Sistemas Operativos

Trabajo Práctico N.º 1: Conceptos generales

13. ¿Qué es el núcleo (kernel) de un SO?

El núcleo (o kernel) de un sistema operativo (SO) es la parte central y más importante del sistema operativo. Actúa como un intermediario entre el hardware del ordenador y los programas de software. Sus principales funciones incluyen:

- **Gestión de procesos:** el kernel controla la creación, ejecución y finalización de procesos. Asigna recursos de la CPU y administra la multitarea, asegurando que los procesos se ejecuten de manera eficiente y sin conflictos.
- **Gestión de memoria:** administra la memoria principal del sistema, asignando espacio de memoria a los procesos en ejecución y garantizando que no haya conflictos entre ellos. También gestiona la memoria virtual, permitiendo que el sistema utilice más memoria de la físicamente disponible.
- **Gestión de dispositivos:** el kernel proporciona una capa de abstracción entre el hardware y el software, controlando y facilitando la comunicación con los dispositivos de hardware como discos duros, impresoras, teclados, etc.
- **Gestión de archivos:** supervisa y controla el acceso al sistema de archivos la computadora, permitiendo que los programas lean y escriban archivos de manera organizada y segura.
- **Seguridad y control de acceso:** implementa mecanismos de seguridad para proteger los datos y recursos del sistema. Esto incluye la gestión de permisos y la autenticación de usuarios.
- **Interrupciones y excepciones:** maneja las interrupciones y excepciones del hardware, respondiendo a eventos como la finalización de una operación de I/O o la ocurrencia de un error de hardware.

(ChatGPT 4o)

14. Defina las propiedades esenciales de los siguientes tipos de sistemas operativos: a) batch, b) interactivo, c) tiempo compartido, d) tiempo real, e) distribuido.

(a) Batch

La idea central detrás del esquema de procesamiento en batch simple es el uso de una pieza de software conocida como monitor. Con este tipo de SO, el usuario no tiene acceso directo al procesador. En su lugar, el usuario presenta el trabajo en tarjetas o cinta a un operador de computadoras, que agrupa (batches) los trabajos juntos secuencialmente y coloca el grupo entero en un dispositivo de entrada, para uso del monitor. Cada programa es construído para ramificarse de nuevo al monitor cuando completa su procesamiento, en ese punto el monitor automáticamente empieza a cargar el próximo programa. *(Stallings, pp. 74, 75)*

(b) Interactivo

Un sistema operativo interactivo es un software que permite a los usuarios interactuar directamente con el sistema a través de interfaces intuitivas, como una interfaz gráfica de usuario (GUI) o una interfaz de línea de comandos (CLI). Está diseñado para proporcionar respuestas rápidas a las acciones del usuario, facilitando la ejecución simultánea de múltiples tareas y proporcionando un entorno de computación que se adapta a las necesidades y preferencias del usuario. Además de gestionar recursos y servicios del sistema, un sistema operativo interactivo suele ofrecer características de personalización, seguridad, conectividad de red y gestión de archivos para mejorar la experiencia del usuario. *(ChatGPT 3.5)*

(c) Tiempo compartido

En un sistema de tiempo compartido, múltiples usuarios acceden de manera simultanea al sistema a través de terminales, con el SO entrelazando la ejecución de cada programa de usuario en una ráfaga corta o un quantum de computación. Entonces, si hay *n* usuarios activamente requeriendo servicios en un tiempo, cada usuario verá solamente en el promedio *1/n* de capacidad de cómputo efectiva, sin contar el *overhead* del SO. *(Stallings, p. 81)*

(d) Tiempo real

Un sistema de tiempo real es usado cuando son definidos requerimientos rígidos de tiempos de operación para el procesador o el flujo de datos; así, generalmente es usado un dispositivo de control en una aplicación dedicada.

Un sistema de tiempo real tiene restricciones de tiempo bien definidas y fijas. El procesamiento debe ser hecho dentro de los límites definidos o el sistema fallará. *(Silberschartz, p. 46)*

La computación en tiempo real puede definir como un tipo de computación en la que la correctitud del sistema depende no solamente en el resultado lógico de la computación, sino también en el tiempo en el que los resultados son producidos. En general, en un sistema en tiempo real, algunas de las tareas son tareas en tiempo real, y esas tiene cierto grado de urgencia para realizarse. Tales tareas intentan controlar o reaccionar a eventos que toman lugar en el mundo exterior. Como estos eventos ocurren en “tiempo real”, una tarea en tiempo real debe ser capaz de mantenerse actualizada con los eventos con los cuales está relacionada. De este modo, es usualmente posible asociar un deadline con una tarea en particular, donde el deadline especifica tanto el tiempo de inicio como el tiempo de completitud. Otra característica de las tareas en tiempo real es que pueden ser periódicas o aperiódicas. *(Stallings, p. 474)*

(e) Distribuído

Un sistema operativo distribuido es un software que gestiona un conjunto de computadoras independientes y las presenta como un único sistema coherente para los usuarios. Facilita la transparencia de acceso, ubicación, migración, replicación y concurrencia, haciendo que los recursos y servicios distribuidos sean accesibles de manera uniforme y sin que los usuarios perciban la complejidad subyacente. Estos sistemas están diseñados para ser escalables, fiables, seguros y capaces de gestionar

eficientemente la comunicación y sincronización entre nodos, proporcionando alta disponibilidad y consistencia de datos en un entorno de computación distribuido. *(ChatGPT 4o)*

15. Enumere y explique las caractertísticas a tener en cuenta en multiprogramación. ¿Qué relación hay entre multiprogramación y tiempo compartido? ¿Existen diferencias con tiempo compartido?

Características a de la multiprogramación:

- **Concurrencia: l**a multiprogramación permite que varios programas se ejecuten concurrentemente en un sistema, aumentando la utilización de la CPU al tener siempre un proceso listo para ejecutarse.
- **Gestión de Memoria:** es esencial una buena gestión de memoria para cargar varios programas en la memoria principal y permitir la ejecución simultánea. Esto incluye la segmentación, la paginación y otras técnicas de administración de memoria.
- **Planificación de CPU:** los algoritmos de planificación determinan cuál de los procesos en memoria debe ser ejecutado por la CPU en un momento dado.
- **Eficiencia de I/O:** la multiprogramación busca maximizar el uso de la CPU minimizando los tiempos de espera de I/O. Esto se logra al tener varios procesos en estado de espera mientras uno está en estado de ejecución.
- **Protección y seguridad:** es crucial asegurar que los procesos no interfieran entre sí. Los sistemas operativos deben proporcionar mecanismos para proteger los datos y los recursos de cada proceso.

Relación entre Multiprogramación y Tiempo Compartido:

Multiprogramación y tiempo compartido son conceptos relacionados pero distintos. La multiprogramación se refiere a la capacidad de un sistema operativo para manejar múltiples programas en la memoria al mismo tiempo, permitiendo que la CPU siempre tenga algún proceso para ejecutar, mejorando así la eficiencia general del sistema.

Tiempo compartido es una extensión de la multiprogramación. En un sistema de tiempo compartido, múltiples usuarios interactúan con el sistema al mismo tiempo, y cada usuario recibe una porción de tiempo de la CPU. Este tiempo se divide en intervalos cortos (llamados "quantum"), y la CPU alterna rápidamente entre los procesos, creando la ilusión de que cada usuario tiene su propia máquina.

Diferencias entre multiprogramación y tiempo compartido:

	Multiprogramación	Tiempo compartido
Objetivo principal	Maximizar el uso de la CPU y los recursos del sistema.	Proporcionar una respuesta rápida y equitativa a múltiples usuarios interactivos.
Interactividad	Enfocado en procesos batch, donde la interacción del usuario no es primordial.	Diseñado para permitir una alta interactividad con múltiples usuarios simultáneamente.
Mecanismos de planificación	Utiliza algoritmos de planificación para maximizar la utilización de la CPU.	Utiliza técnicas de planificación como el Round Robin para asegurar que todos los usuarios reciban un tiempo justo de CPU, minimizando el tiempo de respuesta.

(ChatGPT 4o)

16. En entornos de multiprogramación y tiempo compartido, los usuarios comparten el sistema simultáneamente. Esta situación puede resultar en varios problemas de seguridad. ¿Cuáles son estos problemas? ¿Es posible garantizar el mismo grado de seguridad en una máquina de tiempo compartido como se tiene en una máquina dedicada?

Problemas de seguridad en entornos de multiprogramación y tiempo compartido:

- Acceso no autorizado: los usuarios pueden intentar acceder a datos o recursos que no les pertenecen, aprovechando vulnerabilidades en el sistema operativo o errores en la configuración de permisos.
- Interferencia entre procesos: los procesos pueden interferir entre sí de manera intencionada o accidental, lo que puede llevar a la corrupción de datos o la interrupción de servicios.
- Aislamiento inadecuado: el aislamiento insuficiente entre procesos puede permitir que un proceso malicioso influya en otros procesos, por ejemplo, a través de canales encubiertos o la explotación de vulnerabilidades.
- Robo de información: los procesos maliciosos pueden espiar la memoria de otros procesos para robar información sensible
- Negación de servicio (DoS): los usuarios pueden intentar acaparar recursos del sistema, como CPU, memoria o ancho de banda, para degradar el rendimiento del sistema y denegar el servicio a otros usuarios.
- Explotación de vulnerabilidades del sistema operativo: los atacantes pueden explotar vulnerabilidades en el sistema operativo o en las aplicaciones para ganar acceso no autorizado o ejecutar código malicioso.

Garantía de seguridad en máquinas de tiempo compartido vs. máquinas dedicadas:

Máquinas de tiempo compartido:

En teoría, es más desafiante garantizar el mismo grado de seguridad en una máquina de tiempo compartido que en una máquina dedicada debido a la complejidad adicional de gestionar múltiples usuarios y procesos simultáneamente.

Sin embargo, con medidas de seguridad adecuadas, es posible alcanzar un alto nivel de seguridad. Estas medidas incluyen:

- Control de acceso rigoroso: implementar políticas de control de acceso estrictas para asegurar que los usuarios y procesos solo puedan acceder a los recursos permitidos.
- Aislamiento fuerte: utilizar técnicas de virtualización y contenedorización para aislar completamente los procesos y usuarios entre sí.
- Monitoreo y auditoría: implementar sistemas de monitoreo y auditoría para detectar y responder a actividades sospechosas en tiempo real.
- Encriptación: usar encriptación para proteger los datos en tránsito y en reposo.

Máquinas dedicadas:

Las máquinas dedicadas son intrínsecamente más seguras porque están diseñadas para un solo usuario o propósito, reduciendo la superficie de ataque y la posibilidad de interferencia entre procesos.

Los sistemas dedicados tienen menos necesidad de mecanismos complejos de aislamiento y pueden enfocarse más en la protección específica de su función o usuario. *(ChatGPT 4o)*

18. Describa las diferencias entre multiprocesamiento simétrico y asimétrico. ¿Cuáles son las ventajas y desventajas de un sistema con multiprocesadores?

Multiprocesamiento Simétrico (SMP)

- Distribución de tareas:
  - Cada procesador en el sistema tiene acceso igual al sistema operativo y al espacio de memoria compartida.
  - Tareas distribuidas: todos los procesadores pueden ejecutar tareas y acceder a la memoria compartida independientemente y de manera equitativa.
- Control del sistema:
  - Centralizado: un solo sistema operativo controla todos los procesadores.
  - Simetría: todos los procesadores son iguales y tienen el mismo acceso a los recursos del sistema.

Multiprocesamiento Asimétrico (AMP)

- Distribución de tareas:
  - Procesadores dedicados: uno de los procesadores actúa como el maestro y controla el sistema operativo y la asignación de tareas.
  - Tareas asignadas: los demás procesadores (esclavos) ejecutan las tareas que el maestro les asigna y pueden tener roles específicos, como manejar dispositivos o ejecutar procesos secundarios.
- Control del Sistema:
  - Descentralizado: el maestro tiene más control, y los procesadores esclavos tienen roles más limitados.
  - Asimetría: los procesadores no tienen igualdad en términos de acceso a los recursos y control del sistema.

Ventajas y Desventajas de un Sistema con Multiprocesadores

Ventajas

- Desempeño: si el trabajo a ser realizado por una computadora puede ser organizado de tal manera que algunas porciones del trabajo pueden ser realizadas en paralelo, entonces el sistema con múltiples procesadores produce mayor desempeño que uno con un sólo procesador del mismo tipo.
- Disponibilidad: en un sistema de multiprocesador simétrico, ya que todos los procesadores pueden realizar las mismas funciones, la falla de un sólo procesador no detiene a la máquina. En su lugar, el sistema puede continuar funcionando con un desempeño reducido.
- Crecimiento incremental: un usuario puede mejorar el desempeño de un sistema añadiendo un procesador adicional
- Escalado: los vendedores pueden ofrecer un rango de productos con diferentes precios y características de desempeño basado en el número de procesadores configurados en el sistema. *(Stallings, p. 57)*

Desventajas

- Complejidad: el diseño y la programación de sistemas multiprocesadores son más complejos debido a la necesidad de gestionar la concurrencia y la sincronización entre procesadores.
- Costo: los sistemas multiprocesadores son más caros de construir y operar, ya que requieren más componentes de hardware y consumen más energía.
- Problemas de sincronización: la gestión de recursos compartidos puede llevar a problemas de sincronización como condiciones de carrera y deadlocks, que son difíciles de manejar y depurar. *(ChatGPT 4o)*

19. ¿Cuál es la diferencia entre una interrupción y un trap? ¿Cuál es el uso de cada una?

Una interrupción es causado por un evento que es externo e independiente de un proceso actualmente en ejecución, como la finalización de una operación de I/O. En una interrupción ordinaria, el control es primeramente transferido a un manejador de interrupciones, el cual realiza un poco de gestión interna y luego se ramifica a una rutina del SO que que se ocupa del tipo particular de interrupción que ocurrió.

En cambio, un *trap* se refiere a un error o una condición de excepción generada en la ejecución del proceso actual, tal como un intento de acceso ilegal a un archivo. Con un *trap*, el SO determina si la condición de error o excepción es fatal. Si lo es, entonces el proceso siendo ejecutado actualmente es movido al estado de *Exit* y ocurre un cambio de proceso. Si no, entonces la acción del SO dependerá de la naturaleza del error y el diseño del SO. Puede intentar algún procedimiento de recuperación o simplemente notificar al usuario. Puede realizar un cambio de proceso o reanudar la ejecución del proceso actual.

Mecanismo	Causa	Uso
Interrupción	Externo a la ejecución de la instrucción actual	Reacción a un evento externo asincrónico
Trap	Asociado con la ejecución de la instrucción actual	Manejo de una condición de error o excepción

(Stallings, pp. 160, 161)

20. ¿Porqué es necesaria la operación en "modo dual"?

Ya que el sistema operativos y sus usuarios comparten los recursos de hardware y software del sistema computacional, un sistema operativo diseñado correctamente debe asegurar que un programa incorrecto (o malicioso) no pueda causar que otros programas (o el mismo SO) se ejecuten incorrectamente. Para asegurar la ejecución correcta del sistema, se debe poder distinguir entre la ejecución de código del sistema operativo y de código definido por el usuario.

Se necesitan al menos dos modos separados de operación: modo usuario y modo kernel. Un bit, llamado bit de modo, es añadido al hardware de la computadora para indicar el modo actual: kernel (0) o usuario (1). Con el bit de modo, se puede distinguir entre una tarea que es ejecutada en nombre del sistema operativo y una que es ejecutada en nombre del usuario.

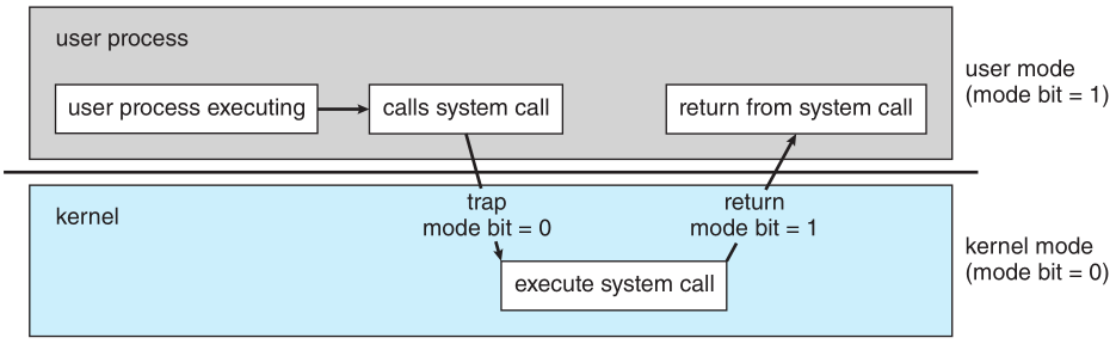


Figure 1.13 Transition from user to kernel mode.

El modo dual de operación provee los medios para la protección del sistemas operativo de usuarios errantes (y usuarios errantes unos de los otros). Se logra esta protección designando algunas de las instrucciones de máquina que pueden causar daño como instrucciones privilegiadas. El hardware permite que las instrucciones privilegiadas sean ejecutadas solamente en modo kernel. Si se intenta ejecutar una instrucción privilegiada en modo usuario, el hardware no la ejecuta, sino que la trata como ilegal y realiza un *trap* al sistema operativo.

(Silberschartz, p. 24)

22. ¿Porqué es necesario que el hardware ofrezca mecanismos de protección de I/0, de memoria y de la CPU? Explique cómo se realiza dicha protección.

Si un sistema computacional tiene múltiples usuarios y permite la ejecución concurrente de múltiples procesos, entonces el acceso a los datos debe ser regulado. Para este propósito, mecanismos aseguran que archivos, segmentos de memoria, CPU y otros recursos puedan ser operadors solamente por esos procesos que han ganado la autorización correspondiente desde el sistema operativo. Por ejemplo, el hardware de direccionamiento de memoria aseugra que un proceso se pueda ejecutar solamente en su propio espacio de memoria. El timer asegura que ningún proceso pueda ganar control de la CPU sin eventualmente renunciar al control. Los registros de control de dispositivo no son accesibles por los usuarios, así la integridad de varios dispositivos periféricos es protegida.

(Silberschartz, p. 33)

27. ¿Cuál es el propósito del intérprete de comandos? ¿Por qué usualmente está separado del kernel?

La función principal función del intérprete de comandos es obtener y ejecutar el siguiente comando especificado por el usuario. Estos comandos pueden ser implementados de dos maneras generales:

- El intérprete de comandos contiene en sí mismo el código para ejecutar el comando. En este caso, el número de comandos que pueden ser dados determina el tamaño del intérprete de comandos, ya que cada comando requiere su propio código de implementación
- Los comandos son implementados a través de programas del sistema. En este caso, el intérprete de comandos no entiende el comando de ninguna manera, solamente usa el comando para identificar un archivo que será cargado en memoria y ejecutado.

(Silberschartz, p. 58)

El motivo por el cual el intérprete de comandos usualmente está separado del kernel se debe a varias razones, principalmente relacionadas con la seguridad, la modularidad y la flexibilidad del sistema operativo:

- Seguridad: si el intérprete de comandos estuviera integrado en el kernel, cualquier fallo o vulnerabilidad en el shell podría comprometer todo el sistema.
- Modularidad: mantener el shell separado del kernel permite un diseño más modular del sistema operativo. Esto significa que diferentes shells pueden ser desarrollados y

utilizados sin necesidad de modificar el kernel. Los usuarios pueden elegir el shell que mejor se adapte a sus necesidades y preferencias, y los desarrolladores pueden actualizar y mejorar el shell sin afectar al núcleo del sistema operativo.

- Flexibilidad y personalización: al tener el shell separado del kernel, se brinda a los usuarios la capacidad de personalizar su entorno de trabajo. Hay una variedad de shells disponibles, cada uno con características y funcionalidades únicas (como Bash, Zsh, Fish, etc.). Los usuarios pueden elegir el shell que mejor se ajuste a su flujo de trabajo y estilo de uso.
- Facilita el desarrollo y la depuración: desarrollar y depurar el shell por separado del kernel es más sencillo y seguro. Los desarrolladores pueden trabajar en mejoras y correcciones del shell sin el riesgo de causar inestabilidad en el núcleo del sistema operativo. Además, los problemas en el shell no afectan la integridad del kernel, lo que permite una mayor estabilidad y confiabilidad del sistema en general.
- Interfaz de usuario: el shell actúa como una interfaz entre el usuario y el kernel, proporcionando una manera amigable y accesible de interactuar con el sistema operativo. Al mantener esta interfaz separada, se puede diseñar y mejorar la experiencia del usuario sin necesidad de alterar el funcionamiento interno del sistema operativo.

(ChatGPT 4o)

28. ¿Cuál es el propósito de los systems calls?

Las system calls son la manera en la cuál un proceso solicita un servicio especial del kernel. Existen varias system calls, la cuales pueden ser agrupadas en seis categorías: sistema de archivos, procesos, planificación, comunicación entre procesos, socket (networking) y misceláneo. *(Stallings, p. 116)*

Toda computadora de un sólo CPU puede ejecutar solamente una instrucción a la vez. Si un proceso está corriendo un proceso de usuario en modo usuario y necesita un servicio del sistema, tal como leer datos de un archivo, debe ejecutar una instrucción trap para transferir el control al sistema operativo. El sistema operativo entonces se da cuenta de qué es lo que el proceso llamador quiere inspeccionando los parámetros. Entonces lleva a cabo la system call y retorna el control a la instrucción siguiente a la system call. En este sentido, hacer una system call es como hacer un especial tipo de llamada a un procedimiento, solamente que solamente las system calls entran al kernel, mientras que los procedimientos no lo hacen. *(Tanenbaum, p. 51)*

29. ¿Cuál es la principal ventaja de usar un microkernel en el diseño de sistemas? ¿Cómo interactúan los programas de usuario y los servicios del sistema en una arquitectura basada en microkernel? ¿Cuáles son las desventajas de usar la arquitectura de microkernel?

La principal función del microkernel es proveer comunicación entre el programa de cliente y los servicios varios que también están corriendo en el espacio de usuario. La comunicación es provista a través de pasaje de mensajes. Por ejemplo, si el programa cliente desea acceder a un archivo, debe interactuar con el servidor de archivos. El

programa de cliente y el servicio nunca interactúan directamente. En su lugar, se comunican indirectamente a través del intercambio de mensajes con el microkernel.

Un beneficio de la aproximación de microkernel es que hace que el sistema operativo sea fácilmente extensible. Todo nuevo servicio es añadido al espacio de usuario y consecuentemente no requiere modificación del kernel. Cuando el kernel no tiene que ser modificado, los cambios tienden a ser pocos, porque el microkernel es un kernel pequeño. El sistema operativo resultante es fácilmente portable de un diseño de hardware a otro. El microkernel también provee mayor seguridad y confiabilidad, ya que la mayoría de los servicios están corriendo como procesos de usuario (en lugar de kernel). Si un servicio falla, el resto del sistema operativo permanece sin ser tocado.

Desafortunadamente, el rendimiento de los microkernels puede sufrir debido al incremento en la sobrecarga (*overhead*) de las funciones del sistema. Cuando dos servicios a nivel de usuario se deban comunicar, los mensajes deben ser copiados entre los servicios, los cuales residen en espacios de memoria separados. Además, el sistema operativo puede tener que cambiar de un proceso al siguiente para intercambiar los mensajes. La sobrecarga (*overhead*) involucrada en el copiado de mensajes y cambiar entre procesos ha sido el mayor impedimento en el crecimiento de los sistemas operativos basados en microkernel.

(Silberschartz, pp. 85, 86)

30. ¿En qué se asemejan la arquilectura kernel modular y la arquitectura en niveles? ¿En qué se diferencian?

Similitudes entre la arquitectura de kernel modular y la arquitectura en niveles

- *Organización y estructura:* ambas arquitecturas tienen una estructura organizada que permite la separación de funciones y componentes dentro del kernel.
- *Facilidad de mantenimiento:* tanto en la arquitectura modular como en la arquitectura en niveles, la separación de componentes facilita el mantenimiento y la actualización del sistema. Las modificaciones o mejoras en un módulo o nivel específico no afectan directamente a los demás componentes.
- *Abstracción:* ambas arquitecturas buscan un cierto grado de abstracción. En la arquitectura en niveles, cada nivel depende solo de las funcionalidades del nivel inferior inmediato, mientras que en la modular, cada módulo funciona como una unidad independiente que interactúa con otros módulos a través de interfaces bien definidas.

Diferencias entre la arquitectura en niveles y la arquitectura de kernel modular

	Arquitectura en niveles	Kernel modular
Estructura jerárquica vs. Independencia de módulos	Tiene una estructura jerárquica clara donde cada nivel solo puede interactuar con el nivel inmediatamente inferior. Este enfoque crea un flujo de control más estricto y definido.	No tiene una jerarquía estricta. Los módulos pueden ser cargados y descargados dinámicamente y pueden interactuar con otros módulos a través de interfaces establecidas sin seguir una estructura jerárquica rígida.
Flexibilidad y extensibilidad	Es menos flexible en comparación con el kernel modular, ya que cualquier cambio o actualización significativa puede requerir modificaciones en varios niveles de la estructura.	Ofrece mayor flexibilidad y extensibilidad. Los módulos pueden ser agregados o eliminados del kernel en tiempo de ejecución sin necesidad de recompilar el kernel completo.
Dependencias y acoplamiento	Tiene un mayor acoplamiento entre niveles, ya que cada nivel depende del nivel inferior inmediato.	Los módulos son relativamente independientes entre sí, lo que reduce el acoplamiento y permite una mayor modularidad.
Carga dinámica	No está diseñada para cargar o descargar componentes en tiempo de ejecución.	Permite la carga y descarga dinámica de módulos, lo que puede mejorar el rendimiento y optimizar el uso de recursos del sistema.

En resumen, la arquitectura de kernel modular y la arquitectura en niveles se asemejan en su enfoque hacia la organización y facilidad de mantenimiento, pero difieren significativamente en su estructura, flexibilidad y capacidad de carga dinámica. La arquitectura en niveles tiene una estructura jerárquica más rígida, mientras que el kernel modular ofrece mayor flexibilidad y extensibilidad gracias a la independencia y capacidad de carga dinámica de los módulos.

(ChatGPT 4o)

Sistemas Operativos

Trabajo Práctico N.º 2: Procesos y Threads

Notas

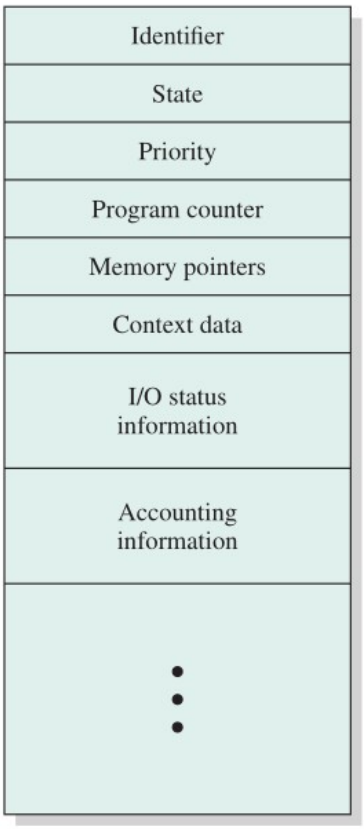


Figure 3.1 Simplified Process Control Block

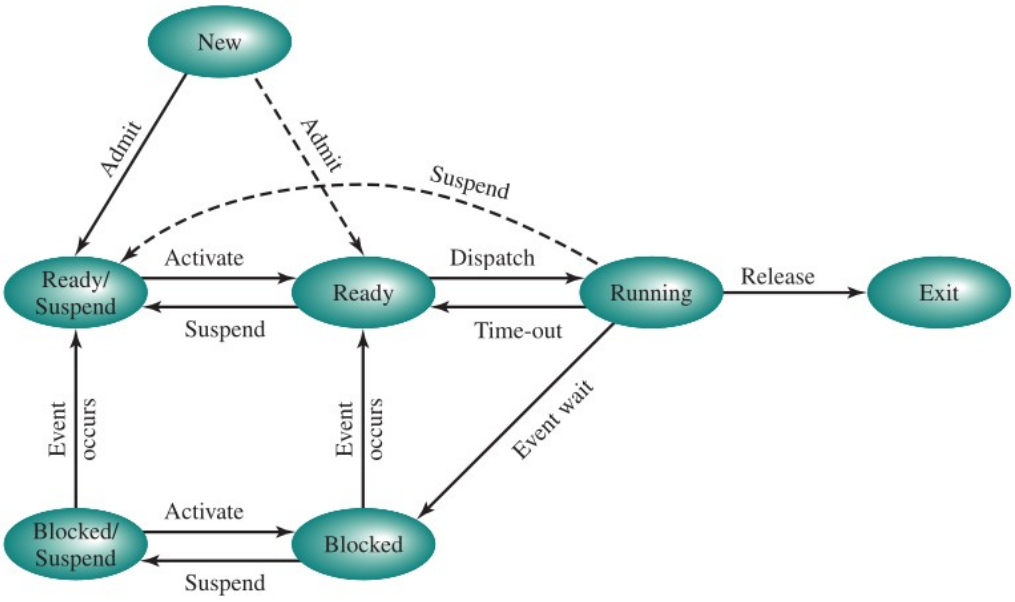
Process Control Block (PCB)

- **Identificador:** un identificador único asociado con el proceso, para distinguirlo de otros procesos
- **Estado:** si el proceso está siendo ejecutado actualmente, está en el estado *running*
- **Prioridad:** el nivel de prioridad con respecto a los otros procesos
- **Program counter:** la dirección de la próxima instrucción a ser ejecutada en el programa
- **Punteros de memoria:** incluye los punteros al código de programa y los datos asociados con el proceso, además de bloques de memoria compartida con otros procesos
- **Datos de contexto:** son los datos presentes en registros en el procesador mientras el proceso está ejecutándose

- **Información de estado de I/O:** incluye las solicitudes I/O pendientes, dispositivos I/O asignados a los procesos, una lista de archivos en uso por el proceso, etc.
- **Información de contabilidad:** puede incluír la cantidad de tiempo de procesador y tiempo de reloj usado, límites de tiempo, contabilidad de números, etc.

La información en la lista anterior es almacenada en una estructura de datos, típicamente llamada process control block, que es creado y administrado por el SO. El punto significativo acerca del process control block es que contiene suficiente información que permite interrumpir un proceso en *running* y luego reanudar su ejecución como si la interrupción nunca hubiese ocurrido. El PCB es la herramienta clave que permite al SO soportar múltiples procesos y proveer para el multiprocesamiento. Cuando un proceso es interrumpido, los valores actuales del program counter y los registros de procesador (datos de contexto) son guardados en los campos apropiados del correspondiente PCB, y el estado del proceso es cambiado a otro valor, como bloqueado o listo. El SO es ahora libre para poner algún otro proceso en el estado de *running*. El program counter y los datos de contexto del proceso son cargados en los registros del procesador, y el proceso ahora comienza a ser ejecutado.

Estados de un proceso



(b) With two Suspend states

Figure 3.9 **Process State Transition Diagram with Suspend States**

1. **Nuevo:** el proceso recién ha sido creado pero aún no ha sido admitido en la pool de procesos ejecutados por el SO. Típicamente, un nuevo proceso aún no ha sido cargado a memoria principal, aunque su PCB (process control block ha sido creado)
2. **Listo:** el proceso está en memoria principal y preparado para ser ejecutado cuando se le de la oportunidad
3. **Listo/Suspendido:** el proceso está en memoria secundaria pero está disponible para la ejecución tan pronto como sea cargado en memoria principal
4. **Corriendo:** el proceso está siendo siendo actualmente ejecutado

5. **Bloqueado:** el proceso está en memoria principal y esperando por un evento
6. **Bloqueado/Suspendido:** el proceso está en memoria secundaria y esperando por evento
7. **Terminado:** el proceso ha sido liberado de la pool de procesos ejecutables por el SO, ya sea porque fue detenido o porque fue abortado por alguna razón.

Transiciones

- **Null → Nuevo:** un nuevo proceso es creado para ejecutar un programa
- **Nuevo → Listo:** el SO moverá un proceso desde el estado *Nuevo* al estado *Listo* cuando esté preparado para tomar un proceso adicional.
- **Listo → Corriendo:** Cuando es tiempo seleccionar un proceso para correr, el SO elige uno de los procesos en el estado *Listo*. Este es el trabajo del planificador o dispatcher.
- **Corriendo → Terminado:** el proceso actualmente corriendo es terminado por SO si el proceso indica que ha completado o si aborta
- **Corriendo → Listo:** la razón más común para esta transición es que el proceso corriendo ha alcanzado el tiempo máximo disponible para la ejecución ininterrumpida.
- **Corriendo → Bloqueado:** un proceso es puesto en estado *Bloqueado* si solicita algo por lo cuál debe esperar. Una solicitud al SO es usualmente en la forma de una llamada al sistema (*system service call*); esto es, una llamada desde el programa corriendo a un procedimiento que es parte del código del sistema operativo.
- **Bloqueado → Listo:** un proceso en el estado *Bloqueado* es movido al estado *Listo* cuando el evento por el cuál estaba esperando ocurre.
- **Listo → Terminado:** por claridad, esta transición no es mostrada en el diagrama. En algunos sistemas, un padre puede terminar un proceso hijo en cualquier momento. Además, si el padre termina, todos los procesos hijos asociados con ese padre pueden ser terminados.
- **Bloqueado → Terminado:** ídem anterior
- **Bloqueado → Bloqueado/Suspendido:** si no hay procesos listos, entonces al menos un proceso bloqueado es *swapped out* (cambiado) para hacer lugar para otros procesos que no están bloqueados. Esta transición puede ser llevada a cabo aún si hay procesos listos disponibles
- **Bloqueado/Suspendido → Listo/Suspendido:** un proceso en el estado *Bloqueado/Suspendido* es movido al estado *Listo/Suspendido* cuando aquel evento por el cual estaba esperando ocurre
- **Listo/Suspendido → Listo:** cuando no hay procesos listos en memoria principal, el SO necesitará traer uno para continuar la ejecución
- **Listo → Listo/Suspendido:** normalmente, el So preferirá suspender un proceso bloqueado en vez de uno listo, porque el proceso listo puede ser ejecutado, mientras que el proceso bloqueado está tomando espacio de memoria principal y no puede ser

ejecutado. Sin embargo, puede ser necesario suspender un proceso listo si es la única manera de liberar un bloque lo suficientemente grande de memoria principal.

- **Nuevo → Listo/Suspendido y Nuevo → Listo:** cuando un nuevo proceso es creado, puede tanto ser agregado a la cola de *Listo* como a la cola de *Listo/Suspendido*.
- **Bloqueado/Suspendido → Bloqueado:** puede ser necesaria en un escenario tal: un proceso termina, liberando algo de memoria principal. Hay un proceso en la cola de *Bloqueado/Suspendido* con mayor pripridad que cualquier otro proceso en la cola de *Listo/Suspendido* y el SO tiene razones para creer que el evento bloqueante para el proceso sucederá pronto.
- **Corriendo → Listo/Suspendido:** normalmente, un proceso corriendo es movido al estado *Listo* cuando su tiempo de asignación expira. Si, sin embargo, el SO está apropiando el proceso porque hay un proceso de mayor prioridad en la cola de *Bloqueado/Suspendido* que recién se ha desbloqueado, el SO puede mover el proceso corriendo directamente a la cola de *Listo/Suspendido* y liberar algo de memoria principal.
- **Cualquier estado → Terminado:** típicamente un proceso termina cuando está corriendo, ya sea porque ha completado o porque una condición de falla fatal. Sin embargo, en algunos sistemas operativos, un proceso puede ser terminado por un proceso que lo creó o cuando un proceso padre es terminado.

2. ¿Cuáles son, normalmente, los sucesos que llevan a la creación de un proceso?

Cuatro principales eventos causan que los procesos sean creados:

1. Inicialización del sistema: cuando un sistema operativo es booteado, típicamente numerosos procesos son creados. Algunos de esos son procesos *foreground*, es decir, procesos que interactúan con usuarios humanos y realizan trabajos para ellos. Otros corren en el *background* y no son asociados a usuarios particulares, sino que en cambio tienen una función específica.
2. Ejecución de una llamada al sistema de creación de proceso de parte de un proceso corriendo: nuevos procesos pueden ser creados luego del tiempo de booteo. Usualmente un proceso corriendo solicitará llamadas al sistema para crear uno o más procesos para ayudarlo en su trabajo.
3. Una solicitud de un usuario de crear un nuevo proceso: en sistemas interactivos, los usuarios pueden lanzar un programa tipeando un comando o clickeando en un ícono. Llevando a cabo cualquiera de esas acciones comienza un nuevo proceso y corre el programa seleccionado en él.
4. Inicio de un trabajo en batch: aplica solamente a los sistemas en batch encontrados en grandes mainframes. Los usuarios pueden sumar trabajos en batch al sistema (posiblemente de manera remota). Cuando el sistema operativo decide que tiene recursos para correr otro trabajo, crea un nuevo proceso y corre el próximo trabajo desde la cola de entrada.

Técnicamente, en todos estos casos, un nuevo proceso es creado teniendo un proceso existente que ejecuta una llamada al sistema de creación del proceso. Lo que este proceso

hace es ejecutar una llamada al sistema para crear un nuevo proceso. Esta llamada al sistema le dice al sistema operativo que cree un nuevo proceso e indica, directa o indirectamente, qué programa correr en él.

3. ¿Cuáles son los pasos que lleva a cabo un sistema operativo para crear un nuevo proceso?

Cuando un nuevo proceso es añadido a los que están actualmente siendo administrados, el SO construye las estructuras de datos usadas para administrar el proceso, y asigna el espacio en memoria principal para el proceso. *(Stallings, p. 137)*

Durante el curso de su ejecución, un proceso puede crear varios nuevos procesos. El proceso creados es llamado proceso padre y los nuevos procesos son llamados hijos de ese proceso.

La mayoría de los sistemas operativos identifican los procesos acorde a un único identificador de proceso (*pid*), que es típicamente un número entero. El pid provee un único valor para cada proceso en el sistema, y puede ser usado como un índice para acceder a varios atributos de un proceso. *(Silberschartz, p 116)*

Aquí están los pasos típicos que un sistema operativo lleva a cabo para crear un nuevo proceso:

1. **Asignación de un identificador de proceso (PID):** el sistema operativo asigna un identificador único (PID) al nuevo proceso. Este PID se utiliza para rastrear y gestionar el proceso durante su vida.
2. **Asignación de recursos:** el sistema operativo asigna los recursos necesarios al nuevo proceso, como memoria, archivos y dispositivos de E/S. Esto puede implicar la asignación de una estructura de control de proceso (PCB) que contendrá toda la información sobre el proceso.
3. **Inicialización de la tabla de procesos:** Se crea una entrada en la tabla de procesos (o lista de procesos) del sistema operativo. Esta entrada incluye el PCB del nuevo proceso, que contiene información como el estado del proceso, registros de la CPU, información de memoria, etc.
4. **Cargar el programa en memoria: e**l código ejecutable del programa se carga en la memoria principal desde el almacenamiento secundario. Esto puede implicar la asignación de espacio de direcciones en la memoria virtual y la carga de las páginas iniciales del programa.
5. **Configuración del contexto inicial: s**e establece el contexto inicial del proceso, incluyendo la inicialización del contador de programa (PC) al punto de inicio del programa y la configuración de los registros de la CPU y el puntero de pila.
6. **Configuración del espacio de direcciones: s**e configura el espacio de direcciones del proceso, que puede incluir la asignación de tablas de páginas en sistemas con memoria virtual y la inicialización de segmentos de código, datos y pila.
7. **Añadir el proceso a la cola de procesos listo:** el nuevo proceso se coloca en la cola de procesos listos para ejecutarse. Esta cola es gestionada por el planificador del sistema operativo. *(ChatGPT 4o)*

5. Describa las diferencias entre planificadores de corto, mediano y largo término.

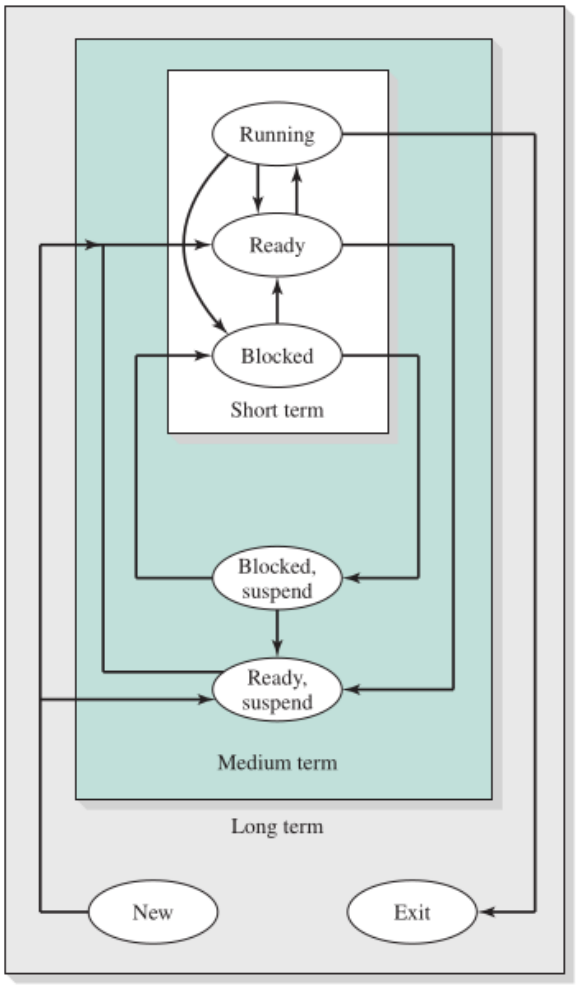


Figure 9.2 **Levels of Scheduling**

En términos de frecuencia de ejecución, el planificador de largo término se ejecuta relativamente de manera infrecuente y hace la decisión general de si tomar o no un nuevo proceso y cuál tomar. El planificador de medio término es ejecutado de manera un poco más frecuente y toma la decisión de swapping. El planificador de corto término, también conocido como dispatcher, se ejecuta frecuentemente y toma las decisiones específicas de qué proceso ejecutar a continuación.

- **Planificador de largo término:** determina qué programas son admitidos en el sistema para ser procesados. Así controla el grado de multiprogramación. Una vez admitido, un trabajo o programa de usuario se convierte en un proceso y es añadido a la cola del planificador de corto término. En algunos sistemas, los nuevo procesos creados empiezan a una condición swapped-out, en ese caso es añadido a la cola del planificador de mediano término.

  La decisión de cuando crear un nuevo proceso es generalmente determinado el grado deseado de multiprogramación. Cuantos más procesos son creados, menor es el porcentaje de tiempo que cada proceso puede ser ejecutado (es decir, más proceso están compitiendo por el mismo monto de tiempo de procesador). Entonces, el planificador de largo término puede limitar el grado de multiprogramación para

proveer un servicio satisfactorio para el conjunto actual de procesos. Cada vez que un trabajo termina, el planificador puede decidir añadir uno a más trabajos nuevos. Adicionalmente, si la fracción de tiempo que el procesador puede estar inactivo excede cierto límite, el planificador de largo término puede ser invocado.

- **Planificador de medio término:** forma parte de la función de swapping. Típicamente, la decisión de swapping-in se basa en la necesidad de administrar el grado de multiprogramación.
- **Planificador de corto término:** es invocado cuando sea que ocurra un evento que puede llevar al bloqueo del proceso actual, o que puede proveer una oportunidad de apropiar (*preempt*) un proceso actualmente corriendo en favor de otro. Ejemplos de esos eventos incluyen:
  - Interrupciones de reloj
  - Interrupciones I/O
  - Llamadas al sistema
  - Señales (por ej. semáforos)

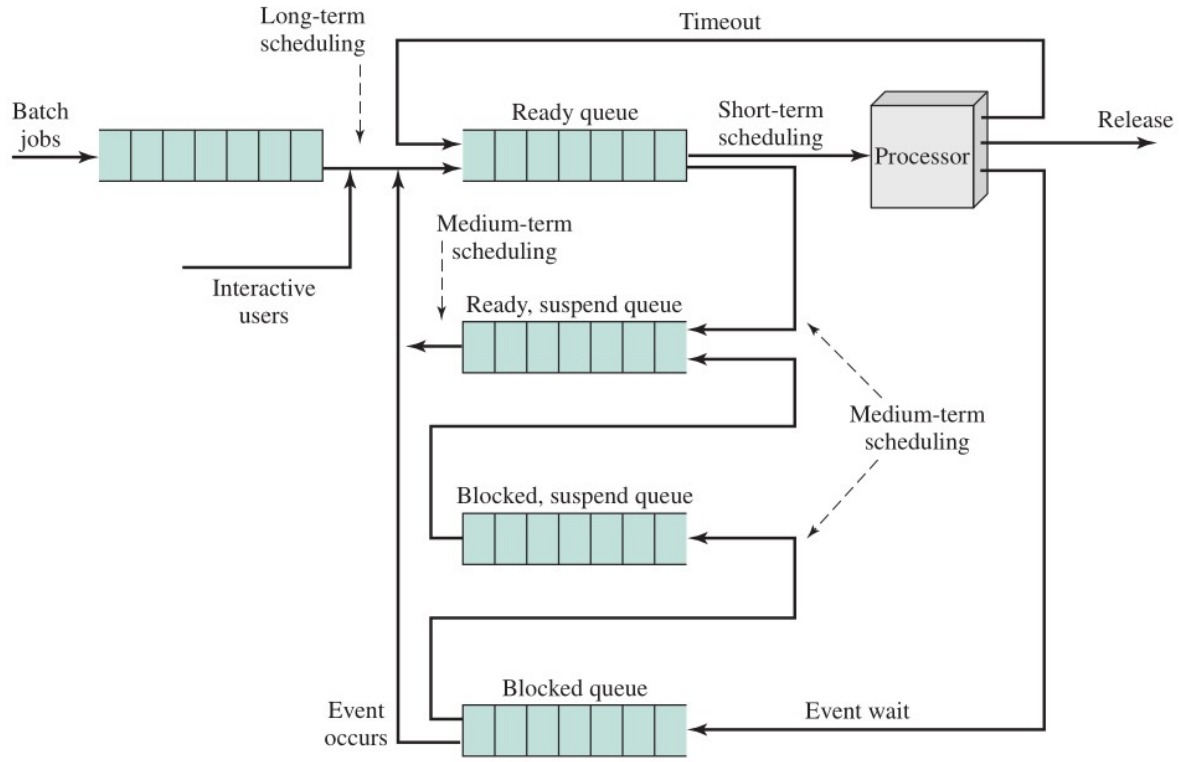


Figure 9.3 **Queueing Diagram for Scheduling**

Planificación de largo término	La decisión de añadir a la lista de procesos a ser ejecutados
Planificación de medio término	La decisión de añadir a la cantidad de procesos que están parcial o completamente en memoria principal
Planificación de corto término	La decisión de qué proceso disponible será ejecutado por el procesador

8. Describa las acciones que realiza el kernel cuando cambia el contexto entre procesos.

Cambiar el núcleo de la CPU a otro proceso requiere realizar un guardado del estado del proceso actual y una restauración de otro proceso. Esta tarea es conocida como cambio de contexto.

El contexto es representado en el PCB del proceso. Incluye el valor de los registros de CPU, el estado del proceso e información de administración de memoria. Generalmente realizamos un guardado de información del estado actual del núcleo de CPU, ya sea que esté en modo kernel o modo usuario, y luego una restauración del estado para reanudar las operaciones.

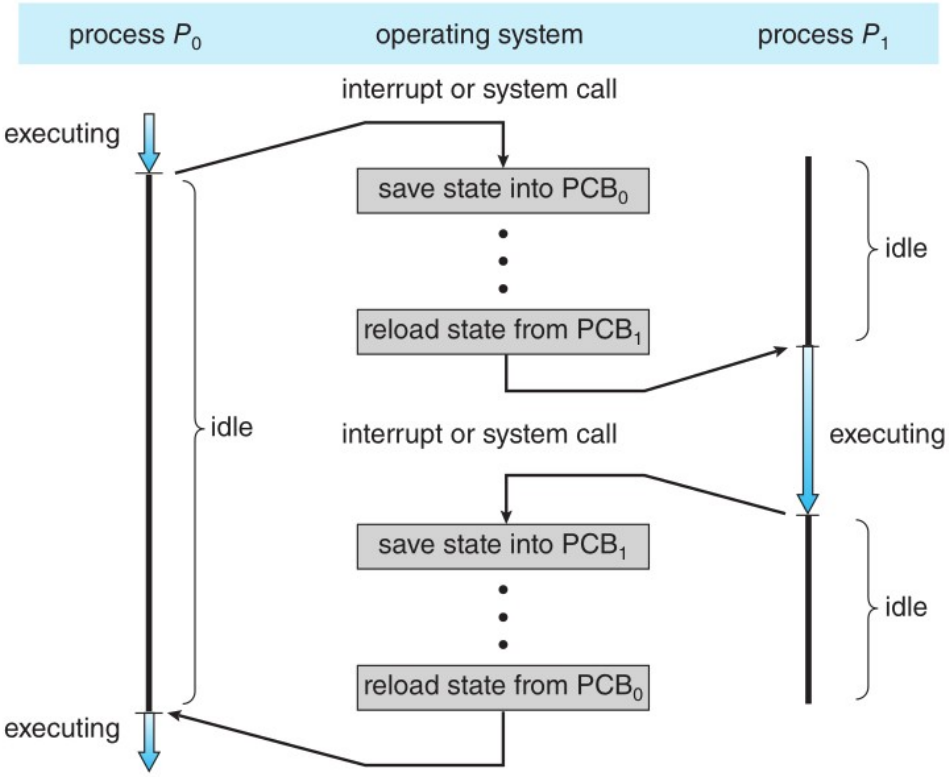


Figure 3.6 Diagram showing context switch from process to process.

Cuando un cambio de contexto ocurre, el kernel guarda el contexto del proceso anterior en su PCB y carga el contexto guardado del nuevo proceso planificado para correr.

El tiempo cambio de contexto es puro malgasto (overhead) porque el sistema no reliza trabajo útil mientras cambia.

9. ¿Un proceso puede esperar simultáneamente a más de un suceso ó evento? Justifique su respuesta.

No, un proceso no puede esperar simultáneamente a más de un suceso o evento de manera directa, ya que cuando un proceso espera por un evento, el sistema operativo lo coloca en una cola de espera asociada a ese evento específico. Si un proceso tuviera que esperar simultáneamente por múltiples eventos, el manejo de estas colas se volvería más complejo y menos eficiente.

Sin embargo, se puede simular una espera simultanea por dos eventos usando hilos. Usar múltiples hilos dentro de un proceso permite que cada hilo espere a diferentes eventos. De esta manera, el proceso en su conjunto puede manejar múltiples eventos concurrentemente.

11. ¿Cómo se pueden comunicar los procesos? Brinde un ejemplo en el cual considere necesario que 2 procesos se comuniquen.

Los procesos cooperativos requieren un mecanismo de comunicación inter-procesos (IPC) que les permitirá intercambiar información, esto es, enviar datos y recibir datos uno con el otro. Hay dos modelos fundamentales de comunicación inter-procesos: memoria compartida e intercambio de mensajes.

- Memoria compartida: en el modelo de memoria compartida, es establecida una región de memoria es compartida entre los procesos cooperativos. Los procesos entonces pueden intercambiar información leyendo y escribiendo esa región compartida.
- Intercambio de mensajes: en el modelo de intercambio de mensajes la comunicación toma lugar por medio de mensajes intercambiados entre los procesos cooperativos.

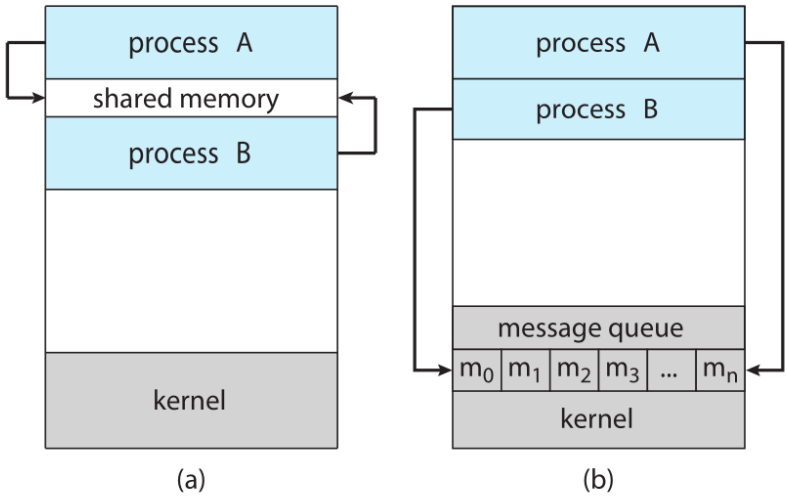


Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

12. Considere la comunicación sincrónica y asincrónica entre procesos. Presente ventajas y desventajas de cada una.

Comunicación Sincrónica

Ventajas:

- **Simplicidad en la sincronización:** En la comunicación sincrónica, los procesos que se comunican están sincronizados. Esto significa que uno de los procesos esperará hasta que el otro esté listo para intercambiar datos, lo que simplifica la sincronización y puede evitar problemas de condición de carrera.

- **Consistencia de datos:** La transmisión de datos es directa y consistente, ya que ambos procesos están en espera hasta que el intercambio de información esté completo.
- **Facilidad de implementación:** En algunos casos, la implementación puede ser más sencilla porque no es necesario manejar buffers intermedios o verificar el estado de los mensajes.

Desventajas:

- **Bloqueo:** Los procesos están bloqueados hasta que ambos estén listos para comunicarse, lo que puede llevar a un uso ineficiente de los recursos. Si un proceso está ocupado o se retrasa, el otro proceso debe esperar.
- **Rendimiento:** Puede afectar negativamente el rendimiento del sistema si los procesos pasan mucho tiempo esperando uno al otro.

Comunicación Asincrónica

Ventajas:

- **No bloqueo:** Los procesos no necesitan esperar uno por el otro para enviar o recibir datos, lo que permite que continúen ejecutándose sin interrupciones. Esto mejora la eficiencia y el uso de recursos.
- **Mejor rendimiento:** Puede aumentar el rendimiento del sistema ya que los procesos pueden continuar con otras tareas mientras esperan la comunicación.
- **Flexibilidad:** Es más flexible y puede manejar mejor los casos en los que los procesos tienen diferentes tiempos de ejecución o tiempos de respuesta.

Desventajas:

- **Complejidad de sincronización:** La implementación puede ser más compleja debido a la necesidad de manejar buffers y posibles problemas de concurrencia, como condiciones de carrera y bloqueos.
- **Consistencia de datos:** Puede haber problemas con la consistencia de los datos si no se manejan adecuadamente los estados de los mensajes y las colas de comunicación. *(ChatGPT 4o)*

14. ¿Cuáles son las características más importantes de los threads? ¿En qué casos los utilizaría? ¿Cuál es la diferencia de los threads a nivel kernel y los que son implementados a nivel usuario?

Características

Las características principales de los threads son:

- **Capacidad de respuesta (responsiveness):** programar una aplicación interactiva con múltiples hilos puede permitir al programa continuar corriendo aún si una parte está bloqueada o está realizando una operación larga, aumentando de este modo la capacidad de respuesta (responsiveness) para el usuario.

- **Compartir recursos:** los procesos pueden compartir recursos solamente a través de técnicas tales como memoria compartida e intercambio de mensajes. Tales técnicas debe ser explícitamente arregladas por el programador. Sin embargo, los threads comparten por defecto la memoria y los recursos del proceso al que pertenecen. El beneficio de compartir código y datos es que ello permite a una aplicación tener varios diferentes threads en actividad con el mismo espacio de memoria.
- **Economía:** asignar memoria y recursos para la creación de un proceso es costoso. Como los threads comparten los recursos del proceso al cual pertenecen, es más económico crear y cambiar de contexto threads.
- **Escalabilidad:** los beneficios del multi-hilado (multithreading) pueden ser incluso más grandes en una arquitectura multiprocesador, donde los threads pueden estar corriendo en paralelo en diferentes núcleos. Un proceso con un sólo thread puede correr solamente en un procesador, sin importar cuántos haya disponibles.

Comparación entre threads de kernel y de usuario

Hilos a Nivel Kernel (Kernel-Level Threads):

- **Gestión:** Estos hilos son gestionados directamente por el sistema operativo (kernel).
- **Visibilidad:** El kernel conoce la existencia de cada hilo y puede programarlos individualmente.
- **Contexto de Ejecución:** Los hilos a nivel kernel tienen su propio contexto de ejecución en el kernel, lo que incluye registros, pila y espacio de direcciones.
- **Cambio de Contexto:** El cambio de contexto entre hilos a nivel kernel es más costoso en términos de rendimiento porque implica cambiar el contexto del kernel.
- **Soporte Multiprocesador:** Son ideales para sistemas multiprocesadores porque el kernel puede asignar hilos a diferentes CPUs.
- **Bloqueo:** Si un hilo a nivel kernel se bloquea (por ejemplo, esperando una I/O), el kernel puede programar otro hilo para que se ejecute.

Hilos a Nivel Usuario (User-Level Threads):

- **Gestión:** Estos hilos son gestionados por una biblioteca de hilos en el espacio de usuario, no por el kernel.
- **Visibilidad:** El kernel no sabe de la existencia de estos hilos, solo ve un proceso único.
- **Contexto de Ejecución:** Los hilos a nivel usuario tienen su propio contexto en el espacio de usuario, pero comparten el mismo contexto del proceso del kernel.
- **Cambio de Contexto:** El cambio de contexto entre hilos a nivel usuario es más rápido porque no implica cambiar el contexto del kernel.
- **Soporte Multiprocesador:** Tienen limitaciones en sistemas multiprocesadores porque el kernel solo ve un proceso, por lo que no puede distribuir hilos a diferentes CPUs.

- **Bloqueo:** Si un hilo a nivel usuario se bloquea, todo el proceso se bloquea porque el kernel no sabe de la existencia de otros hilos dentro del mismo proceso.

Comparación y Consideraciones:

- **Rendimiento:** Los hilos a nivel usuario pueden ser más rápidos en operaciones de cambio de contexto y sincronización porque no requieren intervención del kernel.
- **Flexibilidad:** Los hilos a nivel kernel pueden aprovechar mejor las capacidades de hardware, como múltiples CPUs, y manejar mejor las situaciones en las que los hilos pueden bloquearse.
- **Facilidad de Implementación:** Los hilos a nivel usuario pueden ser más fáciles de implementar en algunos casos, pero pueden requerir una gestión más compleja por parte del desarrollador para manejar situaciones de bloqueo y concurrencia. *(ChatGPT 4o)*

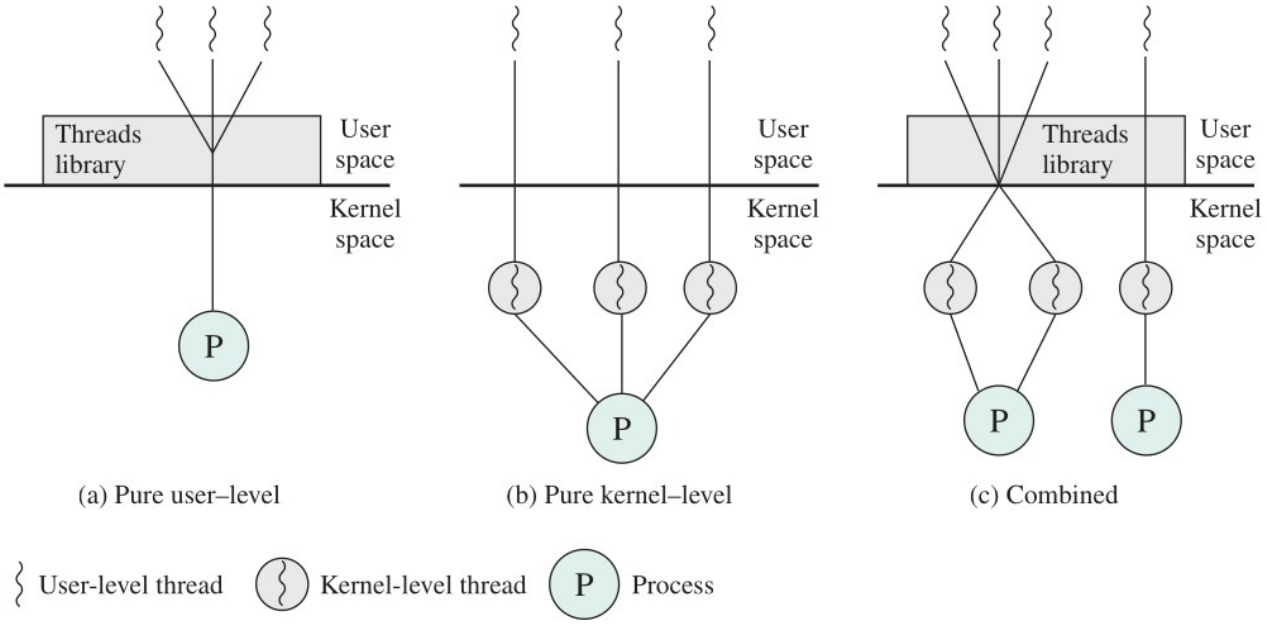


Figure 4.5 User-Level and Kernel-Level Threads

15. ¿Cuáles recursos son utilizados cuando un thread es creado? ¿y en el caso de un proceso?

Creación de un Thread

Un thread es una unidad básica de ejecución dentro de un proceso. Los threads comparten el mismo espacio de direcciones del proceso y algunos otros recursos del proceso, pero cada thread tiene sus propios recursos específicos. Al crear un thread, se utilizan los siguientes recursos:

- **Pila del thread:** cada thread tiene su propia pila para almacenar variables locales, direcciones de retorno y otros datos necesarios para la ejecución de sus funciones.
- **Registro de contexto:** cada thread tiene su propio conjunto de registros de CPU (por ejemplo, contador de programa, registros de datos, etc.) que deben ser mantenidos por el sistema operativo.

- **Estructura de control de thread:** el sistema operativo mantiene información sobre el thread en una estructura de datos, que puede incluir el estado del thread, su prioridad, información de planificación, etc.
- **Recursos de sincronización:** a veces se necesitan recursos adicionales para manejar la sincronización entre threads, como mutexes, semáforos, etc.

Creación de un Proceso

Un proceso es una instancia de un programa en ejecución y tiene su propio espacio de direcciones. Al crear un proceso, se utilizan muchos más recursos que al crear un thread, ya que los procesos son entidades más independientes. Los recursos utilizados son:

- **Espacio de direcciones:** cada proceso tiene su propio espacio de direcciones virtual, lo que significa que tiene su propio conjunto de segmentos de memoria (código, datos, pila, etc.).
- **Pila del proceso:** similar a los threads, pero en el caso de procesos, la pila es parte del espacio de direcciones separado del proceso.
- **Registro de contexto:** similar a los threads, pero para todo el proceso. Incluye todos los registros de CPU relevantes.
- **Estructura de Control de Proceso (PCB - Process Control Block):** es una estructura que mantiene toda la información necesaria para gestionar el proceso, como estado del proceso, contador de programa, punteros a tablas de páginas, etc.
- **Recursos de comunicación:** si el proceso necesita comunicarse con otros procesos, puede requerir recursos adicionales como pipes, colas de mensajes, etc.

En resumen, la creación de un proceso es más costosa en términos de recursos del sistema en comparación con la creación de un thread. Esto se debe a que los procesos son entidades más aisladas y completas, con su propio espacio de direcciones y recursos independientes. Los threads, en cambio, son más ligeros porque comparten el espacio de direcciones del proceso padre y otros recursos, aunque requieren su propia pila y contexto de registro. *(ChatGPT 4o)*

Sistemas Operativos

Trabajo Práctico N.º 3: Sincronización de procesos

NOTA: Hacer los ejercicios prácticos de semáforos

3. Explique la diferencia entre la espera ocupada y el bloqueo.

La espera ocupada malgasta ciclos de CPU que otro proceso podría usar productivamente. Su ciclo continuo es claramente un problema en un sistema de multiprogramación real, donde un único núcleo CPU es compartido entre varios procesos.

EL bloqueo tiene una ventaja ya que no requiere un cambio de contexto cuando un proceso debe esperar por un bloqueo de la sección crítica, y el cambio de contexto puede tomar un tiempo considerable. (Silberschartz, p. 272)

La espera ocupada y el bloqueo son dos técnicas utilizadas en la programación para controlar el acceso a recursos compartidos entre procesos o hilos. Aquí tienes una explicación de ambas:

Espera ocupada (Busy Waiting):

- En la espera ocupada, un proceso o hilo verifica continuamente el estado de un recurso compartido hasta que esté disponible.
- Esto se logra típicamente mediante un bucle de espera activa donde el proceso o hilo consulta repetidamente el estado del recurso.
- La espera ocupada consume recursos de manera ineficiente, ya que el proceso o hilo está activamente ejecutándose y verificando el estado del recurso en lugar de esperar de manera pasiva.

Bloqueo (Locking):

- El bloqueo implica el uso de mecanismos de sincronización, como semáforos, mutex (exclusión mutua) o cerrojos, para controlar el acceso a recursos compartidos.
- Cuando un proceso o hilo necesita acceder a un recurso compartido, primero adquiere el bloqueo asociado con ese recurso.
- Si el bloqueo no está disponible (es decir, está siendo utilizado por otro proceso o hilo), el proceso o hilo que lo solicita se bloquea y espera hasta que el bloqueo esté disponible.
- Una vez que el proceso o hilo ha terminado de utilizar el recurso, libera el bloqueo para que otros procesos o hilos puedan utilizarlo.
- El bloqueo garantiza la exclusión mutua, lo que significa que solo un proceso o hilo puede acceder al recurso compartido en un momento dado, lo que evita condiciones de carrera y garantiza la consistencia de los datos.

En resumen, la diferencia clave entre la espera ocupada y el bloqueo es que la espera ocupada implica una verificación activa y continua del estado del recurso, mientras que el bloqueo utiliza mecanismos de sincronización para permitir que los procesos o hilos esperen de manera pasiva hasta que el recurso esté disponible. El bloqueo es generalmente más eficiente y preferible en la mayoría de los casos, ya que evita la sobrecarga de CPU asociada con la espera ocupada. (ChatGPT 3.5)

4. Muestre que si wait y signal no son ejecutadas atómicamente, la exclusión mutua puede ser violada.

Supongamos que tenemos dos procesos, A y B, que comparten un recurso crítico y utilizan un semáforo para sincronizarse. El semáforo inicialmente se establece en 1, lo que significa que el recurso está disponible. Ambos procesos intentan acceder al recurso crítico de la siguiente manera:

Proceso A:

1. Espera ocupada: Verifica el semáforo y si está ocupado espera.
2. Cuando el semáforo se desocupa, intenta adquirir el semáforo (operación *wait*).
3. Accede al recurso crítico.
4. Libera el recurso crítico (operación *signal*).

Proceso B:

1. Espera ocupada: Verifica el semáforo y si está ocupado espera.
2. Cuando el semáforo se desocupa, intenta adquirir el semáforo (operación *wait*).
3. Accede al recurso crítico.
4. Libera el recurso crítico (operación *signal*).

Ahora, consideremos lo que sucede si las operaciones *wait* y *signal* no son atómicas y pueden ser interrumpidas en el medio:

1. El proceso A verifica el semáforo y encuentra que está disponible (valor 1).
2. Antes de que el proceso A pueda ejecutar la operación *wait*, el planificador del sistema operativo cambia al proceso B.
3. El proceso B verifica el semáforo y también encuentra que está disponible.
4. El proceso B ejecuta la operación *wait*, decrementando el valor del semáforo a 0.
5. El planificador del sistema operativo cambia nuevamente al proceso A.
6. El proceso A ejecuta la operación *wait*, decrementando nuevamente el valor del semáforo a -1 (lo cual no debería ocurrir).
7. Ambos procesos acceden al recurso crítico simultáneamente, violando la exclusión mutua.

Esta violación ocurre porque las operaciones *wait* y *signal* no son atómicas y pueden ser interrumpidas en el medio, lo que permite que otro proceso entre en la sección crítica antes de que se complete la operación de adquisición del semáforo. Para garantizar la exclusión

mutua, es crucial que estas operaciones sean atómicas o se utilicen otras técnicas de sincronización que proporcionen atomicidad, como los mutex.

7. ¿Pueden dos hilos (threads) en el mismo proceso sincronizarse mediante un semáforo de kernel, si los hilos son implementados en el kernel? ¿Qué pasa si se implementan en espacio de usuario? Suponga que ningún hilo de otro proceso tiene acceso al semáforo. Analice sus respuestas.

En un sistema operativo típico, los semáforos de kernel son recursos compartidos a nivel de sistema que pueden ser utilizados para la sincronización entre procesos o hilos, independientemente de si están implementados en el espacio del kernel o del usuario. Sin embargo, hay algunas consideraciones importantes dependiendo de dónde se implementen los hilos:

Implementación de hilos en el kernel:

- Si los hilos están implementados en el kernel, pueden sincronizarse utilizando un semáforo de kernel. Esto se debe a que los hilos del mismo proceso comparten el mismo espacio de direcciones en el kernel y pueden acceder a los recursos del kernel, como los semáforos de kernel, directamente sin necesidad de cambiar al espacio de direcciones del proceso.
- En este caso, la sincronización mediante un semáforo de kernel sería efectiva y adecuada para garantizar la exclusión mutua y coordinar el acceso a recursos compartidos entre los hilos dentro del mismo proceso.

Implementación de hilos en el espacio de usuario:

- Si los hilos están implementados en el espacio de usuario, la situación cambia. En este caso, los hilos no pueden acceder directamente a los recursos del kernel, incluidos los semáforos de kernel, sin realizar una llamada al sistema.
- Si se utiliza un semáforo de kernel para sincronizar hilos en el espacio de usuario, cada operación de espera (`wait`) y señal (`signal`) requerirá una transición del modo usuario al modo kernel y viceversa. Esta transición de modo es costosa en términos de rendimiento.
- Una opción más eficiente en este caso sería utilizar semáforos implementados en el espacio de usuario o técnicas de sincronización más livianas, como variables de condición o mutexes implementados en el espacio de usuario.

En resumen, los hilos en el mismo proceso pueden sincronizarse utilizando un semáforo de kernel si están implementados en el kernel, ya que pueden acceder directamente a los recursos del kernel. Sin embargo, si están implementados en el espacio de usuario, el uso de semáforos de kernel puede ser ineficiente debido a la necesidad de realizar transiciones entre modos para acceder a ellos, por lo que sería preferible utilizar mecanismos de sincronización implementados en el espacio de usuario. (ChatGPT 3.5)

8. ¿Cuáles son las diferencias entre un semáforo binario y un mutex?

Tanto los semáforos binarios como los mutex son mecanismos de sincronización utilizados para garantizar la exclusión mutua en la programación concurrente, pero tienen algunas diferencias importantes:

Semáforo binario:

- Un semáforo binario es un tipo especial de semáforo que solo puede tener dos estados: 0 y 1.
- Se utiliza típicamente para controlar el acceso a recursos compartidos entre procesos o hilos.
- Un semáforo binario se puede utilizar para implementar la exclusión mutua, donde el recurso compartido solo puede ser accedido por un proceso o hilo a la vez.

Mutex (Exclusión Mutua):

- Un mutex (abreviatura de "mutual exclusion") es un mecanismo de sincronización que garantiza que solo un hilo pueda acceder a un recurso compartido a la vez.
- A diferencia de un semáforo binario, un mutex es más simple y está diseñado específicamente para la exclusión mutua.
- Un mutex tiene dos estados: bloqueado y desbloqueado.
- Cuando un hilo adquiere un mutex, bloquea el mutex para evitar que otros hilos accedan al recurso compartido.
- Solo el hilo que ha bloqueado el mutex puede liberarlo.
- Los mutex suelen ser más eficientes que los semáforos binarios porque están optimizados para operaciones de bloqueo y desbloqueo en un solo hilo.

En resumen, mientras que ambos semáforos binarios y mutex pueden utilizarse para lograr la exclusión mutua, los mutex son más específicos y optimizados para esta tarea, mientras que los semáforos binarios son más versátiles y pueden utilizarse para una variedad de situaciones de sincronización y control de acceso a recursos compartidos. (ChatGPT 3.5)

Sistemas Operativos

Trabajo Práctico N.º 4: Planificación de procesos

Notas

- Planificador de CPU: el rol del planificador de CPU (CPU scheduler) es seleccionar de entre los procesos que están en la cola de listos y asignarlo a un núcleo del CPU. El planificador de CPU se ejecuta al menos una vez cada 100 milisegundos
- Cambio de contexto: cuando sucede una interrupción, el sistema necesita guardar el contexto actual del proceso corriendo en la CPU para así poder restaurar ese contexto cuando termine su procesamiento, esencialmente suspendiendo el proceso y luego reanudándolo. El contexto es representado en el PCB (Process Control Block) del proceso. Incluye el valor de los registros del CPU, el estado del proceso e información de la administración de memoria.

  Cambiar el CPU a otro proceso requiere realizar un guardado de estado del proceso actual y restaurar el estado de un proceso diferente. Esta tarea es conocida como *cambio de contexto*. Cuando sucede un cambio de contexto, el kernel guarda el contexto del proceso anterior en su PCB y carga el contexto guardado del nuevo proceso planificado para ser corrido. El cambio de contexto es puramente costo (overhead), ya que el sistema no realiza trabajo útil en el cambio.

Preguntas

1. Explique un esquema de planificación multinivel.

El algoritmo de planificación de cola multinivel con realimentación (multilevel feedback queue) permite que un proceso se mueva entre colas. La idea es separar los procesos de acuerdo a características de sus ráfagas de CPU. Si un proceso usa mucho tiempo de CPU, será movido a una cola de menor prioridad. Un proceso que espera mucho tiempo en una cola de baja prioridad puede ser movido a una cola de mayor prioridad. De esta forma, se evita la inanición.

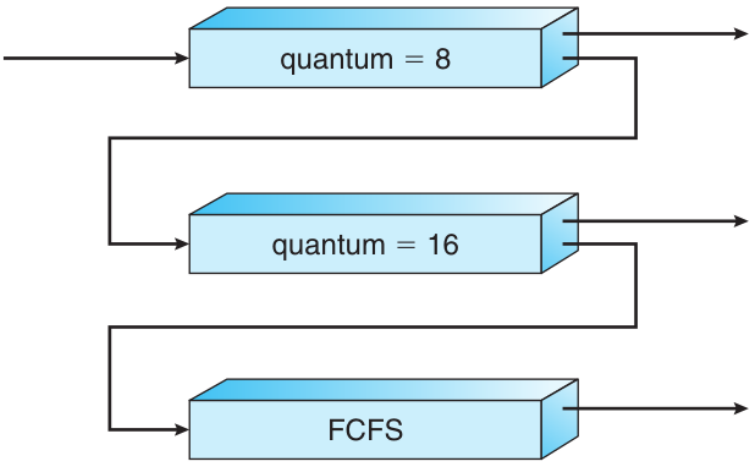


Figure 5.9 Multilevel feedback queues.

Por ejemplo, consideremos un planificador de cola multinivel con realimentación con tres colas, numeradas de 0 a 2. Este planificador primero ejecuta todos los procesos en la cola 0. Solamente cuando la cola 0 está vacía ejecutará los procesos en la cola 1. Similarmente, los procesos en la cola 2 será ejecutados solamente si las colas 0 y 1 están vacías. Un proceso que arriba a la cola 1 se adelantará a un proceso en la cola 2. Un proceso en la cola 1 a su vez será adelantado por un proceso que arribe a la cola 0.

Un proceso entrante es puesto en la cola 0. A un proceso en a cola 0 se le da un quantum de 8 milisegundos. Si no finaliza en ese tiempo, es movido al final de la cola 1. Si la cola 0 está vacía, al proceso en la cabeza de la cola 1 se le da un quantum de 16 milisegundos. Si no se completa, es adelantado y es puesto en la cola 2. Los procesos en la cola 2 son corridos en base a FCFS, pero solamente cuando las colas 0 y 1 están vacías. Para prevenir inanición, un proceso que espera un tiempo largo en una cola de baja prioridad puede ser gradualmente movido a una cola de mayor prioridad.

4. ¿Es posible que ocurra la apropiación de un hilo (thread) mediante una interrupción de reloj? De ser así, ¿bajo qué circunstancias? Si no es así ¿por qué no?

Si. En una planificación con Round Robin y con un LWP (lightweight process), es decir, donde un thread corre sobre un proceso, la apropiación del CPU se hace bajo interrupciones de reloj que indican cuando el tiempo de ejecución para un proceso terminó.

5. En un sistema que soporte hilos (threads), ¿cómo son planificados estos frente al resto de los procesos?

En la mayoría de los sistemas operativos modernos son los threads a nivel de kernel (no los procesos) los que son planificados por el sistema operativo. Los threads a nivel de usuario son administrados por una librería de thread y el kernel los pasa por alto.

Para correr en la CPU, los thread a nivel de usuario deben ser mapeados en última instancia a un thread a nivel de kernel asociado, aunque este maoe puede ser indirecto y puede usar un LWP (lightweight process). En sistemas que implementan modelos muchos-a-uno y muchos-a-muchos, la librería de thread planifica los threads a nivel de usuario para que corran en un LWP disponible. Este esquema es conocido como *process-contention scope* (PCS, alcance de contención del proceso), ya que la competencia por la CPU toma lugar entre los threads pertenecientes al mismo proceso. Típicamente, PCS es hecho de acuerdo a la prioridad: el planificador selecciona el thread ejecutable con la mayor prioridad para ser corrido. Las prioridad de los thread a nivel de usuario son asignados por programados y no son ajustados por la librería de thread. Es importante notar que PCS tipicamente adelantará al thread corriendo actualmente en favor de un thread de mayor prioridad.

Para decidir qué thread a nivel de kernel será planificado en la CPU, el kernel usa *system-contention scope* (SCS, alcance de contención de sistema). Con SCS la competición por el CPU toma lugar entre todos los threads del sistema.

6. ¿Puede presentarse el problema de inversión de prioridades con hilos (threads) en el nivel de usuarios y a nivel de kernel? Justifique su respuesta.

El problema de inversión de prioridades puede presentarse tanto a nivel de usuarios como a nivel de kernel, aunque las causas y las implicaciones pueden variar en cada caso.

1. A nivel de usuario:

En un entorno de usuario, donde los hilos son gestionados por una biblioteca de hilos en el espacio de usuario y no por el kernel del sistema operativo, el problema de inversión de prioridades puede surgir cuando un hilo de alta prioridad está bloqueado esperando un recurso que está siendo utilizado por un hilo de baja prioridad.

Por ejemplo, supongamos que un hilo de alta prioridad necesita acceder a un recurso compartido, pero ese recurso está siendo utilizado por un hilo de baja prioridad. Si el hilo de baja prioridad no libera el recurso de manera oportuna (debido a, por ejemplo, un largo período de tiempo de espera o una operación de E/S prolongada), el hilo de alta prioridad puede quedar bloqueado detrás del hilo de baja prioridad, lo que resulta en una inversión de prioridades.

2. A nivel de kernel:

A nivel de kernel, donde el sistema operativo gestiona directamente los hilos y procesos, el problema de inversión de prioridades puede ocurrir cuando un hilo de alta prioridad es bloqueado por un recurso que está siendo utilizado por un hilo de baja prioridad.

Por ejemplo, en un sistema operativo multitarea preemptivo, un hilo de alta prioridad puede ser interrumpido por el sistema operativo y bloqueado mientras espera un recurso (como un bloqueo de exclusión mutua) que está siendo utilizado por un hilo de baja prioridad. Si el hilo de baja prioridad no libera el recurso de manera oportuna, puede causar que el hilo de alta prioridad quede bloqueado detrás de él, lo que resulta en una inversión de prioridades.

En ambos casos, el problema de inversión de prioridades puede provocar un rendimiento subóptimo y violar las expectativas de prioridad de los hilos. Para mitigar este problema, se pueden utilizar técnicas como el uso de semáforos con prioridad, la programación de bloqueos con inversión de prioridades (como el protocolo de herencia de prioridades) o el diseño cuidadoso de la aplicación para evitar situaciones donde los hilos de alta prioridad se vean bloqueados por hilos de baja prioridad.

9. Defina las diferencias entre planificación con apropiación y sin apropiación. ¿Por qué un algoritmo sin apropiación no es comúnmente utilizado en centros de cómputos?

Las decisiones de planificación de CPU toman lugar bajo las siguientes cuatro circunstancias:

1. Cuando un proceso cambia desde el estado de *running* al estado *waiting* (por ejemplo, como resultado de una solicitud de I/O o la invocación de un `wait()` para la finalización de un proceso hijo)
2. Cuando un proceso cambia del estado *running* al estado *ready* (por ejemplo, cuando ocurre una interrupción)
3. Cuando un proceso cambia desde el estado *waiting* al estado de *ready* (por ejemplo, al completar la I/O)

4. Cuando un proceso finaliza

Para las situaciones 1 y 4, no ha elección en términos de planificación. Un nuevo proceso (si uno existe en la cola de *ready*) debe ser seleccionado para la ejecución. Pero hay opciones para las situación 2 y 3.

Cuando la planificación toma lugar sólo bajo las circunstancias 1 y 4, decimos que el esquema de planificación es no apropiativo (*nonpreemptive*) o cooperativo. De lo contrario, es apropiativo (*preemptive*). Bajo la planificación no apropiativa, una vez que la CPU ha sido asignada a un proceso, el proceso mantiene la CPU hasta que la libera finalizando o cambiando a un estado de *waiting*.

Desafortunadamente, la planificación apropiativa puede resultar en condiciones de carrera cuando datos son compartidos entre varios procesos. Consideremos el caso de dos procesos que tienen datos compartidos. Mientras un procesos está actualizando los datos, es apropiado (*preempted*) así el segundo proceso puede ser corrido. El segundo proceso entonces trata de leer los datos, los cuales están en un estado inconsistente.

Prácticamente todos los sistemas operativos modernos incluyendo Windows, macOS, Linux y UNIX usan algoritmos de planificación apropiativa. (Silberschartz, p. 202)

Si el reloj de hardware provee interrupciones periódicas a cierta frecuencia, una decisión de planificación puede ser hecha a cada interrupción de reloj o cada *k* interrupciones. Los algoritmos de planificación pueden ser divididos en dos categorías con respecto a cómo lidian con las interrupciones de reloj:

- Algoritmo de planificación no apropiativo: selecciona un proceso a correr y luego solamente lo deja correr hasta que se bloquee (ya sea por una I/O o esperando por otro proceso) o voluntariamente libere la CPU. Ya sea que corra por muchas horas, no puede ser suspendido forzosamente. De hecho, no se hacen decisiones de planificación durante las interrupciones de reloj. Después de que el procesamiento de una interrupción de reloj ha sido finalizado, el proceso que estaba corriendo antes de la interrupción es reanudado, a no ser que un proceso con mayor prioridad esté esperando por un timeout ya satisfecho.
- Algoritmo de planificación apropiativo: selecciona un proceso y lo deja correr por un tiempo máximo fijo. Si aún está corriendo al terminar el intervalo de tiempo, es suspendido y el planificador selecciona otro proceso a correr (si hay uno disponible). Hacer planificación apropiativa requiere tener un suceso de interrupción de reloj al finalizar el intervalo de tiempo para darle el control de la CPU nuevamente al planificador. Si no hay un reloj disponible, solamente la planificación no apropiativa es posible. (Tanenbaum, p. 153)

10. Suponga un algoritmo de planificación de corto plazo que favorece a aquellos procesos que han usado poco tiempo de procesador en el más reciente pasado (no solo en el último quantum). ¿Por qué este algoritmo favorece a programas limitados por I/O y no postergará permanentemente programas limitados por la CPU?

Este tipo de algoritmo de planificación de corto plazo, que favorece a los procesos que han utilizado poco tiempo de CPU en el pasado reciente, es conocido como algoritmo de planificación basado en "aging" (envejecimiento). Este enfoque busca evitar la inanición de

los procesos que necesitan más CPU y darles la oportunidad de ejecutarse, incluso si han sido relegados en favor de otros procesos que utilizan menos tiempo de CPU.

La razón por la que este algoritmo favorece a los programas limitados por E/S (entrada/salida) es que, cuando un proceso se bloquea para realizar una operación de E/S, como leer desde o escribir en disco, la CPU queda libre para ser utilizada por otros procesos. Si un proceso utiliza mucho tiempo de CPU y luego se bloquea para realizar una operación de E/S, cuando vuelva a estar listo para ejecutarse, habrá envejecido en la cola de procesos listos. Esto significa que, cuando se desbloquee, tendrá una prioridad más alta para recibir tiempo de CPU en comparación con los procesos que han estado utilizando la CPU constantemente.

Por otro lado, los programas limitados por la CPU también se benefician de este algoritmo porque aunque puedan ser postergados temporalmente en favor de otros procesos, su tiempo de espera en la cola de listos hará que envejezcan y aumenten su prioridad para recibir CPU en el futuro. Esto evita la inanición permanente de los procesos y garantiza una distribución más equitativa del tiempo de CPU entre todos los procesos en ejecución. (ChatGPT 3.5)

Sistemas Operativos

Trabajo Práctico N.º 5

Situaciones de interbloqueo

Puede generarse una situación de interbloqueo (*deadlock*) si se dan las siguientes cuatro condiciones en simultáneo:

1. Exclusión mutua: al menos un recurso debe ser mantenido en un modo de no compartir (*nonsharable mode*), es decir, solamente un *thread* a la vez puede usar el recurso. Si otro thread solicita ese recurso, el thread solicitante debe retrasarse hasta que el recurso sea liberado.
2. Retener y esperar: un thread debe mantener al menos un recurso y esperar para adquirir recursos adicionales que son actualmente retenidos por otros threads.
3. No apropiación: los recursos no pueden ser apropiados, esto es, un recurso puede ser liberado solamente de manera voluntaria por el thread que lo retiene, después de que este thread completó su tarea.
4. Espera circular: un conjunto $\{T_0, T_1, \ldots, T_n\}$ de threads esperando deben existir tales que T_o está esperando por un recurso retenido por T_1, T_1 está esperando por un recurso retenido por T_2, ..., T_{n-1} está esperando por un recurso retenido por T_n, y T_n está esperando por un recurso retenido por T_0.

Existen tres aproximaciones generales para lidiar con los deadlocks. Primero, uno puede **prevenir** el deadlock adoptando una política que elimine una de las condiciones (de la 1 a la 4). Segundo, uno puede **evitar** el deadlock tomando decisiones de manera dinámica basados en el estado actual de la asignación de recursos. Tercero, uno puede intentar **detectar** la presencia de un deadlock (se dan las condiciones 1 a la 4) y tomar una acción para recuperarse.

Grafo de asignación de recursos

Los deadlocks pueden ser descritos de manera más precisa en términos de un grafo dirijido llamado grafo de asignación de recursos del sistema. Este grafo consiste en un conjunto V de vértices y un conjunto E de arcos. El conjunto V de vértices es particionado en dos diferentes tipos de nodos: el conjunto $T=\{T_1, T_2, \ldots, T_n\}$ consistente en todos los threads activos en el sistema, y el conjunto $R=\{R_1, R_2, \ldots, R_n\}$ consistente en todos los tipos de recursos en el sistema.

Un arco directo desde el thread T_i al recurso del tipo R_j es llamado “arco de solicitud”, es denotado $T_i \rightarrow R_j$ y significa que el thread T_i ha solicitado una instancia del recurso del tipo R_j y está actualmente esperando por ese recurso. Un arco directo desde un recurso del tipo R_j al thread T_i es llamado “arco de asignación”, es denotado $R_j \rightarrow T_i$ y significa que una instancia del tipo de recurso R_j ha sido asignado al thread T_i.

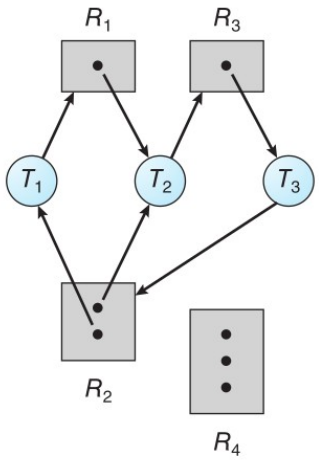


Figure 8.5 Resource-allocation graph with a deadlock.

NOTA: Hacer ejercicio 4, 5, 11. Incluye puntos prácticos.

5. ¿Cómo utilizaría un algoritmo de detección de deadlock (en que momento se ejecutaría)? ¿Qué haría el algoritmo si detecta que hay deadlock?

Un checkeo por un deadlock puede ser hecho tan frecuentemente como para cada solicitud de recurso, o menos frecuentemente, dependiendo de qué tan probable es que ocurra un deadlock. Chechear en cada solicitud de recurso tiene dos ventajas:

- Facilita una detección temprana
- El algoritmo es relativamente simple porque está basado en cambios incrementales del estado del sistema

Por otro lado, checkeos tan frecuentes consumen un tiempo de procesador considerable.

(Stallings, p. 306)

Por su puesto, invocar el algoritmo de detección de deadlocks para cada solicitud de recursos incurrirá en un gasto (overhead) en tiempo de computación. Una alternativa menos costosa es simplemento invocar el algoritmo a intervalos definidos, por ejemplo, una vez por hora o cuando la utilización de la CPU cae debajo del 40% (un deadlock eventualmente paraliza el rendimiento del sistema y causa una caída en la utilización de la CPU).

(Silberschartz, p. 341)

Una vez que un deadlock fue detectado, alguna estrategia es necesaria para recuperarse. Las siguientes son posibles aproximaciones, listados en orden creciente en cuanto a sofisticación:

1. Abortar todos los procesos deadlockeados. Esta es una de las más comunes (si no la más común) soluciones adoptadas en los sistemas operativos.
2. Hacer una copia de seguridad de cada proceso deadlockeado a un checkpoint predefinido, y reiniciar todos los procesos. Esto requiere que sean implementados mecanismos de rollback en el sistema. El riesgo de esta aproximación es que el deadlock original puede volver a ocurrir. Sin embargo, que el procesamiento concurrente sea no-determinístico puede asegurar que esto no sucederá.
3. Abortar sucesivamente procesos deadlockeados hasta que el deadlock deje de existir. El orden en el cuál los procesos son seleccionados para aser abortados debe ser en base a algún criterio de mínimo costo. Después de cada aborto, el algoritmo de detección debe ser reinvocado para ver si el deadlock aún existe.
4. Apropiarse sucesivamente de recursos hasta que el deadlock deje de existir. Como en (3), una selección en base al costo debe ser usado, y la reinvocación del algoritmo de detección es requerido después de cada apropiación. Un procesos que tiene un recurso apropiado debe retroceder (rolled back) a un punto anterior a la adquisición de ese recurso.

Para (3) y (4), el criterio de selección debe ser uno de los siguientes. Selecciona el proceso con:

- el menor tiempo de procesador consumido hasta el momento
- la menor cantidad de salidas producidas hasta el momento
- el mayor tiempo restante estimado
- la menor cantidad de recursos asignados hasta el momento
- la menor prioridad

9. Defina qué es un estado seguro de un sistema.

Un estado es *seguro* si el sistema puede asignar recursos a cada thread (hasta su máximo) en algún orden y aún así evitar un deadlock. Más formalmente, un sistema está en estado seguro sólo si existe una secuencia segura. Una secuencia segura de threads ¿<$T_1, T_2, \ldots, T_n$>¿ es una secuencia segura para un estado de asignación actual si, para cada T_i , las solicitudes de recursos que T_i aún puede hacer pueden ser satisfechas por los recursos disponibles actualmente más los recursos mantenidos por todos los T_j, $j < i$. En esta situación, si los recursos que T_i necesita no está inmediatamente disponibles, entonces T_i puede esperar hasta que T_j haya finalizado. Cuando hayan terminado, T_i puede obtener todos sus recursos necesarios, completar su tarea designada, retornar sus recursos asignados y terminar. Cuando T_i termina, T_{i+1} puede obtener sus recursos necesarios, y así sucesivamente. Si no existe tal secuencia, entonces el estado de sistema es llamado *inseguro*.

Un estado seguro es un estado no en deadlock. En cambio, un estado de deadlock es un estado inseguro. Sin embargo, no todos los estados inseguros son deadlocks. Un estado inseguro **puede** conducir a un deadlock. Mientras el estado es seguro, el sistema operativo puede evitar los estados inseguros y deadlocks. En un estado inseguro, el sistema operativo no puede prevenir que los threads pidan recursos de tal manera que ocurra un deadlock. El comportamiento de los threads controla los estados inseguros.

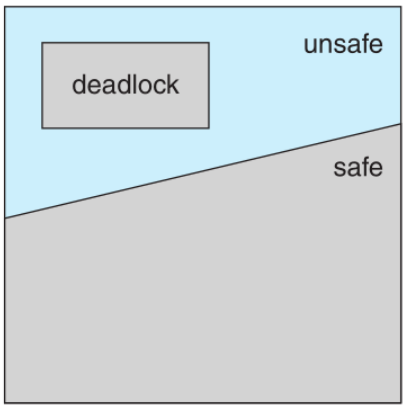


Figure 8.8 Safe, unsafe, and deadlocked state spaces.

10. ¿Qué diferencia hay entre deadlock y starvation?

En un sistema dinámico, las solicitudes por recursos suceden todo el tiempo. Es necesaria alguna política para hacer una decisión acerca de quién obtiene qué recurso y cuando. Esta política, aunque aparentemente razonable, puede llevar a que algunos procesos nunca obtengan el servicio a pesar de que no estén deadlockeados. (Tanenbaum p. 463)

El deadlock y la starvation son dos problemas comunes en los sistemas concurrentes, como los sistemas operativos o los sistemas distribuidos. Aunque ambos implican situaciones en las que los procesos no pueden avanzar, difieren en su causa y en cómo afectan a los procesos.

1. **Deadlock (bloqueo mutuo):**
    - **Causa:** Ocurre cuando dos o más procesos están esperando que otros procesos liberen recursos que necesitan, pero ninguno de ellos puede continuar hasta que el otro libere su recurso. Esto crea un ciclo de espera mutua.
    - **Efecto:** Los procesos quedan atrapados indefinidamente, sin poder avanzar, lo que puede llevar a la inactividad del sistema.
    - **Ejemplo:** Dos procesos A y B necesitan acceso a recursos X e Y, respectivamente. A adquiere X y espera por Y, mientras que B adquiere Y y espera por X. Ninguno puede continuar, ya que ambos están esperando el recurso que el otro tiene.
2. **Starvation (inanición):**
    - **Causa:** Ocurre cuando un proceso nunca obtiene los recursos necesarios para avanzar debido a la priorización de otros procesos. Esto puede deberse a una mala gestión de la prioridad de los procesos o a la competencia injusta por los recursos.
    - **Efecto:** El proceso afectado queda en un estado de espera indefinida, mientras otros procesos continúan utilizando los recursos que necesita.
    - **Ejemplo:** Un proceso de baja prioridad constantemente se ve superado por procesos de alta prioridad cuando compiten por recursos compartidos, lo que resulta en que el proceso de baja prioridad nunca pueda ejecutarse.

En resumen, el deadlock implica una situación en la que los procesos quedan atrapados mutuamente esperando recursos que el otro posee, mientras que la starvation ocurre cuando un proceso no puede avanzar debido a la falta de acceso a recursos, generalmente debido a la priorización de otros procesos. (Chat GPT 3.5)

Cabe mencionar que algunas personas no hacen distinción entre *starvation* y *deadlock* porque en ambos casos no hay progreso. Otros sienten que son diferentes fundamentalmente porque un proceso puede ser programado para intentar hacer algo *n* veces y, si todo eso falla, intentar alguna otra cosa. Un proceso bloqueado no tiene esa opción. (Tanenbaum, p. 496)

12. ¿Cuáles son las dificultades que se presentan cuando un proceso es 'rolled-back' ? Comente también sus ventajas si las hubiere.

En el contexto de la recuperación de un deadlock, el rollback se utilizaría como una medida para romper el estado de bloqueo mutuo entre los procesos. Cuando se detecta un deadlock, se necesita una estrategia para resolver la situación y permitir que los procesos continúen su ejecución.

Dificultades:

- **Pérdida de progreso:** Al retroceder los procesos implicados en el deadlock, cualquier progreso que hayan hecho hasta el momento se perderá. Esto puede ser especialmente problemático si los procesos estaban realizando operaciones críticas o consumiendo recursos valiosos.
- **Complejidad de implementación** Implementar el rollback para resolver un deadlock puede ser complejo, ya que implica identificar los procesos involucrados, determinar el punto en el que ocurrió el deadlock y revertir los cambios realizados por esos procesos hasta ese momento. Esto puede requerir un manejo cuidadoso de los recursos y del estado del sistema.

Ventajas:

- **Recuperación del sistema:** El rollback puede romper el estado de deadlock al revertir las acciones realizadas por los procesos involucrados, lo que permite que el sistema recupere su funcionalidad normal y que los procesos continúen su ejecución.
- **Restauración de la disponibilidad:** Al resolver el deadlock, se restaura la disponibilidad de los recursos del sistema, lo que permite que otros procesos puedan utilizarlos y que el sistema vuelva a ser operativo.

En resumen, aunque el rollback puede presentar desafíos como la pérdida de progreso y la complejidad de implementación, puede proporcionar ventajas importantes al romper el estado de deadlock y restaurar la funcionalidad del sistema. Su aplicación debe ser cuidadosamente considerada para minimizar el impacto en el sistema y maximizar la eficacia de la recuperación. (Chat GPT 3.5)

Si los diseñadores del sistema y los operadores de la computadora saben que los deadlocks son probables, pueden hacer arreglos para tener los procesos con puntos de control (checkpointed) periódicamente. Colocar checkpoints a un proceso significa que su estado es

escrito en un archivo así puede ser restaurado luego. El checkpoint contiene no solamente la imagen de memoria, sino también el estado del recurso, en otras palabras, qué recursos están actualmente asignados al proceso.

Cuando un deadlock es detectado, es fácil de ver cuáles recursos son necesarios. Para hacer la recuperación, un proceso que se apropió de un recurso necesitado es *rolled back* hasta un punto en el tiempo antes de adquirir el recurso comenzando por uno de sus checkpoints más tempranos. El proceso es reseteado a un momento más temprano cuando no tenía el recurso, el cual es ahora asignado a uno de los procesos deadlockeados. Si el proceso reiniciado intenta adquirir ese recurso nuevamente, tendrá que esperar hasta que esté disponible.

Sistemas Operativos

Trabajo Práctico N.º 6

2. ¿Cuál es la diferencia entre una dirección física y una dirección virtual (dirección lógica)?

- Dirección Lógica: generadas por la CPU; también llamadas direcciones virtuales.
- Dirección Física: dirección vista por la unidad de memoria.

El concepto de espacio de direcciones lógico que está limitado a un espacio de direcciones físicas separado es central a la administración de la memoria.

Las direcciones lógicas y físicas son las mismas en tiempo de compilación y en los esquemas de mapeo de direcciones en tiempo de carga; las direcciones lógicas (virtuales) y físicas difieren en el esquema de mapeo de direcciones en tiempo de ejecución.

4. ¿Por qué el número de páginas y el tamaño de las páginas es potencia de 2?

Para hacer que el esquema de paginado sea conveniente, el tamaño de página así como el tamaño de frame deben ser una potencia de 2. Con el uso de un tamaño de página que sea potencia de 2 es fácil de demostrar que la dirección relativa (la cuál es definida con referencia al origen del programa) y la dirección lógica (expresada como un número de página y un offset) son la misma.

Las consecuencias de usar un tamaño de página que sea potencia de 2 son dobles. Primero, el esquema de direccionamiento lógico es transparente al programador, el ensamblador y el linkeador. Cada dirección lógica (número de página, offset) de un programa es idéntico a su dirección relativa. Segundo, es relativamente fácil implementar una función en hardware que realice la traducción dinámica en tiempo de ejecución.

7. Analice los modelos de memoria: paginada, segmentada y segmentación con paginación. Discuta las ventajas y desventajas de cada uno de ellos.

Frame (marco)	Un bloque de tamaño fijo de memoria principal
Página	Un bloque de datos de tamaño fijo que reside en memoria secundaria (como un disco). Una página de datos puede ser temporalmente copiada en un frame de memoria principal
Segmento	Un bloque de datos de tamaño variable que reside en memoria secundaria. Un segmento completo puede ser temporalmente copiado en una región disponible de memoria principal (segmentación) o el segmento puede ser dividido en páginas, las cuales pueden ser copiadas individualmente en memoria principal (segmentación y paginación combinadas)

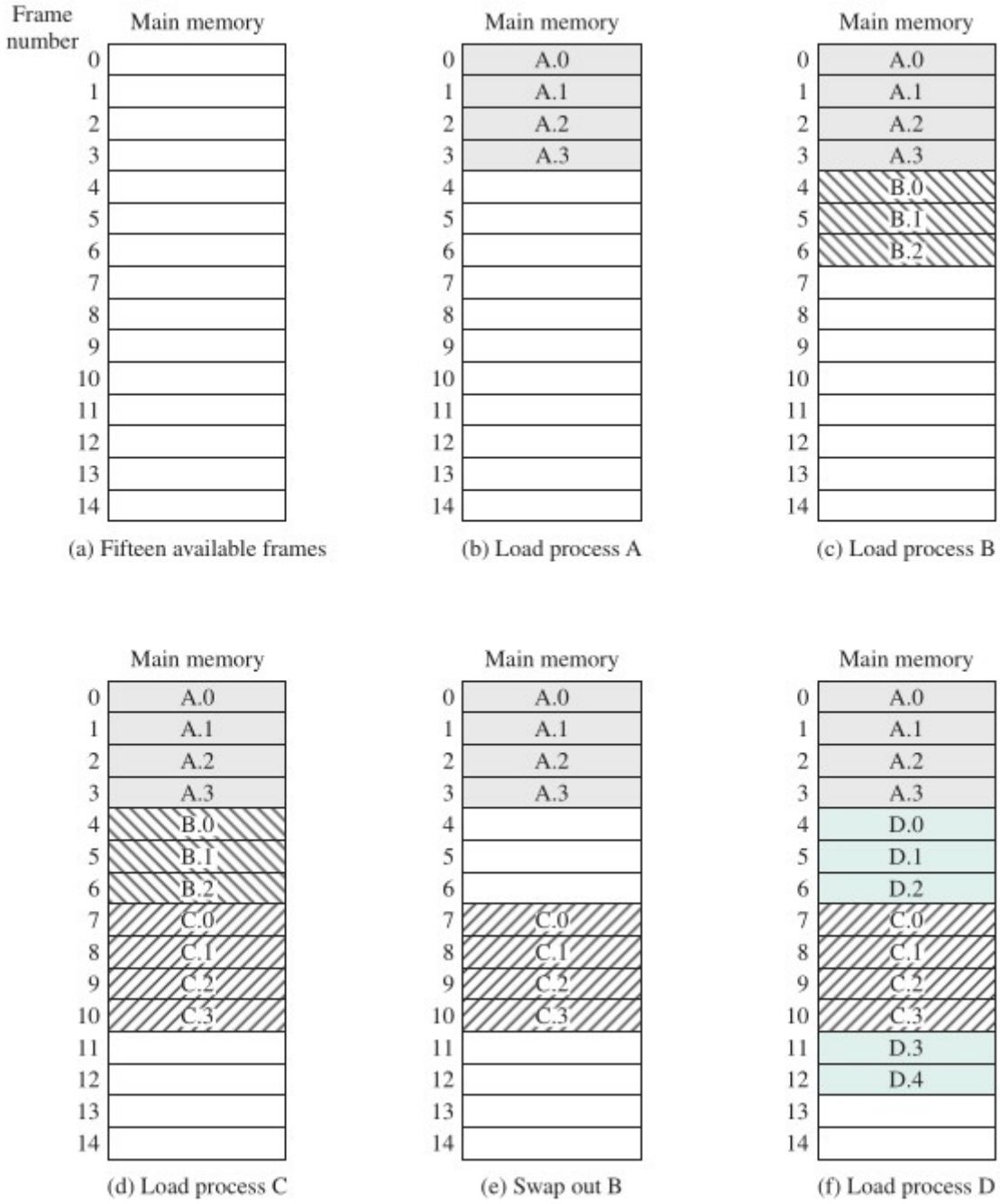
Técnica	Descripción	Fortalezas	Debilidades
Particionado fijo	La memoria principal es dividida en un número de particiones estáticas en el momento de la generación del sistema. Un proceso puede ser cargado en una partición de igual o mayor tamaño.	• Simple de implementar • Poco gasto (overhead) para el sistema operativo	• Uso ineficiente de la memoria debido a la fragmentación interna • El número máximo de procesos es fijo
Particionado dinámico	Las particiones son creadas dinámicamente, así cada proceso es cargado en una partición de exactaente el mismo tamaño que el proceso	• No hay fragmentación interna • Uso de memoria principal más eficiente	• Ineficiente uso de procesos debido a la necesidad de compatación para contrarrestar la fragmentación externa
Paginado (simple)	La memoria principal es dividida en un número de frames de igual tamaño. Cada proceso es dividido e un número de páginas de igual tamaño con el mismo tamaño que los frames. Un proceso es cargado cargando todas sus páginas en frames disponibles, no necesariamente continuos	No hay fragmentación externa	Una pequeña cantidad de fragmentación interna
Segmentación (simple)	Cada proceso es dividido en un número de segmentos. Un proceso es cargado cargando todos sus segmentos en particiones dinámicas que necesitan no ser continuas	• No hay fragmentación interna • Utilización de memoria mejorada y costo (overhead) reducido comparado con particionado dinámico	• Fragmentación externa
Paginado de Memoria Virtual	Como paginado simple, excepto que no es necesario cargar todos las páginas del proceso. Páginas no residentes que son necesitadas son incorporadas	• No hay fragmentación externa • Mayor grado de multiprogramación • Espacio de direcciones	• Costo (overhead) de administración de memoria complejo

Técnica	Descripción	Fortalezas	Debilidades
	automáticamente más adelante.	virtuales amplio	
Segmentación de memoria virtual	Como segmentación simple, excepto que no es necesario cargar todos los segmentos del proceso. Los segmentos no residente que son necesitados son traídos automáticamente más adelante	• No hay fragmentación interna • Mayor grado de multiprogramación • Espacio de direcciones virtuales amplio • Protección y soporte para compartir	• Costo (overhead) de administración de memoria complejo

Paginación

La memoria principal es particionada en chunks de igual tamaño fijo que son relativamente pequeños, y cada proceso es también es dividido en pequeños chunks de tamaño fijo del mismo tamaño. Entonces los chunks de un proceso, llamados páginas, pueden ser asignados a chunks de memoria, llamados frames, o frames de página. Así, el espacio gastado en memoria para cada proceso devido a fragmentación interna consiste en solo una fracción de la última página de un proceso. No hay fragmentación externa.

En la figura se ilustra el uso de páginas y frames. En un punto de tiempo dado, algunso de los frames en memoria están siendo usados y algunos están libres. Una lista de frames libres es mantenida por el OS. El proceso A, almacenado en el disco, consiste en 4 páginas. Cuando es el tiempo de que sea cargado este proceso, el OS encuentra 4 frames libres y carga las 4 páginas del proceso A en los 4 frames (b). El proceso B, consistente de 3 páginas, y el proceso C, consistente de 4 páginas, son luego cargados. Entonces el proceso B es suspendido y swappeado fuera de memoria principal. Luego, todos los procesos en memoria principal son bloqueado, y el OS necesita traer un nuevo proceso, el proceso D, el cual consiste en 5 páginas.



Ahora, supongamos, como en este ejemplo, que no hay suficientes frames continuos sin usar para contener el proceso ¿Esto le impide al sistema operativo cargar D? La respuesta es no, porque podemos usar nuevamente el concepto de direcciones lógicas. El sistema operativo mantiene una tabla de páginas para cada proceso. La tabla de páginas muestra la locación del frame para cada página del proceso. Dentro del programa, cada dirección lógica consiste en un número y un offset dentro de la página. La traducción de las direcciones de lógica a física es hecha por el hardware del procesador. El procesador debe saber cómo acceder a la tabla de páginas del proceso actual. Presentado con una dirección lógica (número de página, offset), el procesador usa la tabla de páginas para producir una dirección física (número de frame, offset).

Continuando con el ejemplo, las 5 páginas del proceso D están cargadas en los frames 4, 5, 6, 11 y 12.

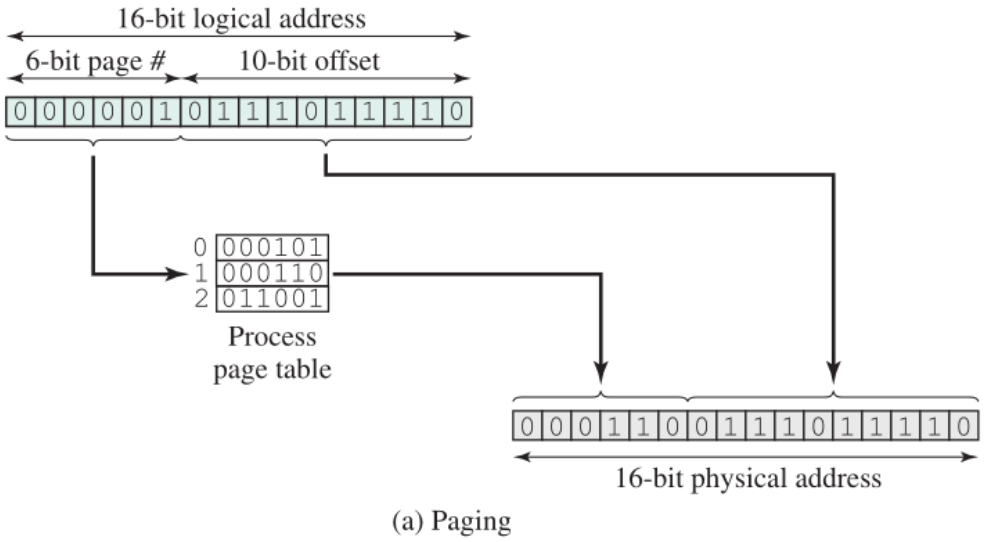


Figura 1: Ejemplo de traducción de dirección lógica a física

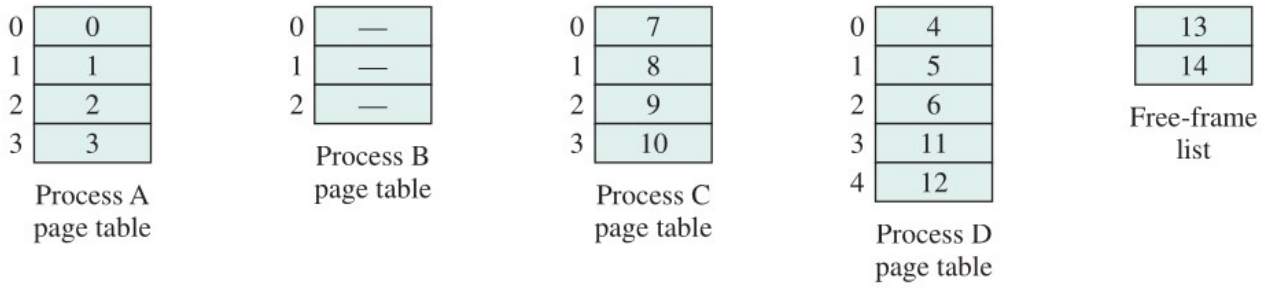


Figure 7.10 Data Structures for the Example of Figure 7.9 at Time Epoch (f)

Una tabla de páginas contiene una entrada por cada página de los procesos, así la tabla es fácilmente indexada por el número de página (empezando por la página 0). Cada tabla de páginas contiene el número de frame en memoria principal, si tiene, que contiene la correspondiente página. Además, el OS mantiene una única lista de frames libres de todos los frames en memoria principal que están actualmente sin ocupar y disponible para páginas.

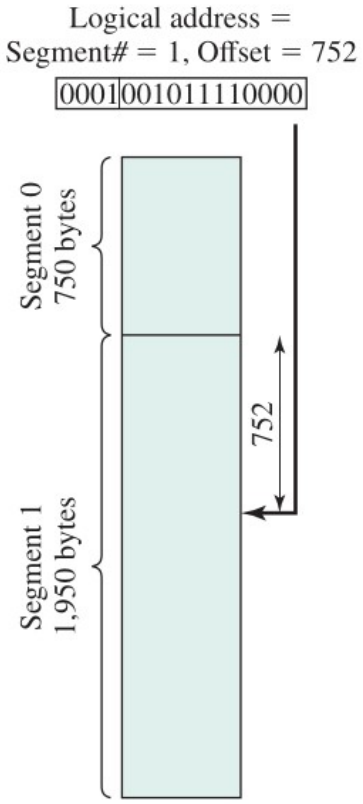
Segmentación

Un programa de usuario puede ser subdividido usando segmentación, en la cuál el programa y sus datos asociados son divididos en un número de segmentos. No es requerido que todos los segmentos de todos los programas sean del mismo largo, aunque hay un largo de segmento máximo. Como con paginación, una dirección lógica usando segmentación consiste en dos partes, en este caso, un número de segmento y un offset.

La segmentación elimina la fragmentación interna pero sufre de fragmentación externa. Sin embargo, como un proceso es dividido en un número de piezas más pequeñas, la fragmentación externa será menor.

Aunque la paginación es invisible para el programador, la segmentación es usualmente visible y es provista según convenga para la organización de programas y datos.

Otra consecuencia de que los segmentos sean de diferente tamaño es que no hay una relación simple entre las direcciones lógicas y las direcciones físicas. Análogamente a la paginación, un esquema simple de segmentación hará uso de una tabla de segmentos para cada proceso y una lista de bloques libres de memoria principal. Cada entrada de la tabla de segmentos tendrá que dar una dirección de inicio en memoria principal del segmento correspondiente. La entrada deberá proveer además la longitud el segmento para asegurar que las direcciones inválidas no son usadas.



(c) Segmentation

Cuando un proceso entra en estado Corriendo, la dirección de su tabla de segmentos es cargada en un registro especial usado por el hardware de administración de memoria. Considera una dirección de n+m bits, donde los n bits más a la izquierda son el número de segmento y los m bits más a la derecha son el offset. En la figura, n = 4 y m = 12. Entonces, el tamaño máximo de segmento es 2^{12} = 4096. Los pasos siguientes son necesarios para la traducción de la dirección:

1. Extraer el número de segmento de los n bits más a la izquierda en la dirección lógica.
2. Usar el número de segmento como un índice en la tabla de segmentos del proceso para encontrar la dirección física inicial del segmento.
3. Comparar el offset, expresado en los m bits más a la derecha, con el tamaño del segmento. Si el offset es mayor o igual al tamaño, la dirección es inválida.
4. La dirección física deseada es la suma de la dirección física inicial del segmento más el offset.

En resumen, con segmentación simple, un proceso es dividido en un número de segmentos que no necesariamente son de igual tamaño. Cuando un proceso es traído, todos sus

segmentos son cargados en direcciones disponibles de memoria, y una tabla de segmentos es establecida.

Segmentación con paginado

Si los segmentos son largos, puede ser inconveniente, o incluso imposible, mantenerlos en memoria principal enteros. Esto lleva a la idea de paginarlos, así solamente aquiellas páginas de un segmento que son realmente necesitadas tienen que estar accesibles.

Para implementarlo, los diseñadores de MULTICS eligieron tratar cada segmento como una memoria virtual y paginarla, combinando las ventajas del paginado (tamaño de página uniforme y no tener que mantener todo el segmento en memoria si solamente una parte de él es usada) con las ventajas la de segmentación (fácil de programar, modularidad, protección y compartimiento [sharing]).

Cada programa MULTICS tenía una tabla de segmentos, con un descriptor por segmento. Ya que había potencialmente más de un cuarto de millón de entradas en la tabla, la tabla de segmentos era ella misma un segmento y estaba paginada. Un descriptor de segmento contenía un indicador de si el segmento estaba o no en memoria principal. Si alguna parte del segmento estaba en memoria, el segmento era considerado como cargado en memoria y tu tabla de páginas estaba en memoria. El descriptor además contenía el tamaño del segmento, los bits de protección y otros items.

Cada segmento era un espacio de direcciones virtual orginario y era paginado de la misma manera que la memoria paginada no segmentada.

Una dirección en MULTICS consistía en dos partes: el segmento y la dirección dentro del segmento. La dirección en el segmento era además dividida en un número de página y una palabra dentro de la página. Cuando ocurría una referencia a memoria, el siguiente algoritmo era llevado a cabo:

1. El número de segmento era usado para encontrar el descriptor del segmento.
2. Un checkeo era realizado para ver si la tabla de segmentos de página estaba en memoria. Si lo estaba, era ubicada. Sino, ocurría un segment fault.
3. La entrada en tabla de páginas para la página virtual requerida era examinada. Si la página misma no estaba en memoria, un page fault era lanzado. Si estaba en memoria, la dirección de inicio de la página en memoria principal era extraída de la entrada de tabla de páginas.
4. El offset era sumado al origen de la página para dar la dirección en memoria principal donde la palabra estaba ubicada.
5. La lectura o almacenamiento finalmente tenía lugar.

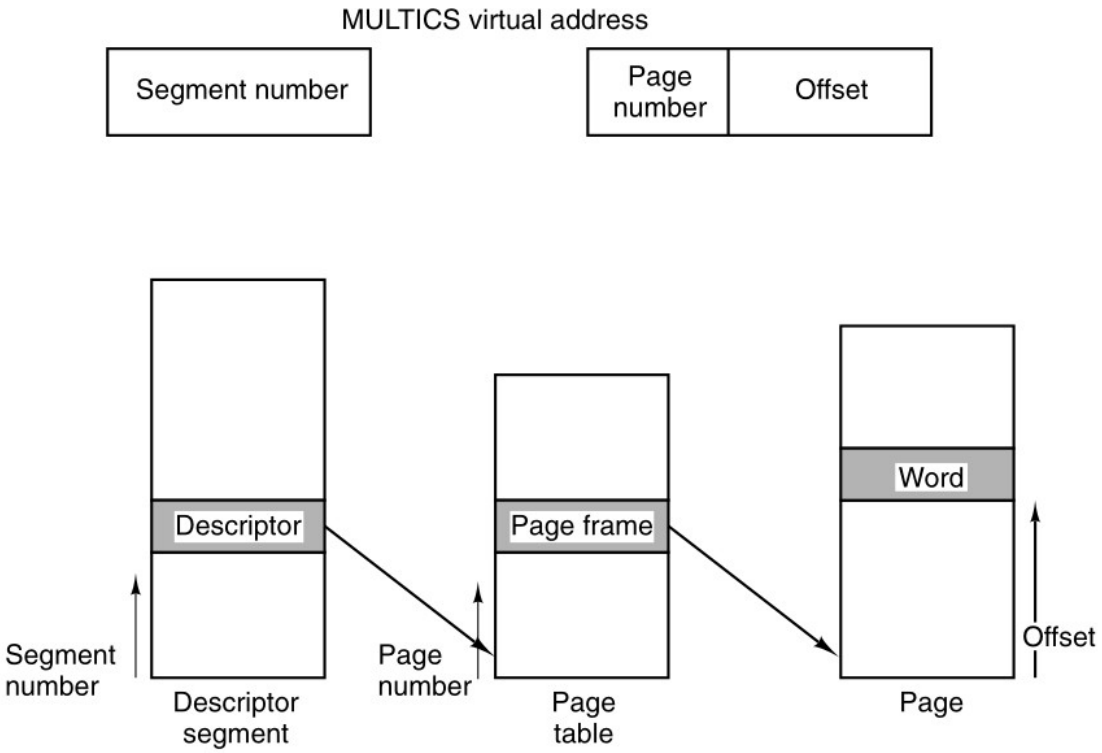


Figure 3-36. Conversion of a two-part MULTICS address into a main memory address.

Consideración	Paginado	Segmentación
¿Necesita que el programador tenga en cuenta que esta técnica está siendo usada?	No	Si
¿Cuántas espacios lineales de direcciones tiene?	1	Muchos
¿Puede el espacio total de direcciones exceder el tamaño de la memoria física?	Si	Si
¿Pueden procedimientos y datos ser distinguidos y protegidos separadamente?	No	Si
¿Pueden las tablas cuyo tamaño fluctúa ser acomodadas fácilmente?	No	Si
¿Se facilita compartir procedimientos entre usuarios?	No	Si
¿Por qué esta técnica fue inventada?	Para tener un espacio de direcciones lineal sin tener que comprar más memoria física	Para permitir a los programas y datos ser divididos en espacios de direcciones lógicos independientes y para ayudar en la protección y compartir

En la manera más simple, cada espacio de dirección de cada proceso es dividido en bloques de tamaño uniforme llamados “páginas”, las cuales pueden ser ubicadas en cualquier frame de página disponible en memoria.

La segmentación ayuda a manejar estructuras de datos que pueden cambiar de tamaño durante la ejecución y simplifica el compartir y el linkeo. También facilita proveer diferentes protecciones para diferentes segmentos. A veces la segmentación y el paginado son combinados para proveer una memoria virtual de dos dimensiones.

Hoy en día, aún la versión de 64 bits de la x86 ya no soporta segmentación real.

Segmentación con paginado (v2)

En un sistema combinado de paginación/segmentación, el espacio de direcciones de usuario es divido en un número de segmentos, a criterio del programador. Cada segmento es, a su vez, divido en un número de páginas de tamaño fijo, las cuales tienen igual longitud que los frames de memoria principal. Si un segmento tiene una longitud menor que la de una página, el segmento ocupa solamente una página. Desde el punto de vista del programador, una dirección lógica aún consiste en un número de segmento y un *offset*. Desde el punto de vista del sistema, el *offset* de segmento es visto como un número de página y un *offset* de página para una página en el segmento especificado.

La Figura 8.12 sugiere una estructura para soportar la combinación paginado/segmentación. Asociado con cada proceso hay una tabla de segmento y un número de tablas páginas, una por segmento de proceso. Cuando un proceso en particular está corriendo, un registro mantiene la dirección de inicio de la tabla de segmentos para ese proceso. Presentado con una dirección virtual, el procesador usa la porción de número de segmento para indexar dentro de la tabla de segmentos del proceso para encontrar la tabla de páginas para ese segmento. Entonces la porción de número de página de la dirección virtual es usada para indexar la tabla de páginas y buscar el número de *frame* correspondiente. Esto es combinado con la porción de *offset* de la dirección virtual para producir la dirección real deseada.

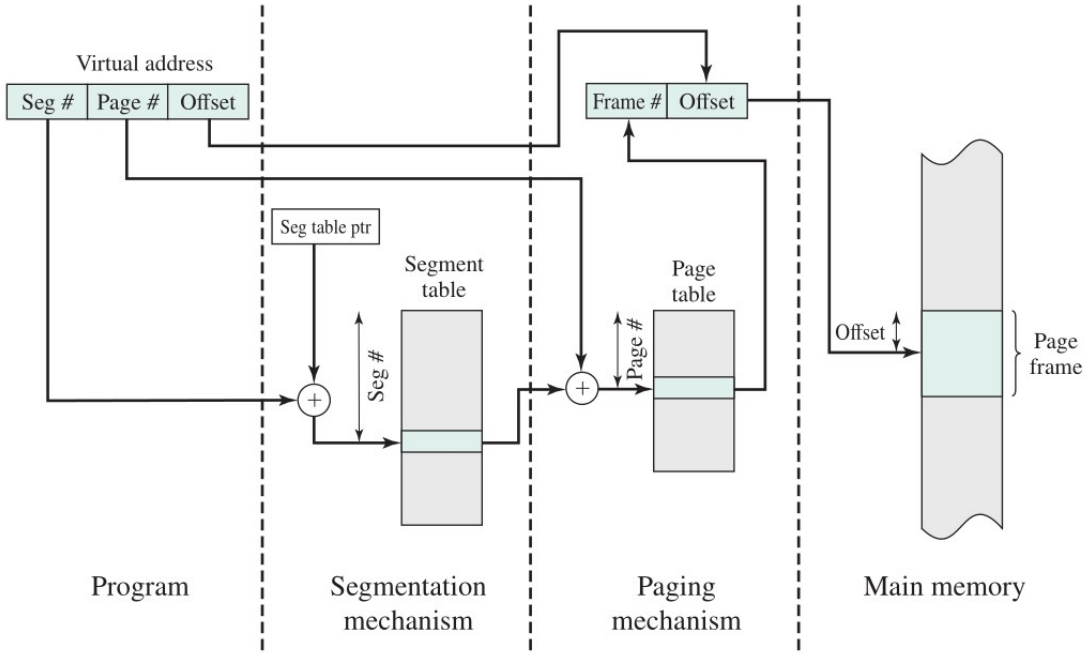


Figure 8.12 Address Translation in a Segmentation/Paging System

Cada entrada de la tabla de segmentos contiene la longitud del segmento. También contiene un campo base, el cual ahora referencia a la tabla de páginas. Los bits de presencia y modificación no son necesarios porque esos asuntos son manejados a nivel de página. Otros bits pueden ser usados para propósitos de compartir y protección. Cada número de página es mapeado en un número de *frame* correspondiente si la página está presente en memoria principal. El bit de modificado indica si la página necesita ser escrita de nuevo cuando el *frame* es asignado a otra página. Puede haber otros bits de control que lidean con protección u otros aspectos de administración de memoria.

8. ¿Cómo se protegen los esquemas de asignación del ejercicio 7?

La protección de memoria en un ambiente de paginado es logrado por los bits de protección asociados a cada frame. Normalmente, estos bits son mantenidos en la tabla de páginas.

Un bit puede definir si la página es de lectura-escritura o de sólo-lectura. Cada referencia a memoria pasa a través de la tabla de páginas para encontrar el número de frame correcto. Al mismo tiempo que la dirección física es calculada, los bits de protección pueden verificar que no hay escrituras siendo hechas a una página de sólo-lectura. Un intento de escritura en una página de sólo-lectura causa una señal del hardware al sistema operativo (o una violación de protección de memoria).

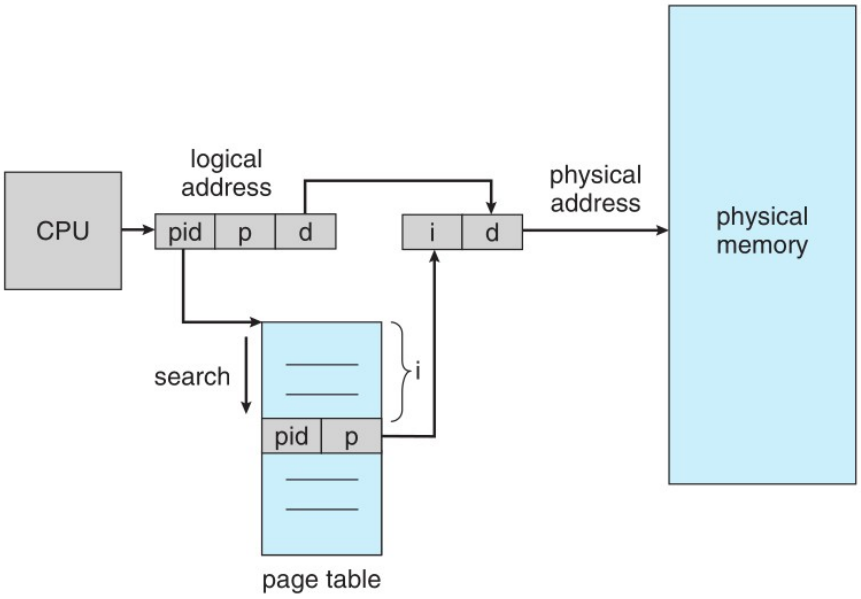
Un bit adicional es generalmente adjuntado a cada entrada en la tabla de páginas: un bit de válido-inválido. Cuando este bit es seteado en *válido*, la página asociada está en el espacio de direcciones lógicas del proceso y entonces es una página legal (o válida). Cuando el bit es seteado a *inválido*, la página no está en el espacio de direcciones lógicas del proceso. Las direcciones ilegales son capturadas por el uso del bit de válido-inválido. El sistema operativo asigna este bit a cada página para permitir o inhabilitar el acceso a la página.

14. ¿En qué caso utilizaría página invertida? ¿Por qué?

Usualmente, cada proceso tiene su tabla de páginas asociada. La tabla de páginas tiene una entrada por cada página que el proceso está usando. Ya que la tabla está ordenada por dirección virtual, el sistema operativo es capaz de calcular dónde está alocada la entrada de dirección física asociada dentro de la tabla y de usar su valor directamente. Una de las desventajas de este método es que cada tabla de páginas puede consistir de millones de entradas. Estas tablas pueden consumir grandes cantidades de memoria física solamente para hacer seguimiento de cómo otra memoria física es usada.

Para resolver este problema, podemos usar una tabla de páginas invertida. Una tabla de páginas invertida tiene una entrada por cada página real (o frame) de memoria. Cada entrada consiste de direcciones virtuales de la página almacenada en esa locación de memoria real. Entonces, solamente hay una tabla de páginas en el sistema, y tiene solamente una entrada por cada página en memoria física. Las tabla de páginas invertida generalmente requiere que un identificador de espacio de direcciones sea almacenado en cada entrada de la tabla de páginas, ya que usualmente contiene varios espacios de direcciones diferentes mapeando a memoria física. Almacenando el identificador de espacio de memoria asegura que una página lógica para un proceso en particular sea mapeado al frame de página físico correspondiente.

Aunque este esquema reduce la cantidad de memoria requerida para almacenar cada tabla de páginas, incremente la cantidad de tiempo necesario para buscar la tabla cuando una referencia a una página ocurre.



17. Suponga que tiene un tamaño de memoria de 1024 KB y la memoria se asigna utilizando el algoritmo de Buddy (del compañero).

En un sistema buddy, los bloques de memoria están disponibles del tamaño de 2^k palabras, $L \leq K \leq U$, dónde:

2^L= el bloque de menor tamaño que está alocado

2^U = el bloque de mayor tamaño que está asignado; generalmente 2^U es el tamaño de toda la memoria disponible para asignación

En principio, todo el espacio disponible para asignación es tratado como un único bloque de tamaño 2^U. Si es hecha una solicitud de tamaño s tal que $2^{(U-1)} < s \leq 2^U$, entonces todo el bloque es asignado. De lo contrario, el bloque es dividido en 2 buddies iguales de tamaño $2^{(U-1)}$. Si $2^{(U-2)} < s \leq 2^{(U-1)}$, entonces la solicitud es asignada a uno de los 2 buddies. Si no, uno de los buddies es dividido en 2 nuevamente. Este proceso continúa hasta que el bloque más pequeño más grande o igual a s es generado y asignado a la solicitud. En todo tiempo, el sistema buddy mantiene una lista de los agujeros (bloques sin asignar) de cada tamaño 2^i. Un agujero puede ser eliminado de la lista ($i + 1$) dividiéndolo a la mitad para cread dos buddies de tamaño 2^i en la lista i. Cuando un par de buddies de la lista i se vuelven no asignados, son removidos de la lista y fusionados en un único bloque en la lista ($i + 1$).

La Figura 7.6 da un ejemplo usando un bloque inicial de 1 Mbyte. La primer solicitud, A, es de 100 Kbytes, para la cual un bloque de 128K es necesitado. El bloque inicial es dividido en 2 buddies de 512K. El primero de esos es dividido en 2 buddies de 256K, y el primero de esos en 2 buddies de 128K, uno de los cuales es asignado a A. La siguiente solicitud, B, requiere un bloque de 256K. Ese bloque ya está disponible y es asignado. El proceso continúa dividiendo y fusionando según sea necesario. Cuando E es liberado, 2 buddies de 128K son fusionados en un bloque de 256K, el cuál es inmediatamente fusionado con su buddy.

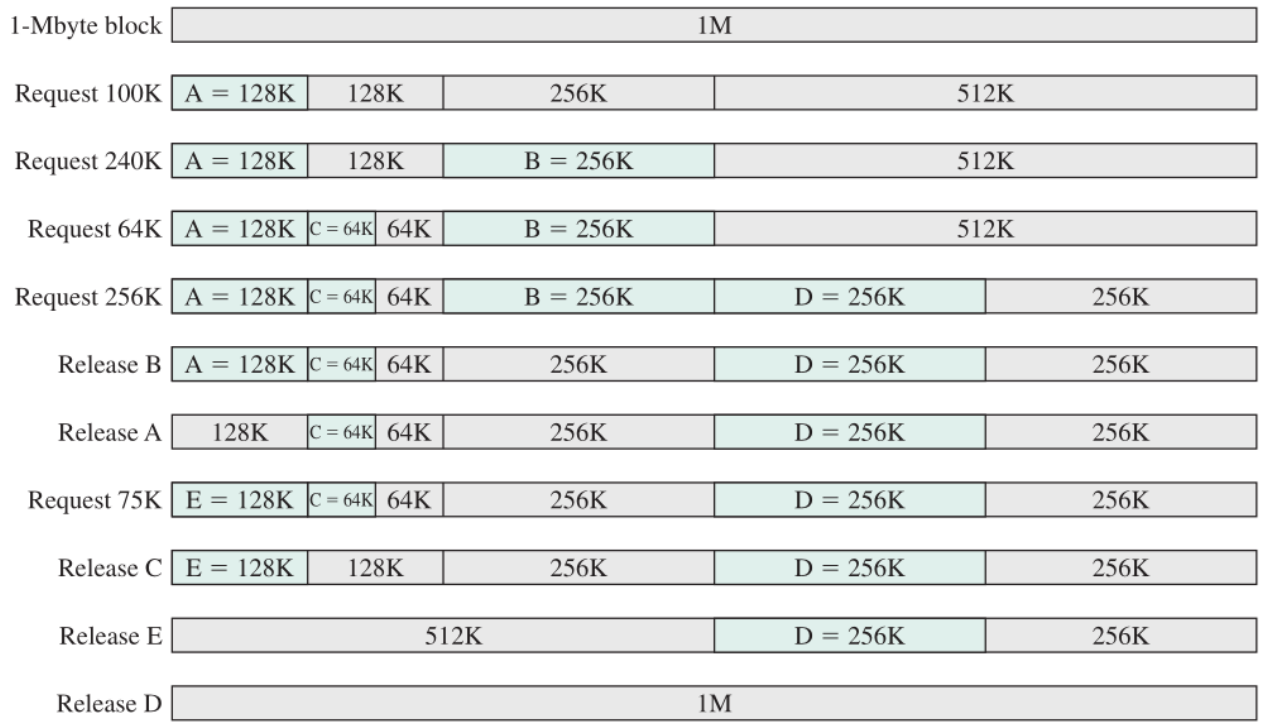


Figure 7.6 **Example of the Buddy System**

18. Si se utiliza el reemplazo FIFO con cuatro marcos y ocho páginas, ¿Cuántos fallos ocurrirán con la cadena de referencia 0172327103 si los cuatro marcos están vacíos al principio? Repita el problema con LRU.

FIFO

El sistema de reemplazo de páginas más simple es el algoritmo de primero en llegar, primero en salir (FIFO, first-in, first-out). Un algoritmo de reemplazo FIFO asocia con cada página el tiempo cuando esta página fue traída a memoria. Cuando una página debe ser remplazada, la página más antigua es elegida.

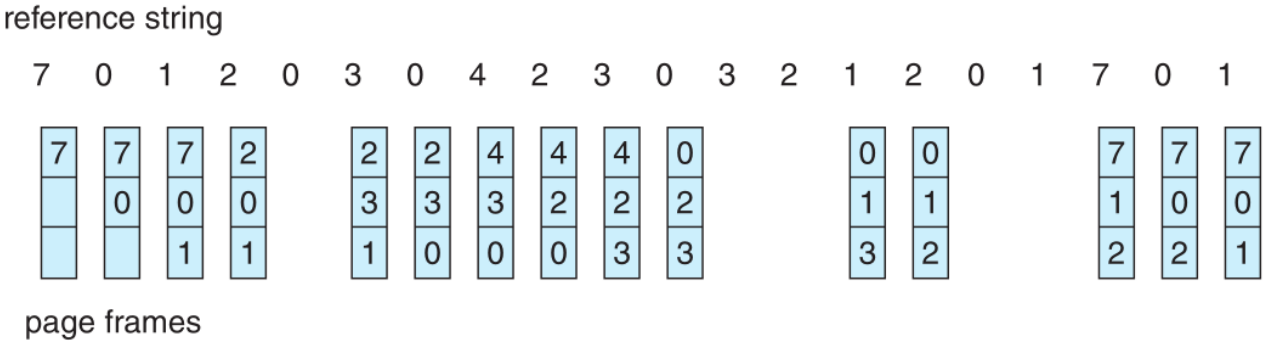


Figure 10.12 FIFO page-replacement algorithm.

En nuestro ejemplo, los tres frames están inicialmente vacíos. Las primeras tres referencias (7, 0, 1) causa *page faults* y es traída en esos frames vacíos. La siguiente referencia (2) reemplaza la página 7, porque la página 7 fue traída primero. Ya que 0 es la siguiente referencia, y 0 está ya en memoria, no tenemos *fault* por esta referencia. La primera

referencia a 3 resulta en un reemplazo de la página 0, ya que está ahora primera en la línea. A causa de este reemplazo, la siguiente referencia, a 0, fallará. La página 1 es entonces reemplazada por la página 0. Este proceso continúa como es mostrado en la Figura 10.12. Cada vez que una *fault* ocurre, se muestra qué páginas están en los tres frames. Hay 15 *faults* en total.

LRU

La aproximación del algoritmo usado menos recientemente (LRU, *least recently used*) es poder reemplazar la página que no ha sido usada por el mayor periodo de tiempo.

El reemplazo LRU asocia con cada página el tiempo del último uso de la página. Cuando una página debe ser reemplazada, LRU elige la página que no ha sido usada por el mayor periodo de tiempo.

El resultado de aplicar el reemplazo LRU a nuestro ejemplo de referencia es mostrado en la Figura 10.15. El algoritmo LRU produce 12 fallas.

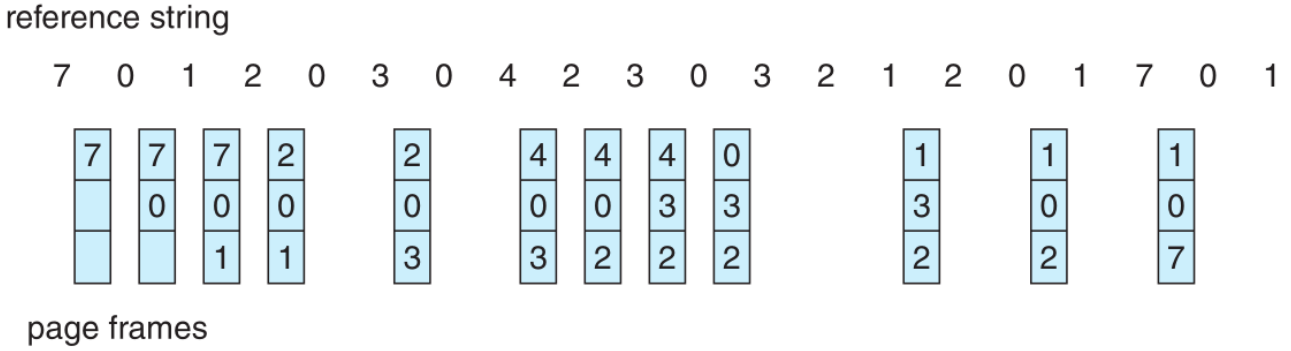


Figure 10.15 LRU page-replacement algorithm.

Cuando la referencia a la página 4 ocurre, el reemplazo LRU ve que de los tres frames en memoria, la página 2 fue la menos recientemente usada. Entonces, el algoritmo LRU reemplaza la página 2, no sabiendo que la página 2 está por ser usada. Cuando falla por la página 2, el algoritmo LRU reemplaza la página 3, ya que ahora es la menos recientemente usada de las tres páginas en memoria.

A pesar de esos problemas, el reemplazo LRU con doce faltas es mucho mejor que el reemplazo FIFO con quince.

20. ¿Es necesario proteger la memoria en un sistema de memoria virtual? ¿Por qué?

La segmentación se presta por sí misma a la implementación de políticas de compartir y protección. Ya que cada entrada de tabla de segmentos incluye una longitud así como su dirección base, un programa no puede acceder de manera inadvertida a una locación de memoria principal más allá de los límites del segmento.

Para el paginado, la estructura de los programas y datos de la página no son visibles para el programador, haciendo la especificación de protección y compartir más incómoda.

Pueden ser provistos mecanismos más sofisticados. Un esquema común es usar una estructura de anillo de protección. En este esquema, los anillos con numeración baja (o

internos) gozan de mayores privilegios que los con numeración alta (o externos). Típicamente, el anillo 0 es reservado para las funciones del kernel del SO, con aplicaciones en los niveles superiores. Algunas utilidades o servicios del SO pueden ocupar un anillo intermedio. Los principios básicos para el sistema de anillos son:

- Un programa puede acceder solamente a información que resida en el mismo anillo o en uno con menos privilegios
- Un programa puede llamar servicios que residan en el mismo anillo o en uno con mayores privilegios

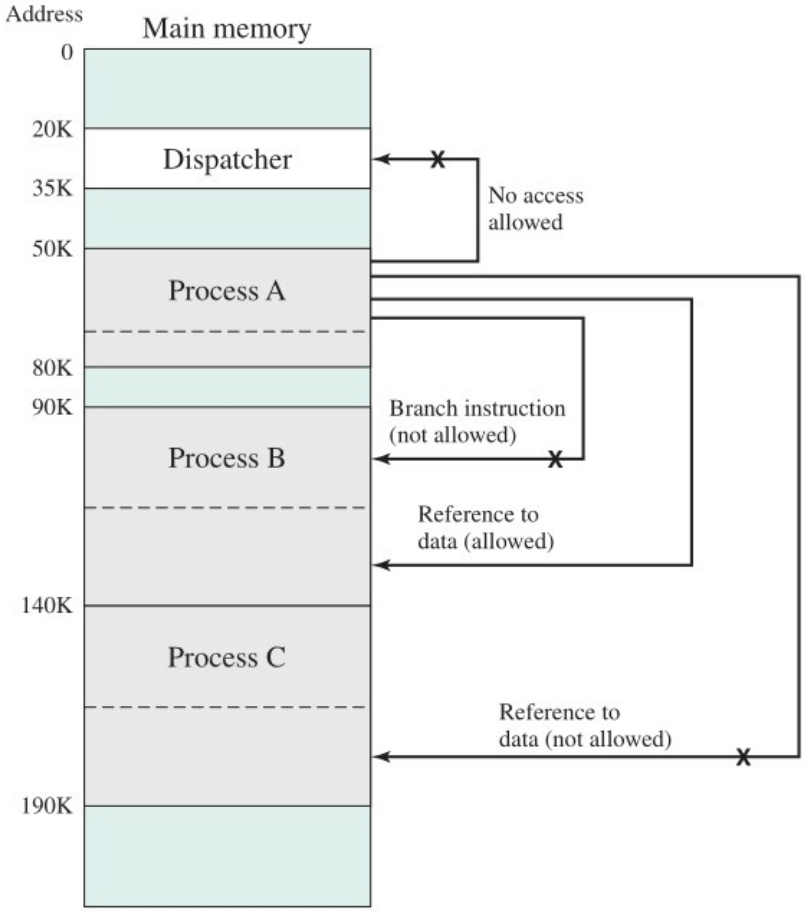


Figure 8.13 Protection Relationships between Segments

21. Un sistema de 32 bits utiliza una tabla de páginas de dos niveles. Las direcciones virtuales se dividen en un campo de 9 bits para la tabla de nivel superior y un campo de 11 bits para la tabla de nivel secundario, además de un ajuste. ¿Cuál es el tamaño de las páginas y cuál es el número en el espacio de direcciones virtuales?

Un ejemplo simple de una tabla de páginas multinivel es mostrada en la Fig. 3-13. En Fig. 3-13(a) tenemos una dirección virtual de 32 bits que es particionada en un campo *PT1* de 10 bits, un campo *PT2* de 10 bits, y un campo *offset* de 12 bits. Como los *offsets* son de 12 bits, las páginas son de 4 KiB, y hay un total del 2^{20} de ellos.

El secreto del método de tabla de páginas multinivel es evitar mantener todas las tablas de páginas en memoria todo el tiempo. En particular, aquellas que no son necesarias no deben mantenerse cerca.

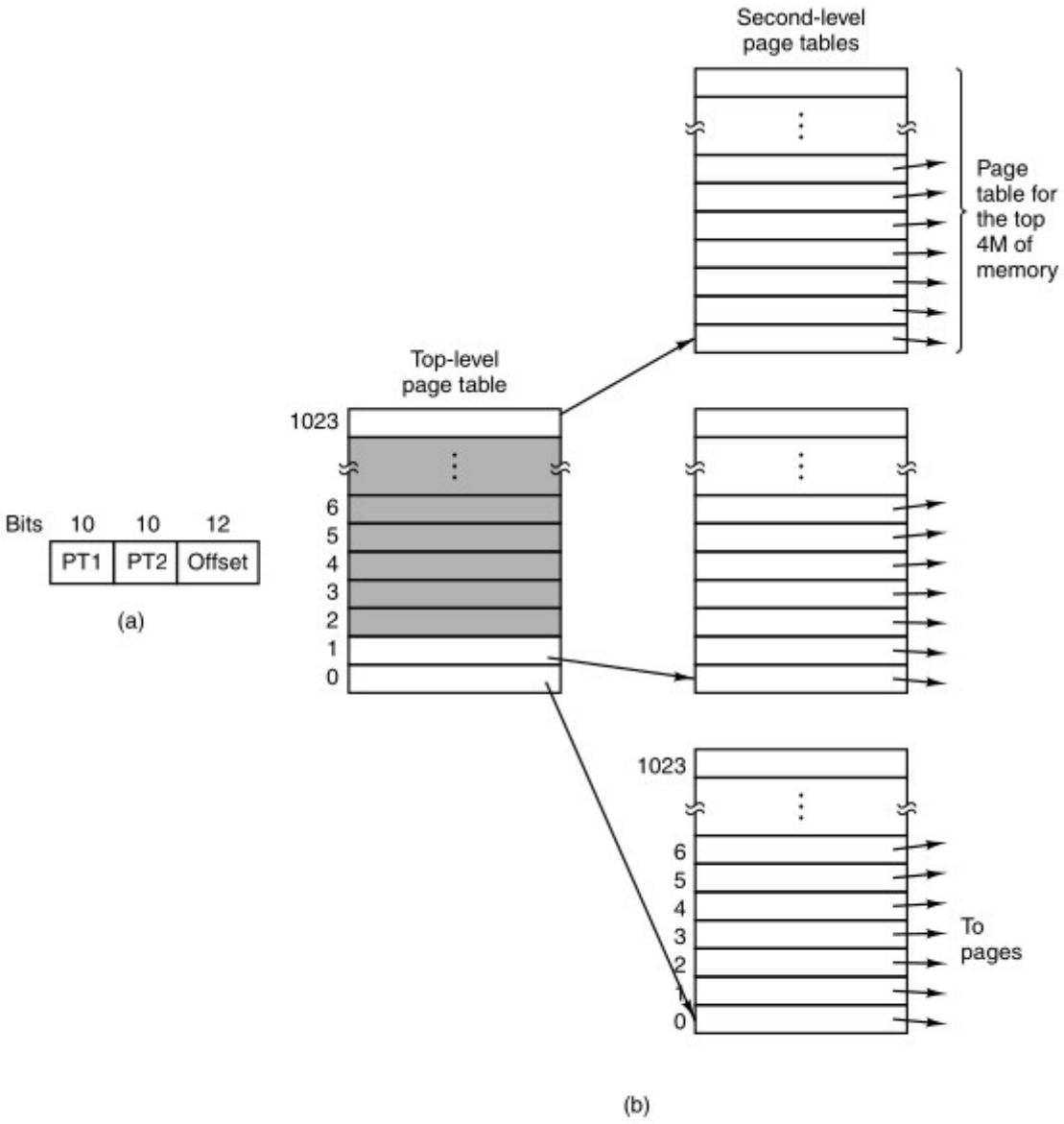


Figure 3-13. (a) A 32-bit address with two page table fields. (b) Two-level page tables.

En Fig. 3-13(b) vemos como trabaja una tabla de páginas de dos niveles. En la izquierda vemos que la tabla de páginas de alto nivel, con 1024 entradas, correspondiente a los 10 bits del campo *PT1*. Cuando una dirección virtual es presentada al MMU, primero extrae el campo *PT1* y usa su valor como un índice en la tabla de páginas de alto nivel. Cada una de esas 1024 entradas en la tabla de páginas de alto nivel representa 4M porque el espacio de direcciones virtual completo de 4 gigabytes (es decir, 32 bits) fueron divididos en trozos de 4096 bytes.

La entrada asignada indexando en la tabla de páginas de alto nivel produce la dirección o el número de frame de la página de una tabla de páginas de segundo nivel. El campo *PT2* es ahora usado como un índice en la tabla de páginas de segundo nivel seleccionada para encontrar el número de frame de la página para la página en sí.

El sistema de tabla de páginas de dos niveles mostrado en Fig. 3-13 puede ser expandido a tres, cuatro o más niveles. Niveles adicionales ofrecen mayor flexibilidad. Por ejemplo, el procesador de 32 bits de Intel 30386 (lanzado en 1985) era capas de direccionar 4 GiB de memoria usando una tabla de páginas de dos niveles que consistía en un directorio de páginas cuyas entradas apuntaban a tablas de páginas, las cuales, cada una, apuntaba concretamente a frame de páginas de 4 KiB. Tanto el directorio de páginas y las tablas de páginas cada una contenía 1024 entradas, dando un total de $2^{10} \cdot 2^{10} \cdot 2^{12} = 2^{32}$ bytes direccionables.

Sistemas Operativos

Trabajo Práctico N.º 7

3. Suponga que el movimiento de cabezas de un disco con 200 tracks numerados de 0 a 199, esta actualmente en el track 143. Si la cola de solicitudes ordenada es: 147, 91, 177, 94, 150, 102, 175, 130, 125. ¿Cuál es el número de cabezas necesarios para realizar estos requerimientos utilizando los siguientes algoritmos de planificación?:

El tiempo de búsqueda es el tiempo en el que el brazo del dispositivo mueva los cabezales al cilindro que contiene el sector deseado. La latencia rotacional es el tiempo adicional en el que el plato rota el sector deseado al cabezal. El ancho de banda del dispositivo es el número total de bytes transferidos, divididos por el tiempo total entre la primer solicitud del servicio y la finalización de la última transferencia.

- FCFS: la forma más simple de planificación de disco es el algoritmo first-come, first-served (FCFS, o FIFO). Este algoritmo es intríncecamente justo, pero generalmente no provee el servicio más rápido. Consideremos, por ejemplo, una cola de disco con solicitudes para I/O para bloques en los cilindros 98, 183, 37, 122, 14, 124, 65, 67, en ese orden. Si el cabezal del disco está inicialmente en el cilindro 53, primero se moverá de 53 a 98, luego a 183, 37, 122, 14, 124, 65 y finalmente a 67, con un movimiento de cabezal total de 640 cilindros.

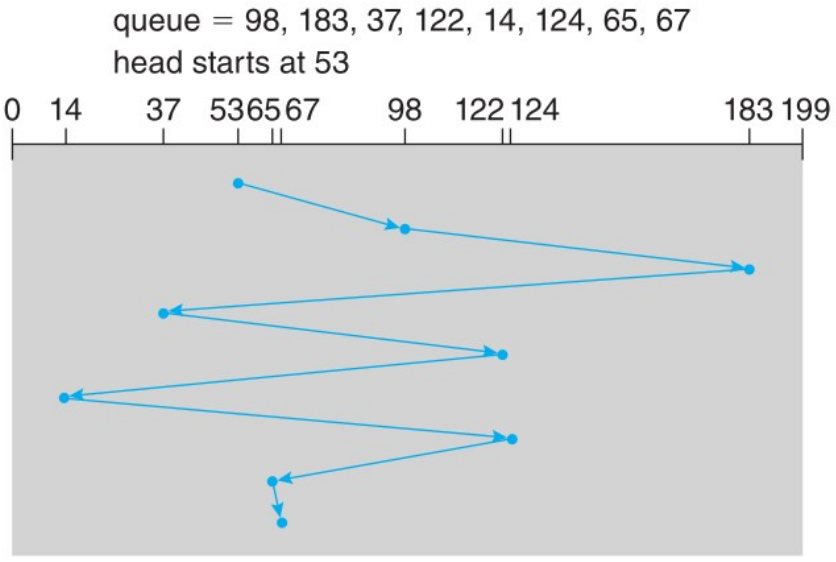


Figure 11.6 FCFS disk scheduling.

Las idas y venidas del 122 a 14 y luego de nuevo al 124 ilustra el problema con esta planificación. Si la solicitud por los cilindros 37 y 14 se servieran juntas, antes o después de las solicitudes para 122 y 124, la cantidad de movimientos total del cabezal podría disminuír sustancialmente, y la performance podría entonces mejorar.

- SSTF: la política de Shortest-Service-Time-First es seleccionar la solicitud de I/O de disco que requiera el menor movimiento del brazo del disco desde su posición actual. De este modo, siempre se eligirá incurrir en el menor tiempo de búsqueda. Por su puesto, siempre elegir el menor tiempo de búsqueda no garantiza que el tiempo promedio de búsqueda sobre una cantidad de movimientos del brazo será mínimo. Sin embargo, proveerá mejor performance que FIFO.
- SCAN: en el algortimo SCAN, el brazo del disco comienza en un final del disco y se mueve hacia el otro final, sirviendo solicitudes a medida que llega cada cilindro, hasta que alcanza el otro final del disco. En el otro final, la dirección del movimiento de la cabeza se invierte, y el servicio continúa. El cabezal continuamente escanea hacia adelante y atrás del disco.

  Volviendo a el ejemplo anterior, antes de aplicar SCAN para planificar las solicitudes en los cilindros 98, 183, 37, 122, 14, 124, 65 y 67, necesitamos saber la dirección de movimiento del cabezal además de la posición actual del cabezal. Asumiendo que el brazo del disco se está moviendo hacia 0 y que la posición inicial del cabezal es 53, siguiente sercivio del cabezal será 37 y luego 14. En el cilindro 0, el brazo invertirá su movimiento y se moverá hacia el otro final del disco, sirviendo las solicitudes 65, 67, 98, 122, 124 y 183.

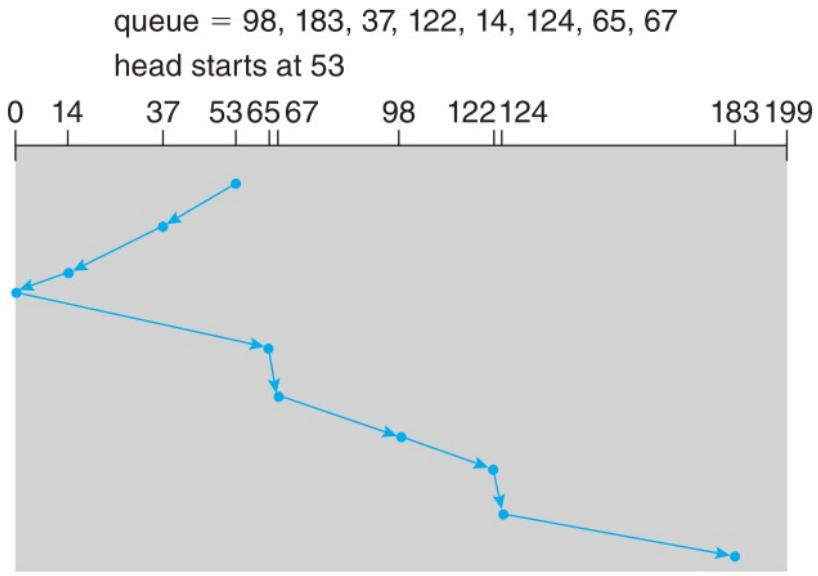


Figure 11.7 SCAN disk scheduling.

  Si una solicitud llega a la cola justo en frente al cabezal, será servida inmediatamente; una solicitud que arriba justo atrás del cabezal tendrá que esperar hasta que el brazo se mueva al final del disco, invierta su dirección, y vuelva hacia atrás.
- LOOK: es un algoritmo de planificación de I/O de HDD que modifica SCAN para detener el cabezal después de que la última solicitud es completada (en vez de detenerlo en el cilindro más al interior o exterior)
- C-SCAN: la planificación SCAN Circular es una variante de SCAN diseñada para proveer un tiempo de espera más uniforme. Como SCAN, C-SCAN mueve el cabezan

desde un final al otro del disco, sirviendo las solicitudes a lo largo del camino. Sin embargo cuando el cabezal alcanza el otro final, inmediatamente vuelve al principio del disco sin servir ninguna solicitud en el camino de regreso.

Volviendo a nuestro ejemplo para ilustrar. Antes de aplicar C-SCAN para planificar las solicitudes en los cilindros 98, 183, 37, 122, 14, 124, 65 y 67, necesitamos saber la dirección de movimiento del cabezal en la cual las solicitudes son planificadas. Asumiento que las solicitudes son planificadas cuando el brazo del disco se está moviendo de 0 a 199 y que la posición inicial del cabezal es 53, las solicitudes serán

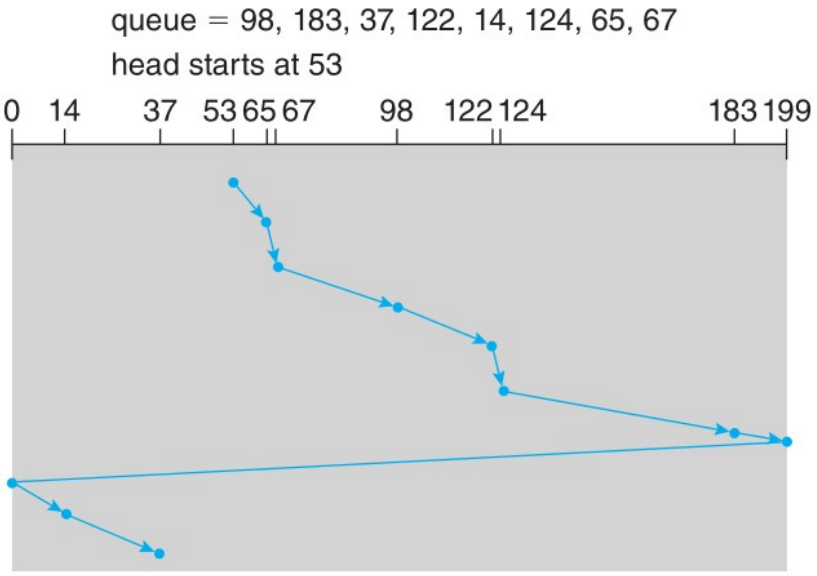


Figure 11.8 C-SCAN disk scheduling.

servidas como se muestra en la figura.

Table 11.2 Comparison of Disk Scheduling Algorithms

(a) FIFO (starting at track 100)		**(b) SSTF** (starting at track 100)		**(c) SCAN** (starting at track 100, in the direction of increasing track number)		**(d) C-SCAN** (starting at track 100, in the direction of increasing track number)	
Next track accessed	**Number of tracks traversed**	**Next track accessed**	**Number of tracks traversed**	**Next track accessed**	**Number of tracks traversed**	**Next track accessed**	**Number of tracks traversed**
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Average seek length	55.3	**Average seek length**	27.5	**Average seek length**	27.8	**Average seek length**	35.8

5. Calcule cuánto espacio en disco (en sectores, pistas y superficie) será requerido para almacenar 250000 registros lógicos de 200 bytes, si el tamaño del sector es de 1024 bytes/sector, con 108 sectores/pista, 140 pistas por superficies y 12 superficies utilizables. Asuma que los registros no se pueden dividir entre dos sectores.

$(250\,000\,registros\,lógicos \cdot 200\,bytes)/1024\,b/s = 48\,828{,}125 = 48\,829\,sectores$

$48\,829\,sectores/108\,s/t = 452{,}12\,tracks$

$452{,}12\,tracks/(140\,t/sur \cdot 12\,sur) = 0{,}27\,superficies$

Cada plato de disco tiene una forma cicular. Ambas caras del plato está cuviertas con material magnético.

Una cabeza lecto-escritora “vuela” sobre cada superficie de cada plato. Las cabezas están unidas a un brazo de disco que mueve todas las cabezas como una unidad. La superficie del plato está logicamente dividida en pistas circulares, las cuales está subdivididas en sectores. Un conjunto de pistas en una posición de brazo dada conforman un cilindro. Cada sector tiene un tamaño fijo y es la unidad de transferencia más pequeña.

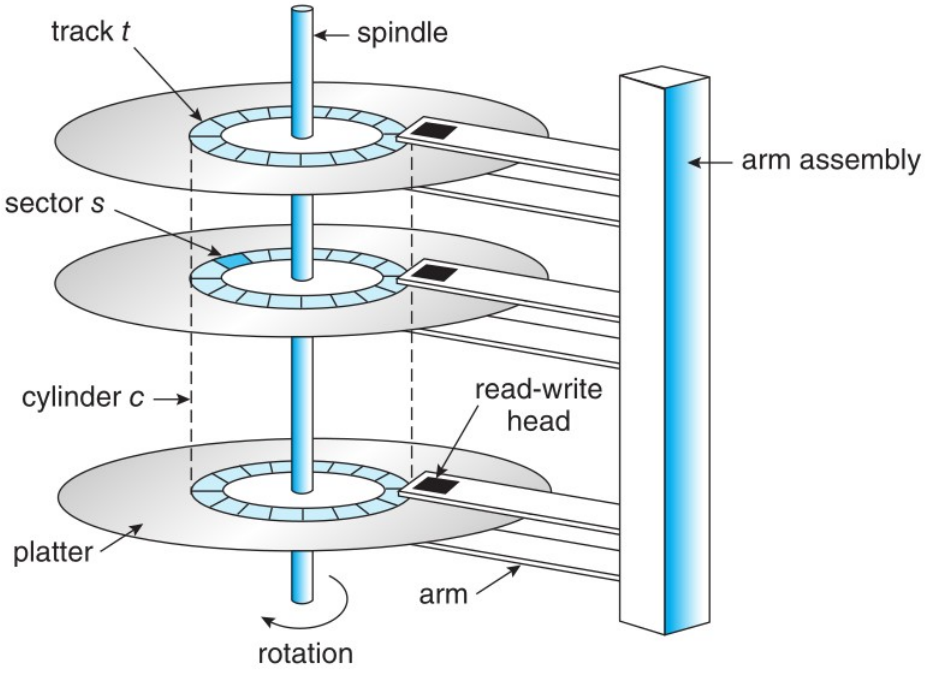


Figure 11.1 HDD moving-head disk mechanism.

Sistemas Operativos

Trabajo Práctico N.º 8

1. ¿Es absolutamente esencial la llamada al sistema open en UNIX? ¿Cuáles serían las consecuencias de no tenerla?

Los i-nodos y file objects son mecanismos usados para acceder a los archivos. Un i-nodo es una estructura de datos que contiene punteros a los bloques del disco que contienen el contenido del archivo, y un file object representa un puntero de acceso a los datos en un archivo abierto. Un thread no puede acceder al contenido de un i-nodo sin primero obtener el file object que apunta al i-nodo. El file object realiza seguimiento de dónde en el archivo el proceso está leyendo o escribiendo, para hacer seguimiento del archivo secuencial de I/O.

Los file objects generalmente pertenecen a un sólo proceso, pero los objetos i-nodo no. Hay una sola instancia de file object para cada instancia de un archivo abierto, pero siempre solamente un solo objecto i-nodo. Aún cuando un archivo no es usado por ningún proceso, su objeto i-nodo puede aún estar cacheado por el VFS (Virtual File System) para mejorar la performance si el archivo es usado nuevamente en el futuro.

Los archivos directorio son tratados de manera ligeramente diferente de los otros archivos. La interfaz de programación de UNIX define un número de operaciones en directorios, como crear, eliminar y renombrar un archivo en un directorio. Las llamadas al sistema para esas operaciones de directorio no requieren que el usuario abra los archivos en cuestión, a diferencia del caso de leer o escribir datos. El VFS por lo tanto define esas operaciones de directorio en el objeto i-nodo, en lugar de en el file object.

2. Ciertos sistemas proporcionan una llamada al sistema rename para cambiar el nombre de un archivo. ¿Existe alguna diferencia entre el uso de esta llamada para cambiar el nombre a un archivo y un procedimiento en el que primero se copie el archivo a otro nuevo con el nuevo nombre, y después se elimine el archivo antiguo?

Operaciones sobre archivos:

- ***Crear un archivo:*** dos pasos son necesarios. Primero, debe ser hallado un espacio para el archivo en el sistema de archivos. Segundo, una entrada para el nuevo archivo debe ser hecha en un directorio.
- ***Abrir un archivo:*** para no tener que especificar el nombre del archivo en todas las operaciones, causando que el sistema operativo evalúe el nombre, chequee los permisos de acceso, étc., todas las operaciones excepto la creación y la eliminación requieren que se ejecute primero un `open()`. Si es exitosa, la llamada open retorna un manejador de archivo que es usado como argumento en otras llamadas.
- ***Leer un archivo:*** para leer de un archivo, se utiliza una llamada que especifica el manejador de archivo y dónde (en memoria) el próximo bloque del archivo debe ser colocado. Nuevamente, el sistema necesita mantener un *puntero de lectura* a la locación en el archivo donde la próxima lectora está por tomar lugar, si es

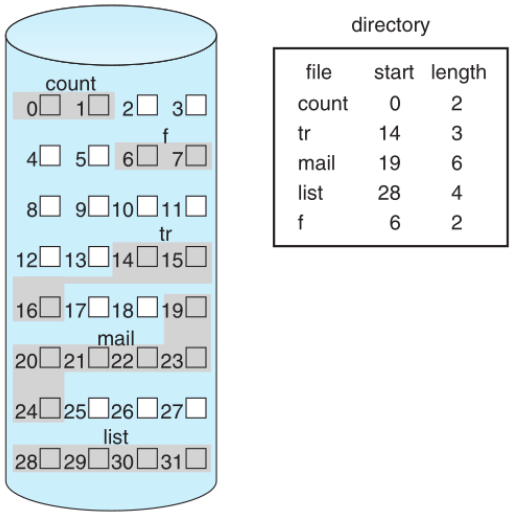
secuencial. Una vez que la lectura fue realizada, el puntero de lectura es actualizado. A causa de que un proceso es usualmente tanto para lectura o escritura de un archivo, la ubicación de la operación actual puede ser mantenida como un puntero de posición actual del archivo (current-file-position pointer). Tanto las operaciones de lectura y escritura usan ese mismo puntero, ahorrando espacio y reduciendo la complejidad del sistema.

- ***Reposicionamiento dentro de un archivo:*** el punteo de posición actual del archivo (current-file-position pointer) del archivo abierto es reposicionado a un valor dado. El reposicionamiento dentro de un archivo necesita no tiene porqué involucrar ningún I/O real. Esta operación es conocida también como búsqueda dentro de un archivo (file seek).
- ***Borrar un archivo:*** para borrar un archivo se debe buscar el directorio para el archivo nombrado. Habiendo encontrado la entrada al directorio asociado, se libera todo el espacio del archivo, así puede ser reutilizado por otros archivos, y se borra o marca como libre la entrada del directorio. Se debe notar que algunos sistemas permiten los hard links (múltiples nombres y entradas de directorio) para el mismo archivo. En ese caso, el contenido real del archivo no se eliminar hasta que el último link es eliminado.
- ***Truncar un archivo:*** el usuario puede querer eliminar los contenidos de un archivo pero mantener sus atributos. En vez de forzar al usuario a eliminar el archivo y entonces recrearlo, esta función permite a todos los atributos permanecer sin cambios (excepto por la longitud del archivo). El archivo entonces puede ser reseteado a la longitud cero y su espacio de archivo ser liberado.

Estas siete operaciones básicas comprender el conjunto mínimo requerido de operaciones sobre archivos.

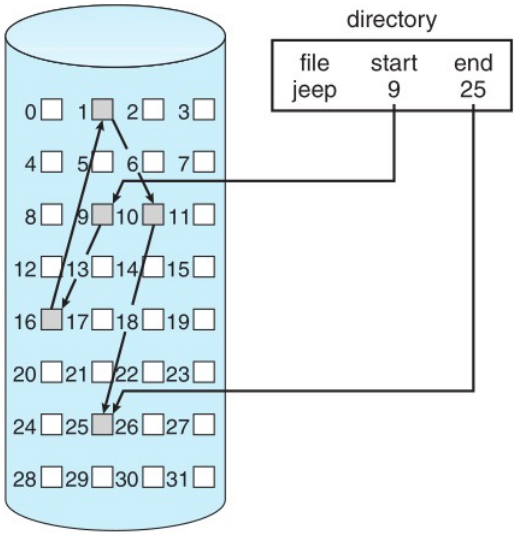
3. La asignación adyacente de los archivos produce fragmentación en el disco. ¿Es esta fragmentación interna o externa?

- ***Alocación continua:*** requiere que cada archivo ocupe un conjunto continuo de bloques en el dispositivo. La alocación continua presenta algunos problemas. Una dificultad es encontrar espacio para un nuevo archivo. El problema de la alocación continua puede ser visto como una aplicación particular del problema general de alocación de almacenamiento continuo, que involucra cómo satisfacer un requerimiento de tamaño *n* desde una lista de agujeros libres. First fit y best fit son las estrategias más comunes usadas para seleccionar un agujero desde un conjunto de agujeros disponibles.



Todos estos algoritmos sufren del problema de la fragmentación externa. A medida que los archivos son alocados y eliminados, el espacio libre de almacenamiento es roto en pequeñas piezas. La fragmentación externa existe cuando el espacio libre es roto en chunks. Se vuelve un problema cuando el chunk continuo más largo es insuficiente para el requerimiento; el almacenamiento es fragmentado en un número de agujeros, ninguno de los cuales es lo suficientemente grande para almacenar los datos.

- ***Alocación enlazada:*** cada archivo es una lista enlazada de bloques de almacenamiento, los cuales pueden estar dispersos en el dispositivo. El directorio contiene un puntero al primer y último bloque del archivo, y cada bloque contiene un puntero al bloque siguiente.



Para crear un nuevo archivo, simplemente se crea una nueva entrada en el directorio, con el puntero inicializado en `null` que significa que es un archivo vacío. Una escritura en el archivo hace que el sistema administrador de espacio libre busque un bloque libre, y este nuevo bloque es escrito y enlazado al final del archivo. Para leer un archivo, simplemente se leen los bloques siguiendo los punteros bloque a bloque.

No hay fragmentación externa con alocación enlazada, y cualquier bloque libre en la lista de espacio libre puede ser usado para satisfacer el requerimiento. El tamaño de un archivo no necesita ser declarado cuando el archivo es creado. Un archivo puede continuar creciendo tanto como haya bloques libres disponibles. En consecuencia, nunca es necesario compactar el espacio del disco.

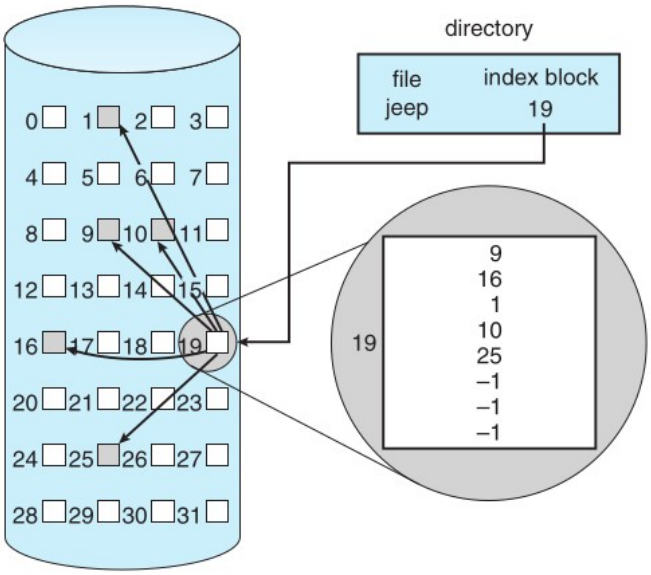
Las desventajas de la alocación enlazada son:

- Puede ser usada efectivamente solamente para archivos de acceso secuencial. Para encontrar el *i*-ésimo bloque de un archivo, se debe comenzar por el principio de ese archivo y seguir los punteros hasta llegar el *i*-ésimo bloque
- Otra desventaja es el espacio requerido para los punteros. Si un puntero requiere 4 bytes de un bloque de 512 bytes, entonces el 0,78% del disco es usado para punteros en vez de para información.

- ***Alocación indexada:*** cada archivo tiene su propio bloque de índices, el cuál es un arreglo de direcciones de bloques de almacenamiento. La *i*-ésima entrada en el bloque de índices apunta al *i*-ésimo bloque del archivo. El directorio contiene la dirección del bloque de índices. Para encontrar y leer el *i*-ésimo bloque, se usa el puntero en la *i*-ésima entrada del bloque de índices (index-block).

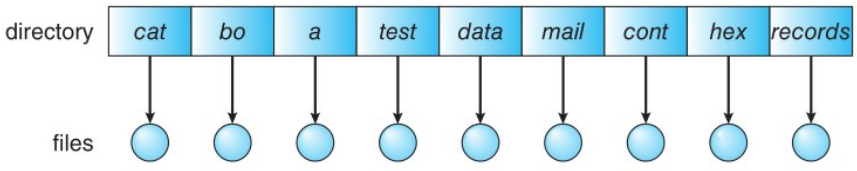
  Cuando el archivo es creado, todos los punteros en el bloque de índice son asignados a `null`. Cuando el *i*-ésimo bloque es escrito por primera vez, un bloque es obtenido desde el administrador de espacio libre, y su dirección es puesta en la *i*-ésima entrada del bloque de índices (index-block).

  La locación indexada soporta acceso directo, sin sufrir de fragmentación externa, porque cualquier bloque libre en el dispositivo de almacenamiento puede satisfacer el requerimiento por más espacio.



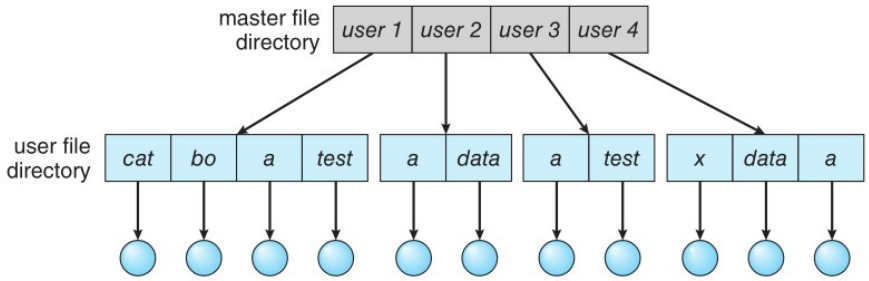
4. Suponga que se tiene un sistema operativo que admite un solo directorio, pero permite que el directorio tenga un número arbitrario de archivos, con nombres de archivos de longitud arbitraria. ¿Puede simularse algo parecido a un sistema jerárquico de archivos? ¿En qué forma?

- ***Directorio de nivel único:*** todos los archivos son contenidos en el mismo directorio. Ésto tiene significativas limitaciones, por ejemplo, cuando el número de archivos incrementa o el sistema tiene más de un usuario. Ya que todos los archivos están en el mismo directorio, deben tener nombres únicos.

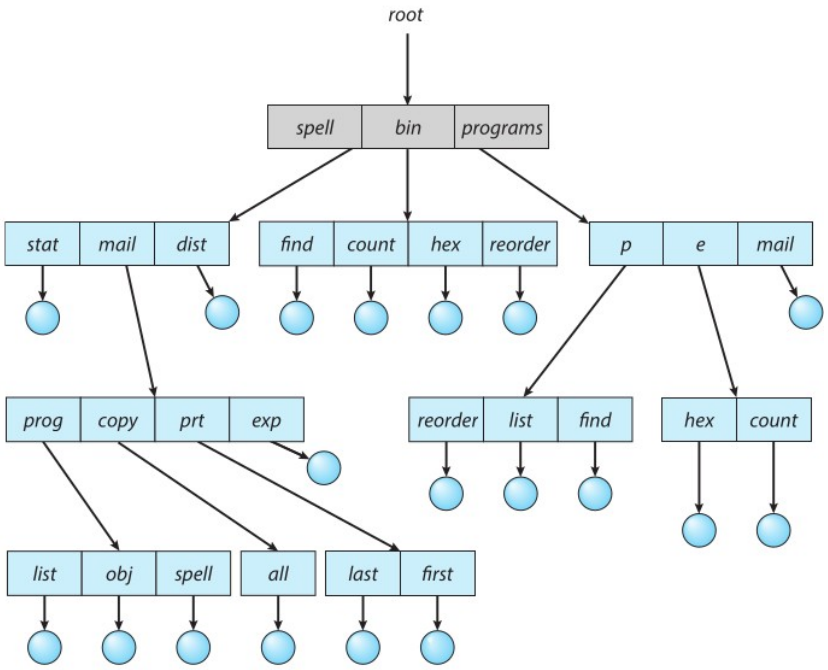


- ***Directorio de dos niveles:*** cada usuario tiene su propio *user file directory* (UFD). Los UFD tienen estructuras similares, pero cada uno lista solamente los archivos de un sólo usuario. Cuando una tarea de usuario comienza o un usuario se loggea, se busca el *master file directory* (MFD) del sistema. El MDF es indexado por nombre de usuario o número de cuenta, y cada entrada apunta al UFD para ese usuario.

Cuando un usuario refiere a un archivo en particular, la búsqueda es realizada solamente en su UFD. Así, diferentes usuarios pueden tener archivos con el mismo nombre.

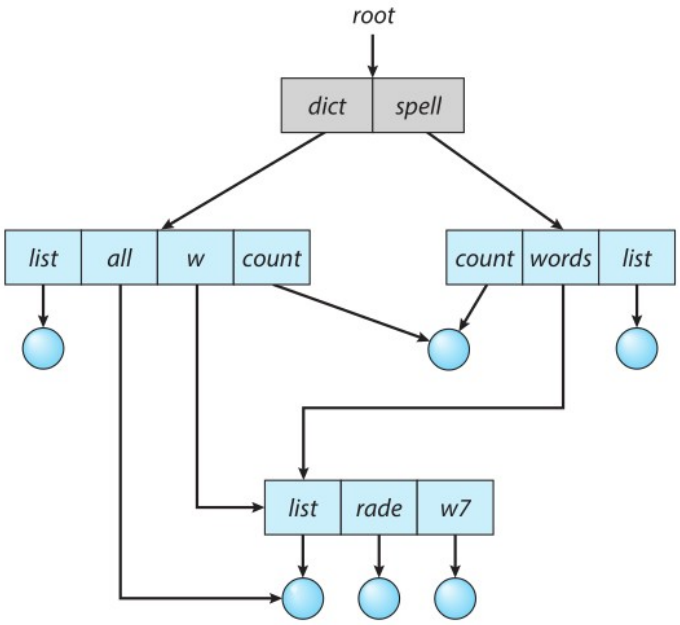


- ***Directorio estructurados en forma de árbol:*** es una generalización de estructura de directorio extendida a un árbol de altura arbitraria. Esta generalización permite que los usuarios creen sus propios subdirectorios. El árbol tiene su directorio raíz y cada archivo en el sistema tiene un único *path name*.

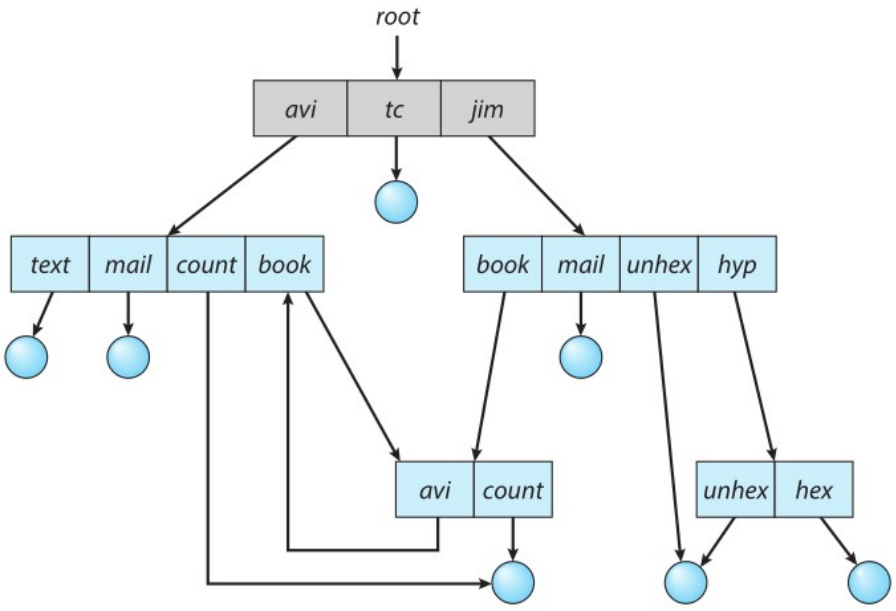


Un directorio (o subdirectorio) contiene un conjunto de archivos o subdirectorios. En muchas implementaciones, un directorio es simplemente otro archivo, pero es tratado de manera especial. Un bit en cada entrada de directorio define si la entrada es un archivo (0) o un subdirectorio (1).

- ***Directorio de grafo acíclico:*** permite a los directorio compartir subdirectorios y archivos. El mismo archivo o subdirectorio puede estar en dos diferentes directorios. Es importante notar que un archivo (o directorio) compartido no es lo mismo que dos copias del archivo.



- ***Directorio de grafo general:*** cuando se agregan links, la estructura de árbol es destruída, resultando en una estructura de un grafo simple.



Si los ciclos son permitidos en un directorio, se necesita evitar que en una búsqueda se busque un componente dos veces, por razones de correctitud así como de performance. Un algoritmo mal diseñado puede resultar en un loop infinito que continuamente busque a través del ciclo y nunca termine.

5. Los directorios pueden ser implementados como archivos especiales que sólo pueden ser accedidos de maneras limitadas ó como archivos de datos comunes. ¿Cuáles son las ventajas y desventajas de estas aproximaciones?

PREGUNTAR EN UNA CONSULTA

7. Considere un sistema de archivos que no soporta operaciones de apertura y cierre de archivos (open y close). Cada operación de lectura y escritura debe especificar la ruta simbólica del archivo.

a) Asuma operaciones de lectura/escritura orientadas a bloques, cada una de las cuales accede a un bloque de longitud fija. ¿Cuáles tareas ejecutadas normalmente por las operaciones open/close deben ser ejecutadas por cada operación de lectura/escritura?

PREGUNTAR EN UNA CONSULTA

b) ¿Es posible soportar lectura/escritura secuencial cuando no está implementada la Tabla de Archivos Abiertos que mantiene información de la posición actual y el buffer de lectura/escritura?

8. ¿Qué problemas pueden ocurrir en un sistema si se permite que el sistema de archivos (file system) sea montado simultáneamente en más de un lugar?

PREGUNTAR EN UNA CONSULTA

9. ¿Por qué el mapa de bits para la asignación de archivos debe mantenerse en el almacenamiento secundario, en lugar de en la memoria principal?

Un bitmap es una cadena de *n* dígitos binarios que pueden ser usados para representar el estado de *n* items. Por ejemplo, supongamos que tenemos varios recursos, y la disponibilidad de cada recurso es indicada por el valor de un dígito binario: 0 significa que el recurso está disponible, mientras que 1 indice que no está disponible (o viceversa). El poder de los bitmaps se torna visible cuando consideramos la eficiencia de su espacio. Si usáramos un valor booleano de 8 bits en lugar de un solo bit, la estructura de datos resultante sería 8 veces más grande. Por eso, los bitmaps son usados comunmente cuando necesitamos representar la disponibilidad de un número grande de recursos.

Frecuentemente, la lista de espacio libre es implementada como un bitmap o vector de bits. Cada bloque es representado por 1 bit. Si el bloque está libre, el bit es 1; si el bloque está alocado, el bit es 0.

Por ejemplo, consideremos un disco donde los bloques 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 y 27 están libres y el resto de los bloques están alocados. El bitmap de espacio libre sería:

001111001111110001100000011100000 ...

La principal ventaja de esta aproximación es su simplicidad relativa y su eficiencia encontrando el primer bloque libre o *n* bloques libres consecutivos en el disco.

Desafortunadamente, los vectores de bits son ineficientes al menos que todo el vector sea mantenido en memoria principal (y esté escrito en el dispositivo que contiene el sistema de

archivos para ocasionales necesidades de recuperación). Mantenerlo en memoria principal es posible para dispositivos pequeños, pero no necesariamente para los grandes. Un disco de 1,3 GB con bloques de 512 bytes necesitaría un bitmap de más de 332 KB para hacer un seguimiento de los bloques libres, mientras que un disco de 1 TB con bloques de 4 KB requerirá 32 MB para almacenar su bitmap. Dado que el tamaño del disco incrementa constantemente, el problema con los vectores continuará escalando también.

10. Un verificador de sistema de archivos tiene integrado sus contadores de la siguiente manera:

Bloques en uso: 1 0 1 0 0 1 0 1 1 0 1 0 0 1 0
Bloques libres: 0 0 0 1 1 1 0 0 0 1 0 1 1 0 1

• ¿Hay algún error? ¿Porqué? ¿Cómo se solucionaría?

• El corregir la estructura del sistema de archivos cuando se detecta un error en el chequeo me asegura que se mantiene una estructura consistente, ¿también me asegura que el significado de la información almacenada en el disco es semánticamente correcta?

El verificador de consistencia compara la información en la estructura del directorio y otra medatada con el estado en el almacenamiento e intenta arreglar cualquier inconsistencia que encuentre.

Pueden ser llevados a cabo dos tipos de checkeos de consistencia: de bloques y de archivos. Para checkear por consistencia de bloque, el programa crea dos tablas, cada una conteniendo un contador para cada bloque, inicialmente seteado en 0. Los contadores en la primer tabla hacen seguimiento de cuantas veces cada bloque está presente en un archivo. Los contadores en la segunda tabla graban con qué frecuencia cada bloque está presente en la lista de libres (o el bitmap de bloques libres).

Si el sistema de archivos es consistente, cada bloque tendrá un 1 ya sea en la primera o la segunda tabla, como se ilustra en la Fig. 4-27(a). Sin embargo, como resultado de una falla, las tablas podrían verse como Fig. 4-27(b), en la cuál el bloque 2 no ocurre en ninguna tabla. Serpa reportado como un bloque faltante (missing block). Mientras que los bloques faltantes no causan daño real, desperdician espacio y por lo tanto reducen la capacidad del disco. La solución para bloques faltantes es sencilla: el verificador de sistema de archivos solamente los agrega a la lista de libres.

Otra situación que puede ocurrir es la de Fig. 4-27(c). Aquí se ve un bloque, el número 4, que ocurre dos veces en la lista de libres (duplicados pueden suceder solamente si la lista de libres es realmente una lista, con un bitmap es imposible). La solución aquí también es simple: reconstruír la lista de libres.

Lo peor que puede ocurrir es que el mismo bloque de datos esté presente en dos o más archivos, así como se muestra en Fig. 4-27(d) con el bloque 5. Si alguno de esos archivos es eliminado, el bloque 5 será puesto en la lista de libres, llevando a una situación en la cuál el mismo bloque está en uso y libre al mismo tiempo. Si ambos archivos son eliminados, el bloque será puesto en la lista de libres dos veces.

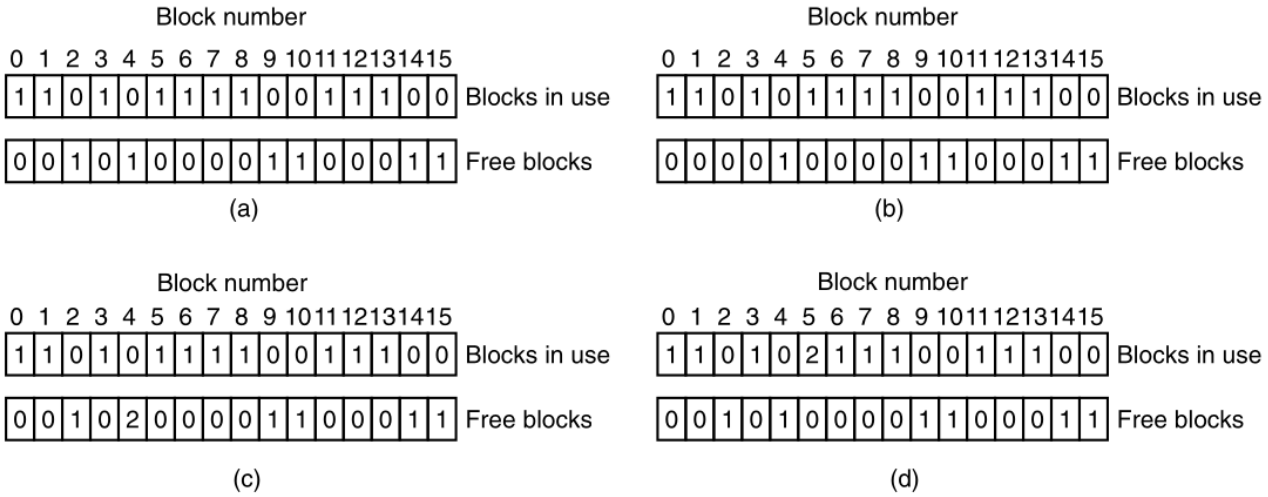


Figure 4-27. File-system states. (a) Consistent. (b) Missing block. (c) Duplicate block in free list. (d) Duplicate data block.

La acción apropiada para el verificador de sistemas de archivos es alocar un bloque libre, copiar los contenidos del bloque 5 en él, e insertar la copia en uno de los archivos. De esta manera, la información contenida de los archivos no cambia (aunque con seguridad una está confusa) pero la estructura del sistema de archivos es hecha consistente.

11. En los sistemas UNIX se sugiere que la primera parte de cada archivo se mantenga dentro del mismo bloque de disco que el i-nodo. ¿Cuáles son las ventajas de esto?

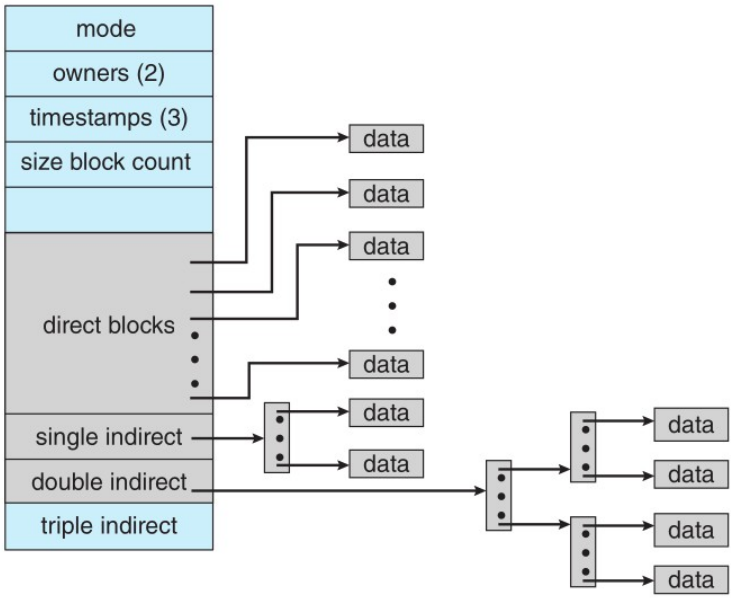


Figure C.7 The UNIX inode.

Un archivo es representado por un inodo, el cuál es un registro que almacena la mayoría de la información acerca de un archivo específico en el disco.

El inodo contiene los identificativos de usuario y grupo del archivo, los tiempos de última modificación y acceso, un contador del número de hard links (entradas de directorio) al archivo, y el tipo de archivo (archivo plano, directorio, link simbólico, carácter del

dispositivo, bloque del dispositivo o socket). Además, el inodo contiene 15 punteros a bloques del disco que contienen la información contenida en el archivo. Los primeros 12 de esos punteros apuntan a bloques directos. Esto es, contienen direcciones de bloques que contienen la información del archivo. De este modo, la información de archivos pequeños (no más de 12 bloques) puede ser referenciada inmediatamente, porque una copia del inodo es mantenida en memoria principal mientras el archivo está abierto.

Los próximos tres punteros en el inodo apuntan a bloques indirectos. Si el archivo es lo suficientemente grande para usar bloques indirectos, cada uno de los bloques indirectos es del tamaño del bloque principal; el tamaño del fragmento aplica solamente a bloques de datos. El primer puntero de bloque indirecto direcciona a un único bloque indirecto (single indirect block). El único bloque indirecto (single indirect block) es un bloque de índices que no contiene datos sino direcciones de bloques que contienen los datos. Luego hay un puntero de bloque indirecto doble (double-indirect-block pointer), la direcciones de un bloque que contiene las direcciones de bloques que contienen punteros a los bloques de datos. El último puntero contendría la dirección de un bloque indirecto triple (triple indirect block), sin embargo, no hay necesidad de él.

12. ¿Cuáles son los diferentes métodos para compartir archivos entre directorios?

En la primera solución, los bloques de disco no son listados en directorios, sino en pequeñas estructuras de datos asociadas con el archivo en sí mismo. Los directorios apuntarían entonces solamente a esta pequeña estructura de datos. Esta aproximación es usada en UNIX (donde la pequeña estructura de datos es el i-nodo).

Como desventaja, al momento en el que *B* linkea a un archivo compartido, el i-nodo registra a *C* como el propietario (owner) del archivo. Crear un link no cambia su propiedad (ownership), pero incrementa el contador de links en el i-nodo, así el sistema sabe cuantas entradas de directorio apuntan al archivo.

Si *C* después trata de remover el archivo, el sistema enfrenta un problema. Si remueve el archivo y limpia el i-nodo, *B* tendrá una entrada de directorio apuntando a un i-nodo inválido. Si el i-nodo es liego reasignado a otro archivo, el enlace de *B* estará apuntando a un archivo erróneo. El sistema puede ver desde el contador del i-nodo que el archivo está aún en uso, pero no hay una manera fácil de encontrar todas las entradas de directorio para ese archivo, con el objetivo de eliminarlo. Los punteros a los directorios no pueden ser almacenados en el i-nodo porque puede haber un número ilimitado de directorios.

Lo único que se puede hacer es remover la entrada de directorio de *C*, pero dejar el i-nodo intacto con el contador seteado a 1. Ahora se tiene una situación en la que *B* es el único usuario teniendo una entrada de directorio para un archivo propiedad de *C*. Si el sistema cuenta o tiene cuotas, *C* continuará siendo facturado por el archivo hasta que *B* decida removerlo, si alguna vez lo hace, y cuando el contador llega a 0 el archivo es eliminado.

En la segunda solución, *B* enlaza a uno de los archivos de *C* haciendo que el sistema cree un nuevo archivo, del tipo LINK, e ingresando ese archivo en el directorio *B*. El nuevo archivo contiene solamente la dirección del archivo al que es linkeado. Cuando *B* lee de un archivo linkeado, el sistema operativo ve que el archivo siendo leido es del tipo LINK, observa el nombre del archivo y lee ese archivo. Esta aproximación es llamada *symbolic linking* (enlazamiento simbólico) en contraste con el *hard linking* (enlazamiento tradicional).

El problema con los links simbólicos es el gasto adicional requerido. El archivo conteniendo la ruta debe ser leído, entonces la ruta debe ser parseada y seguida, componente a componente, hasta que el i-nodo es alcanzado. Toda esta actividad puede requerir un considerable número de accesos a disco extra. Además, un i-nodo extra es neceario para cada link sumbólico, como es un bloque de disco extra para almacenar la ruta, aunque si el nombre de la ruta es corto, el sistema puede almacenarla en el mismo i-nodo, como un tipo de optimización. Los links simbólicos tienen la ventaja de que pueden ser usados para enlazar a archivos en máquinas en cualquier otra parte del mundo, simplemente proveyendo la dirección de red de la máquina donde el archivo reside además de su ruta en essa máquina.

También hat otro problema introducido por los links, simbólicos o de otra manera. Cuando los links son permitidos, los archivos pueden tener dos o más rutas. Los programas que comienzan en un directorio dado y buscan todos los archivos en tal directorio y sus subdirectorios encontrará un archivo linkeado múltiples veces.

14. Explique las diferencias entre un hard y un soft link. Cuáles son las ventajas y desventajas de usar unos u otros.

El linkeo es una técnica que permite a un archivo aparecer en más de un directorio. Ésta llamada al sistema especifica un archivo existente y el nombre de una ruta, y crea un link desde el archivo existente al nombre especificado de la ruta. De esta manera, el mismo archivo puede aparecer en múltiples directorios. Un link de este tipo, el cual incrementa el contador del i-nodo del archivo (para hacer seguimiento del número de entradas de directorio conteniendo el archivo) es usualmente llamado *hard link*.

Una variante de la idea del linkeado de archivos es el link simbólico. En vez de tener dos nombres apuntando a la misma estructura interna representando un archivo, puede ser creado un nombre que apunta a un pequeño archivo nombrando otro archivo. Cuando el primer archivo es usado, por ejemplo, abierto, el sistema de archivos sigue la ruta y encuentra el nombre al final. Entonces empieza el proceso de búsqueda completo usando el nuevo nombre. Los links simbólicos tiene una ventaja de que pueden cruzar los límites de los discos e incluso nombrar archivos en computadoras remotas. Aunque su implementación es un poco menos eficiente que los hard links.

16. ¿En que casos para llevar la lista de bloques libres el mapa de bits es más económico que la lista enlazada?

Frecuentemente, la lista de espacio libre es implementada como un mapa de bits. Cada bloque es representado por 1 bit. Si el bloque está libre, el bit es 1; si el bloque está alocado, el bit es 0.

La principal ventaja de esta aproximación es su simplicidad relativa y su eficiencia buscando el primer bloque libre o *n* bloques libres consecutivos en el disco. Sin embargo, los mapas de bits son ineficientes al menos que el mapa entero sea mantenido en memoria principal. Mantenerlo en memoria principal es posible para dispositivos de almacenamiento pequeños, pero no necesariamente para grandes. Un disco de 1,3 GiB con bloques de 512 bytes necesitará un mapa de bits de más de 332 KiB para hacer seguimiento de los bloques libres, aunque colocando los bloques en cluster en grupos de 4 reduce el

número al rededor de 83 KiB por disco. Un disco de 1 TiB con bloques de 4 KiB requerirá 32 MiB ($2^{(40)}/2^{(12)}=2^{(28)} bits=2^{(25)} bytes=2^5 MiB$, donde 2^{40}=1 TiB, 2^{12} = 4 KiB)

Otra aproximación al manejo del espacio libre es enlazar juntos todos los bloques libres, manteniendo un puntero al primer bloque libre en una locación especial en el sistema de archivos y cachearlo en memoria. El primer bloque contiene un puntero al siguiente bloque libre y así consecutivamente. Este esquema no es eficiente; para atravezar la lista, debemos leer cada bloque, lo cual requiere un tiempo de I/O sustancial en los HDDs. Afortunadamente, atravesar la lista de libres no es una acción frecuente. Usualmente el sistema operativo simplemente necesita un bloque libre así puede alocar el bloque a un archivo.

Sistemas Operativos

Apuntes sobre seguridad, protección y virtualización

- Seguridad: es la medida de confianza que la integridad de un sistema y sus datos serán preservados. Decimos que un sistema es seguro si sus recursos son usados y accedidos de la manera que fueron destinados bajo todas las circunstancias.
- Protección: es el conjunto de mecanismo que controlan el acceso de procesos y usuarios a los recursos definidos por el sistema computacional.

Seguridad

Conceptos generales

- Amenaza: una potencial violación de seguridad, como el descubrimiento de una vulnerabilidad
- Ataque: intento de romper la seguridad

Tipo de violación	Descripción	Ejemplo
Confidencialidad	Este tipo de violación involucra la lectura no autorizada de datos (o robo de información). Típicamente, una brecha de confidencialidad es el objetivo de un intruso	Capturar datos secretos de un sistema o flujo de datos, como información de tarjetas de crédito o información de identidad para un ladrón de identidad, o películas aún no publicadas o código fuente
Integridad	Involucra modificación no autorizada de datos	Puede resultar en asignarle responsabilidad a una parte inocente, o modificar el código fuente de una aplicación importante
Disponibilidad	Involucra la destrucción no autorizada de datos	La mutilación de sitios web es un ejemplo común de este tipo d brecha de seguridad
Robo de servicio	Involucra uso no autorizado de recursos	Un intruso (o programa intruso) puede instalar un daemon en un sistema que actúa como servidor de archivos
Denegación de servicio	Involucra evitar el uso legítimo del sistema	Ataques DOS (Denial-of-service)

Tipo de ataque	Descripción
Enmascaramiento (Masquerading)	Un participante en una comunicación finge ser alguien más (otro host u otra persona). A través del enmascaramiento, los ataques violan la autenticación (correctitud de la identificación); pueden ganar acceso que normalmente no podrían
Ataque de reproducción (Replay Attack)	Consiste en la repetición maliciosa o fraudulenta de datos válidos transmitidos
Modificación de mensajes (Message Modification)	El atacante cambia datos en una comunicación sin que el emisor lo sepa
Ataque Man-in-the-middle	El atacante se sitúa en medio del flujo de datos en una comunicación, enmascarándose como emisor al receptor y viceversa
Toma de control de sesión	Una sesión activa de comunicación es interceptada

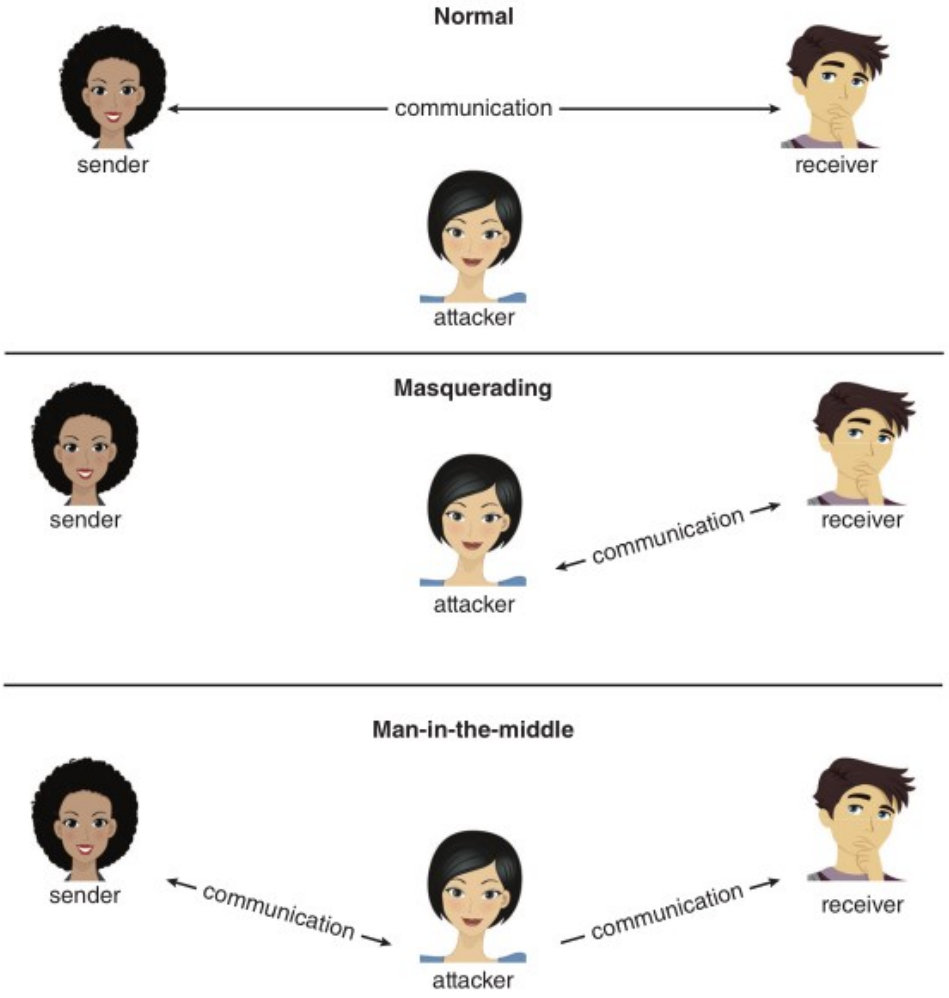


Figure 16.6 Standard security attacks.[1]

Niveles de seguridad

Para proteger un sistema, debemos tomar medidas de seguridad en cuatro niveles:

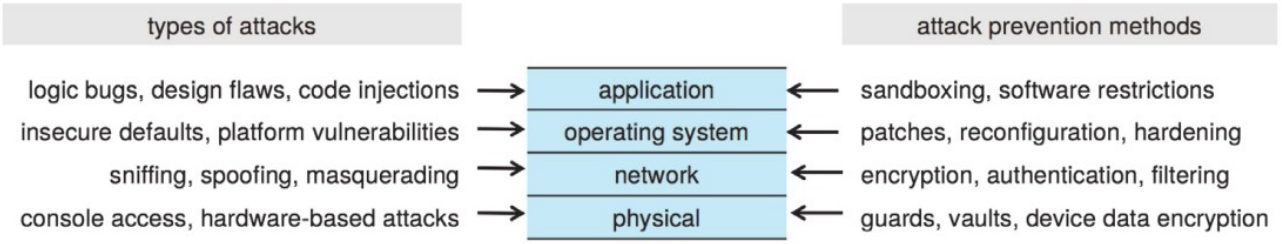


Figure 16.1 The four-layered model of security.

1. **Físico:** el sitio o sitios que contienen los sistemas computacionales deben ser asegurados físicamente contra la entrada de intrusos. Tanto los cuartos de máquinas y las terminales o computadoras que tienen acceso a las máquinas objetivo deben ser aseguradas, por ejemplo, limitando el acceso al edificio en donde se encuentran, o bloqueándolas en el escritorio en el que se encuentran.
2. **Red:** los datos de computadoras en los sistemas modernos frecuentemente viajan sobre líneas privadas alquiladas, líneas compartidas como Internet, conexiones inalámbricas y líneas de acceso telefónico. Interceptar esos datos puede ser tan dañino como entrar en una computadora, y la interrupción de las comunicaciones pueden constituir un ataque de denegación de servicio, disminuyendo el uso de los usuarios y la confianza en el sistema.
3. **Sistema operativo:** los sistemas operativos deben ser mantenidos actualizados (a través de los parches continuos), y configurados y modificados para decrementar la superficie de ataque y evitar la penetración. La superficie de ataque es el conjunto de puntos en los cuales un atacantes puede intentar entrar en el sistema.
4. **Aplicación:** algunas aplicaciones son inherentemente maliciosas, pero aún las aplicaciones benignas pueden contener *bugs* de seguridad. Debido al vasto número de aplicaciones de terceros y sus bases de código dispar, es virtualmente imposible asegurar que todas las aplicaciones sean seguras.

Otro factor que no puede ser ignorado es el factor humano. La autorización debe ser realizada cuidadosamente para asegurar que solamente los usuarios permitidos, confiables tienen acceso al sistema. Aún los usuarios autorizados pueden ser maliciosos o ser “alentados” a permitir a otros usar su acceso, ya sea de manera voluntaria o engañados a través de la ingeniería social, la cual usa el engaño para persuadir a personas para que entreguen información confidencial. Un tipo de ataque de ingeniería social es el *phishing*, en donde un e-mail o página web de apariencia legítima engaña a un usuario para que entregue información confidencial.

Programas peligrosos

El *malware* es un software designado para explotar, inhabilitar o dañar sistemas computacionales.

Malware	Descripción	Ejemplo
Caballo de Troya (*Trojan Horse*)	Es un programa que actúa de manera clandestina o maliciosa, en vez de realizar simplemente su función declarada. Si el programa es ejecutado en otro dominio, puede escalar privilegios Los troyanos especialmente prosperan en casos donde hay una violación del principio de privilegios mínimos	Una aplicación móvil que pretende proveer alguna funcionalidad benigna (como una app de linterna), pero mientras tanto a escondidas accede a los contactos o mensajes del usuario y los contrabanda a un servidor remoto
Puerta trampa (o puerta trasera) (*trap door* ó *back door*)	El diseñador de un programa o sistema deja un agujero en el software que únicamente él es capaz de usar	El código puede checkear por un ID o contraseña específicos y puede evitar los procedimientos normales de seguridad cuando recibe tal ID o contraseña
Bomba lógica (*logic bomb*)	Una puerta trampa que puede ser puesta a operar solamente bajo un conjunto específico de condiciones lógicas	Un administrador de red tiene una reconfiguración destructiva de la red de su compañía para ser ejecutada cuando el programa detecta que ya no es un empleado de la compañía

Rebalse de buffer (buffer overflow)

El siguiente programa ilustra un *overflow*, que ocurre debido a una operación de copia sin límites, la llamada a `strcpy()`.

```
#include <stdio.h>
#define BUFFER_SIZE 0

int main(int argc, char *argv[])
{
   int j = 0;
   char buffer[BUFFER_SIZE];
   int k = 0;
   if (argc < 2) {return -1;}

   strcpy(buffer,argv[1]);
   printf("K is %d, J is %d, buffer is %s\n", j,k,buffer);
   return 0;
   }
}
```

Figure 16.2 C program with buffer-overflow condition.

La función copia sin considerar el tamaño del buffer en cuestión, parando solamente cuando es encontrado un byte `NULL` (`\0`). Si tal byte se encuentra antes de que `BUFFER_SIZE` es alcanzado, el programa se comporta como es esperado. Pero el copy puede fácilmente exceder el tamaño del buffer.

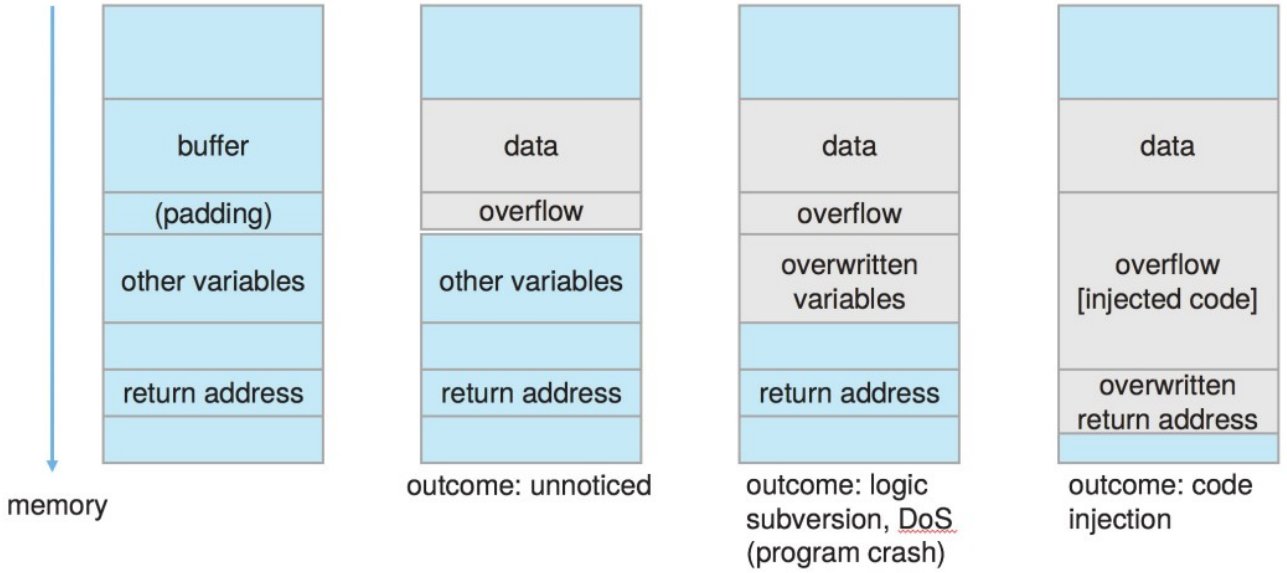


Figure 16.3 The possible outcomes of buffer overflows.

Si el *overflow* excede grandemente el padding, todo el *frame* de la pila de la función actual es sobrescrito. En el tope del *frame* está la dirección de retorno de la función, la cual es accedida cuando la función retorna. El flujo del programa es alterado y puede ser redirigido por el atacante a otra región de memoria, incluyendo memoria controlada por el atacante. El código inyectado es entonces ejecutado, permitiendo que el atacante corra código arbitrario con el ID del proceso efectivo.

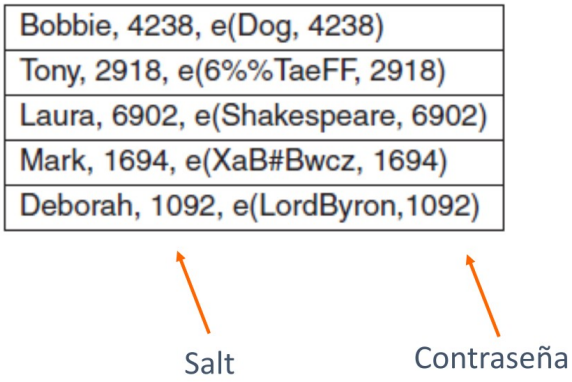
Amenazas al sistema y red

- **Virus:** un virus es un fragmento de código incorporado en un programa legítimo. Los virus son auto-replicantes y están diseñados para "infectar" otros programas. Pueden causar estragos en un sistema modificando o destruyendo archivos y causando fallas del sistema y el mal funcionamiento de programas. Los virus son muy específicos a arquitecturas, sistemas operativos y aplicaciones.
- **Gusanos (worms):** debe ser hecha una distinción entre virus, lo cuales requieren actividad humana, y *worms*, que usan una red para replicarse sin ayuda de humanos.
- **Negación de servicio (Denial of Service):** no tienen como objetivo obtener información o robar recursos, sino interrumpir el legítimo uso de un sistema o instalación. Los ataques de negación de servicio son generalmente basados en la red. Pueden clasificarse en dos categorías:
    - Los de la pirmera categoría usan tantos recursos de la instalación que, en esencia, no puede realizarse ningún trabajo útil.
    - La segunda categoría involucra interrumpir la red de la instalación.

- **Escaneo de puertos (Port Scanning):** no es un ataque en sí mismo, pero es una manera de detectar las vulnerabilidades para atacar el sistema. Es típicamente automatizado, involucrando una herramienta que intenta crear una conexión TCP/IP o enviar un paquete UDP a un puerto específico o a un conjunto de puertos. El escaneo de puertos es usualmente una parte de una técnica de reconocimiento conocida como *fingerprinting*, en la cuál un atacante intenta deducir el tipo de sistema operativo siendo usado y el conjunto de servicios para identificar vulnerabilidades conocidas.

Autenticación usando contraseñas

Para evitar que las contraseñas sean crackeadas por ataques de fuerza bruta, los sistemas incluyen una "*salt*", o un registro de un número random, en el algoritmo de hasheo. El valor de *salt* es añadido a la contraseña para asegurar que si dos contraseñas de texto plano son la misma, el resultaran en diferentes valores hash. Además, el valor de *salt* hace que un diccionario de hashing sea inefectivo, porque cada término del diccionario necesitaría ser combinado con cada valor de *salt* para comparar las contraseñas almacenadas.



Autenticación

La autenticación es la restricción del conjunto de potenciales emisores de un mensaje. La autenticación es entonces complementaria a la encriptación.

Un algoritmo de autenticación usando claves simétricas consiste en los siguientes componentes:

- Un conjunto K de claves
- Un conjunto M de mensajes
- Un conjunto A de autenticadores
- Una función $S : K \rightarrow (M \rightarrow A)$ tal que para cada k ϵ K, S_k es una función para generar autenticadores desde los mensajes. Tanto S como S_k para cada k deben ser funciones eficientemente computables
- Una función $S : K \rightarrow (M \times A \rightarrow true, false)$. Esto es, para cada k ϵ K, V_k es una función para verificar los autenticadores en los mensajes. Tanto V como V_k para cada k deben ser funciones eficientemente computables

La propiedad crítica que un algoritmo de autenticación debe poseer es que para un mensaje m, una computadora puede generar un autenticador a ϵ A tal que $V_k(m,a)=true$ solamente si posee k. Entonces, una computadora que tiene k puede generar autenticadores

en mensajes así cada computadora que posea *k* puede verificarlos. Sin embargo, una computadora que no tiene *k* no puede generar autenticadores en mensajes que puedan ser verificados usando V_k. Como los autenticadores están generalmente expuestos (por ejemplo, siendo enviados en un red con los mensajes), no debe ser factible derivar *k* de los autenticadores. Prácticamente, si $V_k(m,a)=true$, entonces sabemos que *m* no ha sido modificado y que el emisor del mensaje tiene *k*. Si compartimos *k* solamente con una entidad, entonces sabemos que el mensaje es originario de *k.*

Seguridad multinivel

Modelo de Bell-LaPadula

El modelo de Bell-LaPadula tiene reglas acerca de cómo puede ser el flujo de información:

1. **Propiedad de seguridad simple:** un proceso corriendo en el nivel de seguridad *k* puede leer solamente objetos a su mismo nivel o menor. Por ejemplo, un general puede leer los documentos de un teniente, pero un teniente no puede leer los documentos del general.
2. **Propiedad $*$:** un proceso corriendo a un nivel de seguridad puede escribir solamente objetos en su mismo nivel o mayor. Por ejemplo, un teniente puede adjuntar un mensaje a la casilla de correo del general diciéndole todo lo que sabe, pero un general no puede adjuntar un mensaje a la casilla de correo de un teniente diciendo todo lo que sabe porque el general puede haber visto documentos *top-secret* que pueden no haber sido revelados al teniente.

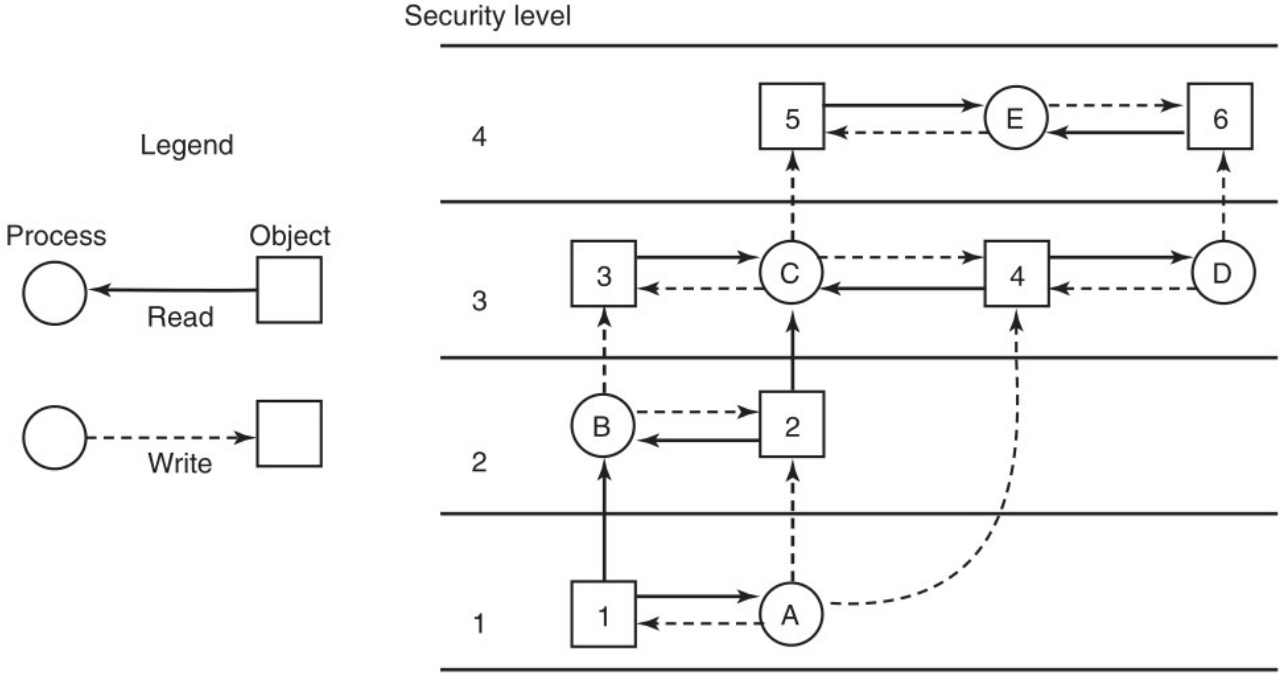


Figure 9-11. The Bell-LaPadula multilevel security model.

En resumen, los procesos pueden escribir hacia abajo y leer hacia arriba, pero no al revés.

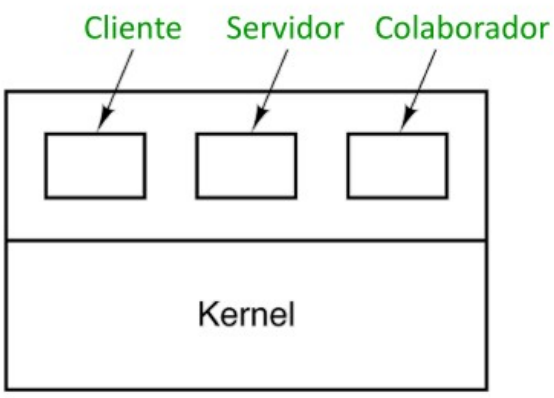
Modelo Biba

El problema con el modelo de Bell-LaPadula es que fue ideado para mantener secretos, no garantizar la integridad de los datos. Para lograr lo último, necesitamos las propiedades contrarias:

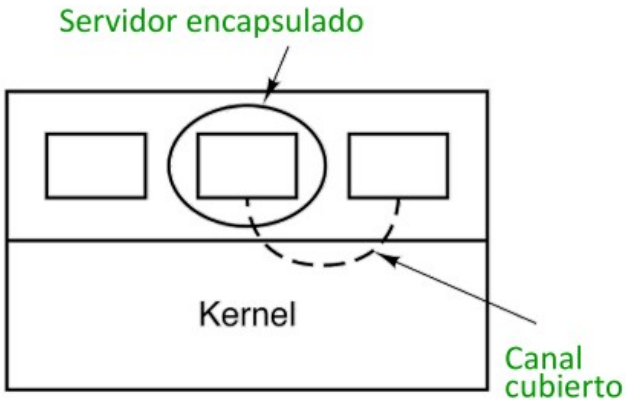
1. **Propiedad de integridad simple:** un proceso corriendo en un nivel de seguridad k puede escribir objetos solamente en su nivel o menor (no escribir hacia arriba)
2. **Propiedad $*$ de integridad:** un proceso corriendo en un nivel de seguridad k puede leer objetos solamente en su nivel o mayor (no leer hacia abajo)

Canales encubiertos

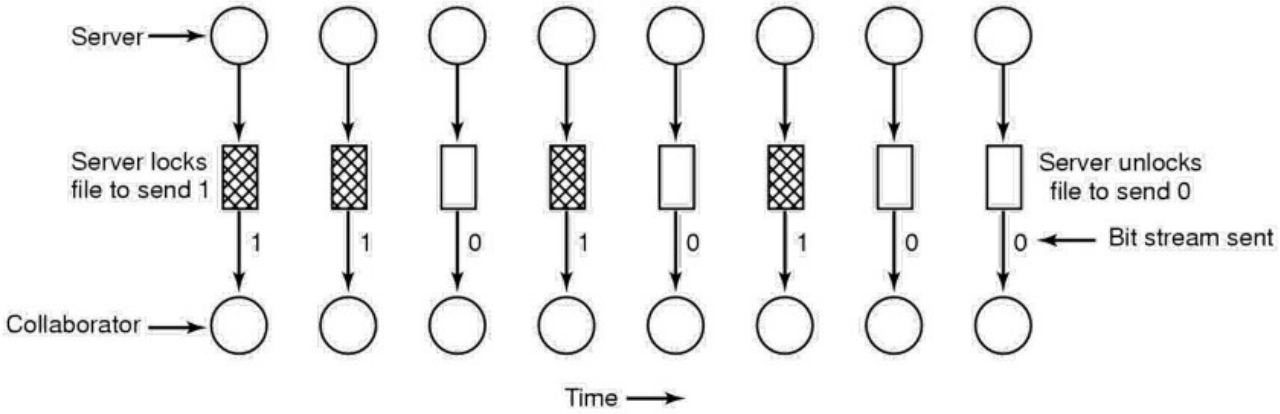
- Procesos cliente, servidor y colaborador



- El servidor encapsulado puede aún fugar datos a un colaborador via canales cubiertos



Un canal cubierto usando bloqueo de archivos (locking)



Protección

- **Principio del menor privilegio:** este principio dicta que los programas, usuarios y aún los sistemas deben tener los privilegios justos para desarrollar sus tareas.
- **Compartimentación:** es el proceso de proteger cada componente del sistema individualmente mediante el uso de permisos específicos y restricciones de acceso. Así, si un componente es alterado, otra línea de defensa entrará en acción y evitará que el atacante comprometa más al sistema
- **Defensa en profundidad:** múltiples capas de protección deben ser aplicadas una sobre la otra (por ejemplo, un castillo con una guarnición, un muro y un foso para protegerlo). Al mismo tiempo, los atacantes usan múltiples maneras de traspasar la defensa en profundidad, resultando en una carrera armamentística en constante aumento.

Anillos de protección

Para lograr la separación de privilegios, es necesario soporte de hardware. Todo hardware moderno soporta la noción de niveles separados de ejecución. Un modelo popular de separación de privilegios es el de anillos de protección. En este modelo la ejecución es definida como un conjunto de anillos concéntricos, con el anillo i proveyendo un subconjunto de funcionalidad del anillo j para todo $j < i$. El anillo más interior, el anillo *0*, por lo tanto provee el conjunto completo de privilegios.

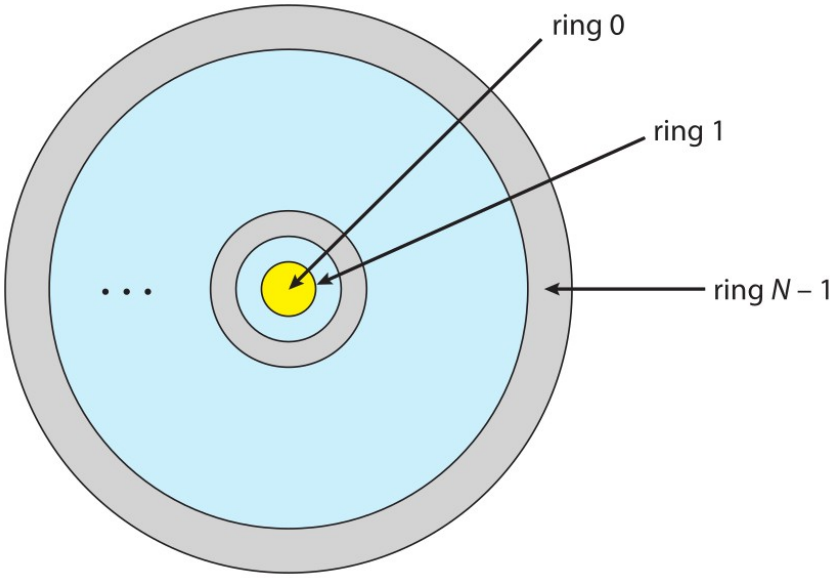


Figure 17.1 Protection-ring structure.

- Kernel: las arquitecturas Intel ubican el código en modo usuario en el anillo *3* y el código en modo kernel en el anillo *0*.
- Hipervisores: con el advenimiento de la virtualización, Intel definió un anillo adicional (*-1*) para permitir a los hipervisores, o administradores de máquinas virtuales, crear y ejecutar máquinas virtuales. Los hipervisores tienen mayores capacidades que los kernels de los sistemas operativos anfitriones.

- Trust Zone y Secure Monitor Call: la arquitectura de procesadores ARM inicialmente permitía solamente los modos USR (usuario) y SVC (kernel, supervisor). En los procesadores ARMv7, ARM intrudujo la TrustZone (TZ), la cual provee un anillo adicional. Este ambiente de ejecución más privilegiado también tiene un acceso exclusivo a características criptográficas respaldadas por hardware, tales como NFC Secure Element y una clave criptográfica en chip, que hace que se manejen contraseñas e información sensible de manera más segura. Aún el kernel en sí mismo no tiene acceso a la clave en crip, y puede solamente solicitar los servicios de encriptado y desencriptado del ambiente TrustZone (por medio de una instrucción especializada, Secure Monitor Call –SMC–) la cual es utilizable solamente en modo kernel. Como con las *system calls*, el kernel no tiene la habilidad de ejecutar directamente a direcciones específicas en la TrustZone, sino solamente a través de los argumentos vía registros.

Dominios de protección

Definimos como *objetos* a tanto objetos de hardware (como la CPU, segmentos de memoria, impresoras, discos y controladores de cinta) y objetos de software (como archivos, programas y semáforos. Cada objeto tiene un único nombre que lo diferencia de todos los otros objetos en el sistema, y cada uno puede ser accedido solamente a través de operaciones bien definidas y significativas. Las operaciones posibles dependen del objeto.

- **Principio de necesidad de saber (*need-to-know*):** en cualquier momento, un proceso debe poder acceder solamente a aquellos objetos que realmente requiere para completar su tarea. Este principio es útil para limitar la cantidad de daño que un proceso en falla o un atacante puede producir al sistema

Comparando *need-to-know* con mínimo privilegio, es fácil notar que *need-to-know* es la política y mínimo privilegio es el mecanismo que logra esa política.

Estructura de dominios

Para facilitar el tipo de esquema anterior, un proceso debe poder operar en un dominio de protección, el cual especifica los recursos que un proceso debe acceder. Cada dominio define un conjunto de objetos y sus tipos de operaciones que pueden ser invocadas en cada objeto. La habilidad de ejecutar una operación en un objeto es un permiso de acceso (*access right*).

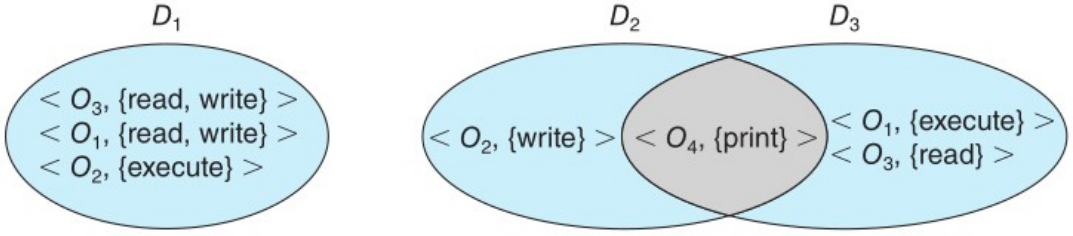


Figure 17.4 System with three protection domains.

Un dominio es una colección de permisos de acceso, cada uno de los cuales es un par ordenado `<object-name, rights-set>`. Por ejemplo, si el dominio *D* tiene el permiso de acceso `<file F, {read, write}>`, entonces un proceso siendo ejecutado en el dominio *D*

puede tanto leer como escribir el archivo *F*, pero no puede realizar ninguna otra operación en el objeto.

Los dominios pueden ser realizador de maneras diferentes

- Cada **usuario** puede ser un dominio. En este caso, el conjunto de objetos que pueden ser accedidos depende de la identidad del usuario. El cambio de dominio ocurre cuando el usuario es cambiado (generalmente cuando un usuario cierra su sesión y otro usuario inicia la suya).
- Cada **proceso** puede ser un dominio. En este caso, el conjunto de objetos que pueden ser accedidos depende de la identidad del proceso. El cambio de domino ocurre cuando un proceso envía un mensaje a otro proceso y espera por la respuesta.
- Cada **procedimiento** puede ser un dominio. En este caso, el conjunto de objetos que puede ser accedido corresponde a las variables locales definidas en el procedimiento. El cambio de dominio ocurre cuando una llamada a un procedimiento es hecha.

Ejemplo de UNIX

En UNIX el usuario root puede ejecutar comandos privilegiados, mientras que otros usuarios no. Restringir ciertas operaciones al usuario root puede perjudicar a otros usuarios en sus operaciones diarias. Por ejemplo, si un usuario quiere cambiar su contraseña inevitablemente requerirá acceso a la base de datos de contraseñas (usualmente `/etc/shadow`), que puede ser accedido solamente por el usuario root.

La solución a este problema es el bit *setuid*. En UNIX, una identificación del *owner* y un bit de dominio, conocidos como ***bit setuid*** son asociados con cada archivo. El bit *setuid* puede estar o no activado. Cuando el bit está activado en un archivo ejecutable (a través de `chmod +s`), quien sea que ejecute el archivo temporalmente asume la identidad del *owner* del archivo. Esto significa que si un usuario logra crear un archivo con el ID de usuario “root” y el bit *setuid* activado, cualquiera que gane permisos de ejecución del archivo se convierte en usuario “root” durante la vida del proceso.

Matriz de acceso

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Figure 17.6 Access matrix of Figure 17.5 with domains as objects.

El modelo general de protección puede ser visto abstractamente como una matriz, llamada matriz de acceso. Las filas de la matriz de acceso representa dominios, y las columnas representan objetos. Cada entrada en la matriz consiste en un conjunto de permisos de acceso. Como cada columna define los objetos explícitamente, podemos omitir el nombre del objeto en el permiso de acceso. La entrada `access(i,j)` define el conjunto de operaciones que el proceso ejecutándose en el dominio D_i puede invocar sobre el objeto O_j.

Las operaciones disponibles para las entradas de la matriz de acceso son:

- Dueño de O_i (*owner*): es el mecanismo que permite agregar nuevos y remover ciertos permisos. Si `access(i,j)` incluye el permiso *owner*, entonces un proceso ejecutándose en el dominio D_i puede agregar y remover cualquier permiso en cualquier entrada en la columna *j*
- Copiar operación desde O_i a O_j (*copy*): la habilidad de copiar un permiso de acceso desde un dominio (o fila) de la matriz de acceso a otra es denotada por un asterisco (*) adjunto al permiso de acceso. El permiso *copy* permite al permiso de acceso ser copiado solamente dentro de la columna (esto es, para el objeto) para el cual el permiso está definido.
- Control (*control*): los permisos *copy* y *owner* permiten a un proceso cambiar las entradas en una columna. Un mecanismo es también necesario para cambiar las entradas en una fila. El permiso *control* es aplicable solamente a objetos de dominio. Si `access(i,j)` incluye el permiso *control*, entonces un proceso ejecutándose en el dominio D_i puede eliminar cualquier permiso de acceso de la fila *j*
- Transferencia (*switch*): cuando cambiamos un proceso de un dominio a otro, ejecutamos la operación *switch* a un objeto (el dominio). Los procesos deben poder transferirse de un dominio a otro. Transferirse del dominio D_i al dominio D_j está permitido solamente si el permiso de acceso `switch ϵ access(i,j)`

Implementación de la matriz de acceso

Tabla global

La implementación más simple de la matriz de acceso es una tabla global consistente de un conjunto de tuplas ordenadas `<dominio, objeto, conjunto de permisos>`.

Esta implementación sufre de varias desventajas

- La tabla usualmente es grande y no puede ser mantenida en memoria principal
- Es difícil tomar ventaja de los agrupamientos especiales de objetos o dominios

Lista de acceso de objetos

Cada columna en la matriz de accesos puede ser implementada como una lista de accesos para cada objeto. La lista resultante para cada objeto consiste en pares ordenados `<dominio, conjunto de permisos>`, que definen todos los dominios con un conjunto no vacío de permisos de accesos para ese objeto.

Lista de capacidad para dominios

En vez de asociar las columnas de la matriz de acceso con los objetos como listas de accesos, podemos asociar cada fila con su dominio. Una lista de capacidades para un dominio es una lista de objetos juntos con las operaciones permitidas en esos objetos. Un objeto es usualmente representado por su nombre físico o dirección, llamado capacidad.

Revocación de derechos de acceso

En un sistema de protección dinámica, a veces es necesario revocar los permisos de accesos a objetos compartidos por diferentes usuarios. Esquemas que implementan la revocación de capacidades incluyen las siguientes:

- **Readquisición:** periódicamente, las capacidades son eliminadas de cada dominio. Si un proceso quiere usar una capacidad, puede encontrar que esa capacidad ha sido eliminada. El proceso puede entonces intentar readquirir la capacidad. Si el acceso ha sido revocado, el proceso no podrá readquirir la capacidad.
- **Punteros hacia atrás:** una lista de punteros es mantenida para cada objeto, apuntando a todas las capacidades asociadas con el objeto. Cuando la revocación es requerida, podemos seguir esos punteros, cambiando las capacidades como sea necesario.
- **Indirección:** las capacidades apuntan indirectamente, no directamente, a los objetos. Cada capacidad apunta a una única entrada en la tabla global, la cual a la vez apunta al objeto. Implementamos revocación buscando en la tabla global para la entrada deseada y eliminándola. Entonces, cuando un acceso es intentado, la capacidad es encontrada apuntando a una entrada ilegal en la tabla.
- **Claves:** una clave es un único patrón de bits que pueden ser asociados con una capacidad. Esta clave es definida cuando la capacidad es creada, y no puede ser modificada ni inspeccionada por el proceso que posee la capacidad. Una clave maestra es asociada con cada objeto, y puede ser definida o reemplazada con la operación `set-key`. Cuando una capacidad es creada, el valor actual de la clave maestra es asociado con la capacidad. Cuando la capacidad es ejercida, su clave es comparada con la clave maestra. Si las claves coinciden, la operación es permitida, de otra manera, se genera una condición de excepción.

  Finalmente, podemos agrupar todas la claves en una tabla global de claves. Una capacidad es válida solamente si su clave coincide con alguna clave en la tabla global. Implementamos revocación removiendo la clave coincidente de la tabla.

Protección basada en lenguajes

Como los sistemas operativos se han vuelto más complejos, las metas de protección se han vuelto mucho más refinadas. Los diseñadores de sistemas de protección se han basado en gran medida en ideas que se originaron en lenguajes de programación y especialmente en conceptos de tipos de datos abstractos y objetos. Los sistemas de protección están ahora preocupados no solamente con la identidad de un recurso al cual es intentado el acceso, sino también la naturaleza funcional del acceso. En los sistemas de protección más nuevos, la preocupación por la función que va a ser invocado se extiende más allá del conjunto de

funciones definidas por el sistema, tales como métodos de acceso a archivos, para incluír funciones que serán definidas por el usuario.

Sandboxing (cajas de arena)

Muchas páginas web contienen pequeños programas llamados *applets*. Cuando una página web que contiene *applets* es descargada, los *applets* son fetcheados y ejecutados.

El sandboxing aisla cada applet en un rango limitado de direcciones virtual aplicadas en tiempo de ejecución. Funciona dividiendo el espacio de direcciones virtuales en regiones de igual tamaño, las cuales son llamadas *sandboxes*. Cada *sandbox* debe tener la propiedad de que todas sus direcciones comparten alguna cadena de los bits de mayor orden.

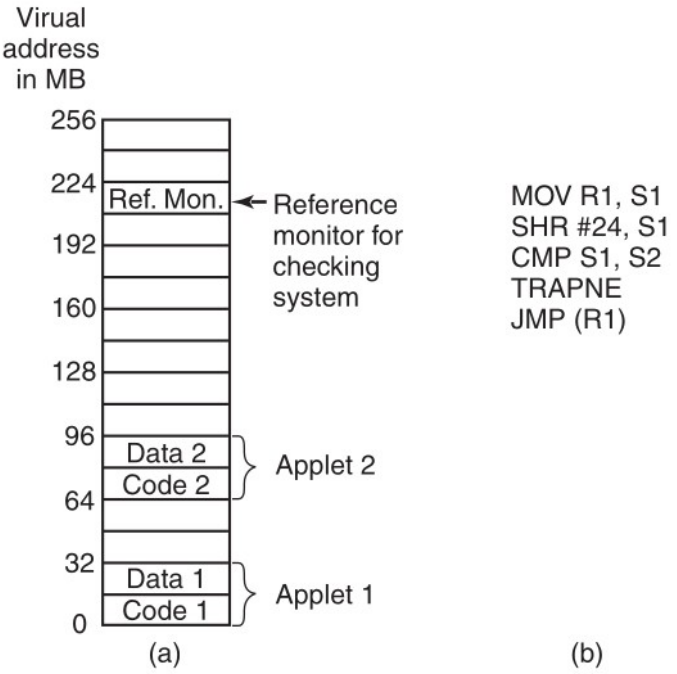


Figure 9-38. (a) Memory divided into 16-MB sandboxes. (b) One way of checking an instruction for validity.

La idea básica detrás de una *sandbox* es garantizar que un *applet* no puede saltar al código exterior a su sandbox de código o referenciar a datos por fuera de su sandbox de datos. La razón para tener dos sandoboxes es prevenir que un applet modifique su propio código durante ejecución para evitar estas restricciones. Previniendo que todas las tiendas (*stores*) entren en el sandbox de código, eliminamos el peligro de código auto modificable. Mientras un applet está confinado de esta manera, no puede dañar el browser u otros applets, plantar viruses en memoria o dañar la memoria de alguna otra manera.

Virtualización

La idea fundamental detrás de una máquina virtual es abstraer el hardware de una computadora (CPU, memoria, discos, interfaces de red, etc) en varios ambientes de ejecución diferentes, creando así la ilusión de que cada ambiente separado corre en su propia computadora privada.

Las implementaciones de máquinas virtuales involucran varios componentes:

- Anfitrión (*host*): sistema de hardware subyacente que corre las máquinas virtuales
- Administrador de máquinas virtuales (VMM, *virtual machine manger*): también conocido como hipervisor (*hypervisor*). Crea y corre las máquinas virtuales proveyendo una interface que es idéntica para el host.
- Invitado (*guest*): proceso provisto con una copia virtual del host. Usualmente, el proceso *guest* es de hecho un sistema operativo.

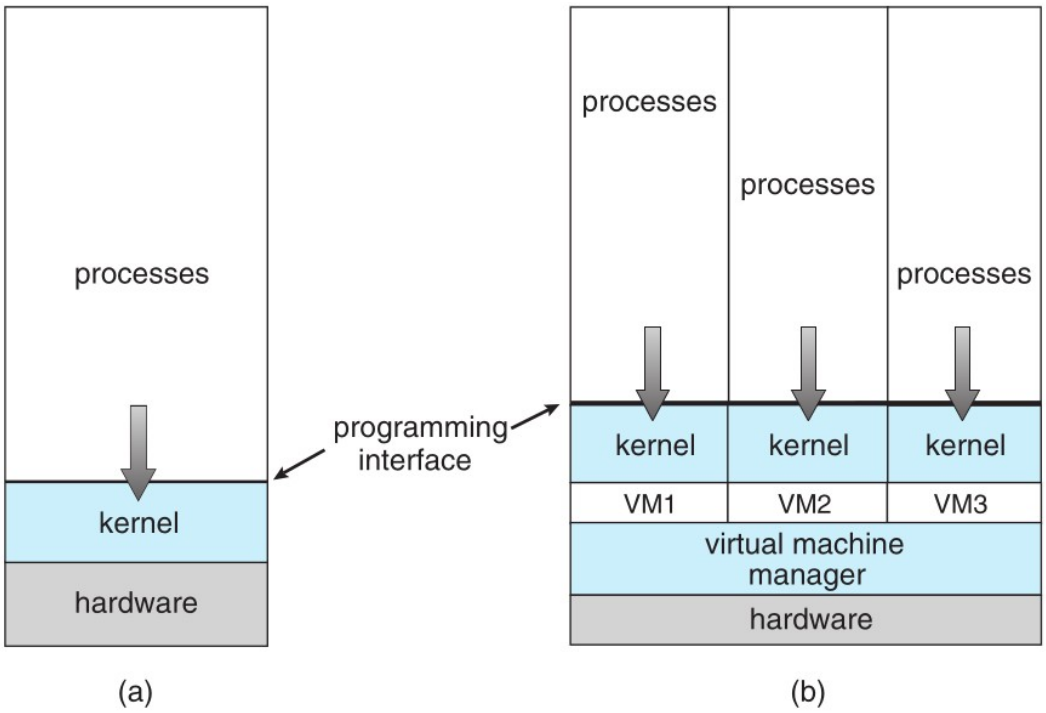


Figure 18.1 System models. (a) Nonvirtual machine. (b) Virtual machine.

Tipos de máquinas virtuales

La implementación de VMMs varían grandemente. Entre las opciones se incluyen:

- **Hypervisors tipo 0:** son soluciones basadas en hardware que proveen soporte para creación y administración de máquinas virtuales por medio de firmware. Estos VMMs son encontrados comúnmente en mainframes y servidores de mediana a larga escala.
- **Hypervisors tipo 1:** software parecido a un sistema operativo (*operating-system-like*) construido para proveer virtualización. Por ejemplo, Vmware ESX, Joyent SmartOS, Citrix XenServer. También se incluyen sistemas operativos que proveen funciones

estándares así como funciones de VMM. Por ejemplo, Microsoft Windows Server con HyperV y Red Hat Linux con KVM

- **Hypervisors tipo 2:** aplicaciones que corren en sistemas operativos estándar, pero proveen características de VMM para sistemas operativos *guest*. Por ejemplo, VMware Workstation, Parallels Desktop y Oracle Virtual Box.
- **Paravirtualización:** una técnica en la cual el sistema operativo *guest* es modificado para trabajar en cooperación con el VMM para optimizar el rendimiento.
- **Virtualización de ambientes de programación:** las VMMs no virtualizan hardware real, sino que crea un sistema virtual optimizado. Por ejemplo, Oracle Java y Microsoft.NET
- **Emuladores:** permiten a aplicaciones escritas para un ambiente de hardware correr en un ambiente de hardware muy diferente, como con un tipo diferente de CPU.
- **Contenedor de aplicaciones:** no es virtualización, pero aún así provee características parecidas a las de virtualización (*virtualization-like*) separando las aplicaciones del sistema operativo. Por ejemplo, Oracle Solaris Zones, BSD Jails e IBM AIX WPARs, Docker, Kubernetes.

Beneficios

Varias ventajas hacen la virtualización atractiva. La mayoría de ellas están fundamentalmente relacionadas con la habilidad de compartir el mismo hardware en varios ambientes de ejecución diferentes de manera concurrente.

- **Protección:** el sistema *host* está protegido de las máquinas virtuales, así como las máquinas virtuales están protegidas unas de otras.
- **Suspender** o freezar una máquina virtual corriendo
- **Snapshots**, los VMMs permiten copias y *snapshots* (instantáneas) para ser hechas del *guest*. La copia pueden ser usadas para crear nuevas VM o mover una VM de una máquina a otra con su estado actual intacto. Las *snapshots* registra un punto en el tiempo, y el *gues* puede resetear a ese punto si es necesario.
- **Resume**, el *guest* puede reanudar (*resume*) donde está, así como en su máquina original
- **Clone**, se pueden crear clones

Implementación

La habilidad de virtualizar depende de las características provistas por la CPU. VMMs usan varias técnicas para implementar virtualización, incluyendo *trap-and-emulate* y traducción binaria.

La VCPU (virtual CPU) no ejecuta código, sino que representa el estado de la CPU como la máquina *guest* cree que está. Para cada *guest*, el VMMs mantiene una VCPU reprsentando el estado actual de la CPU del *guest*. Cuando el *guest* es cambiado de contexto hacia el CPU por la VMM, la información de la VCPU es usada para cargar el contexto correcto.

Trap-and-Emulate

En un sistema típico modo dual, la máquina virtual *guest* puede ejecutarse solamente en modo usuario (excepto que hardware extra de soporte sea provisto). El kernel corre en modo kernel y no es seguro permitir que código a nivel de usuario sea corrido en modo kernel. Así como la máquina física tiene dos modos, así también la máquina virtual (modo usuario virtual y modo kernel virtual), los cuales corren en modo usuario físico.

Cuando el kernel en el *guest* intenta ejecutar una instrucción privilegiada, se produce un error (porque el sistema está en modo usuario) y causa una *trap* al VMM en la máquina real. El VMM gana el control y ejecuta (o emula) la acción que fue intentada por el kernel *guest* en lugar del *guest*. Entonces le devuleve el control a la máquina virtual. Esto es llamado el método *trap-and-emulate*.

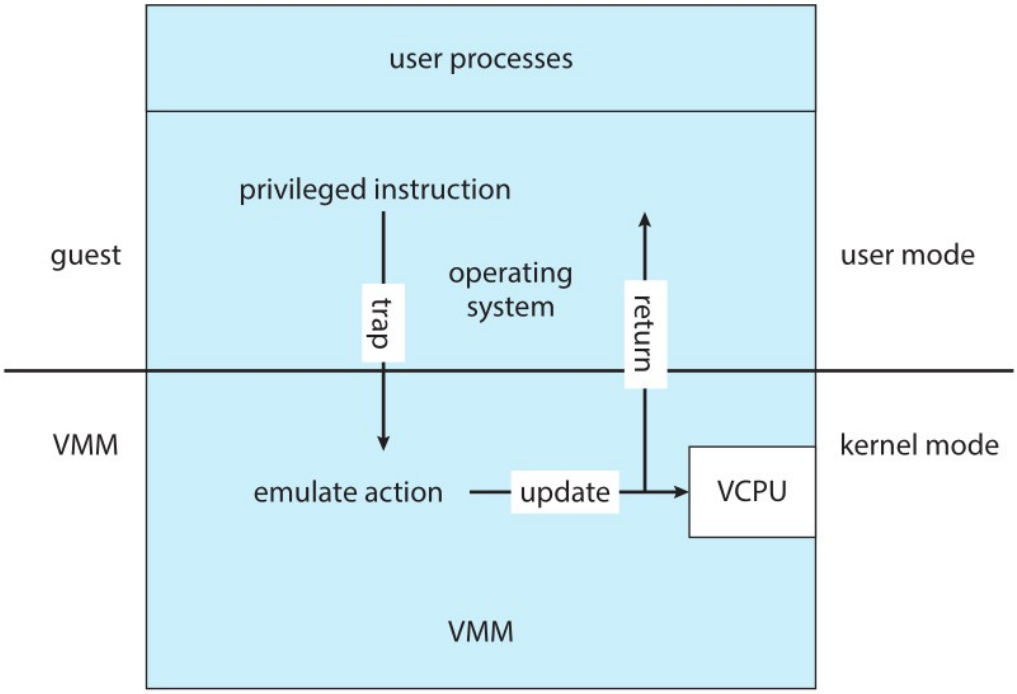


Figure 18.2 Trap-and-emulate virtualization implementation.

Con las instrucciones privilegiadas, el tiempo se vuelve un problema. Todas las instrucciones no privilegiadas corren nativamente en el hardware, proveyendo el mismo rendimiento para los *guest* que para las aplicaciones nativas. Las instrucciones privilegiadas crean un *overhead* (gasto) extra causando que el *guest* corra más lento que lo que lo haría nativamente.

Traducción binaria

Usar el método *trap-and-emulate* para implementar virtualización en arquitecturas x86 es imposible debido a las instrucciones especiales. Debido a esto, se desarrolló la técnica de traducción binaria.

Los pasos básicos son los siguientes:

1. Si la VCPU *guest* está en modo usuario, el *guest* puede correr sus instrucciones nativamente en el CPU físico

2. Si el VCPU *guest* está en modo kernel, entonces el *guest* cree que está corriendo en modo kernel. El VMM examina cada instrucción que el *guest* ejecuta en el modo kernel virtual leyendo las siguientes instrucciones que el *guest* va a ejecutar, basado en el *program counter* del *guest*. Las instrucciones que no son instrucciones especiales son corridas nativamente. Las instrucciones especiales son traducidas en un nuevo conjunto de instrucciones que realizar una tarea equivalente.

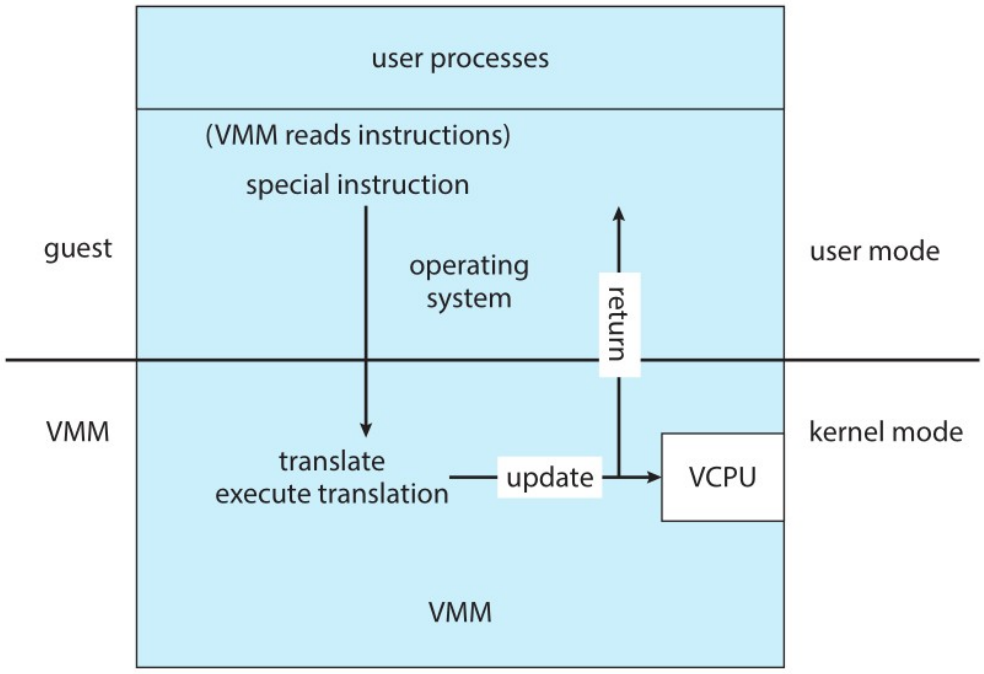


Figure 18.3 Binary translation virtualization implementation.

La traducción binaria es implementada traduciendo código en el VMM. El código lee instrucciones binarias nativas dinámicamente del *guest*, en demanda, y genera código binario nativo que se ejecuta en lugar del código original.

Tipos de VMs y sus implementaciones

Hypervisor tipo 0

El VMM en sí mismo es codificado en el firmware y es cargado en tiempo de booteo. Carga sucesivamente las imágenes *guest* para correr en cada partición. Cada *guest* cree que tiene hardware dedicado porque lo tiene, simplificando muchos detalles de implementación.

El hypervisor administra el acceso compartido o garantiza a todos los dispositivos un control de partición. En el control de partición, un sistema operativo *guest* provee servicios por medio de daemons a otros *guests*, y el hypervisor rutea las solicitudes I/O apropiadamente.

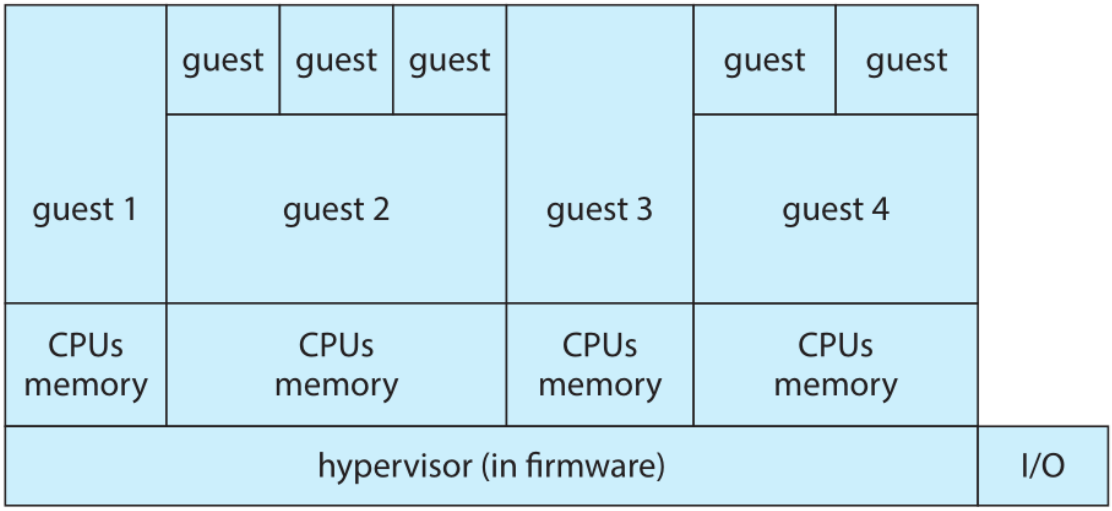


Figure 18.5 Type 0 hypervisor.

Hypervisor tipo 1

Los *hypervisors* de tipo 1 son comunmente encontrados en los data centers de las empresas. Son sistemas operativos de propósito especial que corren en el hardware, pero en lugar de proveer system calls y otras interfaces para correr programas, crean, corren y administran sistemas operativos *guest*.

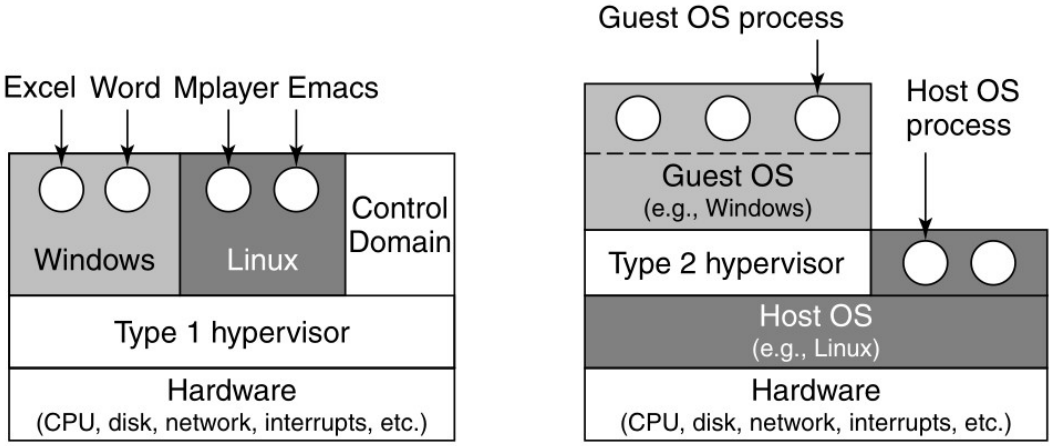


Figure 7-1. Location of type 1 and type 2 hypervisors.

Los *hypervisors* tipo 1 corren en modo kernel, tomando ventaja de la protección de hardware. Donde el CPU del *host* lo permita, usan múltiples modos para dar al sistema operativo *guest* su propio control y rendimiento mejorado. Porque son sistemas operativos, también deben proveer planificación de CPU, manejo de memoria, administración de I/O, protección y aún seguridad.

Hypervisor tipo 2

Este tipo de VMM es simplemente otro proceso siendo ejecutado y administrado por el *host*, y aún el *host* no sabe que la virtualización está sucediendo en el VMM.

Los *hypervisors* tipo 2 tienen límites no asociados con los otros tipos. Por ejemplo, su un usuario necesita privilegios administrativos para acceder a algunas de las características de asistencia de hardware de los CPU modernos. Si el VMM está siendo corriedo por un

usuario estándar sin privilegios adicionales, el VMM no puede tomar ventaja de esas características. Debido a esta limitación, así como el *overhead* (gasto) adicional de correr un sistema operativo de propósito general así como los sistemas operativos *guest*, los *hypervisors* tipo 2 tienden a tener un rendimiento peor que los tipo 0 y 1.

Las limitaciones de los *hypervisors* tipo 2 también proveen algunos beneficios. Corren en una variedad de sistemas operativos de propósito general y ejecutarlos no requiere cambios en el sistema operativo *host*.

Paravirtualización

La paravirtualización presenta el *guest* con un sistema que es similar pero no idéntico al sistema preferido del *guest*. El *guest* debe ser modificado para correr en el hardware virtual paravirtualizado. La ganancia de este trabajo extra es mayor eficiencia en el uso de recursos y una capa de virtualización más pequeña.

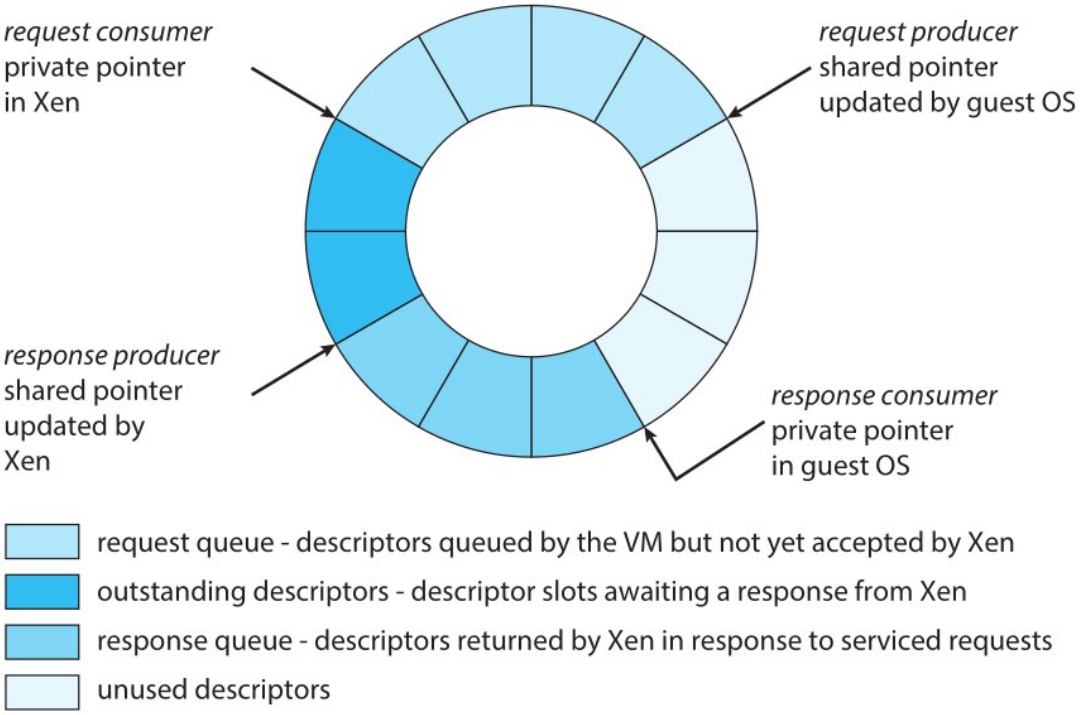


Figure 18.6 Xen I/O via shared circular buffer.[1]