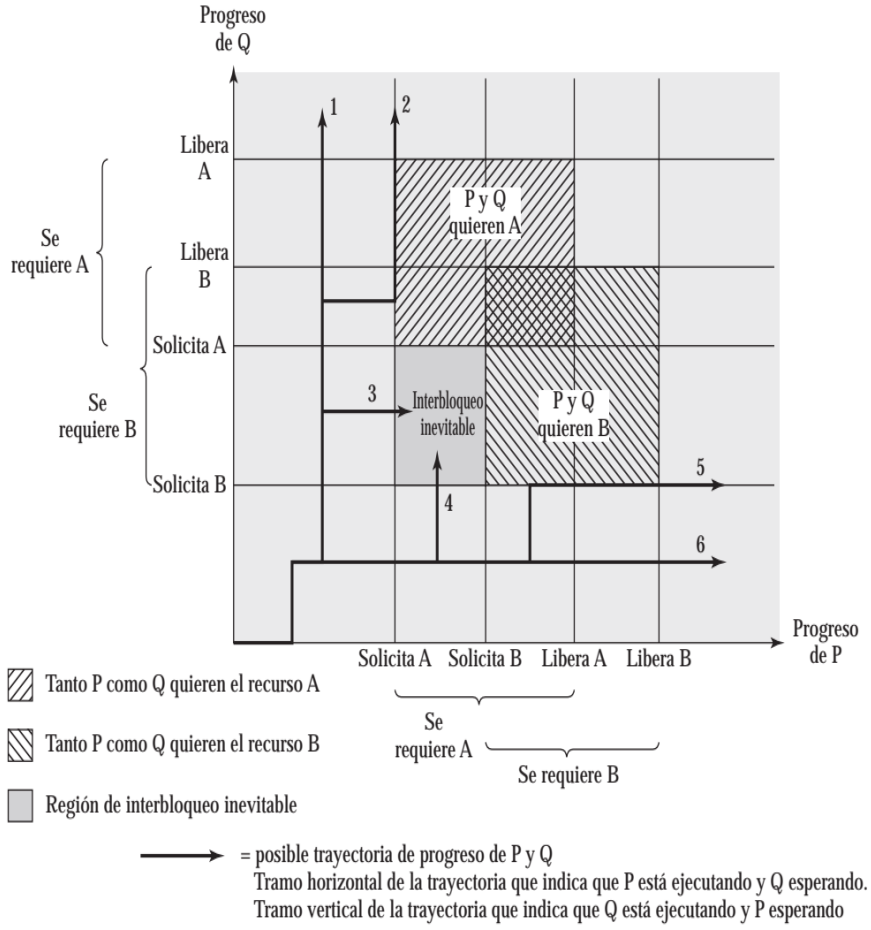


Figura 4.10: Ejemplo de interbloqueo

A continuación, se examina un diagrama de interbloqueo involucrando procesos y recursos del computador. La Figura 4.10, denominada diagrama de progreso conjunto, muestra el progreso de dos procesos compitiendo por dos recursos. Cada proceso necesita el uso exclusivo de ambos recursos durante un cierto periodo de tiempo. Suponga que hay dos procesos, P y Q, que tienen la siguiente estructura general:

Proceso P	Proceso Q
...	...
Solicita A	Solicita B
...	...
Solicita B	Solicita A
...	...
Libera A	Libera B
...	...
Libera B	Libera A
...	...

En la Figura 4.10, el eje x representa el progreso en la ejecución de P, mientras que el eje y representa el de Q. El progreso conjunto de los dos procesos se representa por tanto por una trayectoria que avanza desde el origen en dirección nordeste. En el caso de un sistema uniprocesador, sólo puede ejecutar un proceso cada vez, y la trayectoria consiste en segmentos horizontales y verticales alternados, tal que un segmento horizontal representa un periodo en el que P ejecuta y Q espera, mientras que un segmento vertical representa un periodo en el que Q ejecuta y P espera.

La figura muestra áreas en las que tanto P como Q requieren el recurso A (líneas ascendentes); áreas en las que ambos procesos requieren el recurso B (líneas descendentes); y áreas en las que ambos requieren ambos recursos. Debido a que se asume que cada proceso requiere el control exclusivo de un recurso, todas éstas son regiones prohibidas; es decir, es imposible que cualquier trayectoria que represente el progreso de la ejecución conjunta de P y Q entre en una de estas regiones.

La figura muestra seis diferentes trayectorias de ejecución, que se pueden resumir de la siguiente manera:

1 Q adquiere B y, a continuación, A, y, más tarde, libera B y A. Cuando P continúe su ejecución, será capaz de adquirir ambos recursos.
2 Q adquiere B y, a continuación, A. P ejecuta y se bloquea al solicitar A. Q libera B y A. Cuando P continúe su ejecución, será capaz de adquirir ambos recursos.
3 Q adquiere B y, a continuación, P adquiere A. El interbloqueo es inevitable, puesto que cuando la ejecución continúe, Q se bloqueará a la espera de A y P a la de B.
4 P adquiere A y, a continuación, Q adquiere B. El interbloqueo es inevitable, puesto que cuando la ejecución continúe, Q se bloqueará a la espera de A y P a la de B.
5 P adquiere A y, a continuación, B. Q ejecuta y se bloquea al solicitar B. P libera A y B. Cuando Q continúe su ejecución, será capaz de adquirir ambos recursos.
6 P adquiere A y, a continuación, B, y, más tarde, libera A y B. Cuando Q continúe su ejecución, será capaz de adquirir ambos recursos.

El área sombreada en gris en la Figura 4.10, que puede denominarse región fatal, está relacionada con el comentario realizado sobre las trayectorias 3 y 4. Si una trayectoria de ejecución entra en esta región fatal,

el interbloqueo es inevitable. Nótese que la existencia de una región fatal depende de la lógica de los dos procesos. Sin embargo, el interbloqueo es sólo inevitable si el progreso conjunto de los dos procesos crea una trayectoria que entra en la región fatal.

La aparición de un interbloqueo depende tanto de la dinámica de la ejecución como de los detalles de la aplicación. Por ejemplo, supóngase que P no necesitase ambos recursos al mismo tiempo de manera que los dos procesos tuvieran la siguiente estructura:

Proceso P	Proceso Q
...	...
Solicita A	Solicita B
...	...
Libera A	Solicita A
...	...
Solicita B	Libera B
...	...
Libera B	Libera A
...	...

La situación se refleja en la Figura 4.11. Si analiza la figura, el lector se convencerá de que, con independencia de la temporización relativa de los dos procesos, no puede ocurrir un interbloqueo.

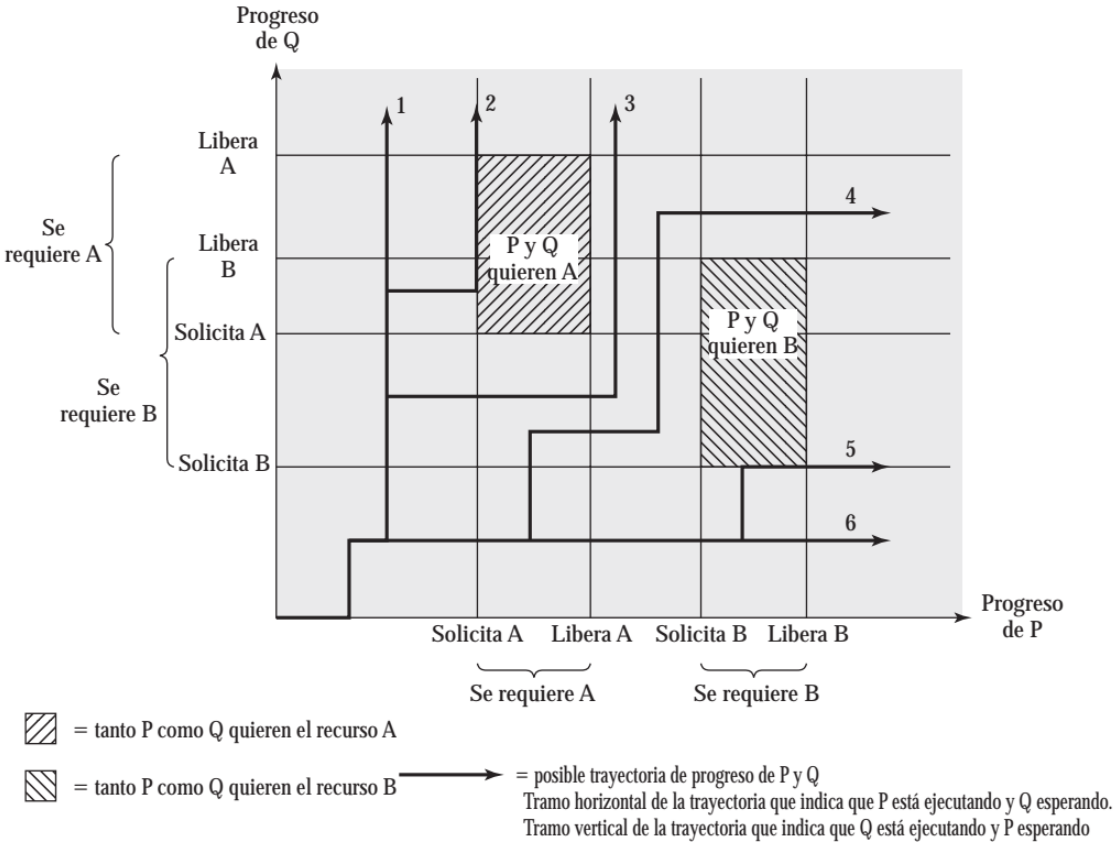


Figura 4.11: Ejemplo donde no hay interbloqueo

Como se ha mostrado, se puede utilizar el diagrama de progreso conjunto para registrar la historia de la ejecución de dos procesos que comparten recursos. En los casos donde más de dos procesos pueden competir por el mismo recurso, se requeriría un diagrama con más dimensiones. En cualquier caso, los principios concernientes con las regiones fatales y los interbloqueos permanecerían igual.

Prevención y evitación del interbloqueo

Prevención

La estrategia de prevención del interbloqueo consiste, de forma simplificada, en diseñar un sistema de manera que se excluya la posibilidad del interbloqueo. Se pueden clasificar los métodos de prevención del interbloqueo en dos categorías. Un método indirecto de prevención del interbloqueo es impedir la aparición de una de las tres

condiciones necesarias listadas previamente (las tres primeras). Un método directo de prevención del interbloqueo impide que se produzca una espera circular (cuarta condición). A continuación, se examinan las técnicas relacionadas con las cuatro condiciones.

Exclusión mutua

En general, la primera de las cuatro condiciones no puede eliminarse. Si el acceso a un recurso requiere exclusión mutua, el sistema operativo debe proporcionarlo. Algunos recursos, como los ficheros, pueden permitir múltiples accesos de lectura pero acceso exclusivo sólo para las escrituras. Incluso en este caso, puede ocurrir un interbloqueo si más de un proceso requiere permiso de escritura.

Retención y espera

La condición de retención y espera puede eliminarse estableciendo que un proceso debe solicitar al mismo tiempo todos sus recursos requeridos, bloqueándolo hasta que se le puedan conceder simultáneamente todas las peticiones. Esta estrategia es insuficiente en dos maneras. En primer lugar, un proceso puede quedarse esperando mucho tiempo hasta que todas sus solicitudes de recursos puedan satisfacerse, cuando, de hecho, podría haber continuado con solamente algunos de los recursos. En segundo lugar, los recursos asignados a un proceso pueden permanecer inutilizados durante un periodo de tiempo considerable, durante el cual se impide su uso a otros procesos. Otro problema es que un proceso puede no conocer por anticipado todos los recursos que requerirá.

Hay también un problema práctico creado por el uso de una programación modular o una estructura multihilo en una aplicación. La aplicación necesitaría ser consciente de todos los recursos que se solicitarán en todos los niveles o en todos los módulos para hacer una solicitud simultánea.

Sin expropiación

Esta condición se puede impedir de varias maneras. En primer lugar, si a un proceso que mantiene varios recursos se le deniega una petición posterior, ese proceso deberá liberar sus recursos originales y, si es necesario, los solicitará de nuevo junto con el recurso adicional. Alternativamente, si un proceso solicita un recurso que otro proceso mantiene actualmente, el sistema operativo puede expropiar al segundo proceso y obligarle a liberar sus recursos. Este último esquema impediría el interbloqueo sólo si no hay dos procesos que posean la misma prioridad.

Esta estrategia es sólo práctica cuando se aplica a recursos cuyo estado se puede salvar y restaurar más tarde, como es el caso de un procesador.

Espera circular

La condición de espera circular se puede impedir definiendo un orden lineal entre los distintos tipos de recursos. Si a un proceso le han asignado recursos de tipo R, posteriormente puede pedir sólo aquellos recursos cuyo tipo tenga un orden posterior al de R.

Para comprobar que esta estrategia funciona correctamente, se puede asociar un índice a cada tipo de recurso, de manera que el recurso Ri precede al Rj en la ordenación si $i < j$. A continuación, supóngase que dos procesos, A y B, están involucrados en un interbloqueo debido a que A ha adquirido Ri y solicitado Rj, y B ha adquirido Rj y solicitado Ri. Esta condición es imposible puesto que implica que $i < j$ y $j < i$.

Como ocurría en el caso de la retención y espera, la prevención de la espera circular puede ser ineficiente, ralentizando los procesos y denegando innecesariamente el acceso a un recurso.

Evitación

Una estrategia para resolver el problema del interbloqueo que difiere sutilmente de la prevención del interbloqueo es la predicción del interbloqueo. En la **prevención del interbloqueo**, se restringen las solicitudes de recurso para impedir al menos una de las cuatro condiciones de interbloqueo. Esto se realiza o bien indirectamente, impidiendo una de las tres condiciones de gestión necesarias (exclusión mutua, retención y espera, y sin expropiación), o directamente, evitando la espera circular. Esto conlleva un uso ineficiente de los recursos y una ejecución ineficiente de los procesos. La **predicción del interbloqueo**, por otro lado, permite las tres condiciones necesarias, pero toma decisiones razonables para asegurarse de que nunca se alcanza el punto del interbloqueo. De esta manera, la predicción permite más concurrencia que la prevención. Con la predicción del interbloqueo, se decide dinámicamente si la petición actual de reserva de un recurso, si se concede, podrá potencialmente causar un interbloqueo. La predicción del interbloqueo, por tanto, requiere el conocimiento de las futuras solicitudes de recursos del proceso.

Existen dos técnicas para evitar el interbloqueo:

- No iniciar un proceso si sus demandas pudieran llevar al interbloqueo.
- No conceder una petición adicional de un recurso por parte de un proceso si esta asignación pudiera provocar un interbloqueo.

Detección y recuperación del interbloque

Detección

Las estrategias de prevención de interbloqueo son muy conservadoras: resuelven el problema del interbloqueo limitando el acceso a los recursos e imponiendo restricciones a los procesos. En el extremo contrario, la

estrategia de detección del interbloqueo no limita el acceso a los recursos ni restringe las acciones de los procesos. Con la detección del interbloqueo, los recursos pedidos se conceden a los procesos siempre que sea posible. Periódicamente, el sistema operativo realiza un algoritmo que le permite detectar la condición de espera circular descrita anteriormente.

Recuperación

Una vez que se ha detectado el interbloqueo, se necesita alguna estrategia para recuperarlo. Las siguientes estrategias, listadas en orden de sofisticación creciente, son posibles:

1 Abortar todos los procesos involucrados en el interbloqueo. Esta es, se crea o no, una de las más usuales, si no la más, solución adoptada en los sistemas operativos.

2 Retroceder cada proceso en interbloqueo a algún punto de control (*checkpoint*) previamente definido, y rearrancar todos los procesos. Esto requiere que se implementen en el sistema mecanismos de retroceso y rearranque. El riesgo de esta técnica es que puede repetirse el interbloqueo original. Sin embargo, el indeterminismo del procesamiento concurrente puede asegurar que probablemente esto no suceda.

3 Abortar sucesivamente los procesos en el interbloqueo hasta que éste deje de existir. El orden en que seleccionan los procesos para abortarlos debería estar basado en algunos criterios que impliquen un coste mínimo. Después de cada aborto, se debe invocar de nuevo el algoritmo de detección para comprobar si todavía existe el interbloqueo.

4 Expropiar sucesivamente los recursos hasta que el interbloqueo deje de existir. Como en el tercer punto, se debería utilizar una selección basada en el coste, y se requiere una nueva invocación del algoritmo

de detección después de cada expropiación. Un proceso al que se le ha expropiado un recurso debe retroceder a un punto anterior a la adquisición de ese recurso.

Para los Puntos (3) y (4), el criterio de selección podría ser uno de los siguientes. Se elige el proceso con:

- la menor cantidad de tiempo de procesador consumida hasta ahora
- la menor cantidad de salida producida hasta ahora
- el mayor tiempo restante estimado
- el menor número total de recursos asignados hasta ahora
- la menor prioridad

Algunas de estas cantidades son más fáciles de medir que otras. El tiempo restante estimado es particularmente de difícil aplicación. Además, con excepción del criterio basado en la prioridad, no hay ninguna otra indicación del «coste» para el usuario, en contraposición con el coste para el sistema como un todo.

8 Preguntas de repaso

Para complementar su aprendizaje, conteste las siguientes preguntas.

1. Para lograr la exclusión mutua de una sección crítica donde se accede a un recurso compartido inicialmente disponible.

a. Los semáforos no sirven para lograr la exclusión mutua de las secciones críticas

b. La inicialización del semáforo binario depende del recurso que se comparta

c. El semáforo binario debe inicializarse en cero

d. El semáforo binario debe inicializarse a uno

2. El análisis de un grafo de asignación de recursos sirve para:

a. La evitación de interbloqueos

b. La detección de interbloqueos

c. La recuperación de interbloqueos

d. La prevención de interbloqueos

3. Si se usa un semáforo para la sincronización de procesos

a. Este se debe inicializar al número de procesos que se desean sincronizar

b. Las operaciones **wait y signal** se utilizan dentro de un mismo proceso

c. Se deben incluir variables de condición pues el semáforo únicamente proporciona exclusión mutua

d. Las operaciones wait y signal se utilizan en procesos separados

4. Los procesos concurrentes esperan que los usuarios puedan ejecutar varios procesos a la vez y no verse obligados a ejecutar los programas de forma secuencial.

Seleccione una:

Verdadero

Falso

5. La condición de carrera ocurre cuando solo un proceso accede a un recurso compartido debidamente controlado

Seleccione una:

Verdadero

Falso

6. Considerando el siguiente ejemplo:

Que un proceso ocupe un recurso y lo marque como ocupado y que termine sin marcarlo como desocupado. Si un segundo proceso pide ese recurso, lo verá ocupado.

¿Cuánto tiempo deberá esperar el segundo proceso a que se desocupe el recurso??

a. No tiene que esperar, el recurso está disponible

b. Ninguna respuesta es valida

c. Hasta que el temporizador de ocupado llegue a 0

d. Indefinidamente

7. Considerando la condición de carrera:

Dos procesos P1 y P2 comparten las variables globales con valores iniciales **a=1** y **b=3** , en algun momento de la ejecución P1 realiza la siguiente asignación **a=a+b** y P2 realiza la siguiente asignación **b=a+b**, Sí P2 ejecuta su setencia de asignación primero, ¿Cuales serán los valores de **a** y **b**?

a. a=4, b=3

b. a=4, b=7

c. a=1, b=4

d. a=5, b=4

8. ¿Al Hablar de interacción de procesos y decir que "¿Los resultados de un proceso pueden depender de la información obtenida de otros", de qué tipo de relación entre procesos se trata?

a. Competencia

b. Ninguna opción es correcta

c. Cooperación por compartición

d. Cooperación por comunicación

9. Cuales dos condiciones, son parte del conjunto de cuatro condiciones que se deben presentar para que se presente un problema serio como es la condición de abrazo mortal (deadlock)?

Seleccione dos

a. condición de apropiación

b. Condición crítica

c. condición de exclusión mutua

d. condición de espera circular

e. Condición de carrera

10. ¿Si la condición de abrazo mortal (deadlock) involucra algunos procesos, éstos quedarán congelados para siempre?

Seleccione una:

Verdadero

Falso

11. ¿En un sistema multiprocesador, deshabilitar interrupciones garantiza exclusión mutua?

Seleccione una:

Verdadero

Falso

12. Indicar si la siguiente afirmación es verdadero o falso

Si dentro de un proceso, la operación **semWait** decrementa el valor del semáforo y este es negativo, el proceso se bloquea

Seleccione una:

Verdadero

Falso

13. Complete:

Él _____ es una construcción del lenguaje de programación que proporciona una funcionalidad equivalente a la de los semáforos, pero es más fácil de controlar

14. Indicar si es verdadero o falso

La prevención del interbloqueo a diferencia de la predicción conlleva un uso eficiente de los recursos y una ejecución eficiente de los procesos

Seleccione una:

Verdadero

Falso

15. Indicar si es verdadero o falso

El interbloqueo NO se puede prevenir directamente evitando la espera circular

Seleccione una:

Verdadero

Falso

Bibliografía

Berzal, F. (2019). *Uso de paralelismo (hebras o procesos) permite mejorar el tiempo de respuesta de una aplicación.*

Google, S. d. (2021, 07 08). *Sistemas Operativos*. Retrieved from https://sites.google.com/site/osupaep2010/administracion-de-procesos/problemas-de-concurrencia/condiciones-de-carrera-o-competencia

Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating Systems Concepts* (10th ed.). Hoboken: Wiley.

Stallings, W. (2018). *Operating Systems, Internals and Design Princiles* (Ninth ed.). Malaysia: Pearson.

CAPÍTULO IV

Gestión de memoria y memoria virtual

Resultado de aprendizaje de la asignatura

Conocer los conceptos fundamentales, partiendo del enfoque clásico de los sistemas operativos como administrador eficiente de recursos, componentes, estructuras y funciones de los sistemas operativos, con la

finalidad que entiendan el funcionamiento y puedan realizar software de sistemas.

Unidad 4: **Gestión de memoria y memoria virtual**

Resultado de aprendizaje de la unidad:

Describir conceptos relacionados con administración de memoria y memoria virtual como base para el uso eficiente de la memoria principal y aumento del grado de multiprogramación de un sistema.

Tema 1: Gestión de memoria

La Memoria Principal es un recurso valioso que se debe asignar y compartir entre los procesos activos. Mantener en memoria la mayor cantidad posible de procesos, permite hacer un uso eficiente del procesador y de los servicios de E/S. El micro no tendrá tiempos ociosos pues siempre habrá un proceso para ejecutar. Por ello, una de las tareas más importantes y complejas de un sistema operativo es la gestión de la memoria. El sistema operativo Linux utiliza memoria virtual y para su administración, las técnicas de intercambio y paginación por demanda. Para ello, se vale de las siguientes estructuras de datos: El propósito principal de un sistema informático es ejecutar programas. Estos programas, junto con los datos a los que acceden, deben estar al menos

parcialmente en la memoria principal durante la ejecución. Los sistemas informáticos modernos mantienen varios procesos en la memoria durante la ejecución del sistema. Existen muchos esquemas de administración de memoria, que reflejan varios enfoques, y la eficacia de cada algoritmo varía con la situación. La selección de un esquema de administración de memoria para un sistema depende de muchos factores, especialmente del diseño de hardware del sistema. La mayoría de los algoritmos requieren algún tipo de soporte de hardware.

- Tabla de páginas: describe las páginas virtuales del proceso
- Descriptor de bloques de disco: describe el bloque de disco que contiene una página determinada.
- Tabla de marcos de páginas: describe cada marco de página de la memoria real.
- Tabla de intercambios: registra las páginas que están en cada uno de los dispositivos de intercambio, pues podemos definir más de uno.

La memoria física se divide en marcos de página y la memoria virtual se divide en páginas. La asignación de una página de memoria virtual a un marco físico se registra en la tabla de páginas. La unidad de gestión de memoria MMU realiza una traducción de la dirección de memoria virtual a la dirección de memoria física y verifica si la página a la cual quiere acceder está cargada o no en memoria. Las búsquedas en las tablas de páginas son costosas para el sistema, por lo que algunas arquitecturas mantienen una caché de búsqueda rápida llamada Translation Lookaside Buffer TLB (buffer de traducción anticipada).

Cualquier dirección de memoria virtual que requiera traducción a la dirección de memoria física, se compara primero con el buffer de traducción para una asignación válida. Cuando una traducción de

dirección válida no está presente en el TLB, se denomina “fallos de TLB”. En ese caso la unidad de gestión de memoria tendrá que consultar la tabla de páginas, y una vez encontrada la página, deberá realizar el intercambio de páginas entre la memoria y el disco. Dada la carga que significa para el sistema las búsquedas realizadas sobre las tablas de página, éste debe procurar reducir los fallos de TLB.

Con la característica HugeTLB en el kernel de Linux, las aplicaciones pueden beneficiarse con la asignación de páginas grandes, especialmente las que demandan mucha memoria tales como aplicaciones de bases de datos, aplicaciones HPC, etc. Esto permite que la cantidad de fallos de TLB se reduzcan y por lo tanto el costo que generan las búsquedas realizadas sobre las tablas de páginas. La información sobre Hugepages se puede consultar en el archivo **/proc/meminfo.**

cat /proc/meminfo | grep Huge

HugePages_Total:4

HugePages_Free:2

HugePages_Rsvd:0

Hugepagesize: 4096 KB

En el ejemplo anterior se muestra que están definidas 4 páginas grandes de 4MB c/u, de las cuales 2 han sido asignadas y 2 están libres. El manejo de páginas grandes aún no es totalmente transparente al usuario, por lo que el administrador deberá definir el conjunto de páginas grandes a utilizar y las aplicaciones a las que serán asignadas.

Fundamentos de la memoria principal

La memoria es fundamental para el funcionamiento de un sistema informático moderno. La memoria consta de un gran arreglo de bytes,

cada uno con su propia dirección. La CPU obtiene instrucciones de la memoria según el valor del contador del programa. Estas instrucciones pueden provocar una carga adicional desde y hacia el almacenamiento en direcciones de memoria específicas.

Un ciclo de instrucción-ejecución típico, por ejemplo, primero recupera una instrucción de la memoria. A continuación, se descodifica la instrucción y puede hacer que los operandos se obtengan de la memoria. Después de ejecutar la instrucción en los operandos, los resultados se pueden almacenar en la memoria. La unidad de memoria solo ve una secuencia de direcciones de memoria; no sabe cómo se generan (por el contador de instrucciones, indexación, direccionamiento indirecto, direcciones literales, etc.) o para qué sirven (instrucciones o datos). En consecuencia, podemos ignorar cómo un programa genera una dirección de memoria. Sólo nos interesa la secuencia de direcciones de memoria generadas por el programa en ejecución.

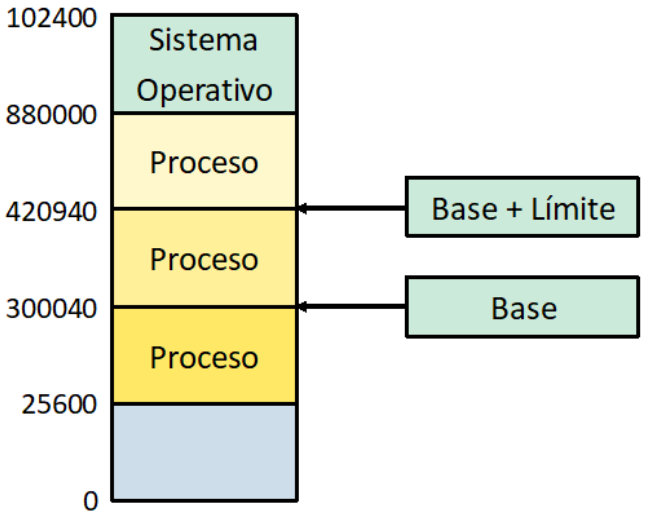
Hardware básico:

La memoria principal y los registros integrados en cada núcleo de procesamiento son los únicos almacenamientos de uso o propósito general a los cuales el CPU tiene acceso directamente. Hay instrucciones de máquina que toman las direcciones de memoria como argumentos, pero ninguna de estas instrucciones, porque no existen, toman direcciones del disco. Por lo tanto, cualquier instrucción en ejecución, y los datos que utilizan las instrucciones, deben estar en uno de estos dispositivos de almacenamiento de acceso directo. Si los datos no están en la memoria, se deben mover allí antes de que la CPU pueda trabajar con ellos.

Los registros que están integrados en cada núcleo de CPU son generalmente accesibles dentro de un ciclo del reloj de la CPU. Algunos núcleos de CPU pueden decodificar instrucciones y realizar operaciones

simples en el contenido del registro a la velocidad de una o más operaciones por pulso de reloj. No se puede decir lo mismo de la memoria principal, a la que se accede a través de una transacción en el bus de memoria. Completar un acceso a la memoria puede tomar muchos ciclos del reloj de la CPU. En tales casos, el procesador normalmente necesita detenerse, ya que no tiene los datos necesarios para completar la instrucción que está ejecutando. Esta situación es intolerable debido a la frecuencia de accesos a la memoria. El remedio es agregar memoria rápida entre la CPU y la memoria principal, típicamente en el chip de la CPU para un acceso rápido. Esta es la memoria caché. Para administrar una memoria caché integrada en la CPU, el hardware acelera automáticamente el acceso a la memoria sin ningún control del sistema operativo. (Recuerde que durante un bloqueo de memoria, un núcleo multiproceso puede cambiar del subproceso de hardware bloqueado a otro subproceso de hardware.)

No sólo nos preocupa la velocidad relativa de acceso a la memoria física, sino que también debemos garantizar un funcionamiento correcto. Para un correcto funcionamiento del sistema, debemos proteger el sistema operativo del acceso de los procesos del usuario, así como proteger los procesos de los usuarios entre sí. Esta protección debe ser proporcionada por el hardware, porque el sistema operativo no suele intervenir entre la CPU y sus accesos a la



memoria (debido a la penalización de rendimiento resultante). El Hardware implementa esta producción de varias maneras diferentes. Aquí, delineamos una posible implementación.

Primero nos aseguramos de que cada proceso tiene un espacio de memoria independiente, apartado o reservado. El espacio de memoria independiente para cada proceso los protege entre sí y además es fundamental para poder cargar varios procesos en memoria y poder ejecutarlos simultáneamente. Para separar los espacios de memoria, se necesita la capacidad de determinar el rango de direcciones legales (se entiendo por dirección legal a aquella que se encuentra instalada físicamente en el equipo) a las que puede acceder el proceso y garantizar que solo pueda acceder a estos espacios. Para proporcionar esta protección se lo puede hacer a través del
uso de dos registros, por lo general un registro base y un registro límite (ver la Figura 4-1). El registro base almacena la dirección de memoria física legal más pequeña, mientras que el registro límite especifica el tamaño del rango. Por ejemplo, si el registro base contiene el valor 300040 y el registro límite es 120900, entonces el programa podrá acceder legalmente a todas las direcciones comprendidas entre 300040 y 420940 (incluyendo los dos extremos).

La protección del espacio de memoria se consigue haciendo que el hardware de la CPU compare todas las direcciones generadas en modo usuario con el contenido de esos registros. Cualquier intento, por parte de un programa que se esté ejecutando en modo usuario, de acceder a la memoria del sistema operativo o a la memoria de otros usuarios hará que se produzca una interrupción hacia el sistema operativo, que tratará dicho intento como un error fatal (Figura 4-2). Este esquema evita que un

programa de usuario modifique (accidental o deliberadamente) el código y las estructuras de datos del sistema operativo o de otros usuarios.

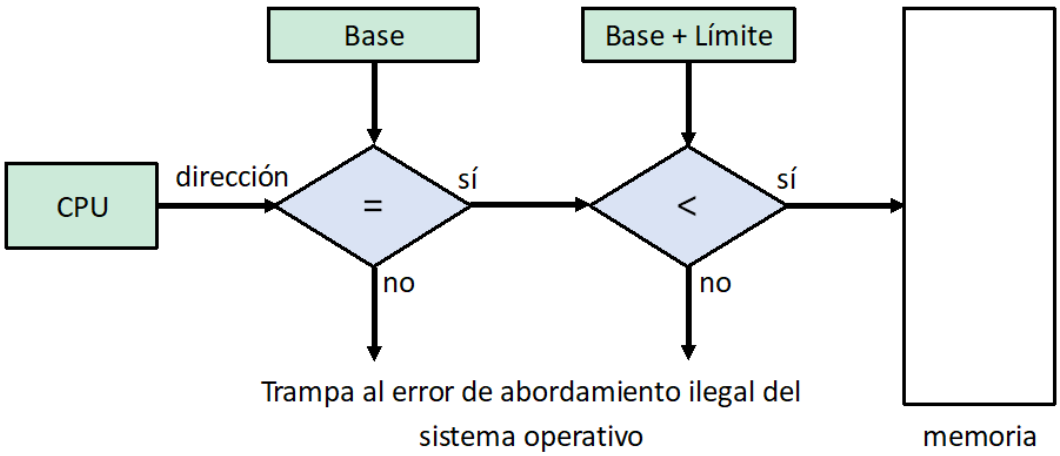
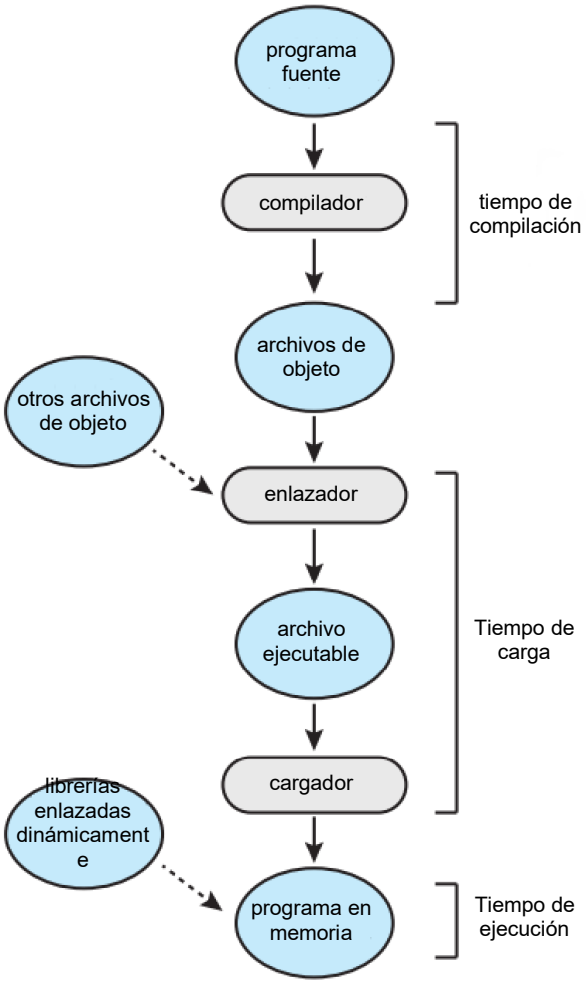


Figura 4-2: Protección hardware de las direcciones, utilizando un registro base y un registro límite

Los registros base y límite sólo pueden ser cargados por el sistema operativo, que utiliza una instrucción privilegiada especial. Puesto que las instrucciones privilegiadas sólo pueden ser ejecutadas en modo kernel y como sólo el sistema operativo se ejecuta en modo kernel, únicamente el sistema operativo podrá cargar los registros base y límite. Este esquema permite al sistema operativo modificar el valor de los registros, pero evita que los programas de usuario cambien el contenido de esos registros.

El sistema operativo, que se ejecuta en modo kernel, tiene acceso sin restricciones tanto a la memoria del sistema operativo como a la memoria de los usuarios. Esta disposición permite al sistema operativo cargar los programas de los usuarios en la memoria de los usuarios, volcar esos programas en caso de errores, acceder y modificar parámetros de llamadas del sistema, realizar E/S hacia y desde la memoria del usuario y proporcionar muchos otros servicios. Considere, por ejemplo, que un sistema operativo para un sistema de multiprocesamiento debe ejecutar modificadores de contexto, almacenando el estado de un proceso de los registros en la memoria principal antes de cargar el contexto del siguiente proceso desde la memoria principal en los registros.



Reasignación de direcciones

Normalmente, un programa reside en un disco como un archivo ejecutable binario. Para ejecutarse, el programa debe colocarse en la memoria y colocarse en el contexto de un proceso (como se describe en la unidad 2), donde se convierte en elegible para la ejecución en una CPU disponible. A

medida que se ejecuta el proceso, accede a instrucciones y datos de la memoria. Eventualmente, el proceso finaliza y su memoria se recupera para su uso por otros procesos.

La mayoría de los sistemas permiten que un proceso de usuario resida en cualquier parte de la memoria física. Por lo tanto, aunque el espacio de direcciones del equipo puede comenzar en 00000, la primera dirección del proceso de usuario no tiene por qué ser 00000. Verá más adelante cómo el sistema operativo realmente coloca un proceso en la memoria física.

En la mayoría de los casos, un programa de usuario pasa por varios pasos, algunos de los cuales pueden ser opcionales, antes de ejecutarse (Figura 4-3). Las direcciones se pueden representar de diferentes maneras durante estos pasos. Las direcciones del programa de origen son generalmente simbólicas (como el recuento de variables). Normalmente, un compilador enlaza estas direcciones simbólicas a direcciones reubicables (como "14 bytes desde el principio de este módulo"). El vinculador o cargador (consulte la Sección
2.5) a su vez enlaza las direcciones reubicables a direcciones absolutas (como 74014). Cada enlace es una asignación de un espacio de direcciones a otro.

Clásicamente, el enlace de instrucciones y datos a direcciones de memoria se puede hacer en cualquier paso del camino:

- Tiempo de compilación

  Si sabe en tiempo de compilación dónde residirá el proceso en la memoria, se puede generar código absoluto. Por ejemplo, si sabe que un proceso de usuario residirá a partir de la ubicación R, el código del compilador generado se iniciará en esa ubicación y se

extenderá desde allí. Si, en algún momento posterior, cambia la ubicación inicial, será necesario volver a compilar este código.

- Tiempo de carga

  Si no se conoce en tiempo de compilación donde el proceso residirá en memoria, el compilador debe generar **código reubicable**. En este caso, el enlace final se retrasa hasta el tiempo de carga. Si la dirección inicial cambia, solo necesitamos volver a cargar el código de usuario para incorporar este valor modificado.

- Tiempo de ejecución

  Si el proceso se puede mover durante su ejecución de un segmento de memoria a otro, el enlace debe retrasarse hasta el tiempo de ejecución. Debe haber hardware especial disponible para que este esquema funcione. La mayoría de los sistemas operativos utilizan este método.

Una parte importante de este capítulo se dedica a mostrar cómo estos diversos enlaces se pueden implementar eficazmente en un sistema informático y a discutir el soporte de hardware adecuado.

Espacios de direcciones lógico y físico

Una dirección generada por la CPU se denomina comúnmente dirección lógica, mientras que una dirección vista por la unidad de memoria (es decir, la que se carga en el registro de direcciones de memoria de la memoria) se denomina comúnmente dirección física.

Los métodos de reasignación en tiempo de compilación y en tiempo de carga generan direcciones lógicas y físicas idénticas. Sin embargo, el esquema de reasignación de direcciones en tiempo de ejecución hace que las direcciones lógica y física difieran. En este caso, usualmente decimos que la dirección lógica es una dirección virtual. A lo largo de este texto,

utilizaremos los términos dirección lógica y dirección virtual de manera intercambiable. El conjunto de todas las direcciones lógicas generadas por un programa es lo que se denomina un espacio de **direcciones lógicas**; el conjunto de todas las **direcciones físicas** correspondientes a estas direcciones lógicas es un espacio de direcciones físicas. Así, en el esquema de reasignación de direcciones en tiempo de ejecución, decimos que los espacios de direcciones lógicas y físicas difieren.

La correspondencia entre direcciones virtuales y físicas en tiempo de ejecución es establecida por un dispositivo hardware que se denomina **unidad de gestión de memoria** (MMU, memory management unit). Podemos seleccionar entre varios métodos distintos para establecer esta correspondencia. Por el momento, vamos a ilustrar esta operación de asociación mediante un esquema MMU simple, que es una generalización del esquema de registro base descrita al comienzo de este documento. El registro base se denominará ahora registro de reubicación. El valor contenido en el registro de reubicación suma a todas las direcciones generadas por un proceso de usuario en el momento de enviarlas a memoria (véase la Figura 4-4). Por ejemplo, si la base se encuentra en la dirección 14000, cualquier intento del usuario de direccionar la posición de memoria cero se reubicará dinámicamente en la dirección 14000; un acceso a la ubicación 346 se convertirá en la ubicación 14346. El sistema operativo MS-DOS que se ejecuta sobre la familia de procesadores Intel 80x86 utiliza cuatro registros de reubicación a la hora de cargar y ejecutar procesos.

El programa de usuario nunca ve las direcciones físicas reales. El programa puede crear un puntero a la ubicación 346, almacenarlo en memoria, manipularlo y compararlo con otras direcciones, siempre como el número 346. Sólo cuando se lo utiliza como dirección de memoria (por

ejemplo, en una operación de lectura o escritura indirecta) se producirá la reubicación en relación con el registro base. El programa de usuario maneja direcciones lógicas y el hardware de conversión (mapeo) de memoria convierte esas direcciones lógicas en direcciones físicas. Esta forma de acoplamiento en tiempo de ejecución ya fue expuesta anteriormente. La ubicación final de una dirección de memoria referenciada no se determina hasta que se realiza esa referencia.

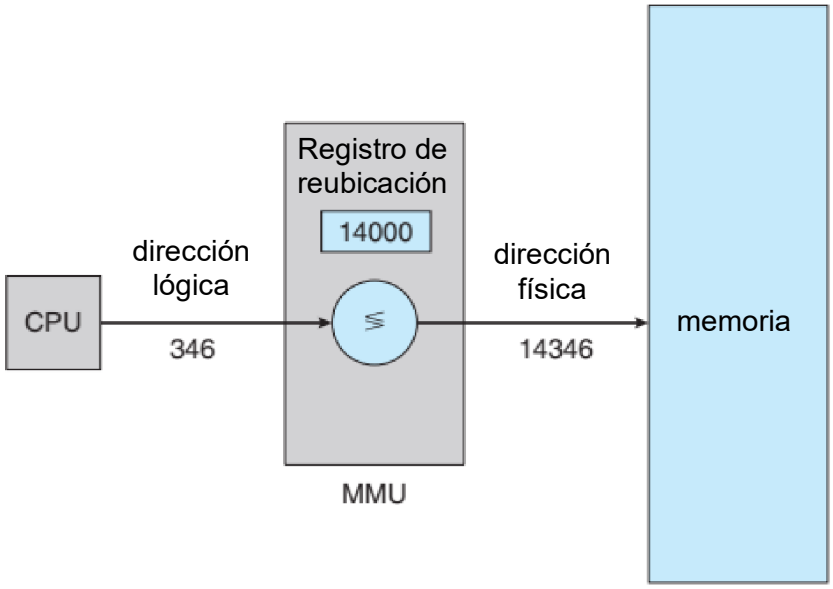


Figura 4-4: Reubicación dinámica, utilizando un registro de reubicación

Ahora tenemos dos tipos diferentes de direcciones: direcciones lógicas (en el rango comprendido entre 0 y max) y direcciones físicas (en el rango comprendido entre R + 0 y R + max para un valor base igual a R). Ei usuario sólo genera direcciones lógicas y piensa que el proceso se ejecuta en las ubicaciones comprendidas entre 0 y max. El programa de usuario suministra direcciones lógicas y estas direcciones lógicas deben ser convertidas en direcciones físicas antes de utilizarlas.

El concepto de un espacio de direcciones lógicas que se acopla a un espacio de direcciones físicas separado resulta crucial para una adecuada gestión de la memoria.

Asignación de memoria contigua

La memoria principal debe acomodar tanto el sistema operativo como los diversos procesos de usuario. Por lo tanto, necesitamos asignar la memoria principal de la manera más eficiente posible. En esta sección se explica un método inicial, la asignación de memoria contigua.

La memoria se divide generalmente en dos particiones: una para el sistema operativo y otra para los procesos del usuario. Podemos colocar el sistema operativo en direcciones de memoria baja o direcciones de memoria alta. Esta decisión depende de muchos factores, como la ubicación del vector de interrupción. Sin embargo, muchos sistemas operativos (incluyendo Linux y Windows) colocan el sistema operativo en alta memoria, y por lo tanto discutimos sólo esa situación.

Normalmente queremos que varios procesos de usuario residan en la memoria al mismo tiempo. Por lo tanto, debemos considerar cómo asignar la memoria disponible a los procesos que están a la espera de ser puestos en la memoria. En la asignación de memoria contigua, cada proceso está contenido en una sola sección de memoria que es contigua a la sección que contiene el siguiente proceso. Sin embargo, antes de seguir discutiendo este esquema de asignación de memoria, debemos abordar el tema de la protección de la memoria.

Protección de la memoria

Podemos evitar que un proceso acceda a la memoria que no posee combinando dos ideas discutidas anteriormente. Si tenemos un sistema con un registro de reubicación, junto con un registro de límite, logramos nuestro objetivo. El registro de reubicación contiene el valor de la dirección física más pequeña; el registro de límites contiene el rango de direcciones lógicas (por ejemplo, reubicación 100040 y límite 74600). Cada dirección

lógica debe estar dentro del intervalo especificado por el registro de límites. La MMU asigna la dirección lógica dinámicamente agregando el valor en el registro de reubicación. Esta dirección asignada se envía a la memoria (Figura 4-5).

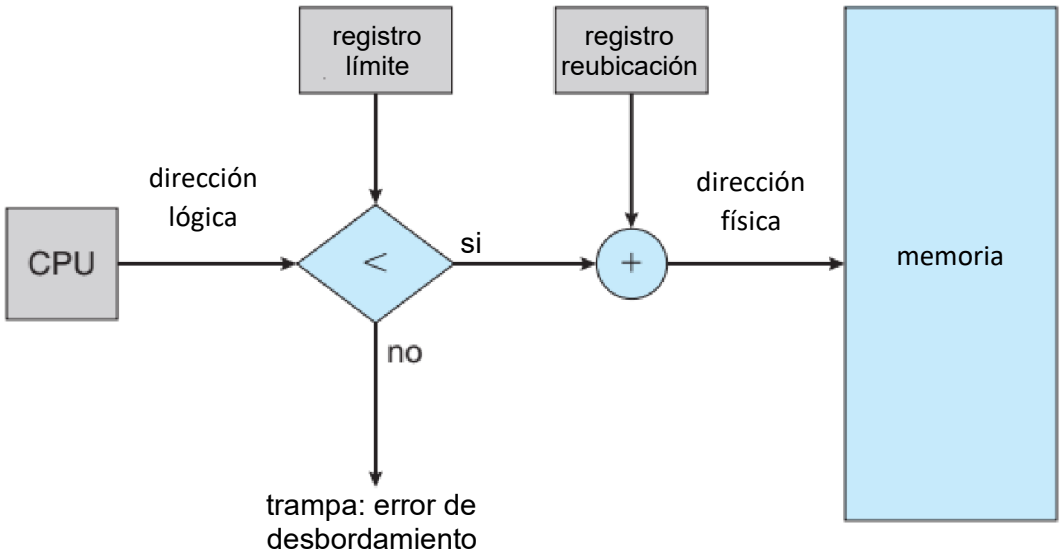


Figura 4-5: Soporte de hardware para registros de reubicación y límites

Cuando el programador de CPU selecciona un proceso para la ejecución, el distribuidor carga los registros de reubicación y límite con los valores correctos como parte del cambio de contexto. Debido a que cada dirección generada por una CPU se comprueba con estos registros, podemos proteger tanto el sistema operativo como los de los demás usuarios y los datos de ser modificados por este proceso en ejecución.

El esquema de reubicación-registro proporciona una manera eficaz de permitir que el tamaño del sistema operativo cambie dinámicamente. Esta flexibilidad es deseable en muchas situaciones. Por ejemplo, el sistema operativo contiene código y espacio de búfer para los controladores de dispositivo. Si un controlador de dispositivo no está actualmente en uso, tiene poco sentido mantenerlo en la memoria; en su lugar, se puede cargar

en la memoria sólo cuando es necesario. Del mismo modo, cuando el controlador del dispositivo ya no es necesario, se puede quitar y asignar su memoria para otras necesidades.

Asignación de memoria

Ahora estamos listos para recurrir a la asignación de memoria. Uno de los métodos más simples de asignación de memoria es asignar procesos a particiones de tamaño variable en la memoria, donde cada partición puede contener exactamente un proceso. En este esquema de **partición-variable**, el sistema operativo mantiene una tabla que indica qué partes de memoria están disponibles y cuáles están ocupadas. Inicialmente, toda la memoria está disponible para los procesos del usuario y se considera un gran bloque de memoria disponible, un **agujero**. Eventualmente, como verá, la memoria contiene un conjunto de agujeros de varios tamaños.

La Figura 4-11 representa este esquema. Inicialmente, la memoria se utiliza completamente, que contiene los procesos 5, 8 y 2. Después del proceso 8 hojas, hay un agujero contiguo. Más adelante, el proceso 9 llega y se asigna memoria. A continuación, el proceso 5 sale, lo que resulta en dos agujeros no contiguos.

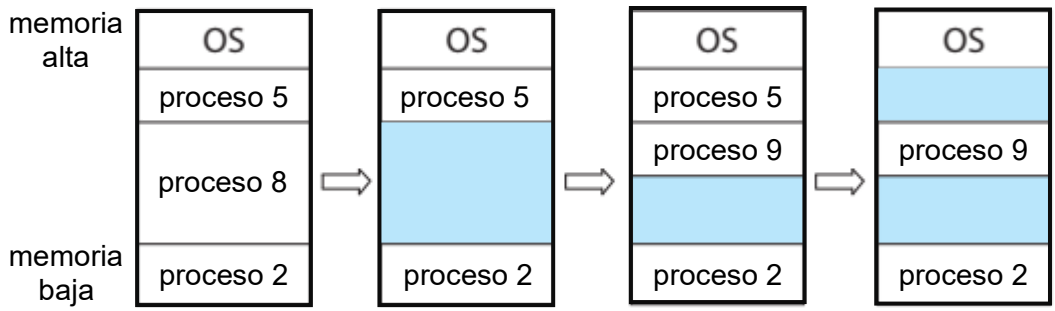


Figura 4-6: Partición Variable

A medida que los procesos entran en el sistema, el sistema operativo tiene en cuenta los requisitos de memoria de cada proceso y la cantidad de espacio de memoria disponible para determinar qué procesos se asignan

memoria. Cuando se asigna espacio a un proceso, se carga en la memoria, donde puede competir por el tiempo de CPU. Cuando un proceso finaliza, libera su memoria, que el sistema operativo puede proporcionar a otro proceso.

¿Qué sucede cuando no hay suficiente memoria para satisfacer las demandas de un proceso de llegada? Una opción es simplemente rechazar el proceso y proporcionar un mensaje de error adecuado. Alternativamente, podemos colocar estos procesos en una cola de espera. Cuando la memoria se libera más adelante, el sistema operativo comprueba la cola de espera para determinar si satisfará las demandas de memoria de un proceso en espera.

En general, como se mencionó, los bloques de memoria disponibles comprenden un conjunto de agujeros de varios tamaños dispersos por toda la memoria. Cuando llega un proceso y necesita memoria, el sistema busca en el conjunto un agujero lo suficientemente grande para este proceso. Si el agujero es demasiado grande, se divide en dos partes. Una parte se asigna al proceso de llegada; el otro se devuelve al conjunto de agujeros. Cuando un proceso finaliza, libera su bloque de memoria, que luego se coloca de nuevo en el conjunto de agujeros. Si el nuevo agujero es adyacente a otros agujeros, estos agujeros adyacentes se fusionan para formar un agujero más grande.

Este procedimiento es una instancia particular del problema de almacenamiento dinámico general, que se refiere a cómo satisfacer una solicitud de tamaño n de una lista de agujeros libres. Hay muchas soluciones a este problema. Las estrategias de primer ajuste, mejor-fi y peor-fi son las más utilizadas para seleccionar un agujero libre del conjunto de agujeros disponibles.

- Primer ajuste

  Asigne el primer agujero que sea lo suficientemente grande. La búsqueda puede comenzar al principio del conjunto de agujeros o en la ubicación donde terminó la búsqueda de primer ajuste anterior. Podemos dejar de buscar tan pronto como encontremos un agujero libre que sea lo suficientemente grande.

- Mejor ajuste

  Asigne el agujero más pequeño que sea lo suficientemente grande. Debemos buscar en toda la lista, a menos que la lista esté ordenada por tamaño. Esta estrategia produce el agujero sobrante más pequeño.

- Peor ajuste

  Asigne el agujero más grande. Una vez más, debemos buscar en toda la lista, a menos que esté ordenada por tamaño. Esta estrategia produce el agujero sobrante más grande, que puede ser más útil que el agujero sobrante más pequeño de un enfoque de mejor ajuste.

Las simulaciones han demostrado que tanto el primer ajuste como el mejor ajuste son mejores que los peores en términos de disminución del tiempo y la utilización del almacenamiento. Ni el primer ajuste ni el mejor ajuste son claramente mejores que los otros en términos de utilización del almacenamiento de información, pero el primer ajuste es generalmente más rápido.

Fragmentación

Tanto las estrategias de primer ajuste como las que mejor se ajustan para la asignación de memoria sufren de fragmentación externa. A medida que los procesos se cargan y se eliminan de la memoria, el espacio de memoria

libre se divide en pequeños trozos. La fragmentación externa existe cuando hay suficiente espacio de memoria total para satisfacer una solicitud, pero los espacios disponibles no son contiguos: el almacenamiento se fragmenta en un gran número de agujeros pequeños. Este problema de fragmentación puede ser grave. En el peor de los casos, podríamos tener un bloque de memoria libre (o desperdiciada) entre cada dos procesos. Si todos estos pequeños fragmentos de memoria estuvieran en un gran bloque libre en su lugar, podríamos ser capaces de ejecutar varios procesos más.

Si estamos utilizando la estrategia de primer ajuste o de mejor ajuste puede afectar la cantidad de fragmentación. (El primer ajuste es mejor para algunos sistemas, mientras que el mejor ajuste es mejor para otros.) Otro factor es qué extremo de un bloque libre se asigna. (¿Cuál es la pieza sobrante, la de arriba o la de la parte inferior?) Sin embargo, no importa qué algoritmo se utilice, la fragmentación externa será un problema.

Dependiendo de la cantidad total de almacenamiento de memoria y el tamaño medio del proceso, la fragmentación externa puede ser un problema menor o importante. El análisis estadístico del primer ajuste, por ejemplo, revela que, incluso con alguna optimización, dados los bloques asignados N, otros bloques de 0,5 N se perderán en la fragmentación. ¡Es decir, un tercio de la memoria puede ser inutilizable! Esta propiedad se conoce como la regla del **50 por ciento**.

La fragmentación de la memoria puede ser tanto interna como externa. Considere un esquema de asignación de varias particiones con un agujero de 18.464 bytes. Supongamos que el siguiente proceso solicita 18.462 bytes. Si asignamos exactamente el bloque solicitado, nos quedamos con un agujero de 2 bytes. La sobrecarga para realizar un seguimiento de este agujero será sustancialmente más grande que el agujero en sí. El enfoque

general para evitar este problema es dividir la memoria física en bloques de tamaño fijo y asignar memoria en unidades basadas en el tamaño del bloque. Con este enfoque, la memoria asignada a un proceso puede ser ligeramente mayor que la memoria solicitada. La diferencia entre estos dos números es la **fragmentación interna,** memoria no utilizada que es interna a una partición.

Una solución al problema de la fragmentación externa es la compactación. El objetivo es barajar el contenido de la memoria para colocar toda la memoria libre en un bloque grande. Sin embargo, la compactación no siempre es posible. Si la reubicación es estática y se realiza en el momento del montaje o de la carga, no se puede realizar la compactación. Sólo es posible si la reubicación es dinámica y se realiza en tiempo de ejecución. Si las direcciones se reubican dinámicamente, la reubicación solo requiere mover el programa y los datos y, a continuación, cambiar el registro base para reflejar la nueva dirección base. Cuando la compactación es posible, debemos determinar su costo. El algoritmo de compactación más simple es mover todos los procesos hacia un extremo de la memoria; todos los agujeros se mueven en la otra dirección, produciendo un gran agujero de memoria disponible. Este esquema puede ser costoso.

Otra posible solución al problema de fragmentación externa es permitir que el espacio de direcciones lógico de los procesos no sea contiguo, permitiendo así que un proceso se asigne memoria física dondequiera que dicha memoria esté disponible. Esta es la estrategia utilizada en la paginación, la técnica de administración de memoria más común para sistemas informáticos. Describimos la paginación en la siguiente sección.

La fragmentación es un problema general en la computación que puede ocurrir dondequiera que debamos gestionar bloques de datos.

Paginación

La administración de memoria discutida hasta ahora ha requerido que el espacio de direcciones física de un proceso sea contiguo. Ahora introducimos la paginación, un esquema de administración de memoria que permite que el espacio de direcciones físicas de un proceso no sea contiguo. La paginación evita la fragmentación externa y la necesidad asociada de compactación, dos problemas que afectan a la asignación de memoria contigua. Debido a que ofrece numerosas ventajas, la paginación en sus diversas formas se utiliza en la mayoría de los sistemas operativos, desde los de servidores grandes a través de los de dispositivos móviles.

La paginación se implementa mediante la cooperación entre el sistema operativo y el hardware del equipo.

Método básico

El método básico para implementar la paginación implica dividir la memoria física en bloques de tamaño fijo denominados marcos y dividir la memoria lógica en bloques del mismo tamaño denominados páginas. Cuando se va a ejecutar un proceso, sus páginas se cargan en cualquier marco de memoria disponible desde su origen (un sistema de archivos o el almacén de respaldo). El almacén de respaldo se divide en bloques de tamaño fijo que tienen el mismo tamaño que los marcos de memoria o clústeres de varios fotogramas. Esta idea bastante simple tiene una gran funcionalidad y amplias ramificaciones. Por ejemplo, el espacio de direcciones lógicas ahora es totalmente independiente del espacio de direcciones físico, por lo que un proceso puede tener un espacio de direcciones lógico de 64 bits aunque el sistema tenga menos de 264 bytes de memoria física.

Cada dirección generada por la CPU se divide en dos partes: un **número de página** (p) y un **desplazamiento de página** (d)

número de página (page number)	desplazamiento de página (page offset)
P	q

Estructura de la tabla de página

En esta sección, exploraremos algunas de las técnicas más comunes para estructurar la tabla de páginas, incluida la paginación jerárquica, las tablas de páginas con hash y las tablas de páginas invertidas.

Paginación jerárquica

La mayoría de los sistemas informáticos modernos admiten un gran espacio de direcciones lógicas (2^{32} a 2^{64}). En un entorno de este tipo, la propia tabla de páginas se vuelve excesivamente grande. Por ejemplo, considere un sistema con un espacio de direcciones lógica de 32 bits. Si el tamaño de página en un sistema de este tipo es de 4 KB (2^{12}), una tabla de páginas puede consistir en más de 1 millón de entradas (2^{20} x $2^{32}/2^{12}$). Suponiendo que cada entrada consta de 4 bytes, cada proceso puede necesitar hasta 4 MB de espacio de direcciones físicas solo para la tabla de páginas. Claramente, no queremos asignar la tabla de páginas de forma contigua en la memoria principal. Una solución simple a este problema es dividir la tabla de páginas en partes más pequeñas. Podemos lograr esta división de varias maneras.

Una forma es utilizar un algoritmo de paginación de dos niveles, en el que también se página la propia tabla de páginas (Figura 4-7). Por ejemplo,

considere de nuevo el sistema con un espacio de direcciones lógicas de 32 bits y un tamaño de página de 4 KB. Una dirección lógica se divide en un número de página que consta de 20 bits y un desplazamiento de página que consta de 12 bits. Dado que paginamos la tabla de páginas, el número de página se divide en un número de página de 10 bits y un desplazamiento de página de 10 bits. Por lo tanto, una dirección lógica es la siguiente:

número de página (page number)		desplazamiento de página (page offset)
p_1	p_2	q
10	10	12

donde p_1 es un índice en la tabla de página externa y p_2 es el desplazamiento dentro de la página de la tabla de páginas interna. El método de traducción de direcciones para esta arquitectura se muestra en la Figura 4-8. Dado que la traducción de direcciones funciona desde la tabla de página externa hacia dentro, este esquema también se conoce como una tabla de páginas **asignada hacia delante**.

Para un sistema con un espacio de direcciones lógicas de 64 bits, un esquema de paginación de dos niveles ya no es adecuado. Para ilustrar este punto, supongamos que el tamaño de página en un sistema de este tipo es de 4 KB (2^{12}). En este caso, la tabla de páginas consta de hasta 2^{52} entradas. Si usamos un esquema de paginación de dos niveles, las tablas de páginas internas pueden tener convenientemente una página o contener 2^{10} entradas de 4 bytes. Las direcciones tienen este aspecto:

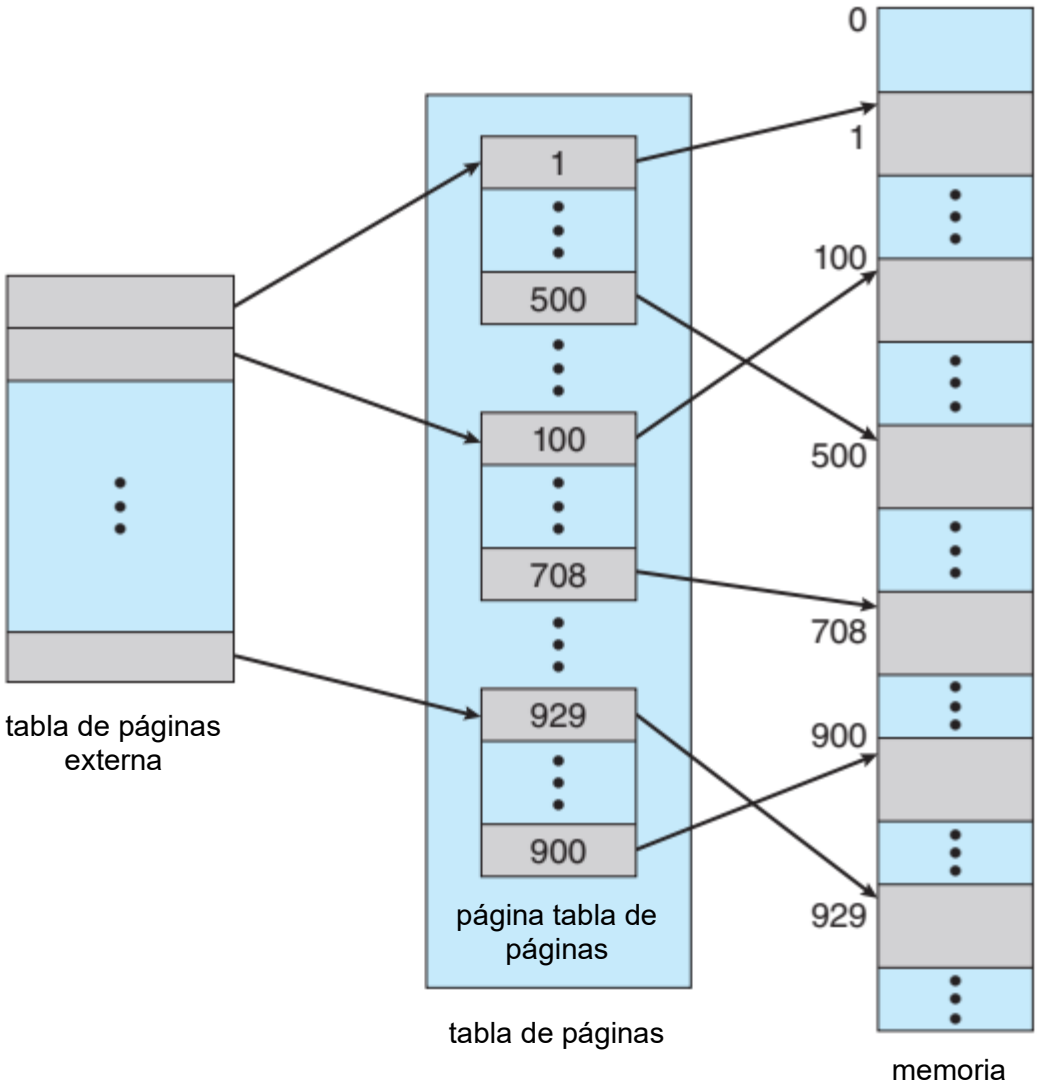


Figura 4-7: Esquema de tabla de páginas de dos niveles

página exterior	página interior	desplazamiento (offset)
p_1	p_2	q
42	10	12

La tabla de páginas externas consta de 2^{42} entradas o 2^{44} bytes. La manera obvia de evitar una tabla tan grande es dividir la tabla de páginas externas en partes más pequeñas. (Este enfoque también se utiliza en algunos procesadores de 32 bits para mayor flexibilidad y eficiencia.)

Podemos dividir la tabla de páginas externas de varias maneras. Por ejemplo, podemos paginar la tabla de página externa, lo que nos da un esquema de paginación de tres niveles. Supongamos que la tabla de página externa se compone de páginas de tamaño estándar (2^{10} entradas o 2^{12} bytes). En este caso, un espacio de direcciones de 64 bits sigue siendo desalentador:

segunda página exterior	página exterior	página interior	desplazamiento (offset)
p_1	p_1	p_3	q
32	10	10	12

La tabla de páginas externas sigue siendo de 2^{34} bytes (16 GB) de tamaño.

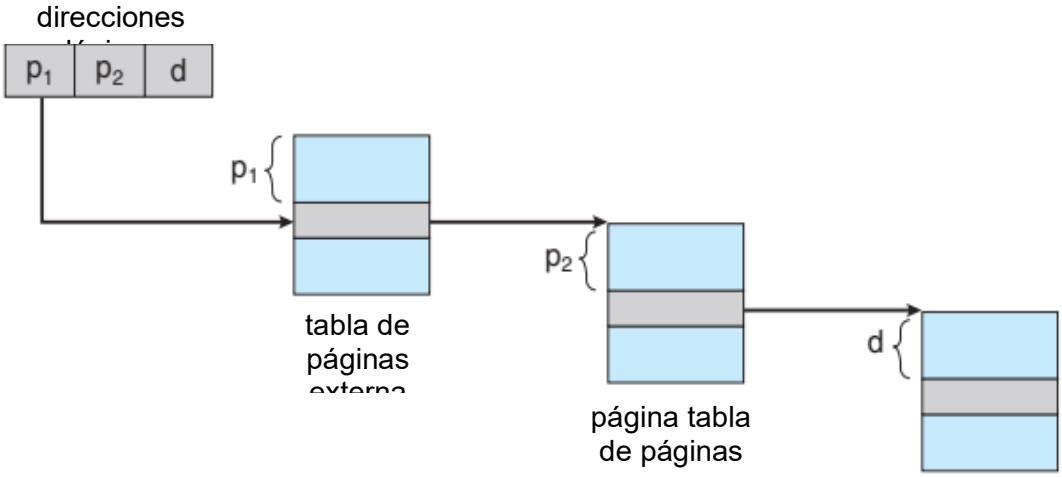


Figura 4-8: Traducción de direcciones para una arquitectura de paginación de 32 bits de dos niveles

El siguiente paso sería un esquema de paginación de cuatro niveles, donde también se página la tabla de página externa de segundo nivel, y así sucesivamente. El UltraSPARC de 64 bits requeriría siete niveles de

paginación (un número prohibitivo de accesos a la memoria) para traducir cada dirección lógica. Puede ver en este ejemplo por qué, para las arquitecturas de 64 bits, las tablas de páginas jerárquicas generalmente se consideran inapropiadas.

Tablas de páginas con hash

Un enfoque para controlar espacios de direcciones mayores de 32 bits es usar una **tabla de páginas con hash**, con el valor hash siendo el número de página virtual. Cada entrada de la tabla hash contiene una lista vinculada de elementos que se aplica hash a la misma ubicación (para controlar las colisiones). Cada elemento consta de tres campos: (1) el número de página virtual, (2) el valor del marco de página asignado y (3) un puntero al siguiente elemento de la lista vinculada.

El algoritmo funciona de la siguiente manera: el número de página virtual de la dirección virtual se aplica un algoritmo hash en la tabla hash. El número de página virtual se compara con el campo 1 del primer elemento de la lista vinculada. Si hay una coincidencia, el marco de página correspondiente (campo 2) se utiliza para formar la dirección física deseada. Si no hay ninguna coincidencia, se buscan entradas posteriores en la lista vinculada para un número de página virtual coincidente. Este esquema se muestra en la Figura 4-9.

Se ha propuesto una variación de este esquema que es útil para espacios de direcciones de 64 bits. Esta variación utiliza **tablas de páginas agrupadas**, que son similares a las tablas de páginas con hash, excepto que cada entrada de la tabla hash hace referencia a varias páginas (como 16) en lugar de a una sola página. Por lo tanto, una sola entrada de tabla de página puede almacenar las asignaciones para varios marcos de página física. Las tablas de páginas agrupadas son especialmente útiles

para **espacios** de direcciones dispersos, donde las referencias de memoria no son contiguas y están dispersas por todo el espacio de direcciones.

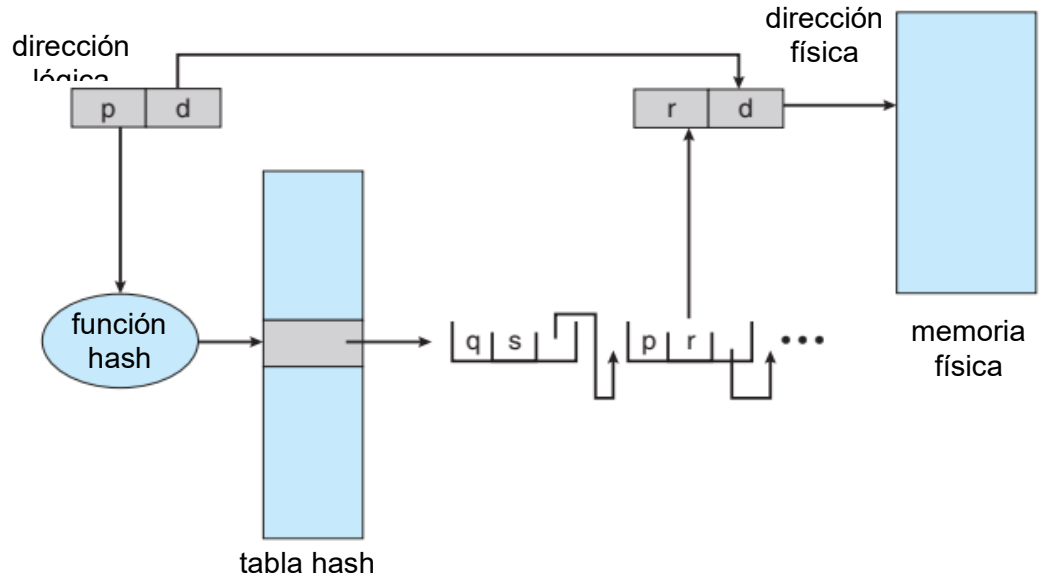


Figura 4-9: Tablas de páginas con hash

Tablas de páginas invertidas

Normalmente, cada proceso tiene una tabla de páginas asociada. La tabla de páginas tiene una entrada para cada página que el proceso está utilizando (o una ranura para cada dirección virtual, independientemente de la validez de esta última). Esta representación de tabla es natural, ya que procesa páginas de referencia a través de las direcciones virtuales de las páginas. A continuación, el sistema operativo debe traducir esta referencia en una dirección de memoria física. Puesto que la tabla se ordena por dirección virtual, el sistema operativo puede calcular dónde se encuentra la entrada de dirección física asociada y usar ese valor directamente. Uno de los inconvenientes de este método es que cada tabla de páginas puede consistir en millones de entradas. Estas tablas pueden consumir grandes cantidades de memoria física sólo para realizar un seguimiento de cómo se utiliza otra memoria física.

Para resolver este problema, podemos usar una tabla de páginas invertida. Una tabla de páginas invertida tiene una entrada para cada página real (o marco) de memoria. Cada entrada consta de la dirección virtual de la página almacenada en esa ubicación de memoria real, con información sobre el proceso que posee la página. Por lo tanto, sólo una tabla de páginas está en el sistema, y sólo tiene una entrada para cada página de memoria. La Figura 9.18 muestra el funcionamiento de una tabla de páginas invertida.

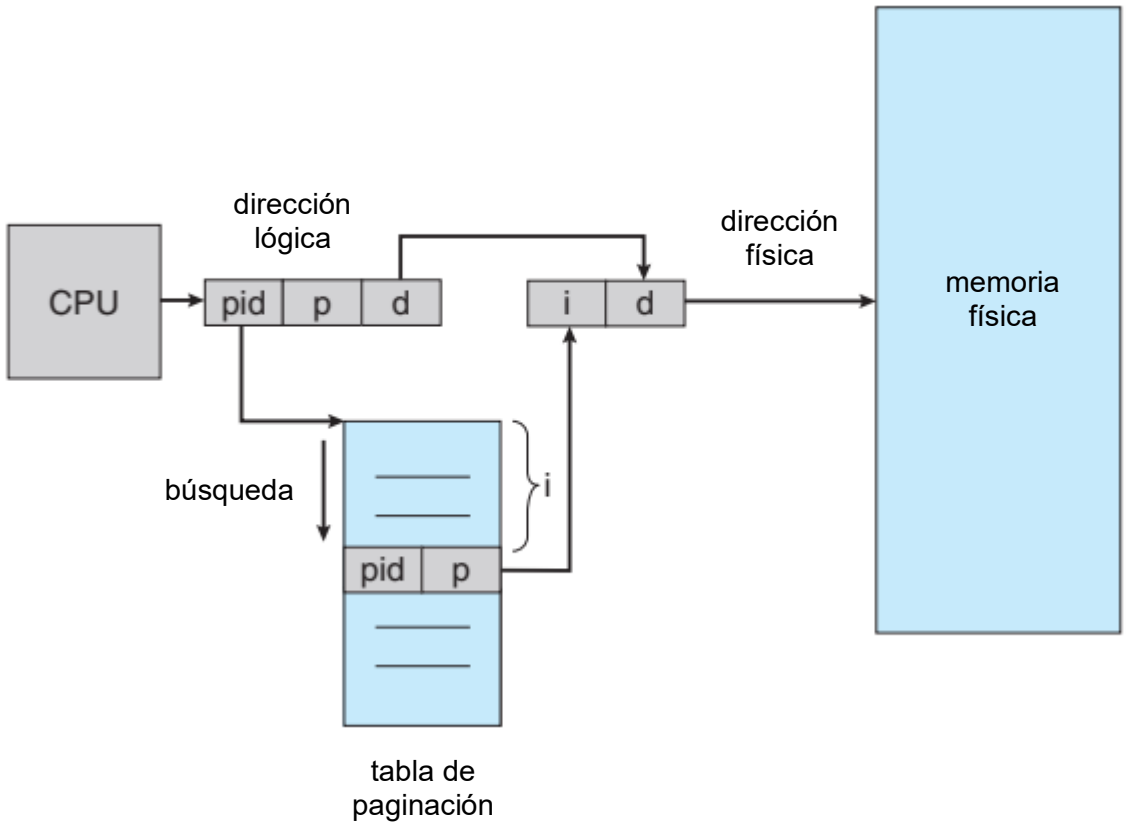


Figura 4-10: tabla de paginación invertida

Segmentación

Un aspecto importante de la gestión de memoria que se volvió inevitable con los mecanismos de paginación es la separación existente entre la vista que el usuario tiene de la memoria y la memoria física real. Como ya hemos comentado, la vista que el usuario tiene de la memoria no es la

misma que la memoria física real, sino que esa vista del usuario se mapea sobre la memoria física. Este mapeo permite la diferenciación entre memoria lógica y memoria física.

Intercambio

Las instrucciones de proceso y los datos en los que operan deben estar en la memoria para ejecutarse. Sin embargo, un proceso, o una parte de un proceso, se puede intercambiar temporalmente fuera de memoria a un almacén de respaldo y, a continuación, volver a la memoria para su ejecución continua (Figura 4-11). El intercambio hace posible que el espacio total de direcciones físicas de todos los procesos supere la memoria física real del sistema, aumentando así el grado de multiprogramación en un sistema.

Intercambio estándar

El intercambio estándar implica mover procesos completos entre la memoria principal y un almacén de respaldo. El almacén de respaldo suele ser un almacenamiento secundario rápido. Debe ser lo suficientemente grande como para acomodar cualquier parte de los procesos que se deben almacenar y recuperar, y debe proporcionar acceso directo a estas imágenes de memoria. Cuando se intercambia un proceso o una pieza en el almacén de respaldo, las estructuras de datos asociadas con el proceso deben escribirse en el almacén de respaldo. Para un proceso multiproceso, todas las estructuras de datos por subproceso también deben intercambiarse. El sistema operativo también debe mantener metadatos para los procesos que se han intercambiado, de modo que se puedan restaurar cuando se intercambian de nuevo en la memoria.

La ventaja del intercambio estándar es que permite que la memoria física se sobresuscriba, de modo que el sistema puede acomodar más procesos

de los que hay memoria física real para almacenarlos. Los procesos inactivos o en su mayoría inactivos son buenos candidatos para el intercambio; cualquier memoria que se haya asignado a estos procesos inactivos se puede dedicar a los procesos activos. Si un proceso inactivo que se ha intercambiado se activa una vez más, debe volver a intercambiarse. Esto se ilustra en la Figura 4-11.

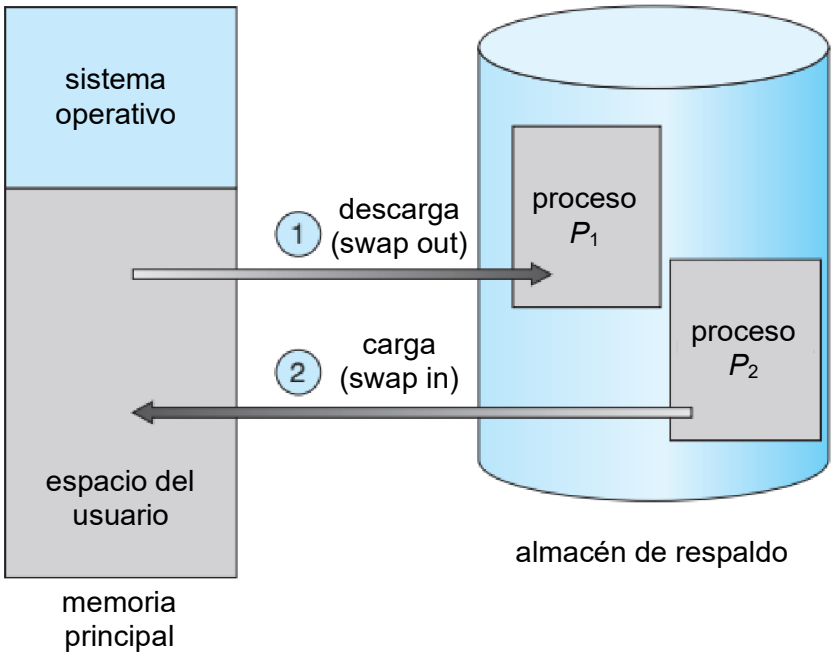


Figura 4-11: Intercambio de dos procesos utilizando un disco como almacén de respaldo

Intercambio con paginación

El intercambio estándar se utilizó en sistemas UNIX tradicionales, pero generalmente ya no se utiliza en sistemas operativos contemporáneos, porque la cantidad de tiempo necesario para mover procesos completos entre la memoria y el almacén de respaldo es prohibitivo. (Una excepción a esto es Solaris, que todavía utiliza el intercambio estándar, sin embargo, sólo en circunstancias terribles cuando la memoria disponible es extremadamente baja.)

La mayoría de los sistemas, incluidos Linux y Windows, ahora utilizan una variación del intercambio en el que se pueden intercambiar las páginas de un proceso, en lugar de un proceso completo. Esta estrategia todavía permite que la memoria física se sobrescriba, pero no incurre en el costo de intercambiar procesos completos, ya que presumiblemente sólo un pequeño número de páginas participarán en el intercambio. De hecho, el término intercambio ahora se refiere generalmente al intercambio estándar, y la paginación se refiere al intercambio con la paginación. Una operación de descarga de página (***page out***) mueve una página de la memoria al almacén de respaldo; el proceso inverso se conoce como carga de página (***page in***). El intercambio con paginación se ilustra en la Figura 4-12, donde un subconjunto de páginas para los procesos A y B se paginan y paginan respectivamente.

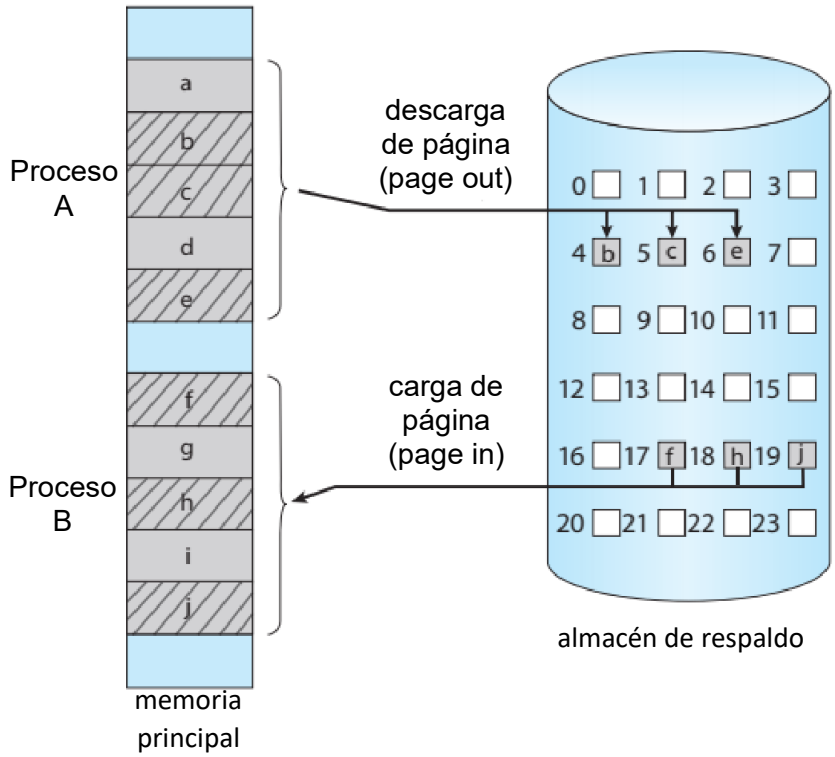


Figura 4-12: Intercambio con paginación

Intercambio en sistemas móviles

La mayoría de los sistemas operativos para PC y servidores admiten el intercambio de páginas. Por el contrario, los sistemas móviles normalmente no admiten el intercambio de ninguna forma. Los dispositivos móviles generalmente utilizan memoria flash en lugar de discos duros más amplios para un almacenamiento no volátil. La restricción de espacio resultante es una de las razones por las que los diseñadores de sistemas operativos móviles evitan el intercambio. Otras razones incluyen el número limitado de escrituras que la memoria flash puede tolerar antes de que se vuelva poco fiable y el bajo rendimiento entre la memoria principal y la memoria flash en estos dispositivos.

En lugar de usar el intercambio, cuando la memoria libre cae por debajo de un cierto umbral, iOS de Apple pide a las aplicaciones que renuncien voluntariamente a la memoria asignada. Los datos de solo lectura (como el código) se eliminan de la memoria principal y se vuelven a cargar posteriormente de la memoria flash si es necesario. Los datos que se han modificado (como la pila) nunca se quitan. Sin embargo, el sistema operativo puede terminar cualquier aplicación que no pueda liberar suficiente memoria.

Android adopta una estrategia similar a la utilizada por iOS. Puede terminar un proceso si no hay suficiente memoria libre disponible. Sin embargo, antes de finalizar un proceso, Android escribe su **estado de aplicación** en la memoria flash para que se pueda reiniciar rápidamente.

Debido a estas restricciones, los desarrolladores de sistemas móviles deben asignar y liberar memoria cuidadosamente para asegurarse de que sus aplicaciones no utilizan demasiada memoria o sufren de pérdidas de memoria.

Práctica de gestión de memoria y dump de memoria

- Extraer una imagen de la memoria RAM con ftk imager
- Pasar la imagen de la memoria RAM a kali Linux
- Luego debemos trabajar con la imagen de la memoria RAM extraída mediante comando en Kali Linux

Ejecución de AccessData FTK Imager

FTK Imager de AccessData es una herramienta para realizar réplicas y visualización previa de datos, la cual permite una evaluación rápida de evidencia electrónica para determinar si se garantiza un análisis posterior con una herramienta forense como AccessData

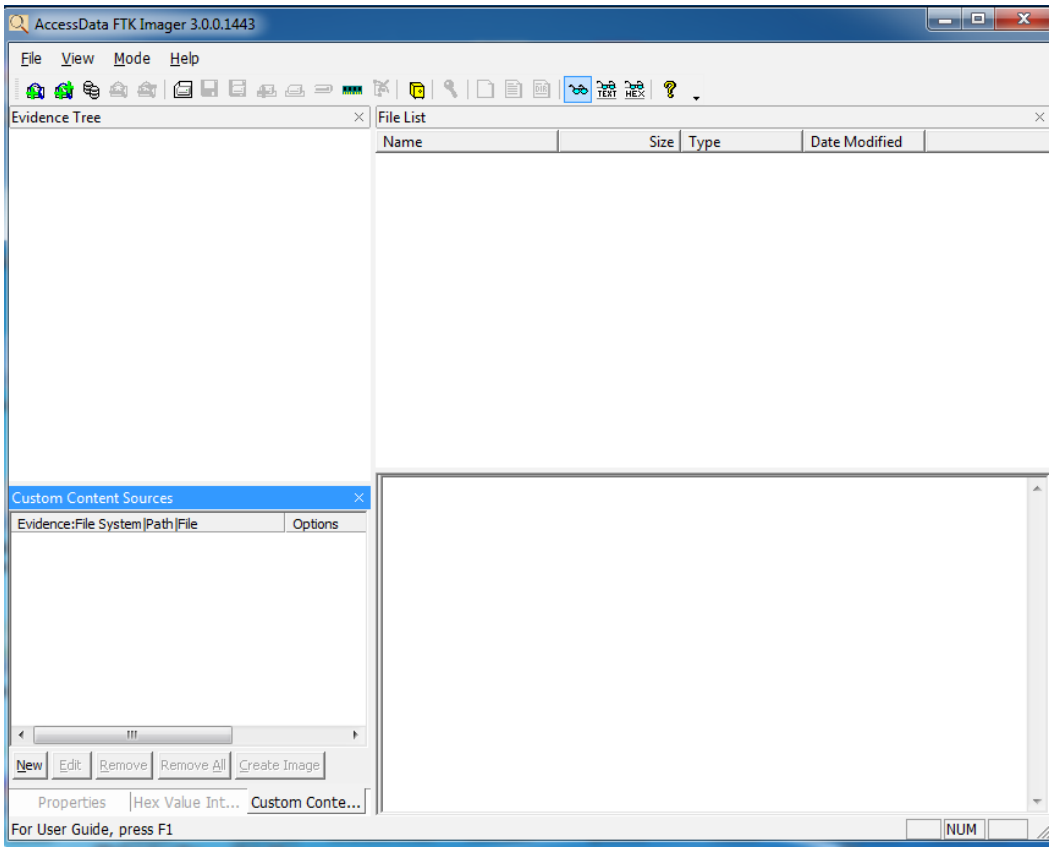


Figura 4-13: Ejecución de FTK Imager para extracción de dump de memoria

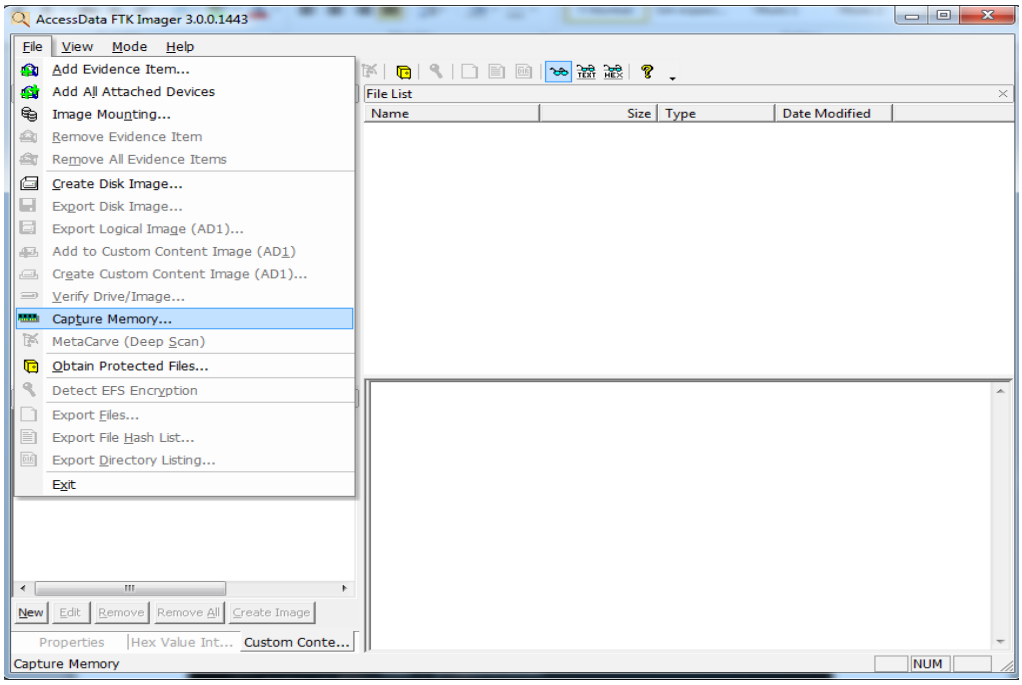


Figura 4-14: Ejecución de FTK Imager para extracción de dump de memoria

Se presentará una nueva ventana donde se requiere definir la Fuente. Para propósito de la presente práctica se creará una imagen forense de toda una unidad USB o Memory Stick, por lo tanto se selecciona la opción "Physical Drive" o Unidad Física. Luego hacer clic en el botón "Siguiente".

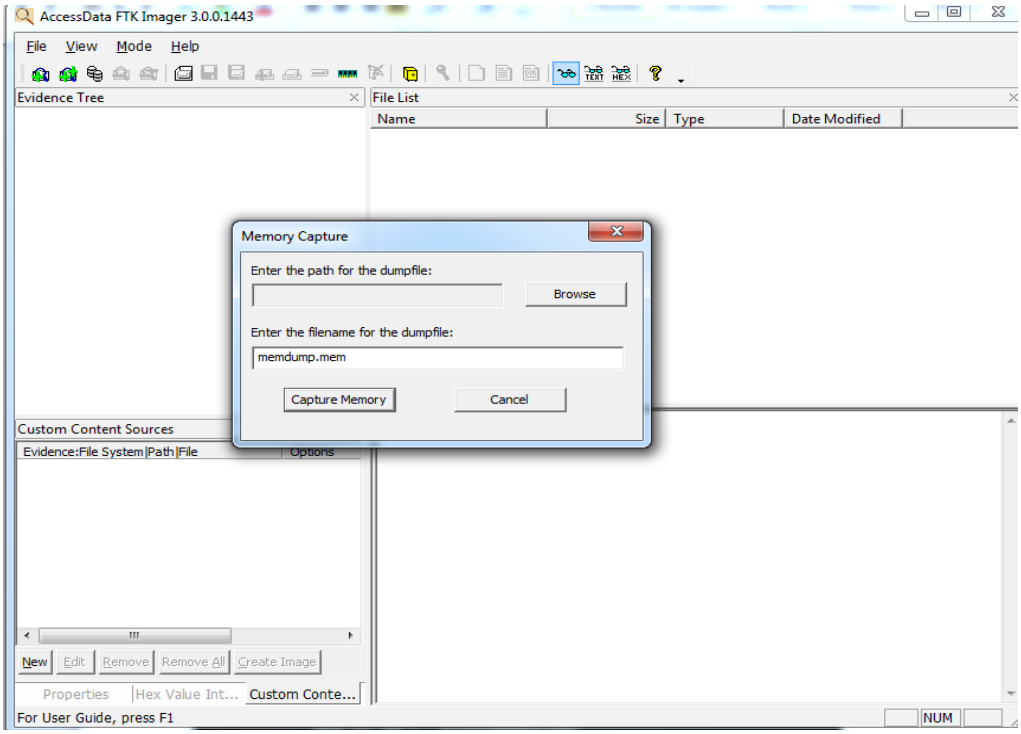


Figura 4-15: Ejecución de FTK Imager para extracción de dump de memoria

En una nueva ventana se muestra un menú desplegable, en el cual se selecciona la Unidad Fuente correspondiente, para luego hacer clic en el botón "Finish" o Finalizar.

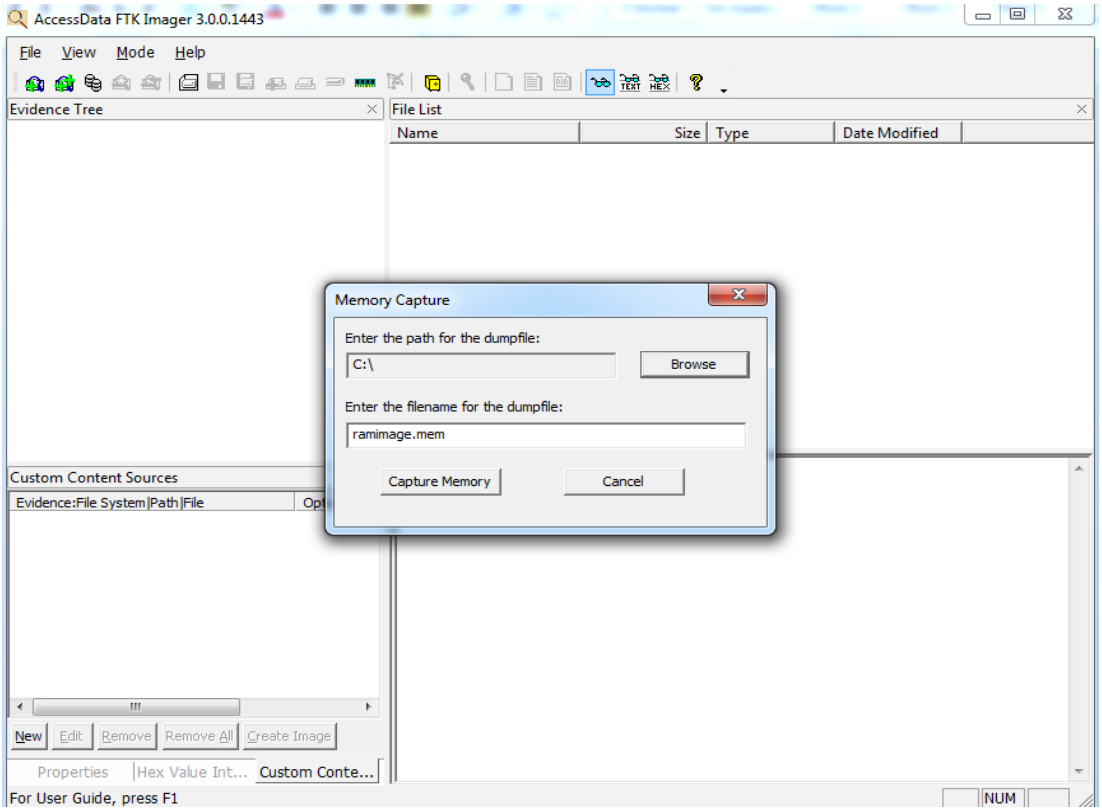


Figura 4-16: Ejecución de FTK Imager para extracción de dump de memoria

Debemos una vez que tengamos el dump de memoria pasarlo a nuestra máquina virtual de kali Linux ejecutar el siguiente comando ver imágenes a continuación del proceso.

Imagen extraída de la memoria RAM, la imagen se encuentra en el escritorio y se llama ramimage.men

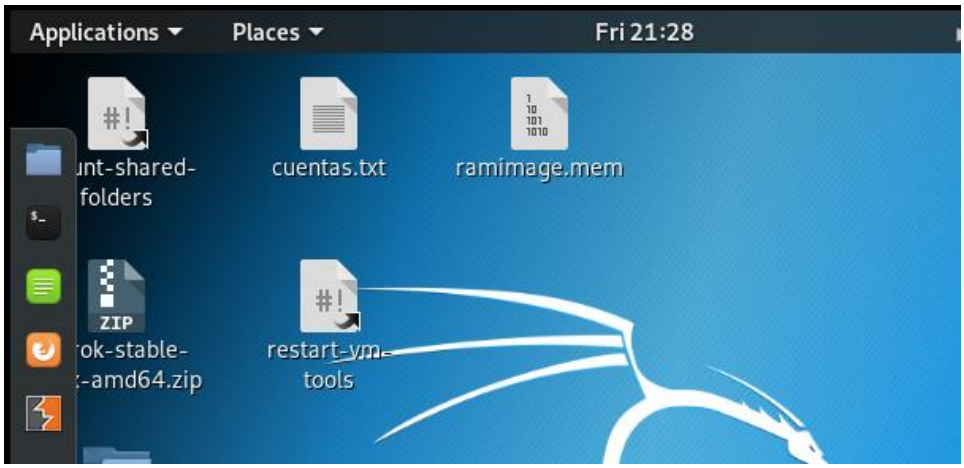


Figura 4-20: Imagen ramimage.men en kali Linux

Luego debemos de utilizar la herramienta volatility que ya está integrada en kali Linux, y el comando siguiente

Para este caso tenemos la imagen de la memoria RAN en el escritorio por lo que mediante consola debemos ubicarnos dentro de esa ruta mediante el comando **cd Desktop/** tal como muestra en pantalla.

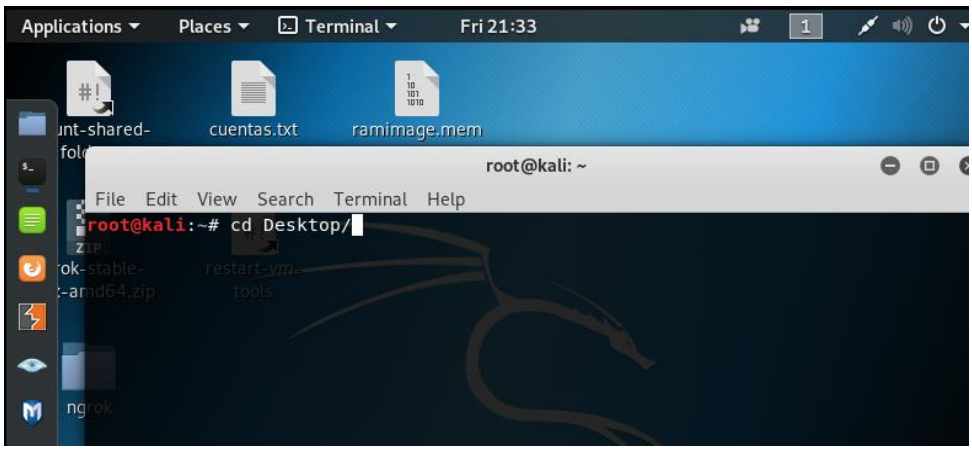


Figura 4-21: accesos a la ruta de la Imagen ramimage.men en kali Linux

Luego con el comando **ls** listamos los ficheros y ubicamos nuestra imagen forense de la memoria RAN extraída, para este caso tiene el nombre de **ramimage.mem**

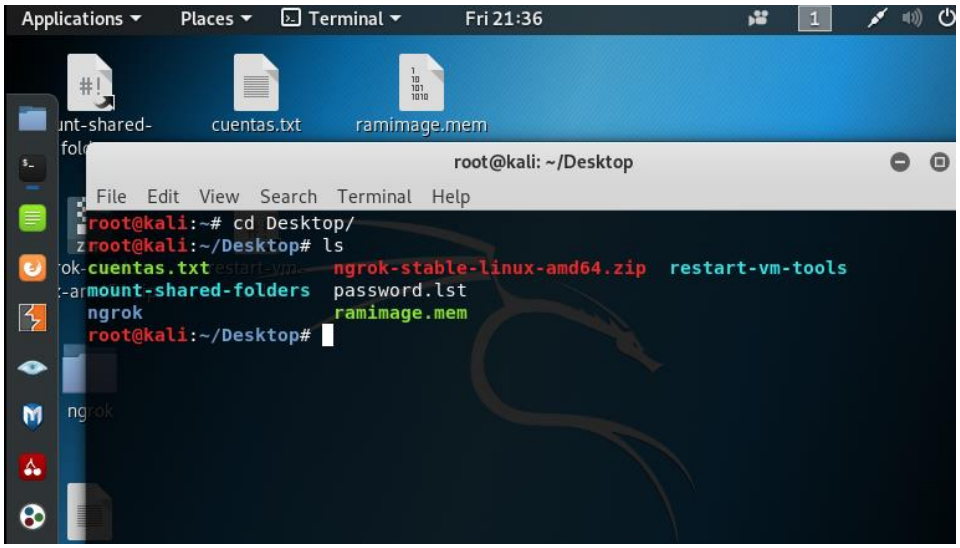


Figura 4-22: Acceso a la Imagen ramimage.men en kali Linux

Como podemos apreciar estamos dentro de la ruta donde tenemos nuestra **imagen forense** extraída de la **memoria RAM**

La herramienta que vamos a utilizar se llama volatility. Volatility es una herramienta forense de código abierto para la respuesta a incidentes y el análisis de malware. Está escrito en Python y es compatible con Microsoft Windows, Mac OS X y Linux

Volatility framework es una completa colección de herramientas open, escrita en Python bajo licencia GNU, para el análisis de la memoria volátil (RAM). Tiene como objetivo introducir a las personas en las complejas técnicas de extracción de artefactos digitales de imágenes de memoria volátil (RAM), y proveer una plataforma para futuro trabajo dentro del área de la investigación.

Con el comando **volatility --help** Vista la ayuda del comando vamos a ver ahora diferentes comandos que nos ayudarán a obtener información del dump así como podría ser el sistema operativo del cual se ha realizado, los procesos que corren en la misma, etc. Tal como se muestra en la figura de abajo.

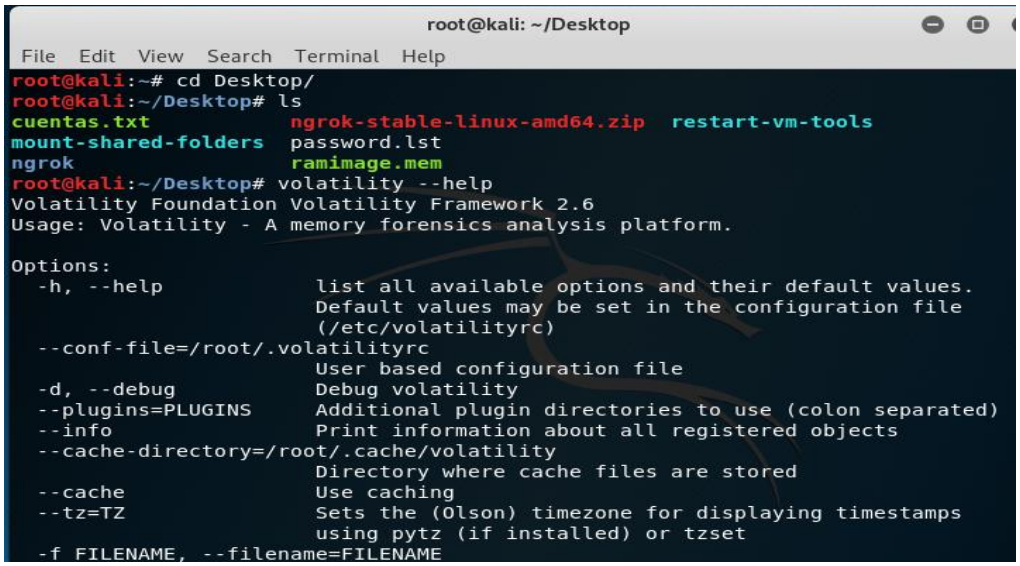


Figura 4-23: Revisión de ayuda de volatility

Identificación de imágenes

Imageinfo

Una de las opciones más sencillas y el primer paso en la mayoría de situaciones es descubrir de qué sistema operativo se ha realizado el dump sobre el cual vamos a realizar el análisis, para ello utilizaremos el módulo imageinfo el cual nos indicará esta información. El comando para lograr esto sería el siguiente: Mediante el comando **volatility imageinfo -f ramimage.mem**

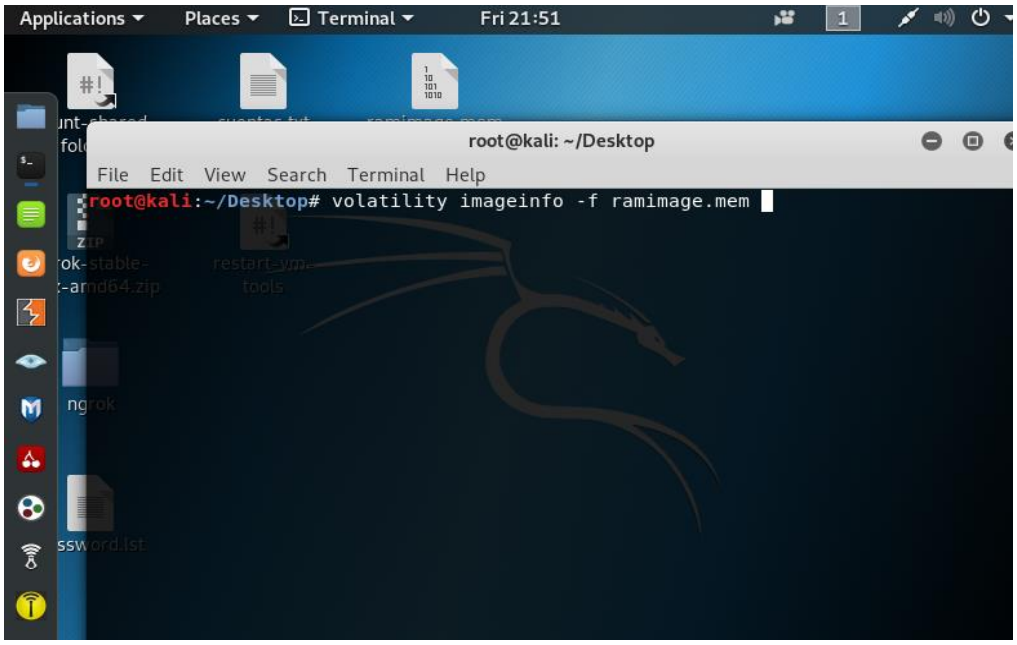


Figura 4-24: Extracción de la imagen de memoria RAM mediante volatility

Resultado que muestra kali de la extracción de la imagen forense de la memoria RAM, mediante la ejecución del comando **volatility imageinfo -f ramimage.mem.** Y que podemos apreciar en la siguiente captura de pantalla donde lo hemos utilizado para conocer el sistema de un dump de memoria de Windows:

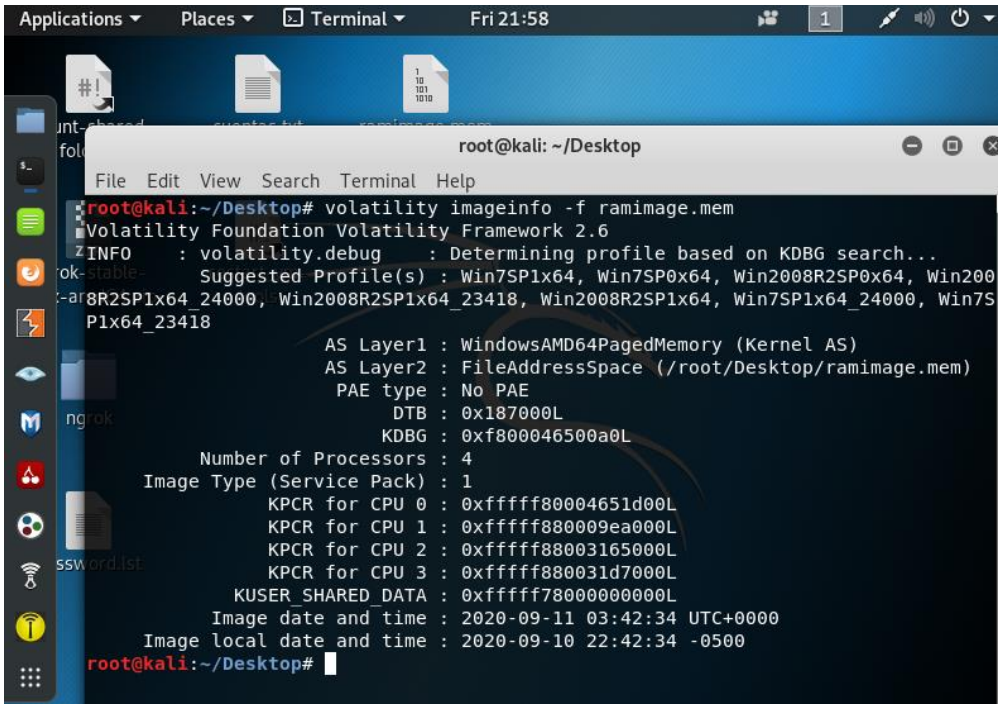


Figura 4-25: Extracción de la imagen de memoria RAM mediante volatility

Como vemos en el ejemplo anterior, al realizar la función de **imageinfo**, este nos ha sugerido varios perfiles a los cuales puede pertenecer el dump, que en este caso pertenece a un sistema Win7SP1x64, Win7SP0x64, Win2008R2SP0x64, Win2008R2SP1x64_24000, Win2008R2SP1x64_23418, Win2008R2SP1x64, Win7SP1x64_24000, Win7SP1x64_23418 Windows y cuya arquitectura es x64.

Conocido el perfil, ahora necesitaremos indicarle el mismo para realizar un análisis más exhaustivo con alguno de los módulos que tengamos disponibles, para indicar el mismo utilizaríamos la opción - - **profile** como vemos a continuación:

De la misma manera podemos utilizar el comando **volatility -f ramimage.mem imageinfo** y se obtendrá el siguiente resultado.

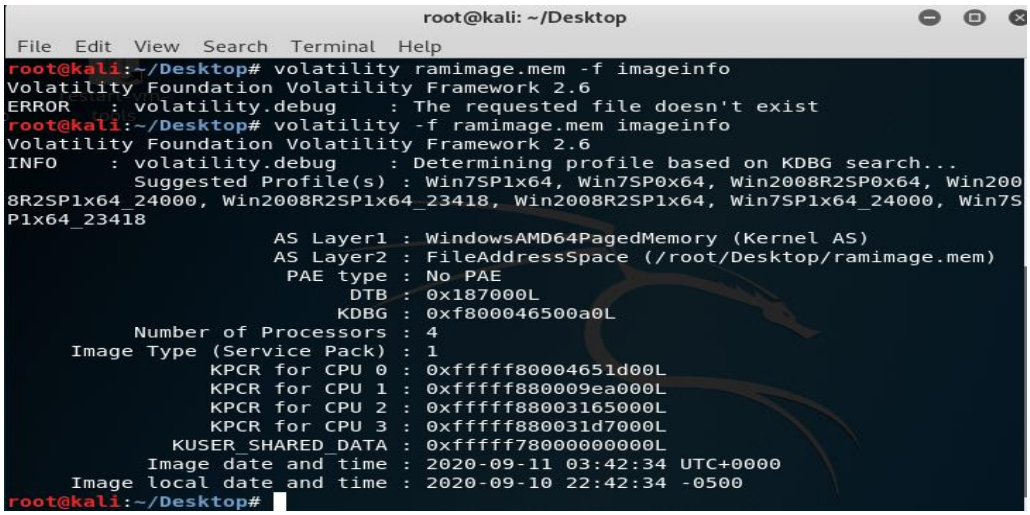


Figura 4-26: Extracción de la imagen de memoria RAM mediante volatility

Resultado de la ejecución del comando **volatility -f ramimage.mem imageinfo** tal como muestra en la captura anterior, cabe indicar que usted puede utilizar ambos comandos, pero obtendrá el mismo resultado.

Conocido el perfil, ahora necesitaremos indicarle el mismo para realizar un análisis más exhaustivo con alguno de los módulos que tengamos disponibles, para indicar el mismo utilizaríamos la opción - -profile como vemos a continuación:

Profile identificado **Win7SP1x64.**

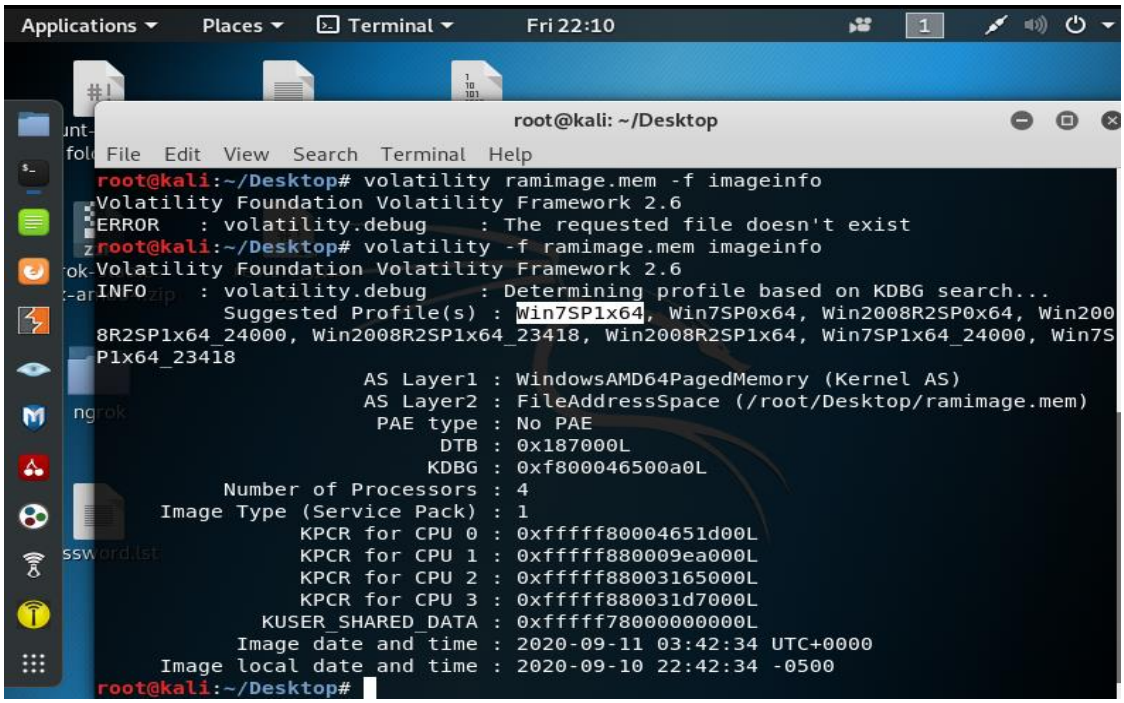


Figura 4-27: Extracción de la imagen de memoria RAM mediante volatility

Una vez identificado el profile debemos de ejecutar un comando más personalizado para conocer con más detalle los diferentes procesos ejecutados dentro de la memoria RAM y de la extracción de la imagen forense.

Kdbgscan

A diferencia de imageinfo, que nos ofrece una serie de perfiles sugeridos, kdbgscan nos ofrece un perfil exacto de la imagen analizada así como el identificador exacto de KDBG. Este plugin escanea las firmas de las cabeceras de la imagen y las contrarresta con los perfiles almacenados en la base de datos de volatility, reduciendo con ello el número de posibles falsos positivos.

Podemos ver un ejemplo en la siguiente imagen donde podemos observar que nos saca el perfil exacto, dando más información de la obtenida anteriormente con imageinfo

Debemos de ejecutar el comando **volatility -f ramimage.mem --profile=Win7SP1x64 kdbgscan** tal como muestra la imagen.

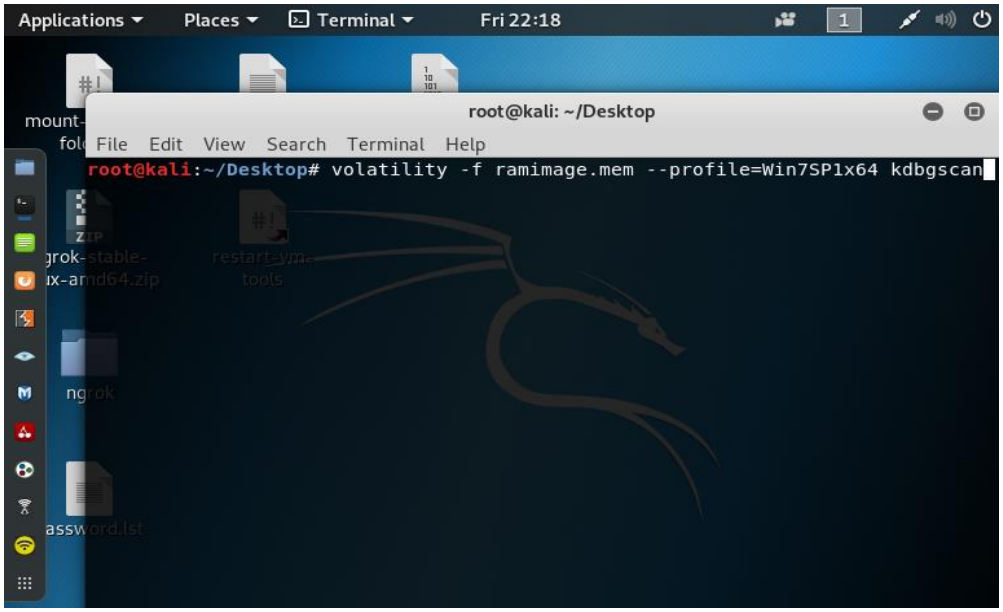


Figura 4-28: Extracción del profile de imagen de memoria RAM mediante volatility

Resultado de la ejecución del comando anterior **volatility -f ramimage.mem --profile= Win7SP1x64 kdbgscan**

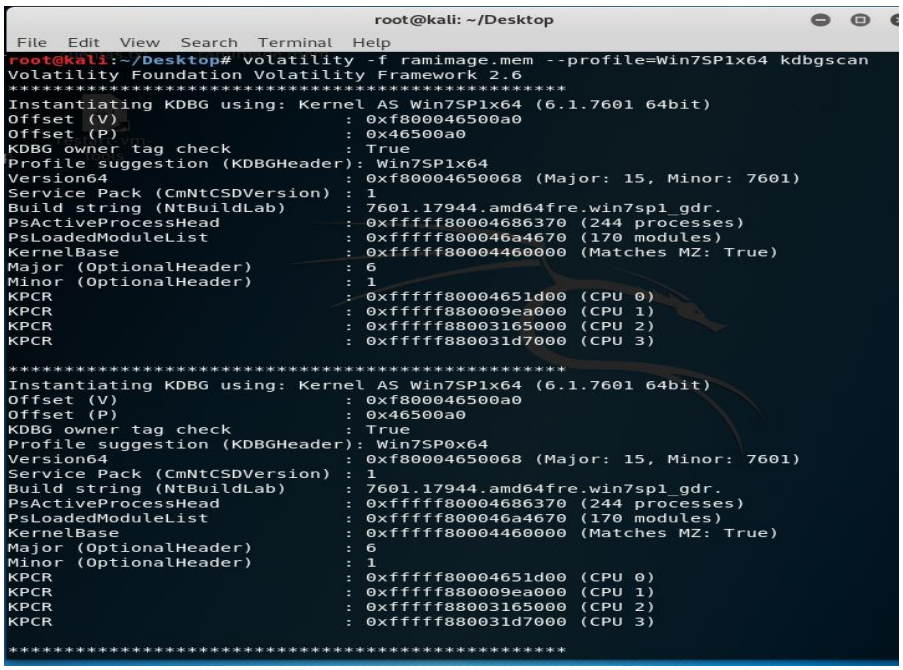


Figura 4-29: Extracción del profile de imagen de memoria RAM mediante volatility

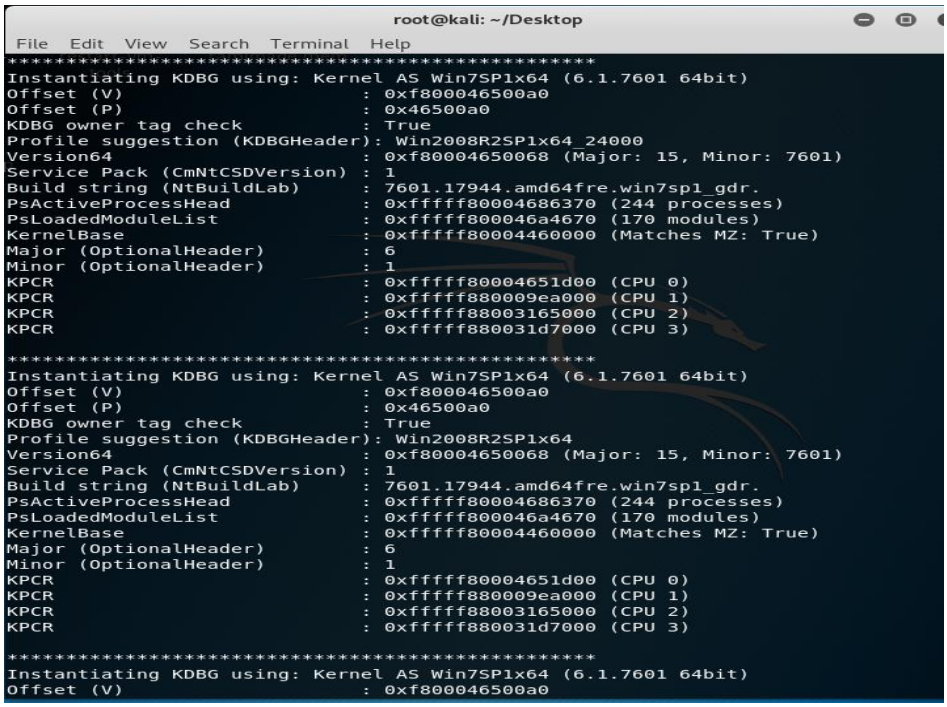


Figura 4-30: Extracción del profile de imagen de memoria RAM mediante volatility

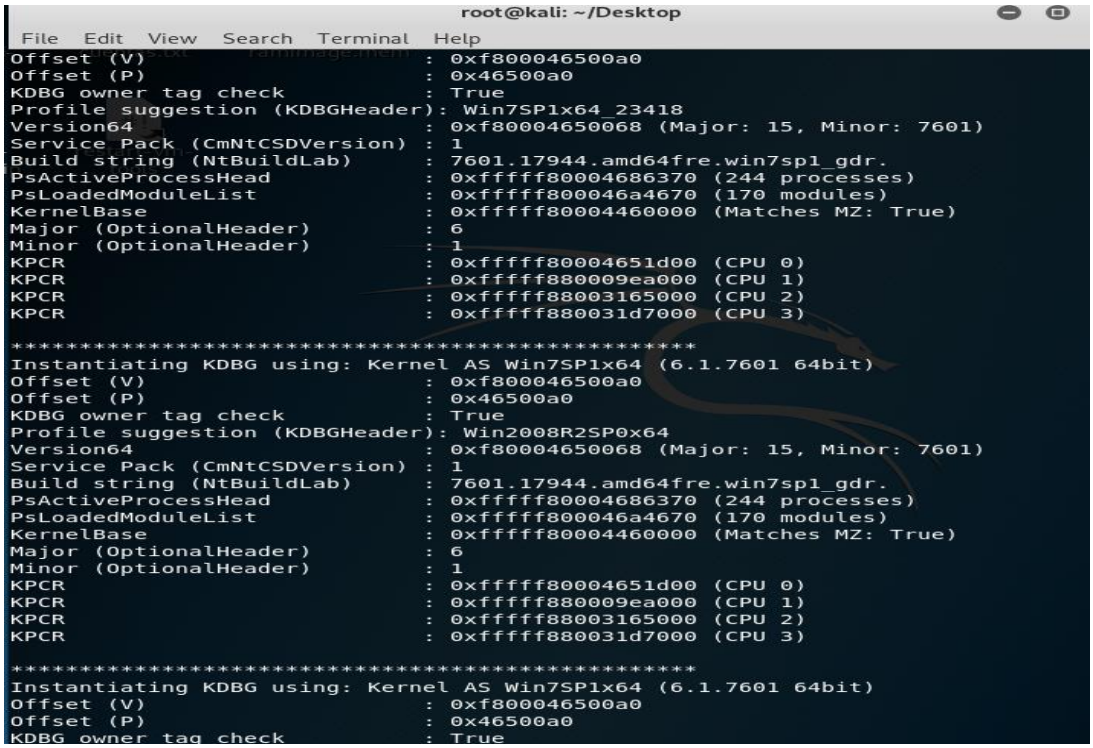


Figura 4-31: Extracción del profile de imagen de memoria RAM mediante volatility

Kpcrscan

Se puede utilizar este comando para buscar posibles estructuras de KPCR comprobando los miembros con autorreferencia. En un sistema multinúcleo, cada procesador tiene su propio KPCR. Por lo tanto, se podrán ver detalles para cada procesador, incluidas las direcciones IDT y GDT; hilos actuales, inactivos y siguientes, número de CPUs, proveedor y velocidad; y valor CR3.

El comando es el siguiente **volatility -f ramimage.mem --profile=Win7SP1x64 kpcrscan**

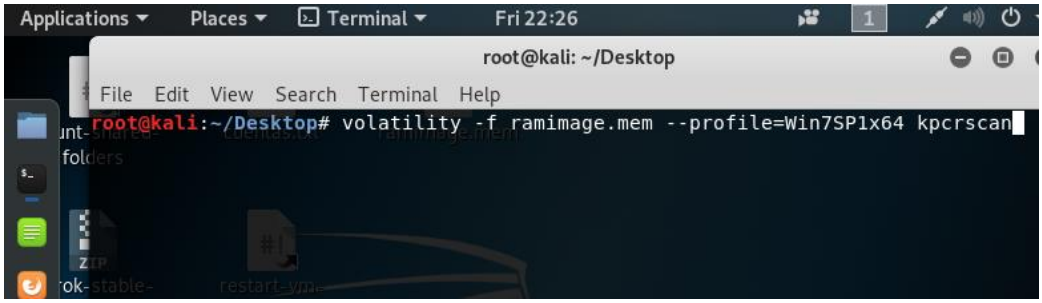


Figura 4-32: Ejecución del comando kpcrscan del profile de imagen de memoria RAM mediante volatility

Identificación de procesos

pslist

Para la enumeración de los procesos de un sistema, utilizaremos el plugin **pslist**. Este recorrerá la lista vinculada que apunta a PsActiveProcessHead y mostrará el desplazamiento, nombre del proceso, ID del proceso e ID del proceso padre del mismo, el número de subprocesos, número de identificadores, así como la fecha y hora en la cual dicho proceso empezó y terminó. A partir de la versión 2.1 muestra también el ID de sesión y si

dicho proceso es un Wow64, es decir, que utiliza un espacio de dirección de 32 bits en un sistema de 64 bits.

Hay que tener en cuenta también que los procesos System y smss.exe no dispondrán de un ID de sesión debido a que el sistema se inicia antes de que se establezcan dichas sesiones. Podemos ver un ejemplo del mismo a continuación:

Para realizar este proceso debemos de ejecutar el siguiente comando **volatility -f ramimage.mem --profile=Win7SP1x64 pslist**

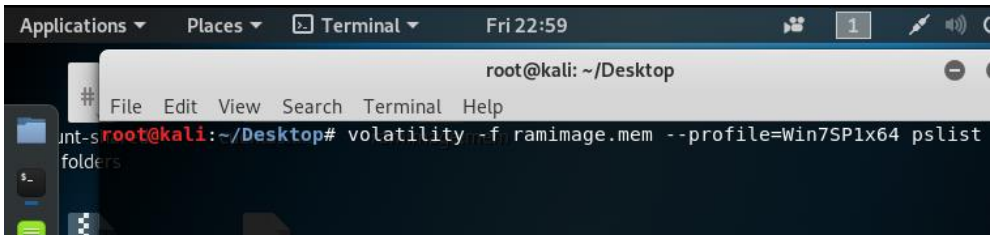


Figura 4-33: Ejecución del comando pslist del profile de imagen de memoria RAM mediante volatility

La ejecución del comando **volatility -f ramimage.mem --profile=Win7SP1x64 pslist nos muestra el siguiente resultado.**

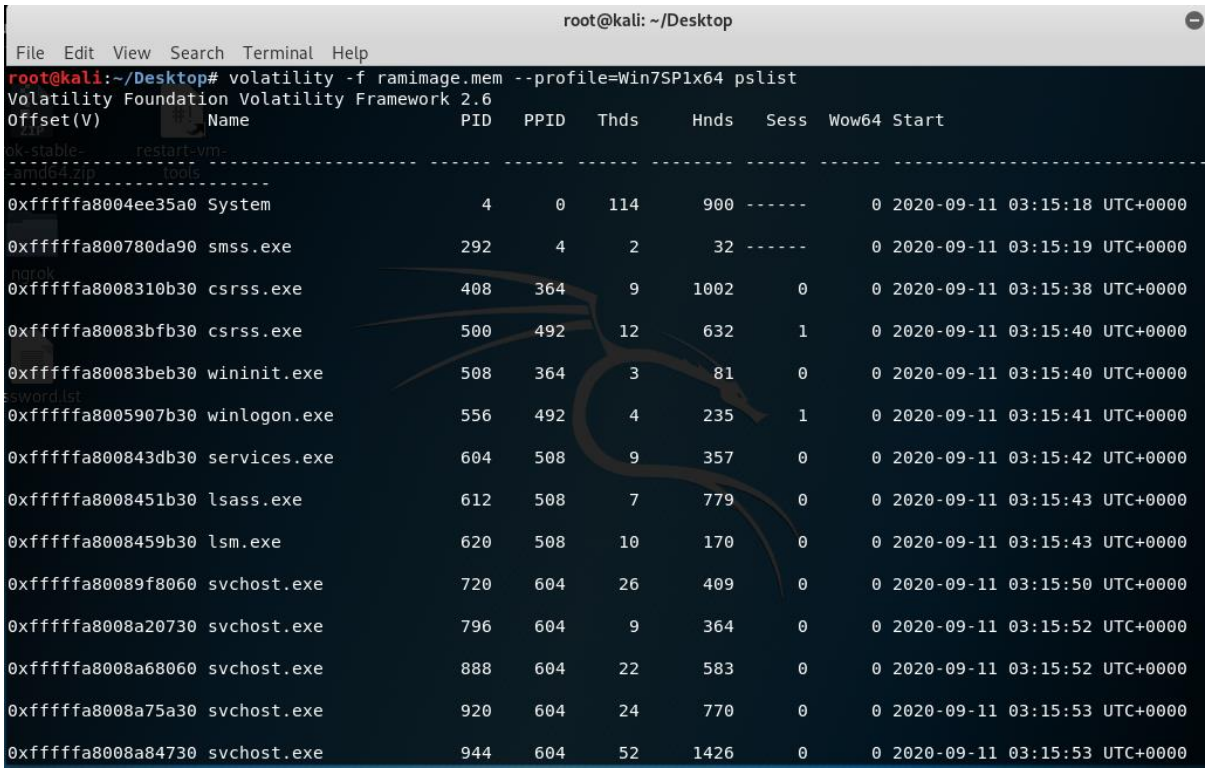


Figura 4-34: Ejecución del comando pslist del profile de imagen de memoria RAM mediante volatility

Pslist por defecto mostrará los offsets virtuales para _EPROCESS pero podemos ver los offsets físicos mediante la opción -P:

El comando que debemos ejecutar es el siguiente

volatility -f ramimage.mem --profile= Win7SP1x64 pslist –P

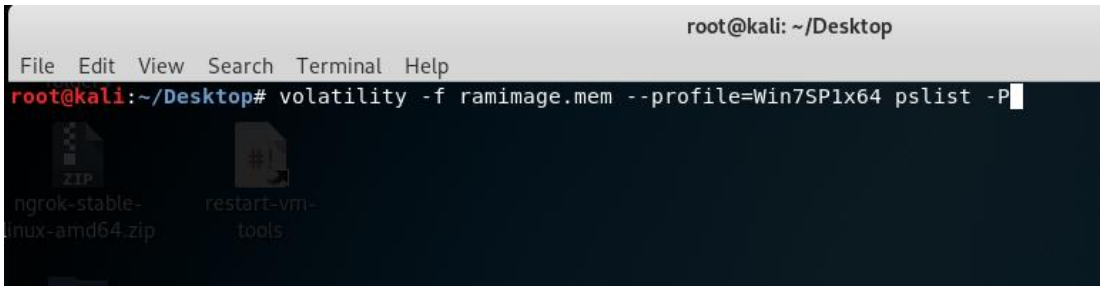


Figura 4-35: Ejecución del comando pslist del profile de imagen de memoria RAM mediante volatility

La ejecución del comando **volatility -f ramimage.mem --profile= Win7SP1x64 pslist –P** muestra los offsets físicos mediante la opción –P como se muestra en la imagen de abajo

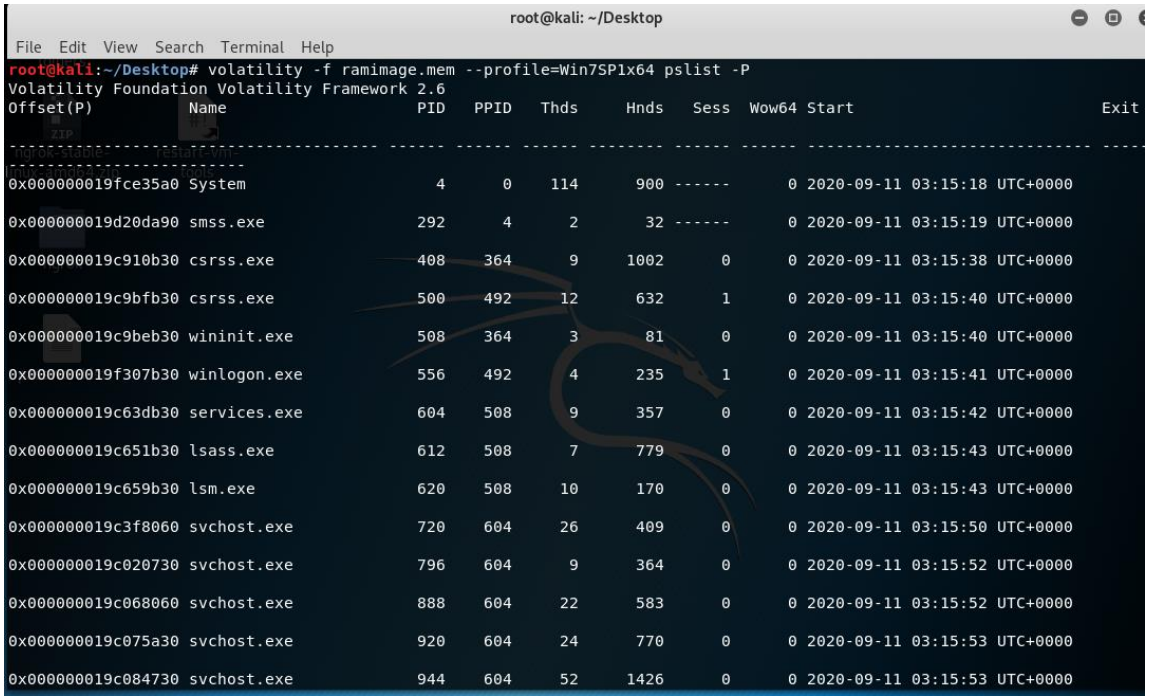


Figura 4-36: Ejecución del comando pslist del profile de imagen de memoria RAM mediante volatility

Resultado de la ejecución del comando **volatility -f ramimage.mem --profile= Win7SP1x64 pslist –P**

pstree

Podemos ver también esta lista de procesos en forma de árbol utilizando el plugin **pstree**, lo que enumerará los procesos con la misma técnica utilizada en **pslist,** pero con una vista diferente de la misma y que puede ayudar a identificar de una forma más rápida el padre de cada uno de estos.

El comando a ejecutar es el siguiente **volatility -f ramimage.mem --profile=Win7SP1x64 pstree**

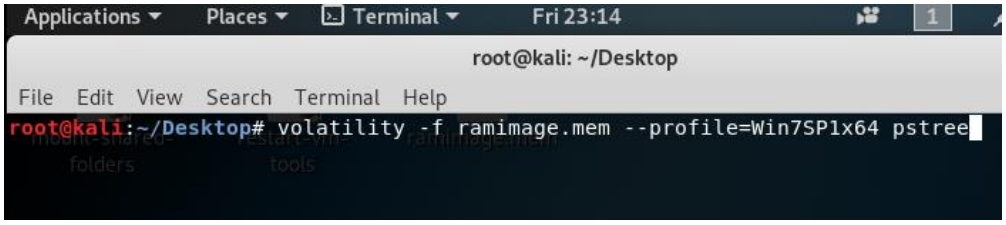


Figura 4-37: Ejecución del comando pstree del profile de imagen de memoria RAM mediante volatility

Resultado de la ejecución del comando **volatility -f ramimage.mem --profile=Win7SP1x64 pstree**

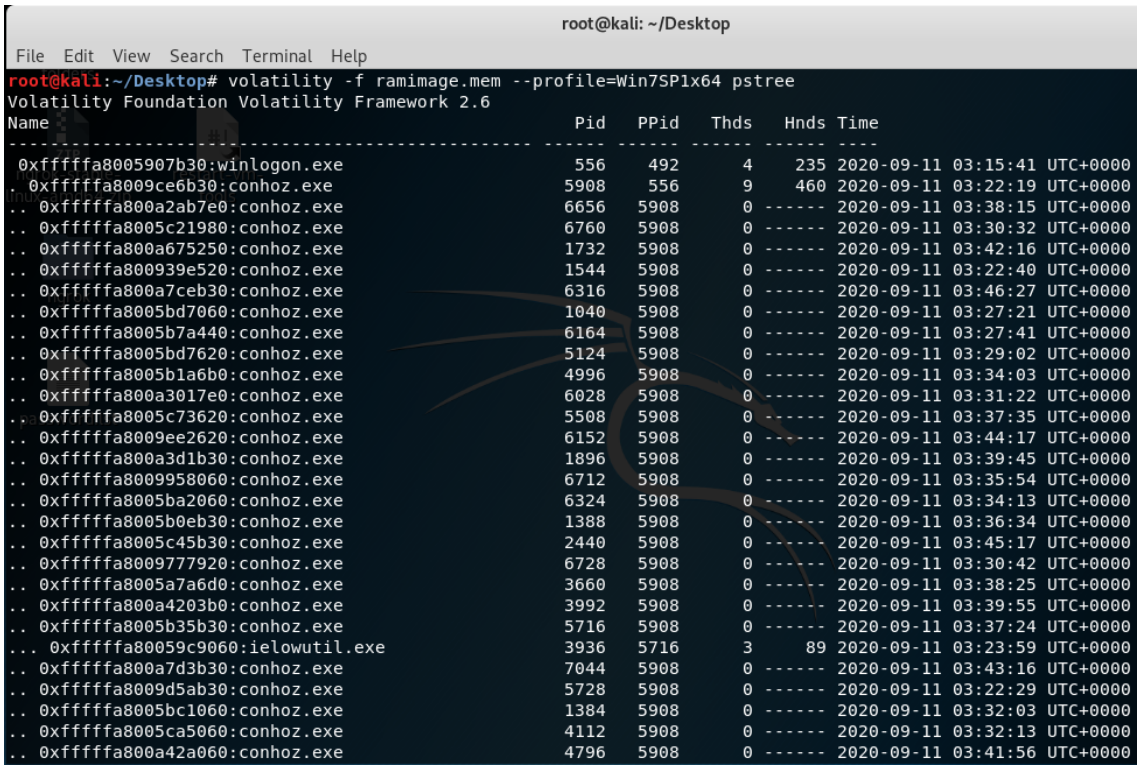


Figura 4-38: Ejecución del comando pstree del profile de imagen de memoria RAM mediante volatility

Como podemos ver ls lista de procesos en forma de árbol utilizando el plugin **pstree**, lo que enumerará los procesos con la misma técnica

utilizada en **pslist,** pero con una vista diferente de la misma y que puede ayudar a identificar de una forma más rápida el padre de cada uno de estos.

psscan

Para escanear los diferentes procesos existentes en la imagen utilizamos el **plugin psscan** que nos permitirá obtener también el **ID del offset físico** de cada uno de los procesos. Este plugin puede encontrar procesos inactivos, es decir, que terminaron previamente, y procesos que han sido ocultos o desvinculados por un **rootkig**.

El comando es el siguiente **volatility -f ramimage.mem --profile=Win7SP1x64 psscan**

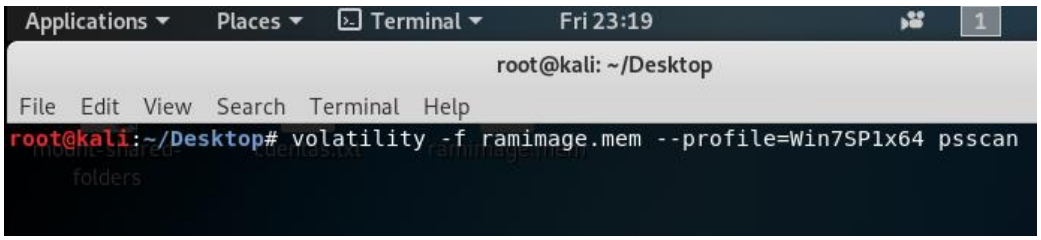


Figura 4-39: Ejecución del comando psscan del profile de imagen de memoria RAM mediante volatility

Resultado de la ejecución del comando **volatility -f ramimage.mem --profile=Win7SP1x64 psscan**

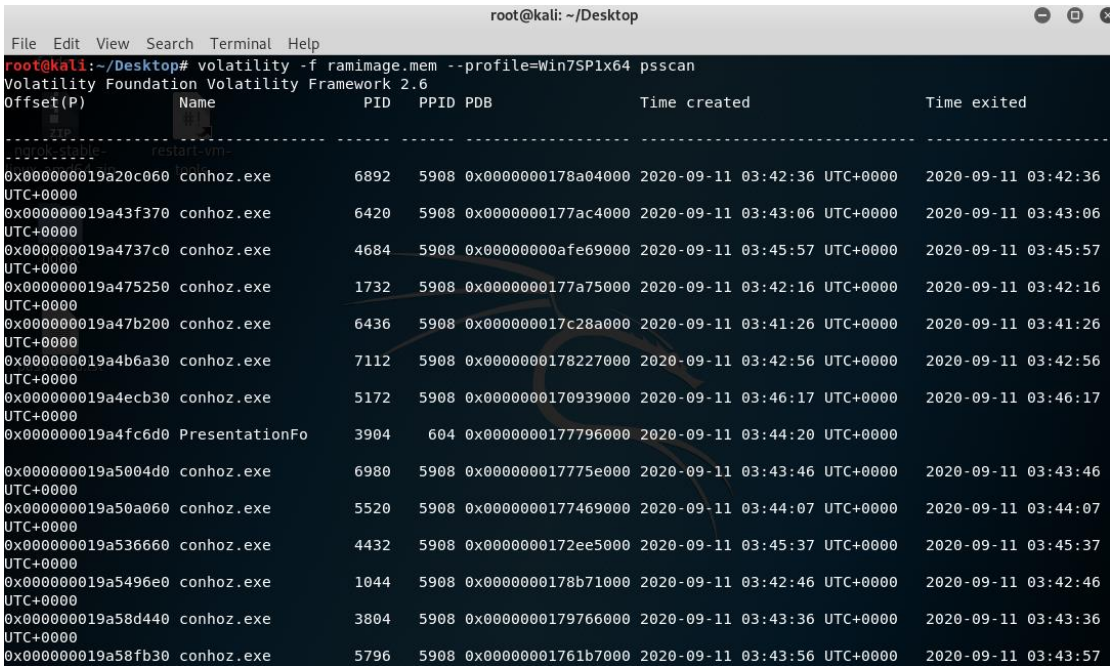


Figura 4-40: Ejecución del comando psscan del profile de imagen de memoria RAM mediante volatility

Resultado de la ejecución de procesos inactivos mediante el comando **volatility -f ramimage.mem --profile=Win7SP1x64 psscan**

Si un proceso ha finalizado previamente, el campo Hora de salida mostrará la hora de salida. Si desea investigar un proceso oculto (como mostrar sus DLL), necesitará un desplazamiento físico del objeto _EPROCESS, que se muestra en la primera columna. Casi todos los complementos relacionados con el proceso tomarán un parámetro offset para que pueda trabajar con procesos ocultos.

dlllist

El **plugin dlllist** nos permite ver que librerías del sistema está utilizando el mismo, así como obtener el offset de las mismas, el tamaño que ocupan, ruta en la cual se almacenan, etc.

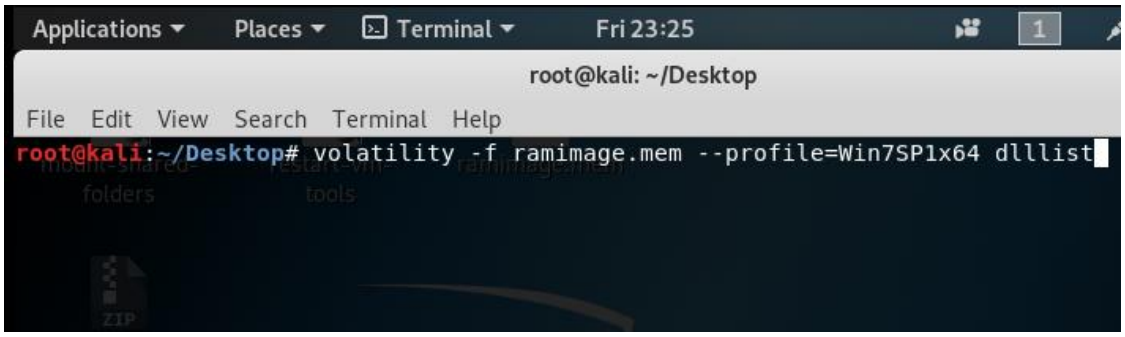


Figura 4-41: Ejecución del comando dlllist del profile de imagen de memoria RAM mediante volatility

Resultado de la ejecución del comando **volatility -f ramimage.mem --profile=Win7SP1x64 dlllist**

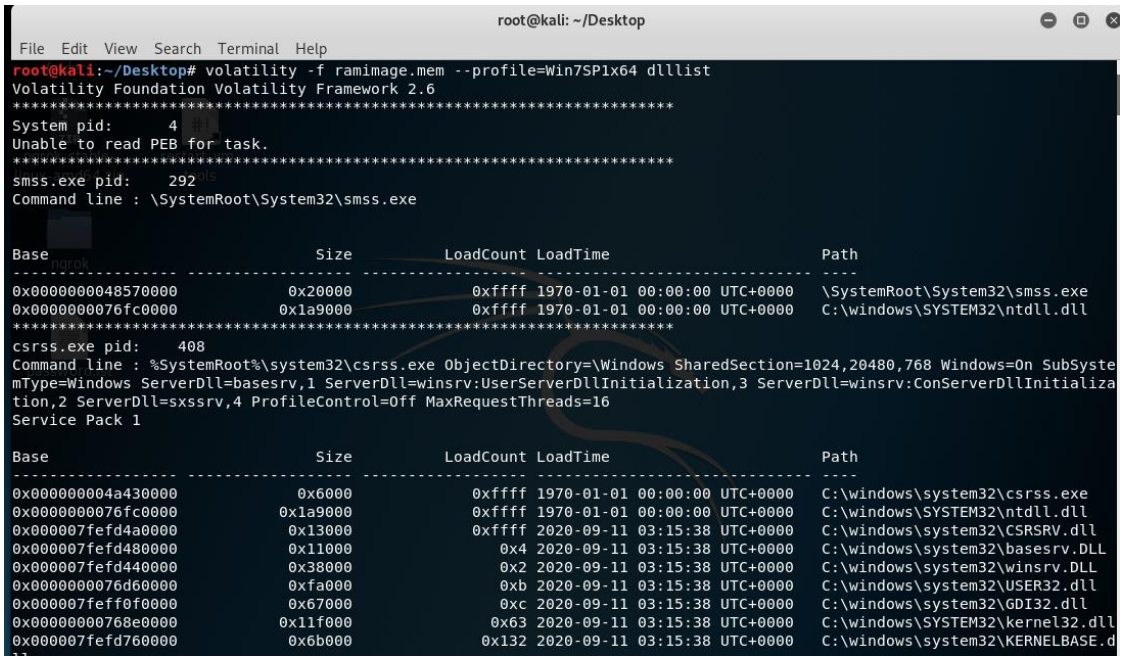


Figura 4-42: Ejecución del comando dlllist del profile de imagen de memoria RAM mediante volatility

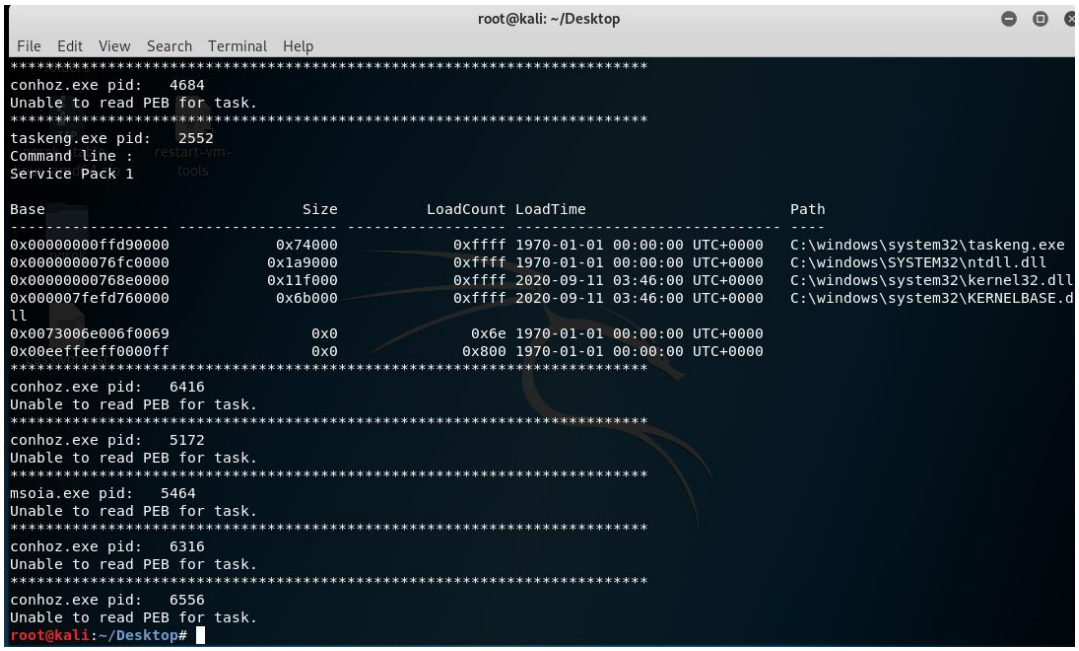


Figura 4-43: Ejecución del comando dlllist del profile de imagen de memoria RAM mediante volatility

Es posible además especificar el ID del proceso sobre el cual queremos realizar este análisis, para ello podemos utilizar la opción -p o –pid como vemos en el siguiente ejemplo:

El comando es el siguiente **volatility -f ramimage.mem --profile=Win7SP1x64 dlllist -p 6416**

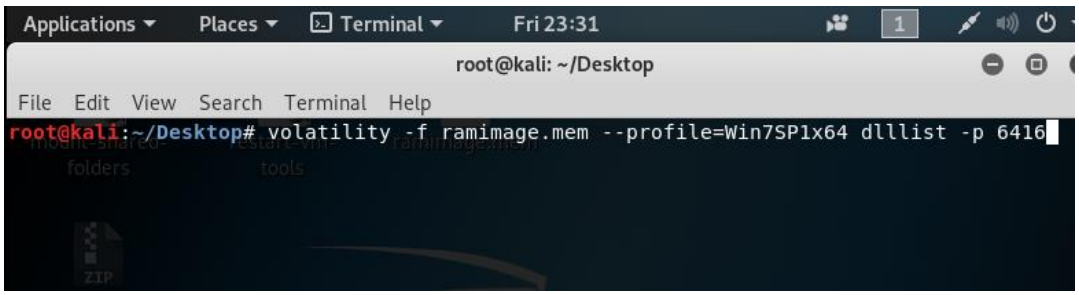


Figura 4-44: Ejecución del comando dlllist del profile de imagen de memoria RAM mediante volatility

Resultado de ña ejecución del comando **volatility -f ramimage.mem --profile=Win7SP1x64 dlllist -p 500**

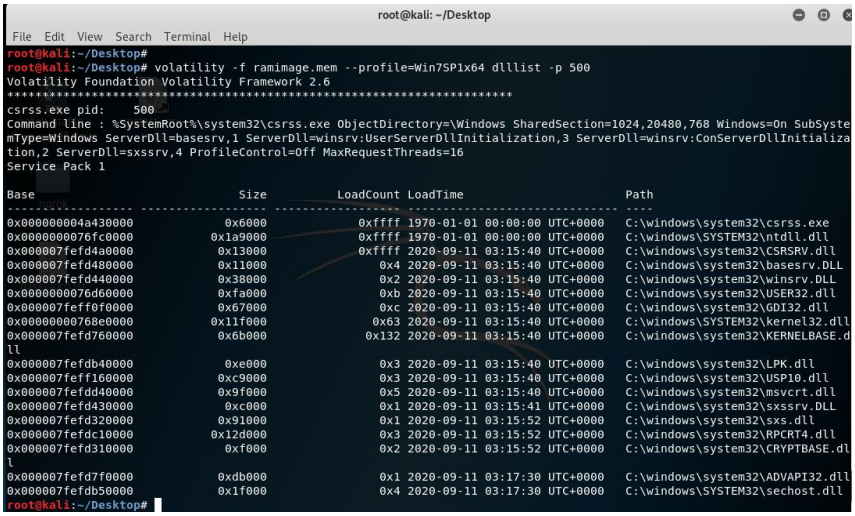


Figura 4-45: Ejecución del comando dlllist del profile de imagen de memoria RAM mediante volatility

Resultado de haber extraído la información de un proceso especifico

dlldump

Para extraer un DLL desde un proceso del espacio de memoria y analizar el mismo podemos utilizar el comando **dlldump**. Este comando dispone además de los siguientes parámetros:

Hacer un dump de todos los DLL de todos los procesos

Hacer un dump de todos los DLL de un determinado proceso con la opción -p pid o –pid=pid

Hacer un dump de los DLL desde un proceso oculto con –offset=OFFSET

Hacer un dump de un PE desde cualquier lugar de la memoria del proceso con –base=BASEDIR, esta opción es útil para extraer archivos DLL ocultos

Hacer un dump de uno o más DLL que coincidan con una expresión regular con la opción –regex=REGEX. También se puede especificar si es case sensitive o no con la opción –ignore-cas

Para especificar un directorio de salida se utilizaría la opción –dump-dir=DIR o -D DIR.

Debemos de ejecutar el siguiente comando **volatility -f ramimage.mem --profile=Win7SP1x64 dlldump -D dlls**

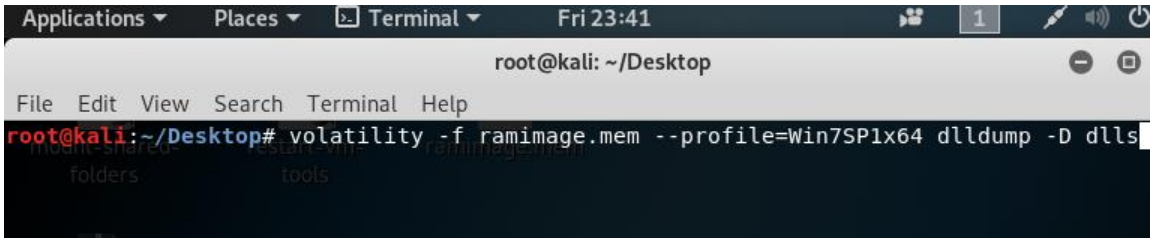


Figura 4-46: Ejecución del comando D dlls del profile de imagen de memoria RAM mediante volatility

cmdscan

El complemento cmdscan busca en la memoria de csrss.exe en XP, 2003, Vista y 2008 y conhost.exe en Windows 7 los comandos que los atacantes ingresaron a través de una consola (cmd.exe). Este es uno de los comandos más poderosos que puede usar para obtener visibilidad de las acciones de los atacantes en un sistema víctima, ya sea que abran cmd.exe a través de una sesión RDP o que ingresen/elaboren un comando desde un backdoor en red.

Este complemento busca estructuras conocidas como COMMAND_HISTORY buscando un valor constante conocido (MaxHistory) y luego aplicando chequeos de saneo. Es importante tener en cuenta que el valor de MaxHistory se puede cambiar haciendo clic derecho en la esquina superior izquierda de una ventana cmd.exe y yendo

a Propiedades. El valor también se puede cambiar para todas las consolas abiertas por un usuario determinado modificando la clave de registro HKCU\Console\HistoryBufferSize. El valor predeterminado es 50 en los sistemas Windows, lo que significa que se guardan los 50 comandos más recientes. Puede ajustarlo si es necesario utilizando el parámetro –max_history=NUMBER.

Las estructuras utilizadas por este complemento no son públicas (es decir, Microsoft no produce PDB para ellas), por lo tanto, no están disponibles en WinDBG ni en ningún otro marco forense. Fueron modificados por Michael Ligh los binarios conhost.exe y winsrv.dll.

Además de los comandos ingresados en un shell, este complemento muestra:

El nombre del proceso de host de la consola (csrss.exe o conhost.exe)

El nombre de la aplicación que usa la consola (cualquier proceso que esté usando cmd.exe)

La ubicación de las memorias intermedias del historial de comandos, incluido el recuento actual de la memoria intermedia, el último comando agregado y el último comando mostrado

El proceso de la solicitud de manejo

Debido a la técnica de escaneo que utiliza este complemento, tiene la capacidad de encontrar comandos de las consolas activas y cerradas.

E comando es el siguiente **volatility -f ramimage.mem --profile=Win7SP1x64 cmdscan**

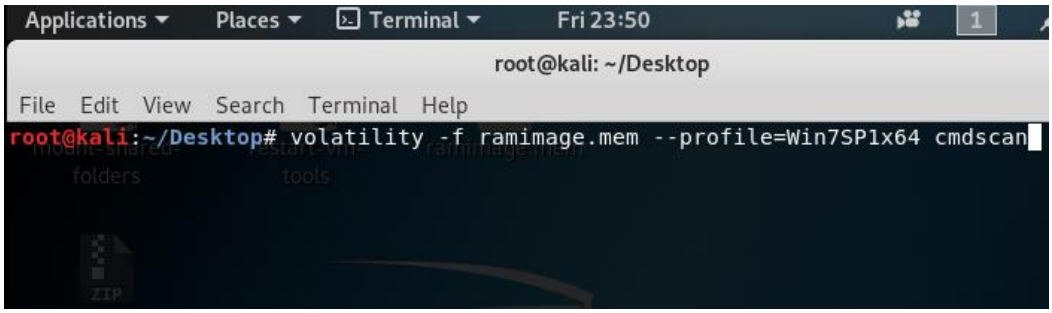


Figura 4-47: Ejecución del comando cmdscan del profile de imagen de memoria RAM mediante volatility

Resultado de la ejecución del comando **volatility -f ramimage.mem --profile=Win7SP1x64 cmdscan**

```
Volatility Foundation Volatility Framework 2.6
**************************************************
CommandProcess: csrss.exe Pid: 592
CommandHistory: 0x544d88 Application: cmd.exe Flags: Allocated, Reset
CommandCount: 2 LastAdded: 1 LastDisplayed: 1
FirstCommand: 0 CommandCountMax: 50
ProcessHandle: 0x410
Cmd #0 @ 0x541eb8: cd Escritorio
Cmd #1 @ 0x10f3338: get_info.bat
**************************************************
CommandProcess: csrss.exe Pid: 592
CommandHistory: 0x11299a0 Application: cmd.exe Flags: Allocated, Reset
CommandCount: 11 LastAdded: 10 LastDisplayed: 10
FirstCommand: 0 CommandCountMax: 50
ProcessHandle: 0x49c
Cmd #0 @ 0x548008: cd ..
Cmd #1 @ 0x541f90: cls
Cmd #2 @ 0x10ffc58: dir
Cmd #3 @ 0x548070: cd "Archivos de programa"
Cmd #4 @ 0x5480b0: dir
Cmd #5 @ 0x5480c0: cls
Cmd #6 @ 0x5480d0: cd MANDIANT
Cmd #7 @ 0x548150: dir
Cmd #8 @ 0x548160: cd Memoryze
Cmd #9 @ 0x548180: dir
Cmd #10 @ 0x10ffbf0: MemoryDD.bat
```

Figura 4-48: Ejecución del comando cmdscan del profile de imagen de memoria RAM mediante volatility

Resultados de la ejecución del comando anterior.

Memoria de procesos

memmap

El comando **memmap** muestra exactamente las páginas que residen en la memoria, dado un proceso específico DTB (o kernel DTB si usa este complemento en el proceso inactivo o del sistema). Muestra la dirección virtual de la página, el desplazamiento físico correspondiente y el tamaño de la misma. La información de mapa generada por este complemento proviene del método get_available_addresses del espacio de direcciones subyacente.

A partir de la versión 2.1, la nueva columna DumpFileOffset ayuda a correlacionar la salida de **memmap** con el archivo de volcado producido por el plugin memdump. Podemos ver un ejemplo de su uso a continuación:

La ejecución del comando es la siguiente **volatility -f ramimage.mem --profile=Win7SP1x64 memmap**

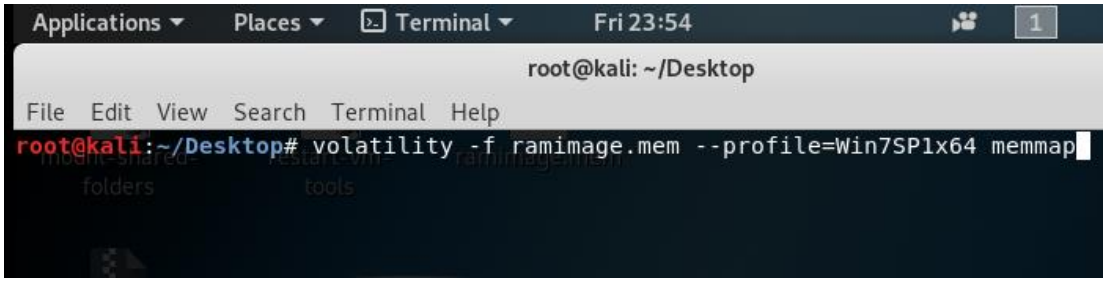


Figura 4-49: Ejecución del comando memmap del profile de imagen de memoria RAM mediante volatility

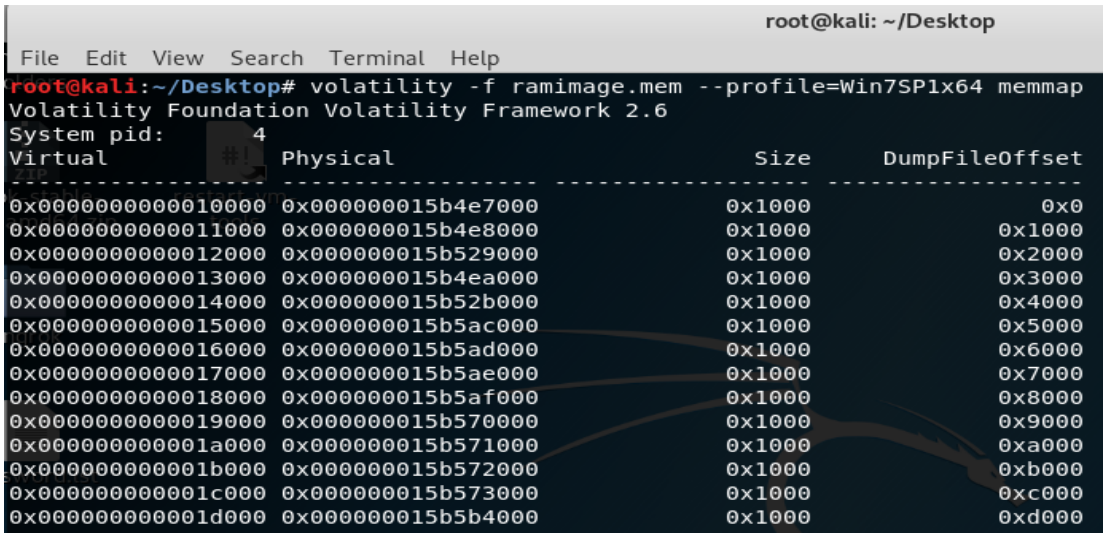


Figura 4-50: Ejecución del comando memmap del profile de imagen de memoria RAM mediante volatility

Memdump

Para extraer todas las páginas residentes en la memoria de un proceso en un archivo individual, existe el comando **memdump**. Se deberá indicar el directorio de salida con -D o –dump-dir = DIR como vemos en el siguiente ejemplo:

El commando es **volatility -f ramimage.mem --profile=Win7SP1x64 memdump -p 4 -D dump/**

procdump

El comando **procdump** se utiliza para extraer el ejecutable utilizado por un proceso. Opcionalmente se pueden añadir las opciones -u o –unsafe para omitir ciertas comprobaciones de saneo utilizadas para analizar el encabedo del mismo. Algunos programas maliciosos podrían falsicicar

intencionalmente los campos de tamaño del encabezado para provocar que herramientas de volcado de memoria fallasen.

Se puede utilizar también la opción –memory para indicar un espacio entre las secciones de PE que no están alineados con la página. Si no se indica este parámetro, el resultado será un arhivo que se parecerá más al archivado en el disco antes de que se expandan las secciones oportunas.

Para ver conexiones remotas en la maquina el comando es el siguiente

volatility --profile=Win7SP1x64 -f ramimage.mem connscan

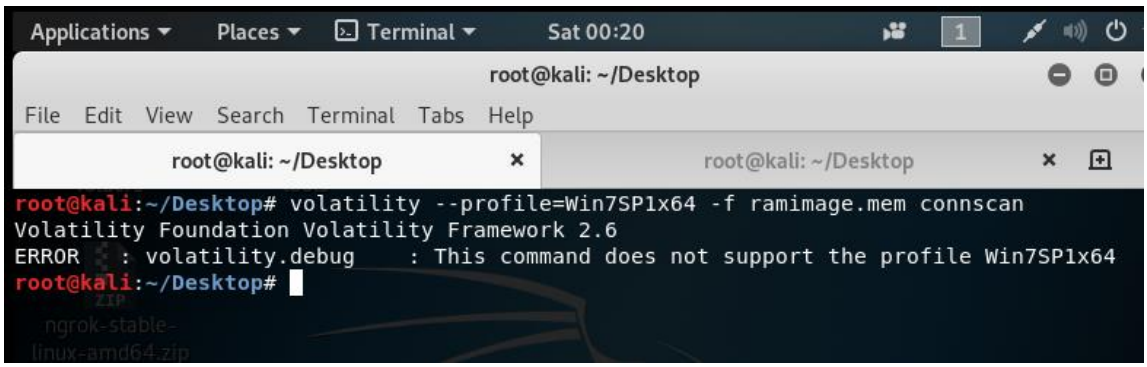


Figura 4-51: Ejecución del comando connscan del profile de imagen de memoria RAM mediante volatility

Para esta práctica no hay conexiones remotas a esa maquina

Para ver todos los hive de memoria de donde están cargados los registros utilizo el comando hivelist

La sintaxis del comando seria la siguiente

volatility -f ramimage.mem --profile=Win7SP1x64 hivelist

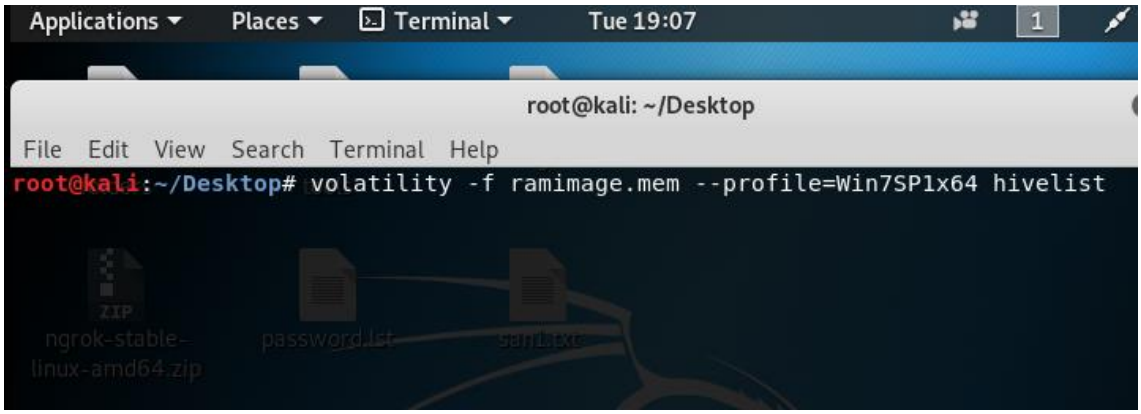


Figura 4-82: Ejecución del comando hivelist del profile de imagen de memoria RAM mediante volatility

Resultado de la ejecución del comando **volatility -f ramimage.mem --profile=Win7SP1x64 hivelist**

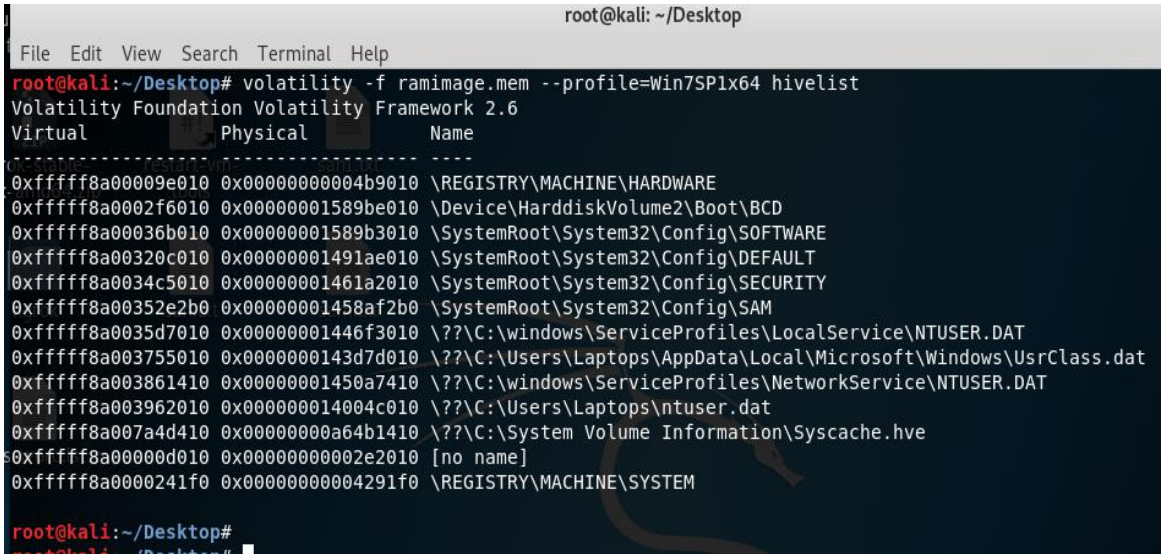


Figura 4-53: Ejecución del comando hivelist del profile de imagen de memoria RAM mediante volatility

Para este apartado debemos de identificar dos archivos muy importantes que son el SYSTEM y el SAM.

Esta información es muy importante ya que nos permite extraer las contraseñas del sistema operativo que se esté utilizando, con la extracción de la imagen forense de la memoria RAM, se puede utilizar cualquier herramienta de password cracking para reventar las contraseñas del sistema operativo.

Vamos a ver los números de procesos mediante el comando psscan, la sintaxis es la siguiente **volatility -f ramimage.mem --profile=Win7SP1x64 psscan**

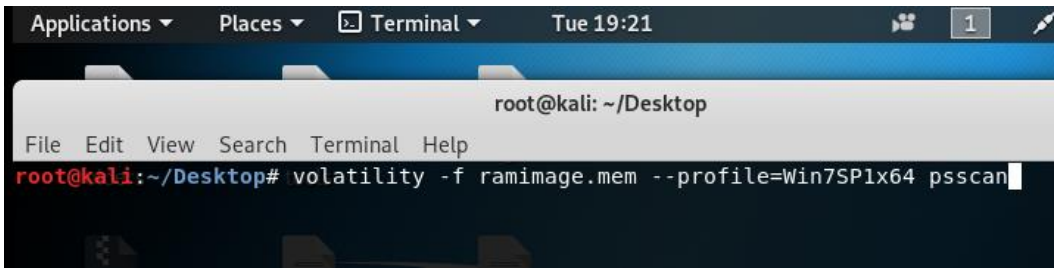


Figura 4-54: Ejecución del comando psscan del profile de imagen de memoria RAM mediante volatility

Resultado de la ejecicion del comando **volatility -f ramimage.mem --profile=Win7SP1x64 psscan**

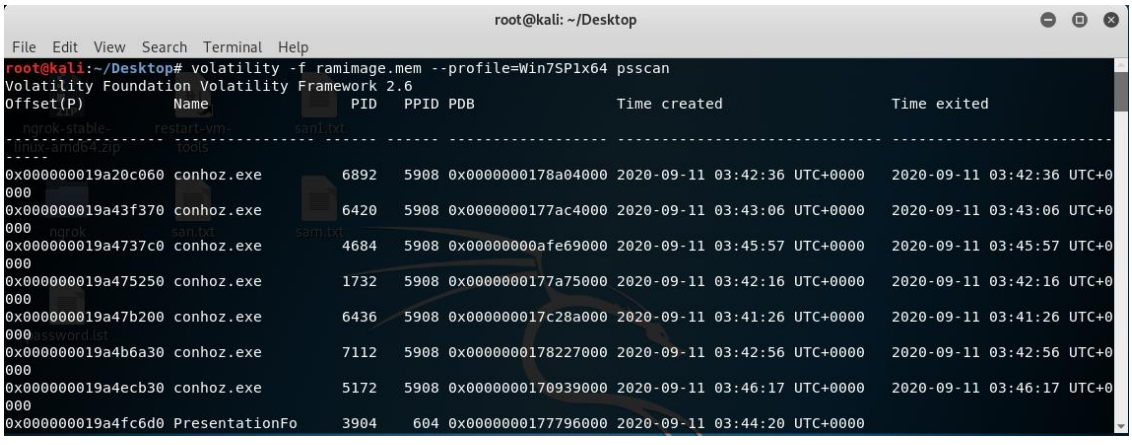


Figura 4-55: Ejecución del comando psscan del profile de imagen de memoria RAM mediante volatility

Luego debemos proceder a revisar las librerías asociadas a un proceso cualquiera, para esto debemos de utiliza el siguiente comando **dlllist,** volatility -f ramimage.mem --profile=Win7SP1x64 dlllist -p 556 donde el numero 556 es el identificador PID de un proceso, en este paso deberá colocar el proceso que va escanear.

volatility -f ramimage.mem --profile=Win7SP1x64 dlllist -p 556

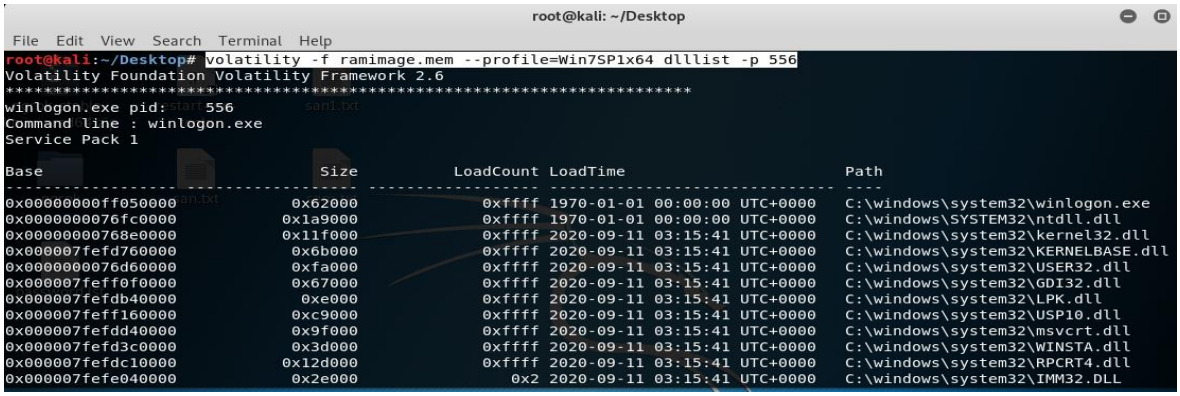


Figura 4-56: Ejecución del comando dlllist del profile de imagen de memoria RAM mediante volatility

Resultado de la ejecución del comando dlllist

Luego procedemos a revisar los identificadores mediante el comando **getsids**, donde podremos observar que usuarios están utilizando ese proceso, el comando es el siguiente **volatility -f ramimage.mem --profile=Win7SP1x64 getsids -p 556**

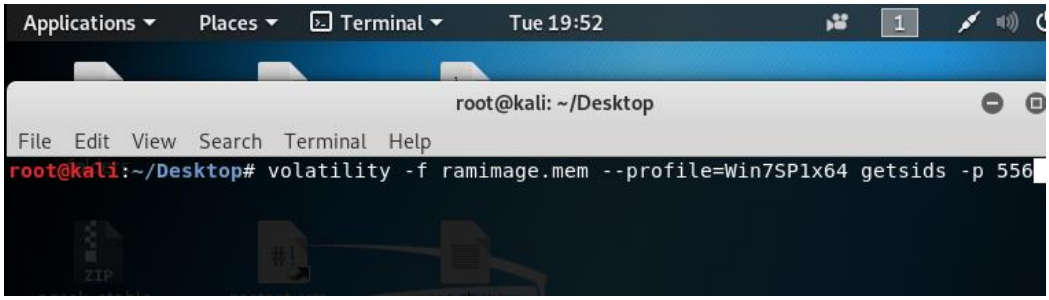


Figura 4-57: Ejecución del comando getsids del profile de imagen de memoria RAM mediante volatility

Resultado de la ejecución del comando anterior

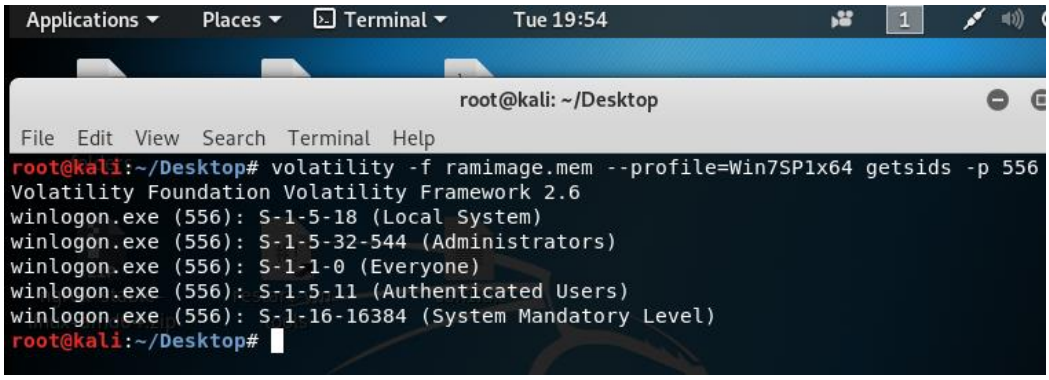


Figura 4-58: Ejecución del comando getsids del profile de imagen de memoria RAM mediante volatility

Como podemos apreciar en la imagen el proceso y los usuarios que están utilizando ese proceso. Este proceso se lo realiza para obtener los system definir y definir usuarios conocidos como identificadores. Esto es muy importante para el análisis de malware.

Como extraer la contraseña de la memora RAM

La forma de extraer la contraseña de la memoria RAM es muy simple debemos de ejecutar el siguiente comando, pero primero necesitamos ver donde están cargados en memoria RAM los espacios del registro para lo cual ejecutamos el siguiente comando

volatility -f ramimage.mem --profile=Win7SP1x64 hivescan

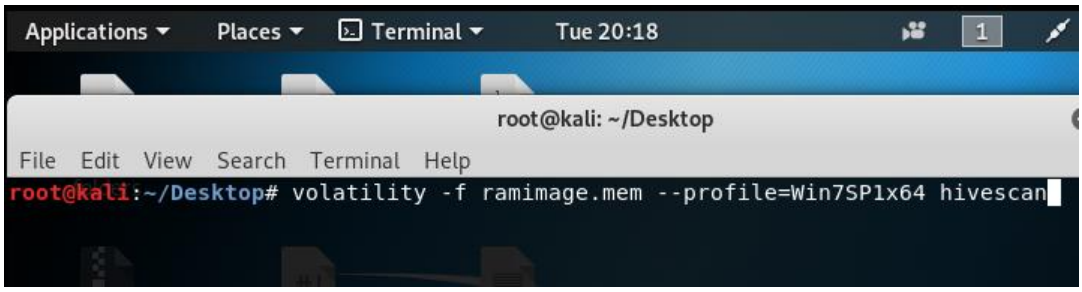


Figura 4-55: Ejecución del comando hivescan del profile de imagen de memoria RAM mediante volatility

Luego nos mostrara el siguiente Resultado

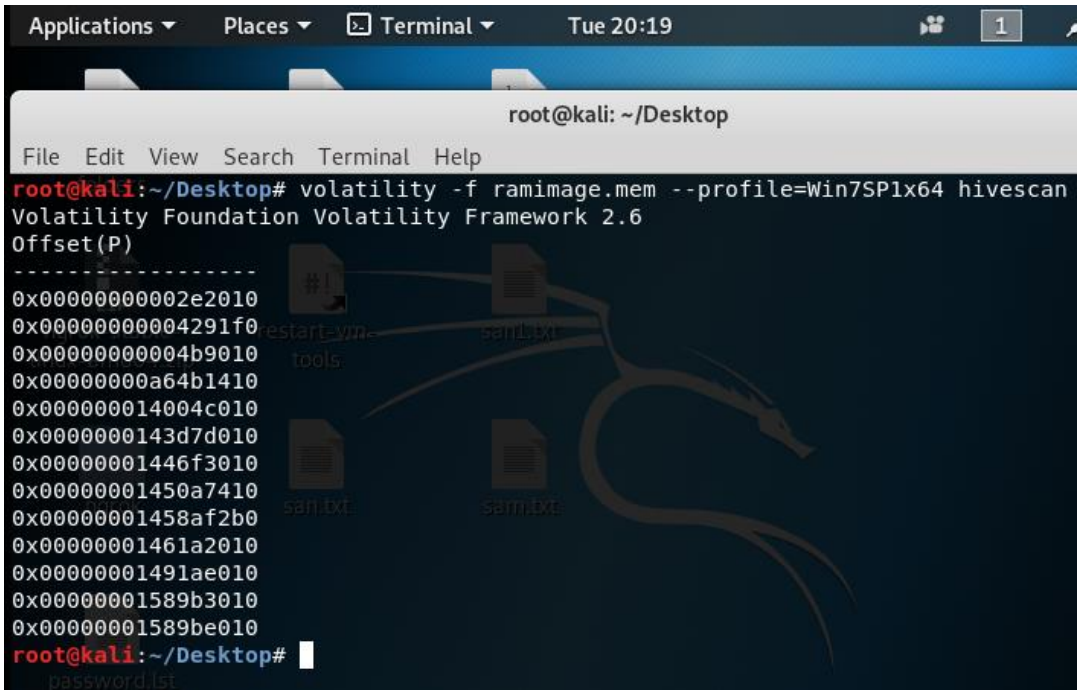


Figura 4-59: Ejecución del comando hivescan del profile de imagen de memoria RAM mediante volatility

Resultado de la ejecución del comando **volatility -f ramimage.mem --profile=Win7SP1x64 hivescan**

En este punto ya sabemos cuáles son los **Offset** donde están los registros cargados, ahora procedemos a mirar dentro de cada **Offset** que existe. Para aquello debemos de ejecutar el siguiente comando hivelist con cualquiera de los Offset cargados anteriormente 0x0000000143d7d010 ejemplo.

volatility -f ramimage.mem --profile=Win7SP1x64 hivelist 0x0000000143d7d010

Cabe indicar que podemos escoger cualquiera de los Offset y daría resultados muy similares

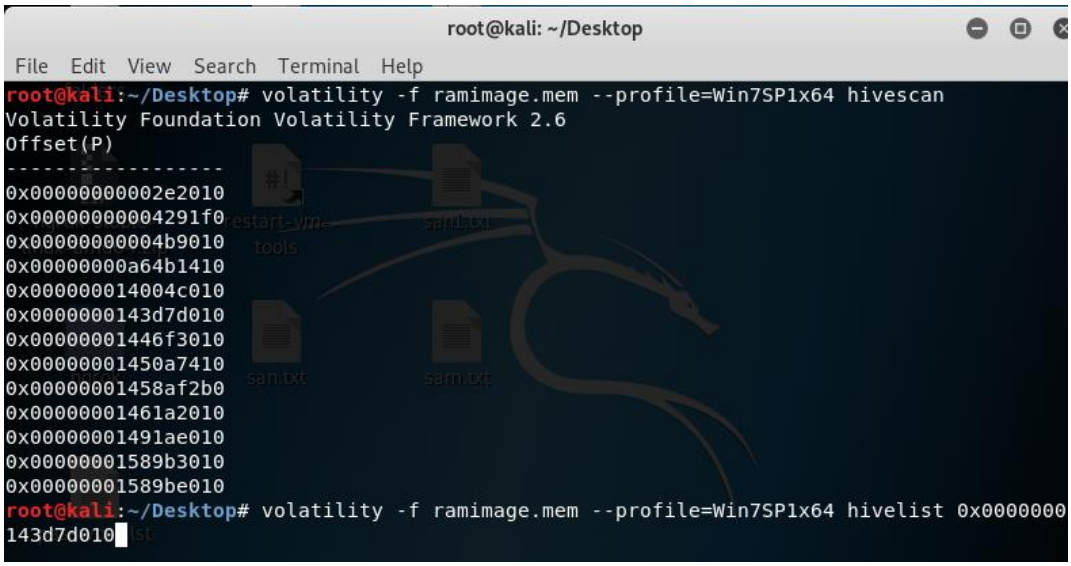


Figura 4-60: Ejecución del comando hivescan del profile de imagen de memoria RAM mediante volatility

volatility -f ramimage.mem --profile=Win7SP1x64 hivelist 0x0000000143d7d010

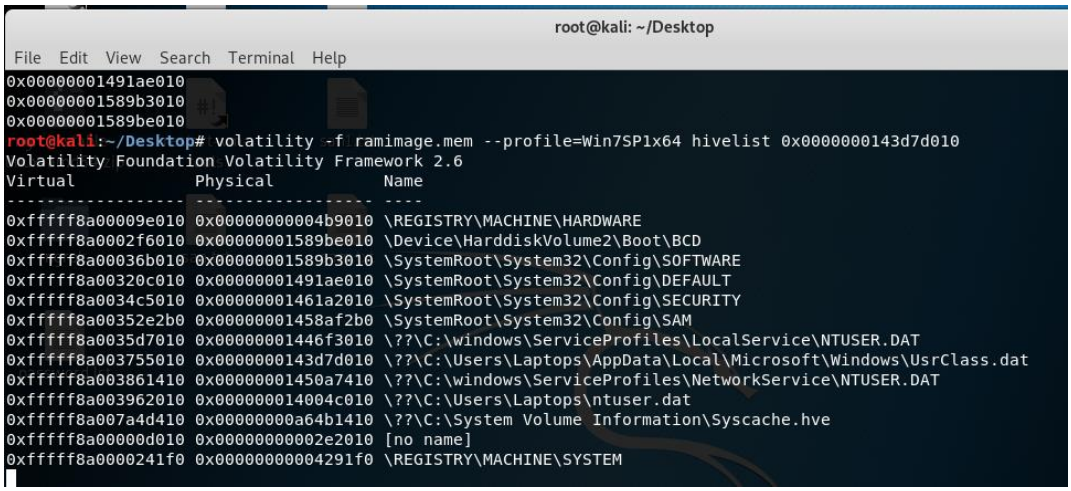


Figura 4-61: Ejecución del comando hivelist del profile de imagen de memoria RAM mediante volatility

Como se puede apreciar en la imagen las que nos interesan para este apartado son SAM y SYSTEM, el comando para extraer el contenido

completo de SAM es el siguiente hashdump, -y registro del archivo SYSTEM –s y ahora si el Offset donde está el SAM para identificar donde está el hive en la memoria –s tal como se muestra a continuación.

volatility -f ramimage.mem --profile=Win7SP1x64 hashdump -y 0xfffff8a0000241f0 -s 0xfffff8a00352e2b0

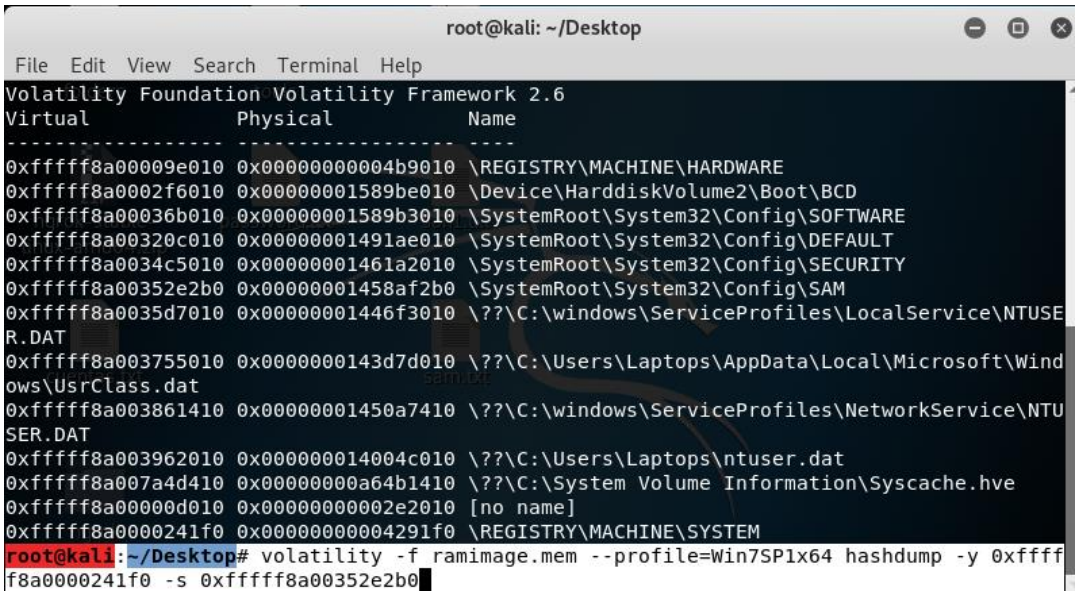


Figura 4-62: Ejecución del comando hashdump del profile de imagen de memoria RAM mediante volatility

La imagen muestra el proceso correspondiente de la extracción del contenido SAM de la memoria RAM.

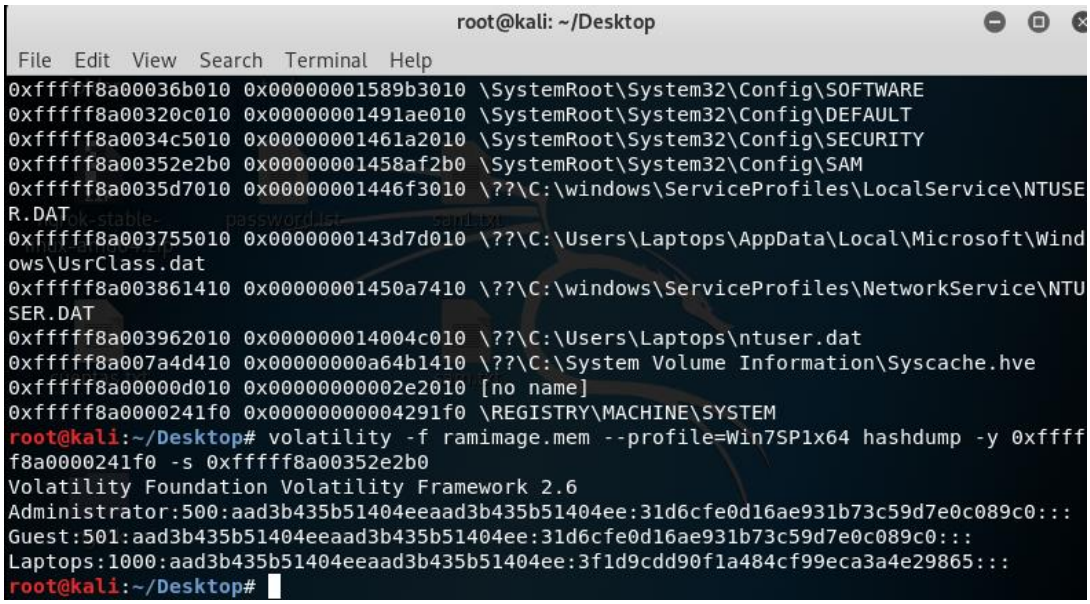


Figura 4-63: Ejecución del comando hashdump del profile de imagen de memoria RAM mediante volatility

Como podemos apreciar en la imagen ya extrajo el contenido de la memoria SAM del registro, donde podemos visualizar los diferentes usuarios del sistema operativo, Administrator, Guest, Laptops con sus respectivos hash

Para recuperar las contraseñas copiamos el contenido y creamos un archivo llamado sam.txt

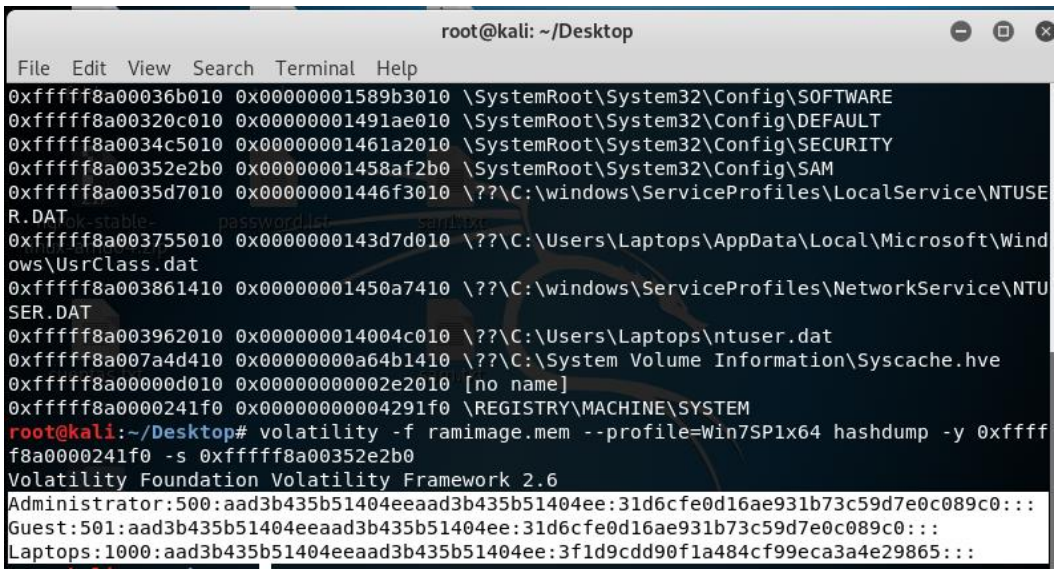


Figura 4-64: Ejecución del comando hashdump del profile de imagen de memoria RAM mediante volatility

Lo que esta resaltado debemos de copiarlo y crear un archivo de nombre sam.txt para aplicar lo que es el password cracking

```
root@kali:~/Desktop# volatility -f ramimage.mem --profile=Win7SP1x64 hashdump -y 0xffff
f8a0000241f0 -s 0xfffff8a00352e2b0
Volatility Foundation Volatility Framework 2.6
Administrator:500:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
Guest:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
Laptops:1000:aad3b435b51404eeaad3b435b51404ee:3f1d9cdd90f1a484cf99eca3a4e29865:::
root@kali:~/Desktop# gedit sam.txt
```

Con el comando gedit sam.txt procedemos a crear el archivo

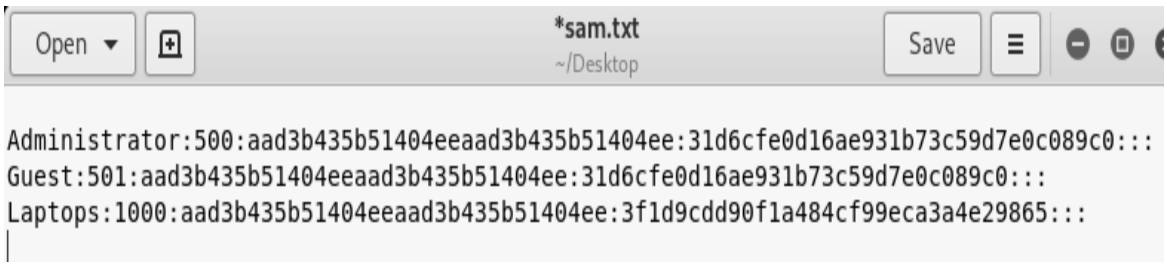


Figura 4-65: Creación del archivo txt para colocar los hash

Luego de haber guardado el contenido dentro del archivo sam.txt podemos utilizar una herramienta de tipo John de ripper para reventar esa clave y poder conocer el password mediante técnicas de password cracking

Comando a utilizar es **john sam.txt**

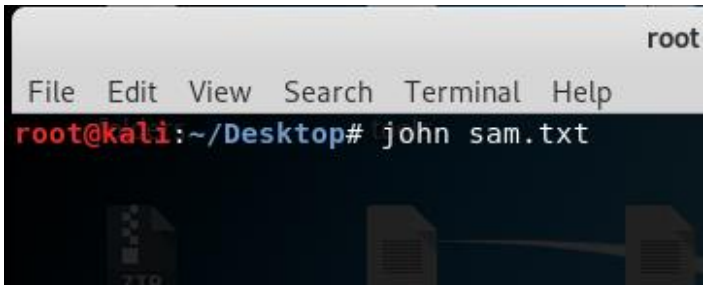


Figura 4-66: Ejecución de la herramienta John ripper

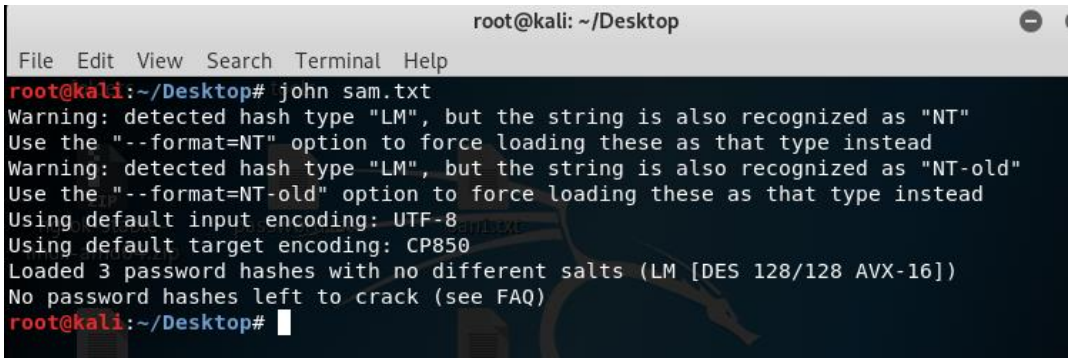


Figura 4-67: Obtención de la clave del sistema operativo

Nos muestra el siguiente resultado para lo cual debemos de ejecutar el formato correcto, pero dice que ha encontrado 3 password

Con el comando **john --format=NT --user=Laptops sam.txt –show** **podemos extraer la clave del sistema operativo**

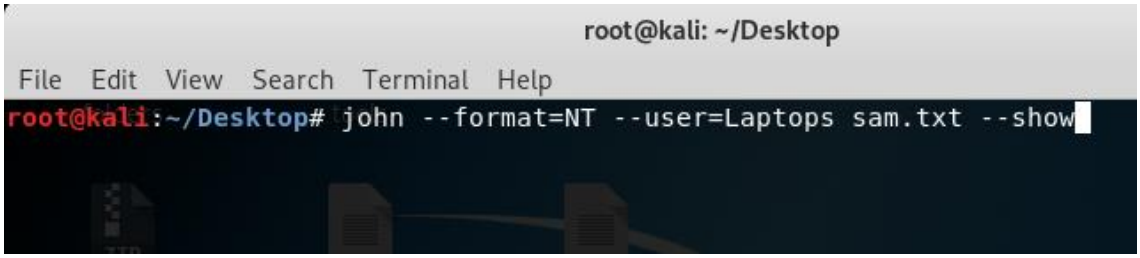


Figura 4-68: Ruta para reventar la clave mediante John ripper

Como podemos apreciar en la imagen de abajo muestra el usuario y su respectiva clave de acceso al sistema operativo

john --format=NT --user=Laptops sam.txt --show

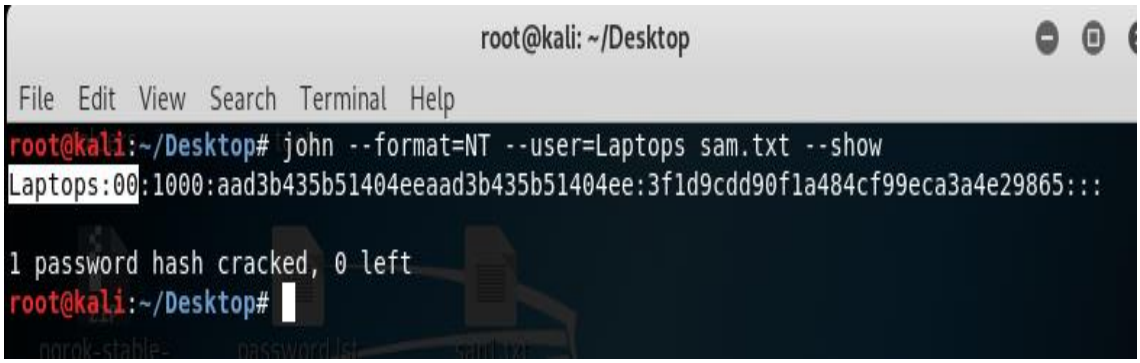


Figura 4-69: Proceso devuelto con la clave actual del sistema operativo

Donde el usuario es **Laptops** y la clave es **00**

Para analizar malware

volatility -f ramimage.mem --profile=Win7SP1x64 malfind

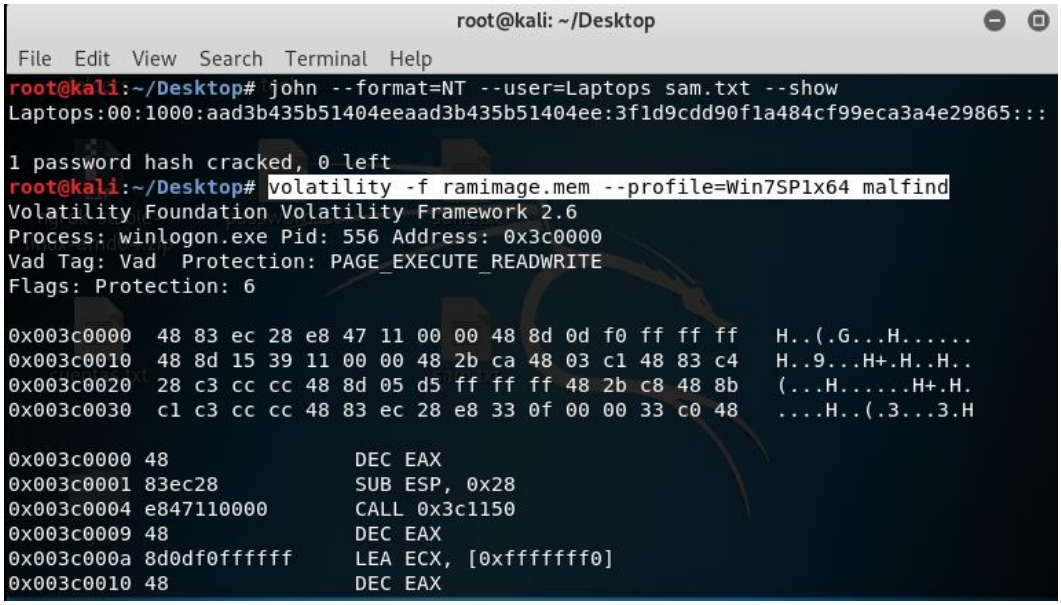


Figura 4-70: Ejecución del comando malfind para encontrar malware en el sistema operativo

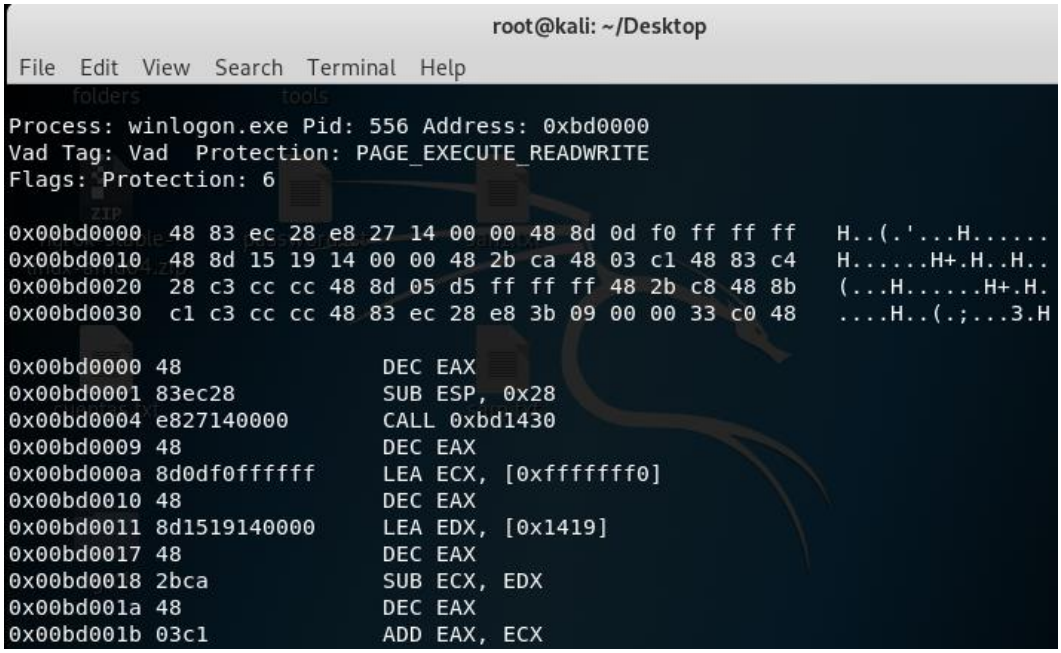


Figura 4-71: Ejecución del comando malfind para encontrar malware en el sistema operativo

Tema 2: Memoria virtual

La memoria virtual es una técnica que permite la ejecución de procesos que no están completamente en la memoria. Una de las principales ventajas de este esquema es que los programas pueden ser más grandes que la memoria física. Además, la memoria virtual abstrae la memoria principal en una matriz de almacenamiento extremadamente grande y uniforme, separando la memoria lógica vista por el programador de la memoria física. Esta técnica libera a los programadores de las preocupaciones de las limitaciones de almacenamiento de memoria. La memoria virtual también permite a los procesos compartir archivos y bibliotecas e implementar memoria compartida. Además, proporciona un mecanismo eficiente para la creación de procesos. La memoria virtual no es fácil de implementar, sin embargo, y puede disminuir sustancialmente el rendimiento si se utiliza descuidadamente. En este capítulo, proporcionamos una descripción detallada de la memoria virtual, examinamos cómo se implementa y exploramos su complejidad y beneficios.

Fundamentos de la memoria virtual

Los algoritmos de administración de memoria descritos en la sección anterior son necesarios debido a un requisito básico: las instrucciones que se ejecutan deben estar en la memoria física. El primer enfoque para cumplir este requisito es colocar todo el espacio de direcciones lógicas en la memoria física. La vinculación dinámica puede ayudar a aliviar esta

restricción, pero generalmente requiere precauciones especiales y trabajo adicional por parte del programador.

El requisito de que las instrucciones deben estar en la memoria física para ser ejecutadas parece necesario y razonable; pero también es desafortunado, ya que limita el tamaño de un programa al tamaño de la memoria física. De hecho, un examen de programas reales nos muestra que, en muchos casos, no se necesita todo el programa. Por ejemplo, considere lo siguiente:

- Los programas a menudo tienen código para manejar condiciones de error inusuales. Dado que estos errores rara vez, si es que alguna vez, se producen en la práctica, este código casi nunca se ejecuta.
- Las matrices, listas y tablas a menudo se asignan más memoria de la que realmente necesitan. Una matriz se puede declarar 100 por 100 elementos, aunque rara vez es mayor que 10 por 10 elementos.
- Ciertas opciones y características de un programa se pueden utilizar raramente. Por ejemplo, las rutinas en computadoras del gobierno de los Estados Unidos que equilibran el presupuesto no se han utilizado en muchos años.

Incluso en aquellos casos en los que se necesita todo el programa, es posible que no todo sea necesario al mismo tiempo.

La capacidad de ejecutar un programa que sólo está parcialmente en la memoria conferiría muchos beneficios:

- Un programa ya no estaría limitado por la cantidad de memoria física disponible. Los usuarios podrían escribir programas para un espacio de direcciones virtual extremadamente grande, simplificando la tarea de programación.

- Debido a que cada programa podría tomar menos memoria física, más programas podrían ejecutarse al mismo tiempo, con un aumento correspondiente en la utilización de la CPU y el rendimiento, pero sin aumento en el tiempo de respuesta o tiempo de respuesta.
- Se necesitaría menos E/S para cargar o intercambiar porciones de programas en la memoria, por lo que cada programa se ejecutaría más rápido.

Por lo tanto, ejecutar un programa que no está completamente en la memoria beneficiaría tanto al sistema como a sus usuarios.

La **memoria virtual** implica la separación de la memoria lógica percibida por los desarrolladores de la memoria física. Esta separación permite proporcionar una memoria virtual extremadamente grande a los programadores cuando solo hay una memoria física más pequeña disponible (Figura 4-). La memoria virtual hace que la tarea de programación sea mucho más fácil, ya que el programador ya no necesita preocuparse por la cantidad de memoria física disponible; puede concentrarse en la programación del problema que se va a resolver.

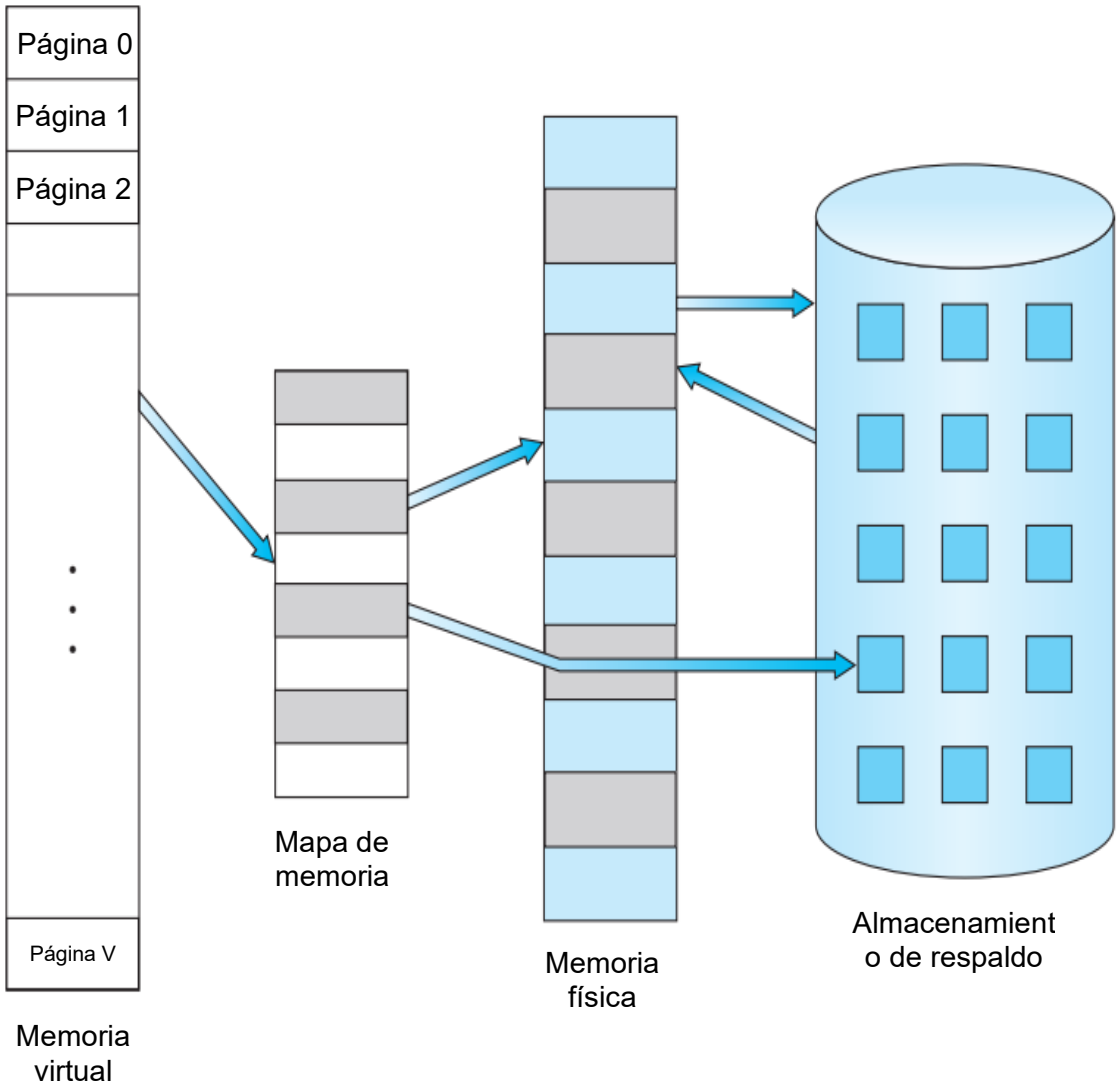
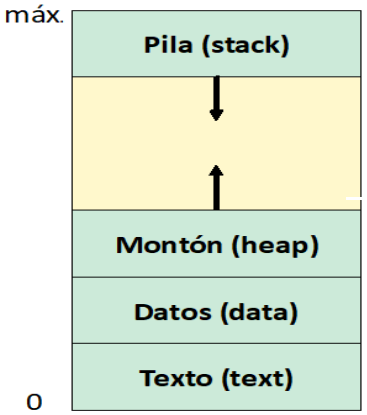


Figura 4-72: Diagrama mostrando la memoria virtual que es más grande que la memoria física

El **espacio de direcciones virtuales** de un proceso hace referencia a la vista lógica (o virtual) de cómo se almacena un proceso en la memoria. Típicamente, esta opinión es que un proceso comienza en una dirección lógica determinada —digamos, la dirección 0— y existe en la memoria contigua, tal y como se muestra en de la Figura 4-. Recuerde de la sección anterior, que, aunque de hecho la memoria física está organizada en

marcos de página y que los marcos de página físicos asignados a un proceso pueden no ser contiguos. Depende de la unidad de gestión de memoria (MMU – memory management unit) para asignar páginas lógicas a marcos de página físicos en la memoria.



Observe en la Figura 4- que permitimos que el montón crezca hacia arriba en la memoria tal como se utiliza para la asignación de memoria dinámica. Del mismo modo, permitimos que la pila crezca hacia abajo en la memoria a través de llamadas de función sucesivas. El gran espacio en blanco (o agujero) entre el montón y la pila forma parte del espacio de direcciones virtuales, pero requerirá páginas físicas reales solo si el montón o la pila crecen. Los espacios de direcciones virtuales que incluyen agujeros se conocen como espacios de direcciones dispersos. El uso de un espacio de direcciones disperso es beneficioso porque los agujeros se pueden rellenar a medida que crecen los segmentos de pila o montón o si deseamos vincular dinámicamente bibliotecas (o posiblemente otros objetos compartidos) durante la ejecución del programa.

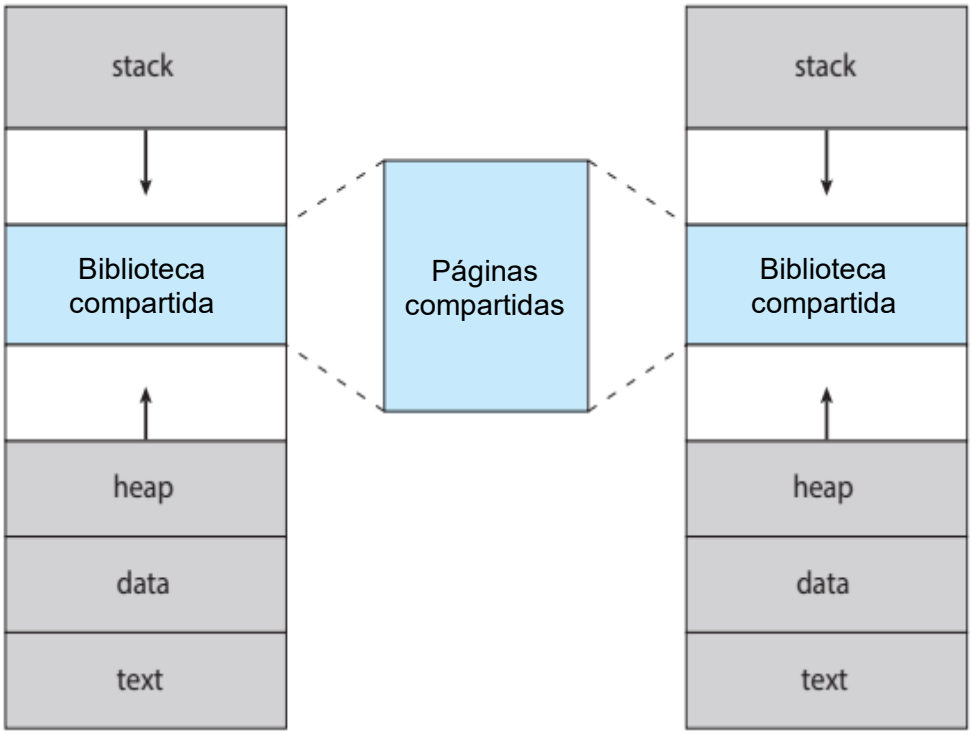


Figura 4-74: Bibliotecas compartidas, usando memoria virtual

Además de separar la memoria lógica de la memoria física, la memoria virtual permite que dos o más procesos compartan archivos y memoria a través del uso compartido de páginas. Esto conduce a los siguientes beneficios:

- Las bibliotecas del sistema, como la biblioteca C estándar, se pueden compartir mediante varios procesos mediante la asignación del objeto compartido en un espacio de direcciones virtual. Aunque cada proceso considera que las bibliotecas forman parte de su espacio de direcciones virtuales, las páginas reales aquí las bibliotecas residen en la memoria física son compartidas por todos los procesos (Figura 4-). Normalmente, una biblioteca se asigna de solo lectura en el espacio de cada proceso que está vinculado con ella.

- Del mismo modo, los procesos pueden compartir memoria. Recuerde en la unidad de procesos, que dos o más procesos pueden comunicarse mediante el uso de memoria compartida. La memoria virtual permite que un proceso cree una región de memoria que puede compartir con otro proceso. Los procesos que comparten esta región la consideran parte de su espacio de direcciones virtuales, sin embargo, las páginas físicas reales de memoria se comparten, al igual que se ilustra en la Figura 4-.
- Las páginas se pueden compartir durante la creación del proceso con la llamada al sistema fork(), acelerando así la creación de procesos.

Exploramos más a fondo estos beneficios (y otros) de la memoria virtual más adelante en este capítulo. En primer lugar, sin embargo, discutimos la implementación de la memoria virtual a través de la paginación de la demanda

Paginación bajo demanda

Considere cómo se puede cargar un programa ejecutable desde el almacenamiento secundario en la memoria. Una opción es cargar todo el programa en memoria física en tiempo de ejecución del programa. Sin embargo, un problema con este enfoque es que es posible que inicialmente no necesitemos todo el programa en la memoria. Supongamos que un programa comienza con una lista de opciones disponibles entre las que el usuario debe seleccionar. La carga de todo el programa en la memoria da como resultado la carga del código ejecutable para todas las opciones, independientemente de si el usuario selecciona o no una opción en última instancia.

Una estrategia alternativa es cargar páginas solo cuando sean necesarias. Esta técnica se conoce como paginación de demanda y se utiliza comúnmente en sistemas de memoria virtual. Con la memoria virtual paginada a petición, las páginas se cargan solo cuando se exigen durante la ejecución del programa. Por lo tanto, las páginas a las que nunca se accede nunca se cargan en la memoria física. Un sistema de paginación de demanda es similar a un sistema de intercambio con paginación (sección anterior) donde los procesos residen en la memoria secundaria (normalmente un dispositivo HDD o NVM). La paginación de demanda explica una de las principales ventajas de la memoria virtual: al cargar solo las partes de los programas que se necesitan, la memoria se usa de forma más eficaz.

Copia durante la escritura

Hasta ahora, hemos ilustramos cómo un proceso puede comenzar rápidamente mediante la paginación de la demanda en la página que contiene la primera instrucción. Sin embargo, la creación de procesos mediante la llamada al sistema fork() puede omitir inicialmente la necesidad de paginación de la demanda mediante una técnica similar al uso compartido de páginas. Esta técnica proporciona una rápida creación de procesos y minimiza el número de páginas nuevas que se deben asignar al proceso recién creado.

Recuerde que la llamada al sistema fork() crea un proceso secundario que es un duplicado de su elemento primario. Tradicionalmente, fork() trabajaba creando una copia del espacio de direcciones del elemento primario para el secundario, duplicando las páginas que pertenecen al elemento primario.

Sin embargo, teniendo en cuenta que muchos procesos secundarios invocan la llamada al sistema exec () inmediatamente después de la creación, la copia del espacio de direcciones del elemento primario puede ser innecesaria. En su lugar, podemos usar una técnica conocida como copia en escritura, que funciona permitiendo que los procesos primarios y secundarios compartan inicialmente las mismas páginas. Estas páginas compartidas se marcan como páginas de copia en escritura, lo que significa que, si cualquiera de los procesos escribe en una página compartida, se crea una copia de la página compartida. La copia en escritura se ilustra en las Figura 4- y 10.8, que muestran el contenido de la memoria física antes y después del proceso 1 modifica la página C (page C).

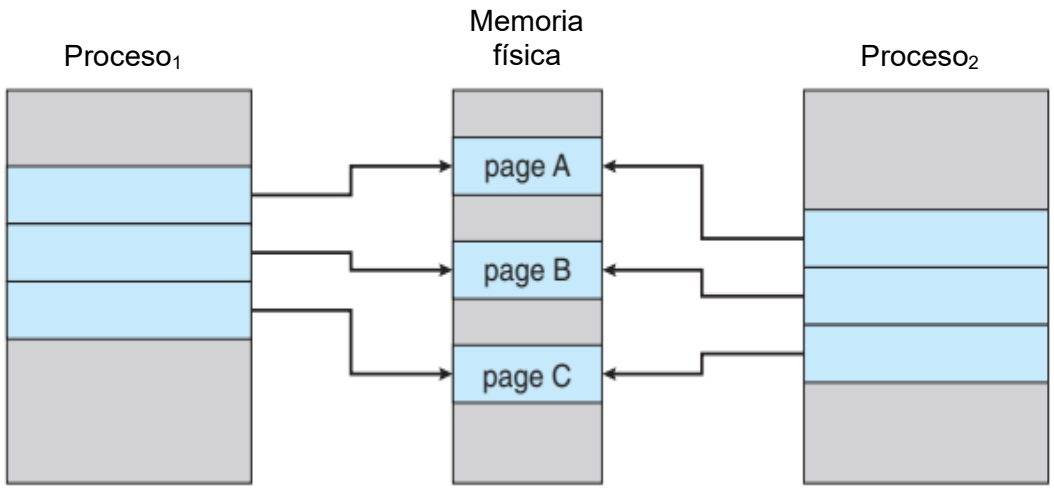


Figura 4-75: Antes que el proceso 1 modifique la página C (page C)

Por ejemplo, supongamos que el proceso secundario intenta modificar una página que contiene partes de la pila, con las páginas configuradas para copiar en escritura. El sistema operativo obtendrá un marco de la lista de fotogramas libres y creará una copia de esta página, asignándolo al espacio de direcciones del proceso secundario.

A continuación, el proceso secundario modificará su página copiada y no la página que pertenece al proceso primario. Obviamente, cuando se utiliza la técnica de copia en escritura, solo se copian las páginas modificadas por cualquiera de los procesos; todas las páginas no modificadas pueden ser compartidas por los procesos primarios e secundarios. Tenga en cuenta, también, que solo las páginas que se pueden modificar deben marcarse como copia en escritura. Las páginas que no se pueden modificar (páginas que contienen código ejecutable) pueden ser compartidas por el elemento primario y el elemento secundario. Copiar en escritura es una técnica común utilizada por varios sistemas operativos, incluidos Windows, Linux y macOS.

Varias versiones de UNIX (incluyendo Linux, macOS y BSD UNIX) proporcionan una variación de la llamada del sistema fork () (vfork () (para la **bifurcación (fork) de memoria virtual**)) que funciona de forma diferente a fork () con copy-on-write. Con vfork(), el proceso primario se suspende y el proceso secundario utiliza el espacio de direcciones del elemento primario. Dado que vfork () no utiliza copy-on-write, si el proceso secundario cambia alguna página del espacio de direcciones del elemento primario, las páginas alteradas serán visibles para el elemento primario una vez que se reanude. Por lo tanto, vfork () debe utilizarse con precaución para asegurarse de que el proceso secundario no modifique el espacio de direcciones del elemento primario. vfork () está pensado para usarse cuando el proceso secundario llama a exec () inmediatamente después de la creación. Debido a que no se realiza ninguna copia de páginas, vfork () es un método extremadamente eficaz de creación de procesos y a veces se utiliza para implementar interfaces de shell de línea de comandos UNIX.

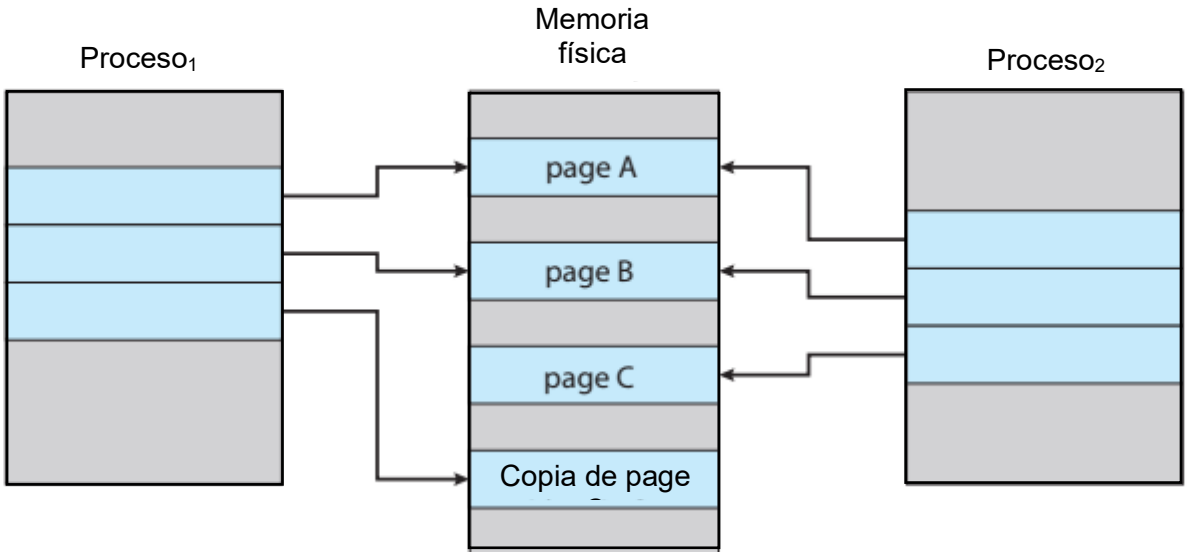


Figura 4-76: Después que el proceso 1 modifique la página C (page C)

Sustitución de páginas

Se suele asumir que cada página falla como máximo una vez, cuando se hace referencia por primera vez. Sin embargo, esta representación no es estrictamente precisa. Si un proceso de diez páginas realmente utiliza solo la mitad de ellas, la paginación de la demanda guarda la E/S necesaria para cargar las cinco páginas que nunca se utilizan. También podríamos aumentar nuestro grado de multiprogramación ejecutando el doble de procesos. Por lo tanto, si tuviéramos cuarenta fotogramas, podríamos ejecutar ocho procesos, en lugar de los cuatro que podrían ejecutarse si cada uno requería diez fotogramas (cinco de los cuales nunca se utilizaron).

Si aumentamos nuestro grado de multiprogramación, estamos asignando en exceso la memoria. Si ejecutamos seis procesos, cada uno de los cuales tiene diez páginas de tamaño, pero en realidad utiliza solo cinco páginas, tenemos un mayor uso y rendimiento de la CPU, con diez fotogramas de sobra. Sin embargo, es posible que cada uno de estos procesos, para un conjunto de datos determinado, intente de repente utilizar las diez

páginas, lo que resulta en la necesidad de sesenta fotogramas cuando sólo cuarenta están disponibles.

Además, tenga en cuenta que la memoria del sistema no se utiliza sólo para contener páginas de programa. Los búferes para E/S también consumen una cantidad considerable de memoria. Este uso puede aumentar la tensión en los algoritmos de colocación de memoria. Decidir cuánta memoria asignar a E/S y cuánto programar páginas es un desafío importante. Algunos sistemas asignan un porcentaje fijo de memoria para los búferes de E/S, mientras que otros permiten que tanto los procesos como el subsistema de E/S compitan por toda la memoria del sistema. El tutor describirá la relación integrada entre los búferes de E/S y las técnicas de memoria virtual.

La sobreasignación de memoria se manifiesta de la siguiente manera. Mientras se ejecuta un proceso, se produce un error de página. El sistema operativo determina dónde reside la página deseada en el almacenamiento secundario, pero luego encuentra que no hay marcos libres en la lista de fotogramas libres; toda la memoria está en uso. Esta situación se ilustra en la Figura 10.9, donde el hecho de que no hay marcos libres se representa mediante un signo de interrogación.

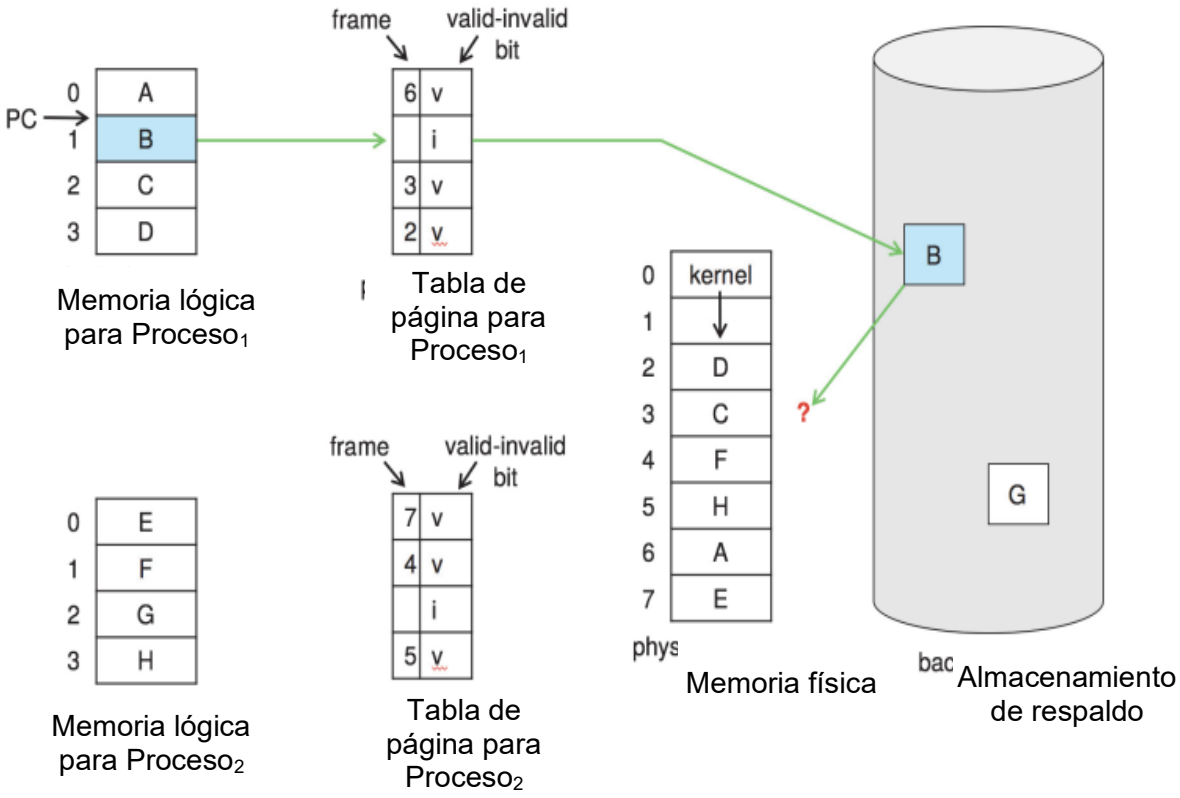


Figura 4-77: Necesidad de reemplazar página

El sistema operativo tiene varias opciones en este punto. Podría terminar el proceso. Sin embargo, la paginación de la demanda es el intento del sistema operativo de mejorar la utilización y el rendimiento del sistema informático. Los usuarios no deben ser conscientes de que sus procesos se ejecutan en un sistema paginado: la paginación debe ser lógicamente transparente para el usuario. Así que esta opción no es la mejor opción. El sistema operativo podría utilizar el intercambio estándar e intercambiar un proceso, liberando todos sus fotogramas y reduciendo el nivel de multiprogramación. Sin embargo, como se describe en la sección de intercambios, la mayoría de los sistemas operativos ya no utilizan el intercambio estándar debido a la sobrecarga de copiar procesos completos entre la memoria y el espacio de intercambio. La mayoría de los sistemas

operativos ahora combinan el intercambio de páginas con el reemplazo de páginas.

Algunos reemplazos que podemos citar son:

- Reemplazo básico de páginas
- Reemplazo de página FIFO
- Reemplazo óptimo de página
- Reemplazo de página LRU
- Reemplazo de página LRU-Aproximación
- Reemplazo de página basado en el recuento

Asignación de marcos

¿Cómo asignamos la cantidad fija de memoria libre entre los diversos procesos? Si tenemos 93 marcos libres y dos procesos, ¿cuántas marcos obtiene cada proceso?

Considere un caso simple de un sistema con 128 marcos. El sistema operativo puede tomar 35, dejando 93 marcos para el proceso del usuario. Bajo la paginación de la demanda pura, los 93 marcos se colocarían inicialmente en la lista de marcos libres. Cuando un proceso de usuario inicia la ejecución, generaría una secuencia de errores de página. Los primeros 93 errores de página obtendrían fotogramas gratuitos de la lista de fotogramas libres. Cuando se agotó la lista de fotogramas libres, se utilizaría un algoritmo de reemplazo de páginas para seleccionar una de los 93 páginas en memoria que se reemplazarán por la 94, y así sucesivamente. Cuando el proceso terminó, los 93 marcos se colocarían una vez más en la lista de fotogramas libres.

Hay muchas variaciones en esta estrategia simple. Podemos requerir que el sistema operativo asigne todo su espacio de búfer y tabla de la lista de marcos libres. Cuando el sistema operativo no utiliza este espacio, se

puede utilizar para admitir la paginación de usuarios. Podemos tratar de mantener tres marcos libres reservados en la lista de fotogramas libres en todo momento. Por lo tanto, cuando se produce un error de página, hay un marco libre disponible para paginar en. Mientras se realiza el intercambio de páginas, se puede seleccionar un reemplazo, que luego se escribe en el dispositivo de almacenamiento a medida que el proceso de usuario continúa ejecutándose. Otras variantes también son posibles, pero la estrategia básica es clara: al proceso de usuario se le asigna cualquier marco libre.

Sobrepaginación

Si el número de marcos asignados a un proceso de baja prioridad cae por debajo del número mínimo requerido por la arquitectura de la máquina, deberemos suspender la ejecución de dicho proceso y a continuación descargar de memoria todas sus restantes páginas, liberando así todos los marcos que tuviera asignados. Este mecanismo introduce un nivel intermedio de planificación de la CPU, basado en la carga y descarga de páginas.

De hecho, pensemos en un proceso que no disponga de "suficientes" marcos. Si el proceso no tiene el número de marcos que necesita para soportar las páginas que se están usando activamente, generará rápidamente fallos de página. En ese momento, deberá sustituir alguna página; sin embargo, como todas sus páginas se están usando activamente, se verá forzado a sustituir una página que va a volver a ser necesaria enseguida. Como consecuencia, vuelve a generar rápidamente una y otra vez sucesivos fallos de página, sustituyendo páginas que se ve forzado a recargar inmediatamente.

Esta alta tasa de actividad de paginación se denomina sobre paginación. Un proceso entrará en sobre paginación si invierte más tiempo implementando los mecanismos de paginación que en la propia ejecución del proceso.

Causa de la sobre paginación

La sobre paginación provoca graves problemas de rendimiento. Considere el siguiente escenario, que está basado en el comportamiento real de los primeros sistemas de paginación que se utilizaron.

El sistema operativo monitoriza la utilización de la CPU. Si esa tasa de utilización es demasiado baja, se incrementa el grado de multiprogramación introduciendo un nuevo proceso en el sistema. Se utiliza un algoritmo de sustitución global de páginas, que sustituye las páginas sin tomar en consideración a qué proceso pertenecen. Ahora suponga que un proceso entra en una nueva fase de ejecución y necesita más marcos de memoria. El proceso comenzará a generar fallos de página y a quitar marcos a otros procesos. Dichos procesos necesitan, sin embargo, esas páginas por lo que también generan fallos de página, quitando marcos a otros procesos. Los procesos que generan los fallos de páginas deben utilizar los procesos de paginación para cargar y descargar las páginas en memoria y, a medida que se ponen en cola para ser servidos por el dispositivo de paginación, la cola de procesos preparados se vacía. Como los procesos están a la espera en el dispositivo de paginación, la tasa de utilización de la CPU disminuye.

El planificador de la CPU ve que la tasa de utilización de la CPU ha descendido e incrementa como resultado el grado de multiprogramación. El nuevo proceso tratará de iniciarse quitando marcos a los procesos que se estuvieran ejecutando, haciendo que se provoquen más fallos de página

y que la cola del dispositivo de paginación crezca. Como resultado, la utilización de la CPU cae todavía más y el planificador de la CPU trata de incrementar el grado de multiprogramación todavía en mayor medida. Se ha producido una sobre paginación y la tasa de procesamiento del sistema desciende vertiginosamente, a la vez que se incrementa enormemente la tasa de fallos de página. Como resultado, también se incrementa el tiempo efectivo de acceso a memoria y no se llegará a realizar ningún trabajo útil, porque los procesos invertirán todo su tiempo en los mecanismos de paginación(Allende et al., 2019).

Preguntas de repaso

Para complementar su aprendizaje, conteste las siguientes preguntas.

1. Se considera “almacenamiento secundario” o “almacenamiento auxiliar” al generalmente soportado en

a. Discos
b. La memoria principal
c. Cassettes
d. La memoria principal

2. En la transformación de bloques: cuanto mayor sea el bloque de AM virtual, menor será la fracción del almacenamiento real que debe dedicarse a contener la información del mapa. Es decir, "bloques" grandes guardan más información por lo que se necesita menos de ellos, en consecuencia para hacer saber sus localizaciones (mapas) se necesita menos espacio en el AM principal.

Seleccione una:
Verdadero
Falso

3. Un proceso solo puede ejecutarse si su segmento actual está en el AM primario en un sistema de "segmentación".

Seleccione una:
Verdadero
Falso

4. En un sistema con memoria virtual con política de preasignación de swap, una memoria física de 32Mbytes y un área de swap de disco de 500 Mbytes. ¿Cuál es el límite máximo de memoria virtual que pueden ocupar los procesos?

a. 532 Mbytes

b. 232 Mbytes.

c. 500 Mbytes

d. 32 Mbytes

5. Son dos tareas del administrador de memoria.

a. Asignar memoria a los procesos; almacenar archivos

b. Asignar memoria de los procesos; llevar un registro de los procesos en espera

c. Desasignar memoria a los procesos; crear procesos

d. Llevar un registro de la memoria que está siendo utilizada; asignar memoria a los procesos

6. Indicar si el siguiente concepto es verdadero o falso:

La memoria física se divide en páginas y la memoria virtual se divide en marcos de página.

Seleccione una:

Verdadero

Falso

7. ¿Cuándo ocurre un fallo de página?

a. Cuando se hacer referencia a una página que no se encuentra en la memoria virtual

b. Cuando no se pueden crear páginas en la memoria virtual.

c. Cuando se hace referencia a una página que no se encuentra en la memoria física

d. Cuando no se pueden crear páginas en la memoria física.

8. ¿Qué sucede cuando no hay suficiente memoria para satisfacer las demandas de un proceso de llegada?

Seleccione dos

a. Se coloca el proceso en una cola de espera

b. Se rechaza el proceso y se proporciona un mensaje de error

c. El proceso se ejecuta en segundo plano

d. El proceso de llegada se ejecuta en primer plano

9. Una dirección generada por la CPU se denomina comúnmente:

a. dirección física

b. dirección logica

c. dirección IP

d. dirección virtual

e. dirección postal

10. En el siguiente comando:

volatility -f ramimage.mem --profile=Win7SP1x64 malfind

El parámetro **malfind** realiza:

a. Encontrar la dirección de paginación

b. Inyectar malware en el sistema operativo

c. Encontrar malware en el sistema operativo

d. Encontrar la asignación de memoria

11. Un aspecto importante de la gestión de memoria que se volvió inevitable con los mecanismos de paginación es la

a. Segmentación

b. Intercambio

c. Paginación

d. Intercambio con paginación

12. Cuál de los siguientes enunciados relacionados con la *paginación* son correctos:

(seleccione dos)

a. Es un esquema de administración de memoria que permite que el espacio de direcciones físicas de un proceso no sea contiguo.

b. Requiere que el espacio de direcciones físicas de un proceso sea contiguo.

c. Evita la fragmentación externa y la necesidad asociada de compactación

d. No se implementa mediante la cooperación entre el sistema operativo y el hardware del equipo.

Bibliografía

Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating Systems Concepts* (10th ed.). Hoboken: Wiley.

Stallings, W. (2018). *Operating Systems, Internals and Design Princiles* (Ninth ed.). Malaysia: Pearson.

Allende, S. L., Gibellini, F. A., Sánchez, C. B., & Serna, M. M. (2019). Sistemas Operativos Linux Teoria y Practica 2da Edicion (REUN).