Go up to Top
Go forward to Procesos

Introducción

¿Qué es un sistema operativo?

A pesar de que todos nosotros usamos sistemas operativos casi a diario, es difícil definir qué es un sistema operativo. En parte, esto se debe a que los sistemas operativos realizan dos funciones diferentes.

- Proveer una **máquina virtual**, es decir, un ambiente en el cual el usuario pueda ejecutar programas de manera **conveniente**, protegiéndolo de los detalles y complejidades del hardware.
- Administrar **eficientemente** los recursos del computador.

El sistema operativo como máquina virtual

Un computador se compone de uno o más procesadores o CPUs, memoria principal o RAM, memoria secundaria (discos), tarjetas de expansión (tarjetas de red, modems y otros), monitor, teclado, mouse y otros dispositivos. O sea, es un sistema complejo. Escribir programas que hagan uso correcto de todas estas componentes no es una tarea trivial. Peor aún si hablamos de uso óptimo. Si cada programador tuviera que preocuparse de, por ejemplo, como funciona el disco duro del computador, teniendo además siempre presentes todas las posibles cosas que podrían fallar, entonces a la fecha se habría escrito una cantidad bastante reducida de programas.

Es mucho más fácil decir `escriba "Chao" al final del archivo "datos"', que

1. Poner en determinados registros del controlador de disco la dirección que se quiere escribir, el número de bytes que se desea escribir, la posición de memoria donde está la información a escribir, el *sentido* de la operación (lectura o escritura), amén de otros parámetros;
2. Decir al controlador que efectué la operación.
3. Esperar. Decidir qué hacer si el controlador se demora más de lo esperado (¿cuánto es "lo esperado"?).
4. Interpretar el resultado de la operación (una serie de bits).
5. Reintentar si algo anduvo mal.
6. etc.

Además, habría que reescribir el programa si se instala un disco diferente o se desea ejecutar el programa en otra máquina.

Hace muchos años que quedó claro que era necesario encontrar algún medio para aislar a los programadores de las complejidades del hardware. Esa es precisamente una de las tareas del sistema operativo, que puede verse como una capa de software que maneja todas las partes del sistema, y hace de *intermediario* entre el hardware y los programas del usuario. El sistema operativo presenta, de esta manera, una interfaz o **máquina virtual** que es más fácil de entender y de programar que la máquina "pura". Además, para una misma familia de máquinas, aunque tengan componentes diferentes (por ejemplo, monitores de distinta resolución o discos duros de diversos fabricantes), la máquina virtual

puede ser idéntica: el programador ve exactamente la misma interfaz.

El sistema operativo como administrador de recursos

La otra tarea de un sistema operativo consiste en administrar los recursos de un computador cuando hay dos o más programas que ejecutan simultáneamente y requieren usar el mismo recurso (como tiempo de CPU, memoria o impresora).

Además, en un sistema multiusuario, suele ser necesario o conveniente compartir, además de dispositivos físicos, información. Al mismo tiempo, debe tenerse en cuenta consideraciones de seguridad: por ejemplo, la información confidencial sólo debe ser accesada por usuarios autorizados, un usuario cualquiera no debiera ser capaz de sobreescribir áreas críticas del sistema, etc. (En este caso, un usuario puede ser una persona, un programa, u otro computador). En resumen, el sistema operativo debe llevar la cuenta acerca de quién está usando qué recursos; otorgar recursos a quienes los solicitan (siempre que el solicitante tenga derechos adecuados sobre el recurso); y arbitrar en caso de solicitudes conflictivas.

Evolución de los sistemas operativos

Los sistemas operativos y la arquitectura de computadores han evolucionado de manera interrelacionada: para facilitar el uso de los computadores, se desarrollaron los sistemas operativos. Al construir y usar los sistemas operativos, se hace obvio que ciertos cambios en la arquitectura los simplificaría. Por eso, es conveniente echar una mirada a la historia.

La primera generación (1945-1955): tubos de vacío

Lo cierto es que el primer computador digital fue diseñado por el matemático inglés Charles Babbage hace cerca de siglo y medio. Era un computador totalmente mecánico, que Babbage nunca pudo construir, principalmente debido a que la tecnología de la época no era capaz de producir las piezas con la precisión requerida.

Después de eso, poco se hizo hasta la segunda guerra: alrededor de 1940 se construyeron las primeras máquinas calculadoras usando tubos de vacío. Estas máquinas de varias toneladas de peso eran diseñadas, construidas, programadas y operadas por el mismo grupo de personas. No había ni lenguajes de programación, ni compiladores; mucho menos sistema operativo. Cada programa se escribía en lenguaje de máquina, usando tableros con enchufes e interruptores y tenía que manejar todo el sistema (lo que era factible gracias a que el programador era el mismo que diseñó y construyó la máquina). Con el tiempo se introdujeron las tarjetas perforadas, para reemplazar al tablero, pero el sistema era esencialmente el mismo.

La segunda generación (1955-1965): transistores y procesamiento por lotes

La introducción de los transistores permitió aumentar sustancialmente la confiabilidad de los computadores, lo que a su vez hizo factible construir máquinas comerciales. Por primera vez hubo una separación entre diseñadores, constructores, y programadores.

La aparición de los primeros compiladores (de FORTRAN) facilitó la programación, a costa de hacer mucho más compleja la operación de los computadores. Por ejemplo, para probar un programa en FORTRAN recién escrito, el programador debía esperar su turno, y:

1. Cargar compilador de FORTRAN, típicamente desde una cinta magnética.
2. Poner el alto de tarjetas perforadas correspondientes al programa FORTRAN y correr el compilador.
3. El compilador genera código en assembler, así que hay que cargar ahora el ensamblador para traducirlo a lenguaje de máquina. Este paso requiere poner otra cinta con el ensamblador. Ejecutar el ensamblador, para generar el programa ejecutable.
4. Ejecutar el programa.

Si hay errores en cualquiera de estos pasos, el programador debía corregir el programa y comenzar todo de nuevo. Obviamente, mientras el programdor ponía cintas y tarjetas, y mientras se devanaba los sesos para descubrir por qué el programa no funciona, la CPU de millones de dólares de costo se mantenía completamente ociosa. Más que rápido, se idearon mecanismos para mantener a la CPU siempre ocupada. El primero fue separar el rol de programador del rol de operador. Un operador profesional demoraba menos en montar y desmontar cintas, y podía acumular lotes de trabajos con requerimientos similares: por ejemplo, si se acumula la compilación de varios programas FORTRAN, entonces el compilador de FORTRAN se carga una sola vez para todos los trabajos.

Aún así, en la transición de un trabajo a otro la CPU se mantenía desocupada. Aparecieron entonces los **monitores residentes**, que fueron los precursores de los sistemas operativos. Un monitor residente es un pequeño programa que está siempre en la memoria del computador, desde que éste se enciende. Cuando un programa termina, se devuelve el control al monitor residente, quien inmediatamente selecciona otro programa para ejecutar. Ahora los programadores, en vez de decirle al operador qué programa cargar, debían informárselo al monitor (mediante tarjetas de control especiales):

```
$JOB
$FTN
programa FORTRAN
$LOAD
$RUN
datos para el programa
$END
```

Esto se conoce como procesamiento por lotes: el programador deja su alto de tarjetas, y después vuelve a retirar la salida que se emite por la impresora (y que puede ser nada más que la notificación de que el programa tenía un error de sintaxis).

La tercera generación (1965-1980): circuitos integrados y multiprogramación

El procesamiento por lotes evita que la CPU tenga que esperar tareas ejecutadas por lentos seres humanos. Pero ahora el cuello de botella se trasladó a los dispositivos mecánicos (impresoras, lectoras de tarjetas y de cinta), intrínsecamente más lentos que que las CPUs electrónicas. Para resolver esto, aparece, dentro de la tercera generación de computadores, la multiprogramación: varios trabajos se mantienen permanentemente en memoria; cuando uno de ellos tiene que esperar que una operación (como grabar un registro en cinta) se complete, la CPU continúa con la ejecución de otro trabajo. Si se mantiene un número suficiente de trabajos en la memoria, entonces la CPU puede estar siempre ocupada.

Pero el sistema sigue siendo esencialmente un sistema de procesamiento por lotes; los programadores no interactúan *en línea* con el computador, los tiempos de respuesta desde que se deja un trabajo para ejecución hasta conocer el resultado son demasiado grandes. (¡En cabio, los computadores de primera generación eran interactivos!) De ahí nace el concepto de **tiempo compartido** que es una variante de la

multiprogramación en la cual una CPU atiende simultáneamente los requerimientos de varios usuarios conectados en línea a través de terminales. Ya que los usuarios humanos demoran bastante entre la emisión de un comando y otro, una sola CPU es capaz de atender, literalemente, a cientos de ellos simultáneamente (bueno, en realidad, uno después de otro, pero los usuarios tienen la ilusión de la simultaneidad). Por otra parte, cuando no hay ningún comando que ejecutar proveniente de un usuario interactivo, la CPU puede cambiar a algún trabajo por lote.

La cuarta generación (1980-): computadores personales

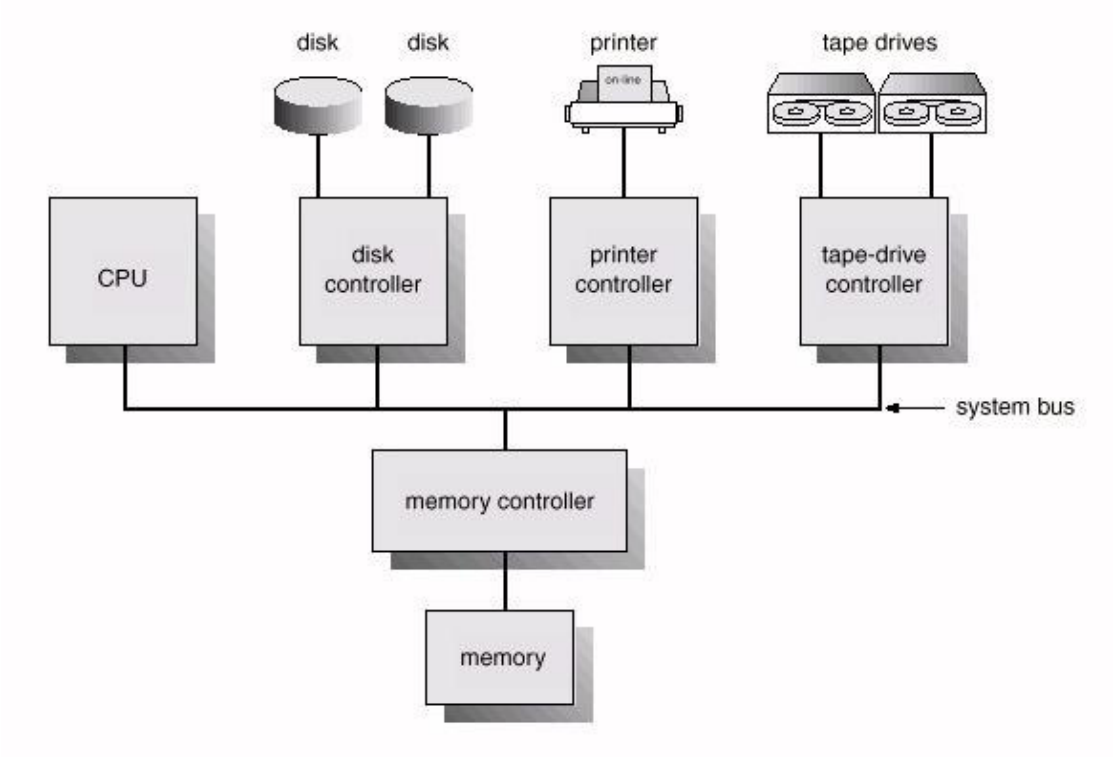
Con el advenimiento de la integración a gran escala, que permitió concentrar en un solo chip miles, y luego millones de transistores, nació la era de la computación personal. Los conceptos utilizados para fabricar los sistemas operativos de la tercera generación de computadores siguen siendo, en general, adecuados para la cuarta generación. Algunas diferencias destacables:

- Dado los decrecientes costos de las CPUs, ya no es nada de grave que un procesador esté desocupado.
- La creciente capacidad de las CPUs permite el desarrollo de interfaces gráficas; buena parte del código de los sistemas operativos de hoy es para manejar la interfaz con el usuario.
- Los sistemas paralelos (un computador con varias CPUs), requieren sistemas operativos capaces de asignar trabajos a los distintos procesadores.
- Las redes de computadores y sistemas distribuidos abren nuevas posibilidades e imponen nuevas obligaciones a los sistema operativo.

Apoyo del hardware

Interrupciones: la base de los sistemas operativos modernos

Un computador moderno se compone de una CPU (a veces más de una) y una serie de controladores de dispositivos, conectados a través de un bus común a la memoria. Cada controlador está a cargo de un tipo específico de dispositivo.



Cuando se enciende el computador, se ejecuta automáticamente un pequeño programa, cuya única tarea es cargar el sistema operativo en la memoria, y entregarle el control (ejecutarlo). El sistema operativo hace las inicializaciones del caso, y luego simplemente espera a que algún evento ocurra. La ocurrencia de un evento es, por lo general, señalizada por una interrupción de software o hardware. Los controladores de dispositivo pueden gatillar una interrupción en cualquier momento, enviando una señal a la CPU a través del bus del sistema. Las interrupciones de software son gatilladas por procesos, por la vía de ejecutar una instrucción especial que se conoce como **llamada al sistema**.

Ejemplos de eventos que gatillan interrupciones: terminación de una operación de I/O, llamadas al sistema, división por cero, alarmas del reloj. Para cada tipo de interrupción, debe existir una rutina que la atienda: el servidor de la interrupción.

Cuando la CPU es interrumpida:

1. La CPU deja de hacer lo que estaba haciendo, que puede haber sido nada, (si simplemente estaba esperando), o la ejecución de un proceso.
2. Determina el tipo de interrupción.
3. Ejecuta el servidor de interrupción correspondiente.
4. Continúa lo que fue interrupido.

Más en detalle, una forma de hacer todo esto es la siguiente: Dado que existe una cantidad limitada (y reducida) de tipos de interrupciones posibles, se usa una tabla con las direcciones de los servidores de cada interrupción, típicamente en las primerísimas posiciones de la memoria. Esta tabla se conoce como **vector de interrupciones**. Así por ejemplo, si las direcciones ocupan 4 bytes, la dirección del servidor de la interrupción *i* se guarda en los 4 bytes a partir de la posición *4i* de la memoria.

Cuando ocurre una interrupción:

1a.
Se deshabilitan las interrupciones para que la CPU no sea interrumpida mientras atiende una interrupción (situación que podría generar caos).

1b.
La CPU debe memorizar lo que estaba haciendo para continuarlo después: se guarda el PC (y posiblemente otros registros) en el stack del sistema.

2.
Se determina la dirección del servidor de la interrupción, según el vector de interrupciones.

3.
Se transfiere el control a esa dirección.

4.
La rutina que atiende la interrupción debe finalizar con una instrucción IRET, que restablece el estado que la CPU tenía cuando fue interrumpida, usando los datos almacenados en el stack.

Esquemas más sofisticados *enmascaran* (deshabilitan selectivamente) las interrupciones en vez de deshabilitarlas totalmente (punto 1b), de manera que una interrupción de alta prioridad **SI** pueda interrumpir a la CPU cuando está atendiendo una de menor prioridad.

Todos los sistemas operativos modernos se basan en interrupciones (*interrupt driven*). Si no hay procesos que ejecutar, ni dispositivos ni usuarios que atender, el sistema operativo no hace nada: se "sienta" a esperar que algo pase. Cuando algo pasa, se señaliza una interrupción, y el sistema operativo entra en actividad (o sea, los servidores de interrupción son parte del sistema operativo).

Ejemplo: multiprogramación (timesharing)

1. El sistema operativo inicializa el *timer* o reloj en una tajada de tiempo, y lo echa a andar.
2. El sistema operativo entrega el control a un proceso.
3. El proceso ejecuta.
4. Concluido el tiempo prefijado, el timer provoca una interrupción.
5. El manejador de interrupciones del timer (que es parte del sistema operativo), guarda la información del proceso interrumpido necesaria para poder reanudarlo después.
6. Se repite el ciclo, escogiendo ahora otro proceso.

Otro ejemplo: manejo de dispositivos de I/O

Para comenzar una operación de I/O, el sistema operativo escribe los registros del caso en el controlador del dispositivo. A su vez, el controlador determina qué hacer mirando el contenido de esos registros. Por ejemplo, si se trata de una solicitud de lectura, inicia la transferencia de datos desde el dispositivo hacia la memoria. Cuando finaliza la transferencia, el controlador le avisa a la CPU a través de una interrupción. Esto permite que la CPU, en vez de esperar que la transferencia termine (lo que sería I/O sincrónico), en el intertanto puede realizar otra tarea (I/O asincrónico). Una secuencia típica:

1. El proceso que esá ejecutando en la CPU solicita una operación de I/O, mediante una llamada al sistema.
2. El sistema operativo suspende el proceso, poniéndolo en la cola de procesos suspendidos, y comienza la operación de I/O.
3. El sistema operativo escoge otro proceso para que siga ejecutando en la CPU, mientras la operación de I/O se completa.
4. Cuando la operación de I/O se completa, el control vuelve al sistema operativo gracias a que el controlador del dispositivo provocó una interrupción.

5. El sistema operativo, determina, según la interrupción y sus tablas internas, que el proceso que había sido suspendido ahora puede reanudarse, así que lo pone en la cola de procesos listos para ejecutar.
6. El sistema operativo reanuda ese, o tal vez otro proceso de la cola de procesos listos.

Protección

En los primeros computadores el operador tenía el control completo del sistema. Con el tiempo, se fueron desarrollando los sistemas operativos, y parte del control se traspasó a ellos. Además, para mejorar la utilización del sistema, se comenzó primero a manejar varios programas en memoria en forma simultánea, y luego a atender simultáneamente a varios usuarios. Pero el remedio trajo su propia enfermedad, ya que un programa podía por error o por mala intención de su creador, modificar datos de otros programas, o peor aún, modificar el propio sistema operativo, lo que podía botar todo el sistema. En general, un programa puede alterar el normal funcionamiento del sistema de las siguientes formas:

- Ejecutando una instrucción de I/O ilegal (por ejemplo, que borre todos los archivos del disco).
- Sobreescribiendo áreas de la memoria que pertenecen al sistema operativo.
- No devolviendo el control de la CPU al sistema operativo.

Para evitar esto, es indispensable el apoyo del hardware, a través de los siguientes mecanismos.

Operación dual

Para asegurar una operación adecuada, se debe proteger al sistema operativo y todos los programas del malfuncionamiento de cualquier otro programa. Para eso, el hardware debe proveer al menos dos modos de operación.

- Modo usuario.
- Modo sistema (o privilegiado, protegido o supervisor).

El bit de modo indica el modo de operación actual. Cuando se enciende el computador y se carga el sistema operativo, se comienza en modo sistema. El sistema operativo siempre cambia a modo usuario antes de pasar el control a un proceso de usuario. Cuando ocurre una interrupción, el hardware siempre cambia a modo sistema. De esta menera, el sistema operativo siempre ejecuta en modo sistema. ¿Cual es la gracia? Que hay ciertas operaciones críticas que sólo pueden ser ejecutadas en modo sistema.

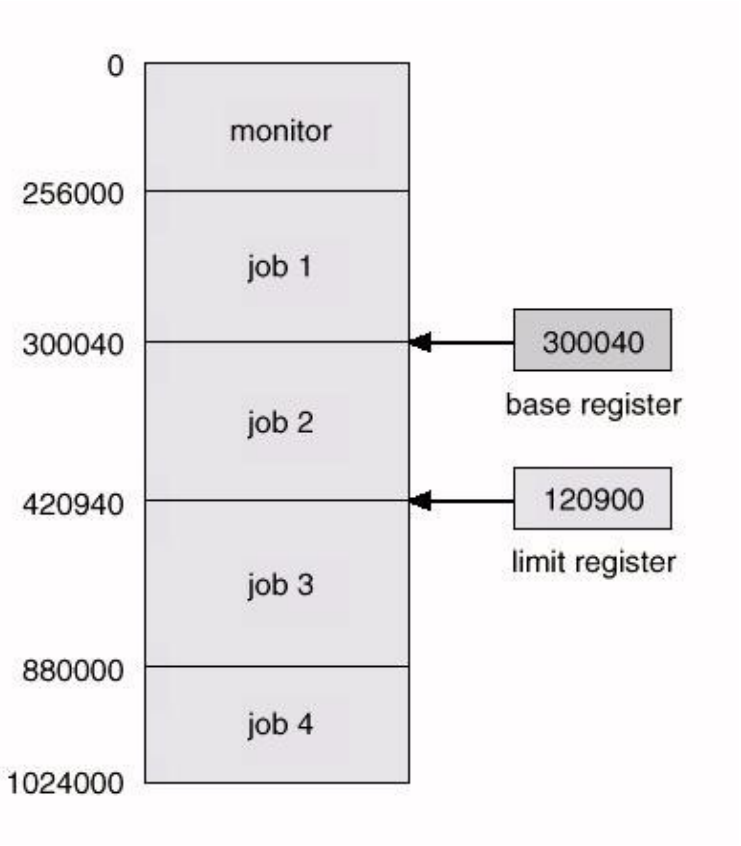
Protección de I/O

Para prevenir que un usuario ejecute instrucciones de I/O que puedan provocar daño, la solución es simple: las instrucciones de I/O sólo pueden ejecutarse en modo sistema. Así los usuarios no pueden ejecutar I/O directamente, sino que deben hacerlo a través del sistema, quien puede filtrar lo que sea del caso.

Para que este mecanismo de protección sea completo, hay que asegurarse que los programas de usuario no puedan obtener acceso a la CPU en modo sistema. Por ejemplo, un programa de usuario podría poner una dirección que apunte a una rutina propia en el vector de interrupciones. Así, cuando se produzca la interrupción, el hardware cambiaría a modo sistema, y pasaría el control a la rutina del usuario. O, también, el programa de usuario podría reescribir el servidor de la interrupción. Se requiere entonces...

Protección de memoria

No sólo hay que proteger el vector de interrupciones y las rutinas servidoras, sino que, en general, queremos proteger las áreas de memoria utilizadas por el sistema operativo y por otros programas. Esto lo vamos a ver con más detalle más adelante, pero una forma es usando dos registros: **base** y **límite**, que establecen el rango de direcciones de memoria que el programa puede accesar legalmente.



Para que esto funcione, hay que comparar cada dirección accesada por un programa en modo usuario con estos dos registros. Si la dirección está fuera del rango, se genera una interrupción que es atendida por el sistema operativo, quien aborta el programa (usualmente con un lacónico mensaje "segmentation fault"). Esta comparación puede parecer cara, pero teniendo presente que se hace en hardware, no lo es tanto. Obviamente, el programa usuario no debe poder modificar estos registros: las instrucciones que los modifican son instrucciones protegidas.

Protección de CPU

La único que falta, es asegurarse que el sistema operativo mantenga el control del buque, o sea, hay que prevenir, por ejemplo, que un programa entre en un ciclo infinito y nunca devuelva el control al sistema operativo. Esto se logra con un timer, tal como vimos en el ejemplo de multiprogramación.

Instrucciones privilegiadas

- Instrucción para cambiar a modo protegido
- Instrucciones para modificar registros de administración de memoria (por ej., base y limite)
- Instrucciones para hacer I/O
- Instrucción para manejar el timer

- Instrucción para deshabilitar interrupciones

Entonces, ¿cómo hace I/O un programa de usuario, si sólo el sistema operativo puede hacerlo? El programa le pide al sistema operativo que lo haga por él. Tal solicitud se conoce como **llamada al sistema**, que en la mayoría de los sistemas operativos se hace por medio de una interrupción de software o *trap* a una ubicación específica del vector de interrupciones. Ejemplo hipotético, (parecido a MSDOS), para borrar un archivo:

1. Poner en registro A el código de la operación: 41h = borrar archivo
2. Poner puntero al nombre del archivo en registro B
3. INT 21h (generar interrupción de software)

Como la llamada es a través de una interrupción, se hace lo de siempre: cambiar a modo protegido, y pasar control a la dirección en la posición 4*21h del vector de interrupciones, o sea, a la rutina que maneja las llamadas al sistema (o tal vez, la rutina que maneja las llamadas que tienen que ver con archivos; puede que otras clases de llamadas sean atendidas por medio de otras interrupciones). Dicha rutina examina el registro A para determinar qué operación debe ejecutar, y después determina el nombre del archivo a través del registro B. En seguida, decide si la operación es válida: puede que el archivo no exista o pertenezca a otro usuario; según eso, borra el archivo o "patalea" de alguna manera.

Cuando uno hace esto en un lenguaje de alto nivel, como C, simplemente escribe:

```
char* Nombre = "Datos";
remove(Nombre);
```

El compilador de C es quien se encarga de traducir esto a una llamada al sistema, ocultando el detalle de la interfaz con el sistema operativo.

En resumen: los sistemas operativos modernos se apoyan ampliamente en el hardware. Las necesidades recién vistas (interrupciones, instrucciones privilegiadas, protección de memoria) fueron impuestas a los diseñadores de hardware por los requerimientos de multiprogramación de los sistemas operativos. Por eso dijimos que la historia de los sistemas operativos está interrelacionada con la historia de la arquitectura de computadores.

IIC2332: Sistemas Operativos

Ultima modificación: July 01, 1998, por Juan E. Navarro (jnavarro@ing.puc.cl)

Go backward to Introducción
Go up to Top
Go forward to Planificación (*scheduling*) de procesos

Procesos

Hasta ahora hemos hablado de programas y procesos un poco vagamente, y como si fueran sinónimos, pero son conceptos distintos. Un proceso es un programa en ejecución, incluyendo el valor actual del *program counter* (PC), registros y variables. Un programa es pasivo (es sólo código o texto) y un proceso es activo y dinámico (varía en el tiempo).

Analogía: preparar una receta de una torta. El programa es la receta, el proceso es la actividad que consiste en leer la receta, mezclar los ingredientes y hornear la torta.

Varios procesos pueden estar ejecutando el mismo programa, por ejemplo, si dos o más usuarios están usando simultáneamente el mismo editor editor de texto. El programa es el mismo, pero cada usuario tiene un proceso distinto (y con distintos datos).

Conceptualmente cada proceso tiene su propia CPU virtual. En la práctica, hay una sola CPU real, que cambia periódicamente la ejecución de un proceso a otro, pero para entender el sistema es más fácil modelarlo como una colección de procesos secuenciales que ejecutan concurrentemente (*pseudoparalelismo*).

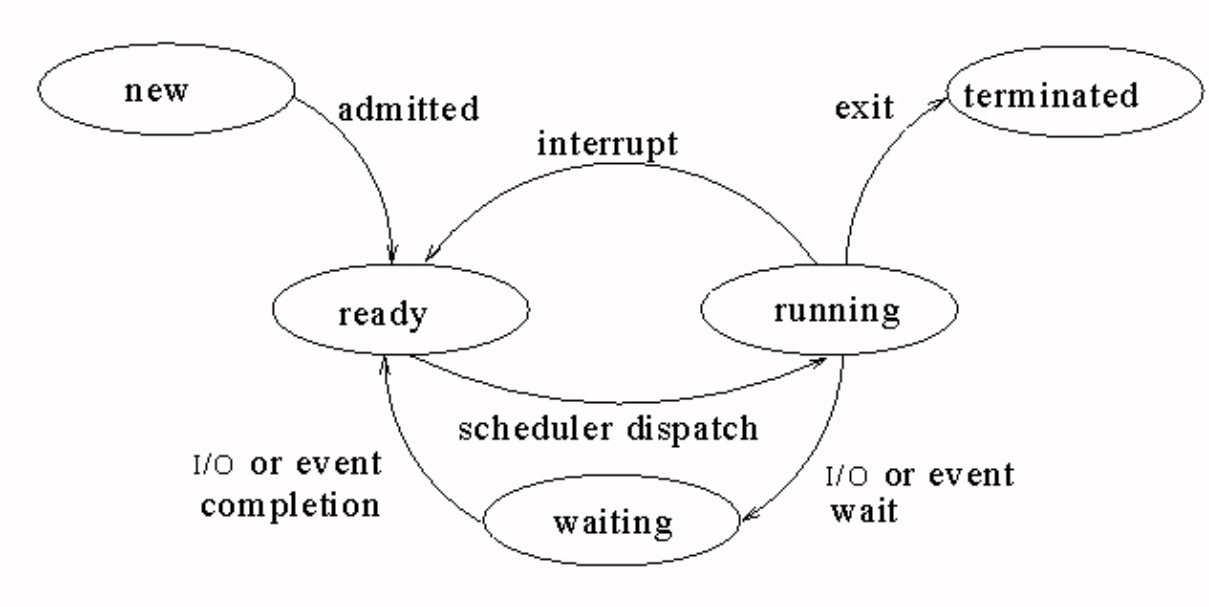
Estado de los procesos

Para efectos del sistema operativo, cada proceso puede estar en uno de los siguientes estados:

Ejecutando.
: El proceso está siendo ejecutado en la CPU. Por lo tanto a lo más un proceso puede estar en este estado en un computador uniprocesador.

Listo.
: El proceso está en condiciones de ejecutarse, pero debe esperar su turno de CPU.

Bloqueado.
: El proceso no está en condiciones de ejecutarse. Está esperando que algún evento ocurra, como la finalización de una operación de I/O. También se dice que está suspendido o en espera.



Implementación de procesos

El sistema operativo mantiene para cada proceso un **bloque de control** o *process control block* (PCB), donde se guarda para cada proceso la información necesaria para reanudarlo si es suspendido, y otros datos.

- Estado (ejecutando, listo, bloqueado)
- Program counter
- Registros de CPU
- Información para planificación (p.ej., prioridad)
- Información para administración de memoria (p.ej., registros base y límite)
- Información de I/O: dispositivos y recursos asignados al proceso, archivos abiertos, etc.
- Estadísticas y otros: tiempo real y tiempo de CPU usado, identificador del proceso, identificador del dueño, etc.

El sistema mantiene una cola con los procesos que están en estado LISTO. Los procesos suspendidos se podrían poner todos en otra cola, pero suele ser más eficiente manejar colas distintas según cuál sea la condición por la cual están bloqueados. Así, se maneja una cola de procesos para cada dispositivo.

Cambios de contexto

Cuando el sistema operativo entrega la CPU a un nuevo proceso, debe guardar el estado del proceso que estaba ejecutando, y cargar el estado del nuevo proceso. El estado de un proceso comprende el PC, y los registros de la CPU. Además, si se usan las técnicas de administración de memoria que veremos más adelante, hay más información involucrada. Este cambio, que demora de unos pocos a mil microsegundos, dependiendo del procesador, es puro *overhead* o sobrecosto, puesto que entretanto la CPU no hace trabajo útil (ningún proceso avanza). Considerando que la CPU hace varios cambios de contexto en un segundo, su costo es relativamente alto.

Algunos procesadores tienen instrucciones especiales para guardar todos los registros de una vez. Otros

tienen varios conjuntos de registros, de manera que un cambio de contexto se hace simplemente cambiando el puntero al conjunto actual de registros. El problema es que si hay más procesos que conjuntos de registros, igual hay que apoyarse en la memoria. Como sea, los cambios de contexto involucran un costo importante, que hay que tener en cuenta.

Creación de procesos

Un proceso `padre' puede crear nuevos procesos `hijos' mediante llamadas al sistema. A su vez, estos hijos también pueden crear otros procesos.

Por ejemplo, en Unix, cuando se carga el sistema operativo, se inicializa un proceso *init*. Este proceso lee un archivo que dice cuántos terminales hay, y crea un proceso *login* para cada terminal, que se encarga de solicitar nombre y clave. Cuando un usuario entra, *login* determina qué shell le corresponde al usuario, y crea otro proceso para ejecutar esa shell. A su vez, la shell crea más procesos según los comandos que ejecute el usuario, generándose así todo un *árbol* de procesos: cada proceso tiene cero o más hijos, y exactamente un padre (salvo *init*, que no tiene padre).

Un proceso se destruye cuando éste hace una llamada al sistema, pidiéndole que lo elimine. También hay otras formas en que un proceso puede ser destruido. Un proceso puede destruir a otro (mediante una llamada al sistema). Obviamente, deben existir ciertas restricciones (p.ej., para que un usuario no pueda matar procesos de otro usuario, o del sistema). Por lo general, un proceso puede ser destruido sólo por su padre o por el sistema. Otra situación en que un proceso es destruido ocurre cuando el proceso provoca algún error fatal (p.ej., división por cero). Esto causa una interrupción, y el sistema operativo lo destruye.

Hebras (*threads*)

Motivación

Consideremos el problema productor-consumidor: un proceso produce ítems que son consumidos por un proceso consumidor. Para esto se usa un *buffer* que puede contener varios ítems; el productor va depositando los ítems a medida que los produce, y el consumidor los va sacando del buffer a medida que los va consumiendo. Por cierto, hay que preocuparse de que el productor no siga poniendo ítems si el buffer está lleno, y que el consumidor no intente sacar un ítem si el buffer está vacío, o sea, los procesos deben *sincronizarse*.

Por ejemplo, un compilador que genera código intermedio en assembler puede descomponerse en un proceso productor que produce código assembler a partir del archivo fuente, y un consumidor que produce un archivo ejecutable a partir del código assembler.

Las ventajas de resolver esta clase de problemas usando dos procesos, es que tenemos:

- Modularidad, ya que cada proceso realiza una función bien específica.
- Disminución en el tiempo de ejecución. Mientras un proceso espera por I/O, el otro puede realizar trabajo útil. O bien, si se cuenta con más de un procesador, los procesos pueden trabajar en paralelo.

Pero también hay algunas desventajas:

- Los procesos deben sincronizarse entre sí, y deben compartir el buffer. También podría ser necesario (o conveniente) que compartan otra clase de recursos (como archivos abiertos). ¿Cómo puede hacerse esto si los procesos tienen espacios de direccionamiento disjuntos? (Recordemos que el sistema operativo no permite que un proceso escriba en la memoria que le corresponde a otro proceso).
- Los cambios de contexto son caros. En la medida que haya más procesos para resolver un problema, más costos habrá debido a los cambios de contexto. Además, al cambiar de un proceso a otro, como los espacios de direccionamiento son disjuntos, hay otros costos indirectos (que tienen relación con cachés y TLBs, que veremos más adelante).

Alternativa: *threads* o hebras o procesos livianos. Una hebra es un hilo de control dentro de un proceso. Un proceso tradicional tiene sólo una hebra de control. Si usamos threads, entonces podemos tener varias hebras dentro de un proceso. Cada hebra representa una actividad o unidad de computación dentro del proceso, es decir, tiene su propio PC, conjunto de registros y stack, pero comparte con las demás hebras el espacio de direccionamiento y los recursos asignados, como archivos abiertos y otros.

En muchos aspectos los procesos livianos son similares a los procesos pesados: comparten el tiempo de CPU, y a lo más un thread está activo (ejecutando) a la vez, en un monoprocesador. Los otros pueden estar listos o bloqueados. Pero los procesos pesados son independientes, y el sistema operativo debe proteger a unos de otros, lo que acarrea algunos costos. Los procesos livianos dentro de un mismo proceso pesado no son independientes: cualquiera puede accesar toda la memoria correspondiente al proceso pesado. En ese sentido, no hay protección entre threads: nada impide que un thread pueda escribir, por ejemplo, sobre el stack de otro. Bueno, quien debiera impedirlo es quien programa los threads, que es una misma persona o equipo (por eso no se necesita protección).

Comparando hebras con procesos:

- Cambio de contexto entre hebras de un mismo proceso es mucho más barato que cambio de contexto entre procesos; gracias a que comparten el espacio de direccionamiento, un cambio de contexto entre threads no incluye los registros y tablas asociados a la administración de memoria.
- La creación de threads también es mucho más barata, ya que la información que se requiere para mantener un thread es mucho menos que un PCB. La mayor parte de la información del PCB se comparte entre todos los threads del proceso.
- El espacio de direccionamiento compartido facilita la comunicación entre las hebras y el compartimiento de recursos.

Implementación de hebras

Hay librerías de hebras que permiten implementar hebras dentro de procesos normales o pesados sin apoyo del sistema operativo. La planificación y manejo de las hebras se hace dentro del proceso. El sistema operativo no tiene idea de las hebras; sólo ve y maneja un proceso como cualquier otro.

La alternativa es que el propio sistema operativo provea servicios de hebras; así como se pueden crear procesos, se pueden también crear nuevas hebras dentro de un proceso, y éstos son administrados por el sistema operativo.

La ventaja del primer esquema respecto del segundo, es que los cambios de contexto entre hebras de un mismo proceso son extremadamente rápidos, porque ni siquiera hay una llamada al sistema de por medio. La desventaja es que si una de las hebras hace una llamada bloqueante (p.ej., solicita I/O), el sistema operativo bloquea todo el proceso, aún cuando haya otras hebras listos para ejecutar.

Usos

¿Cuál es un uso natural para las hebras? Ya vimos el caso productor-consumidor. Otro uso es en los servidores. Cada hebra en el servidor puede manejar una solicitud. El diseño es más sencillo y este esquema mejora la productividad, porque mientras unos hilos esperan, otros pueden hacer trabajo útil.

Ejemplo: servidor de archivos, que recibe solicitudes para leer y escribir archivos en un disco. Para mejorar el rendimiento, el servidor mantiene un caché con los datos más recientemente usados, en la eventualidad que reciba algún requerimiento para leer esos datos, no es necesario accesar el disco. Cuando se recibe una solicitud, se le asigna a un thread para que la atienda. Si ese thread se bloquea porque tuvo que accesar el disco, otros threads pueden seguir atendiendo otras solicitudes. La clave es que el buffer debe ser compartido por todos los threads, lo que no podría hacerse si se usa un proceso pesado para cada solicitud.

IIC2332: Sistemas Operativos

Ultima modificación: July 01, 1998, por Juan E. Navarro (jnavarro@ing.puc.cl)

Go backward to Procesos
Go up to Top
Go forward to Sincronización

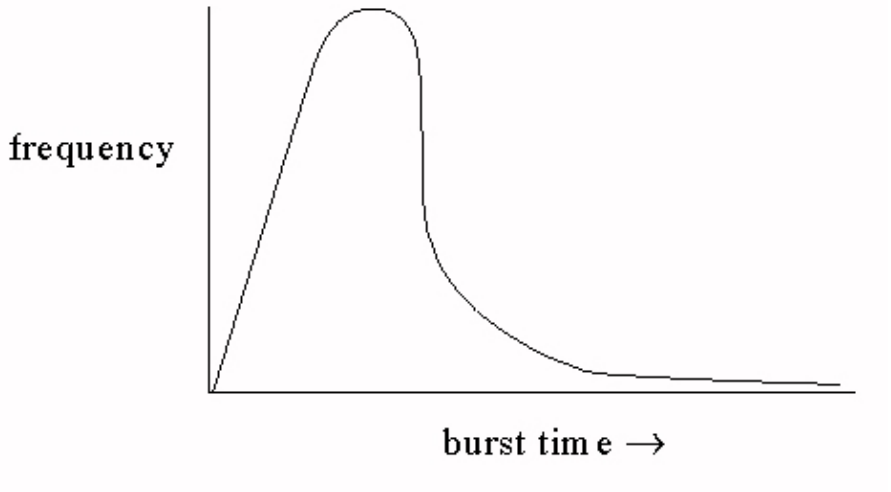
Planificación (*scheduling*) de procesos

Cuando hay más de un proceso que está en condiciones de ejecutar en la CPU, se debe escoger alguno. El encargado de tomar esa decisión es el **planificador** o *scheduler*, y el algoritmo que usa se llama **algoritmo de planificación**.

Posibles objetivos (algunos de ellos contradictorios) del algoritmo de planificación son:

- **Justicia.** Asegurarse que todos los procesos tengan su turno de CPU.
- **Eficiencia.** Mantener la CPU ocupada todo el tiempo.
- **Tiempo de respuesta.** Minimizar el tiempo de respuesta de los usuarios interactivos.
- **Rendimiento o productividad (*throughput*).** Maximizar el número de trabajos terminados por hora.
- **Tiempo de espera.** Minimizar el tiempo medio de espera (en la cola READY) de los procesos.

Una complicación adicional que hay que tener presente es que cada proceso es único e impredecible. Algunos son **intensivos en I/O** (*I/O-bound*), es decir, pierden la mayor parte del tiempo esperando por I/O; otros son **intensivos en CPU** (*CPU-bound*), es decir, requieren principalmente tiempo de CPU. En cualquier caso, todos los procesos alternan entre una fase de ejecución de CPU y otra de espera por I/O. Aunque la duración de las fases de CPU es impredecible y varía mucho entre un proceso y otro, tiende a tener una frecuencia como la de la figura: hay un gran número de fases de CPU cortos, y muy pocos largos. Esta información puede ser importante para seleccionar un algoritmo de planificación adecuado.



¿Cuándo hay que planificar? Recordando el diagrama de transiciones, una decisión de planificación puede o debe tomarse cuando ocurre cualquiera de las siguientes transiciones entre estados de un proceso:

1. EJECUTANDO a BLOQUEADO.
2. EJECUTANDO a TERMINADO.
3. EJECUTANDO a LISTO.
4. BLOQUEADO a LISTO.

En los casos 1 y 2, **necesariamente** hay que escoger un nuevo proceso, pero en los casos 3 y 4 podría no tomarse ninguna decisión de scheduling, y dejar que continúe ejecutando el mismo proceso que estaba ejecutando. En ese caso, se habla de *planificación no-expropiadora* (*nonpreemptive*, como en Windows 3.x). Si en cambio se toma una decisión de scheduling en los casos 3 y 4, entonces se habla de *planificación expropiadora*.

Esta última es más segura y más justa, pero tiene un problema: consideremos dos procesos que comparten información, y que a uno de ellos se le quita la CPU justo cuando estaba en medio de una actualización de los datos compartidos. Cuando sea el turno del segundo proceso, éste podría intentar leer los datos cuando están en un estado inconsistente. Este problema se remedia con sincronización.

El primero que llega se atiende primero

FCFS (First-come, first-served) por sus siglas en inglés. Es un algoritmo que no usa expropiación, y que consiste en atender a los procesos por estricto orden de llegada a la cola READY. Cada proceso ejecuta hasta que termina, o hasta que hace una llamada bloqueante (de I/O), o sea, ejecuta su fase de CPU completa. La gracia es que se trata de un algoritmo muy simple: la cola READY se maneja como una simple cola FIFO. El problema es que el algoritmo es bastante malo.

Tiempo de espera

Consideremos que los procesos P_1, P_2 y P_3 están LISTOS para ejecutar su siguiente fase de CPU, cuya duración será de 24, 3 y 3 milisegundos, respectivamente. Si ejecutan en el orden P_1, P_2, P_3, entonces los tiempos de espera son: 0 para P_1, 24 para P_2 y 27 para P_3, o sea, en promedio, 17 ms. Pero si ejecutan en orden P_2, P_3, P_1, entonces el promedio es sólo 3 ms. En consecuencia, FCFS no asegura para nada que los tiempos de espera sean los mínimos posibles; peor aún, con un poco de mala suerte pueden llegar a ser los máximos posibles.

Utilización de CPU

Ahora supongamos que tenemos un proceso intensivo en CPU y varios procesos intensivos en I/O. Entonces podría pasar lo siguiente: El proceso intensivo en CPU toma la CPU por un período largo, suficiente como para que todas las operaciones de I/O pendientes se completen. En esa situación, todos los procesos están LISTOS, y los dispositivos desocupados. En algún momento, el proceso intensivo en CPU va a solicitar I/O y va a liberar la CPU. Entonces van a ejecutar los otros procesos, pero como son intensivoes en I/O, van a liberar la CPU muy rápidamente y se va a invertir la situación: todos los procesos van a estar BLOQUEADOS, y la CPU desocupada. Este fenómeno se conoce como **efecto convoy**, y se traduce en una baja utilización tanto de la CPU como de los dispositivos de I/O. Obviamente, el rendimiento mejora si se mantienen ocupados la CPU y los dispositivos (o sea, conviene que no haya colas vacías).

El trabajo más corto primero

SJN (shortest-job-next) por sus siglas en inglés.

Supongamos que tenemos tres procesos cuyas próximas fases de CPU son de *a*, *b* y *c* milisegundos de duración. Si ejecutan en ese orden, el tiempo medio de espera es:

$$(0 + a + (a + b))/3 = (2a+b)/3$$

O sea, el primer proceso que se ejecute es el que tiene mayor incidencia en el tiempo medio, y el último, tiene incidencia nula. En conclusión, el tiempo medio se minimiza si se ejecuta siempre el proceso con la menor próxima fase de CPU que esté LISTO. Además, es una buena manera de prevenir el efecto convoy. Lo malo es que para que esto funcione, hay que adivinar el futuro, pues se requiere conocer la duración de la *próxima* fase de CPU de cada proceso.

Lo que se hace es predecir la próxima fase de CPU en base al comportamiento pasado del proceso, usando un *promedio exponencial*. Supongamos que nuestra predicción para la *n*-ésima fase es T_n, y que en definitiva resultó ser t_n. Entonces, actualizamos nuestro estimador para predecir T_{n+1}

$$T_{n+1} = (1-alpha)\, t_n + alpha T_n$$

El parámetro *alpha*, entre 0 y 1, controla el peso relativo de la última fase en relación a la historia pasada.

$$T_{n+1} = (1-alpha)t_n + alpha(1-alpha)t_{n-1} + \ldots + alpha^j(1-alpha)t_{n-j} + \ldots + alpha^{n+1}T_0$$

O sea, mientras más antigua la fase menos incidencia tiene en el estimador.

Un valor atractivo para *alpha* es 1/2, ya que en ese caso sólo hay que sumar los valores y dividir por dos, operaciones especialmente fáciles en aritmética binaria.

Planificación por prioridad

SJN es un caso especial de planificación por prioridad, en el cual a cada proceso se le asigna una prioridad, y la CPU se asigna al proceso con mayor prioridad en la cola READY. SJF es planificación por prioridad donde la prioridad es función del estimador de la duración de la próxima fase de CPU.

Hay muchas criterios para definir la prioridad. Ejemplos:

- Según categoría del usuario.
- Según tipo de proceso: sistema, interactivo, o por lotes; o bien, intensivo en CPU o intensivo en I/O.
- Según cuanto hayan ocupado la CPU hasta el momento
- Para evitar que un proceso de baja prioridad sea postergado en demasía, aumentar prioridad mientras más tiempo lleve esperando: envejecimiento (*aging*).
- Para evitar que un proceso de alta prioridad ejecute por demasiado tiempo, se le puede ir bajando la prioridad.

Carrusel o *round-robin* (RR)

Volviendo a FCFS, una forma obvia de mejorarlo es agregando expropiación o *preemption* de manera que cada proceso no retenga la CPU por más de un **quantum** o tajada de tiempo predefinida. FCFS con tajada de tiempo se conoce como *round-robin*, y se implementa igual que FCFS, sólo que antes de cederle la CPU a un proceso se echa a andar el timer para que provoque una interrupción dentro de un quantum de tiempo. El proceso ejecuta hasta que haga una llamada bloqueante o hasta que use toda su tajada se tiempo. En cualquiera de los dos casos, la CPU se entrega al siguiente en la cola READY.

El punto interesante es encontrar el quantum adecuado. Si es muy grande, la cosa degenera en FCFS, pero tampoco puede ser demasiado pequeño, porque entonces el costo en cambios de contexto es preponderante. Por ejemplo, si un cambio de contexto toma 5 ms, y fijamos el quantum en 20 ms, entonces 20% del tiempo de la CPU se perderá en sobrecosto. Un valor típico es 100 ms. Una regla que suele usarse es que el 80% de las fases de CPU deben ser de menor duración que un quantum.

Con respecto a FCFS, se mejora el tiempo de respuesta y la utilización de la CPU, ya que se mantienen más balanceadas las colas READY y BLOCKED. Pero RR tampoco asegura que los tiempos de espera sean los mínimos posibles. Usando el mismo ejemplo anterior, y considerando un quantum de 4ms, pero sin considerar costos de cambio de contexto, si el orden es P_1, P_2, P_3 entonces el tiempo medio de espera es 5.66ms (¿por qué?).

Múltiples colas

Para complicar más la cosa, podemos agrupar los procesos en distintas clases, y usar distintos algoritmos de planificación intra-clase, más algún algoritmo inter-clases. Por ejemplo, los procesos interactivos y los procesos por lotes tienen distintos requerimientos en cuanto a tiempos de respuesta. Entonces, podemos planificar los procesos interactivos usando RR, y los procesos por lotes según FCFS, teniendo los primeros prioridad absoluta sobre los segundos.

Una forma de implementar este algoritmo es dividiendo la cola READY en varias colas, según la categoría del proceso. Por ejemplo, podemos tener una cola para

- Procesos de sistema.
- Procesos interactivos.
- Procesos de los alumnos.
- Procesos por lotes.

Cada cola usa su propio algoritmo de planificación, pero se necesita un algoritmo de planificación entre las colas. Una posibilidad es prioridad absoluta con expropiación. Otra posibilidad: asignar tajadas de CPU a las colas. Por ejemplo, a la cola del sistema se le puede dar el 60% de la CPU para que haga RR, a la de procesos por lotes el 5% para que asigne a sus procesos según FCFS, y a las otras el resto.

Por otra parte, podríamos hacer que los procesos migren de una cola a otra. Por ejemplo: varias colas planificadas con RR, de prioridad decreciente y quantum creciente. La última se planifica con FCFS. Un proceso en la cola *i* que no termina su fase de CPU dentro del quantum asignado, se pasa al final de la siguiente cola de menor prioridad, pero con mayor quantum. Un proceso en la cola *i* que sí termina su fase de CPU dentro del quantum asignado, se pasa al final de la siguiente cola de mayor prioridad, pero con menor quantum. Ejemplo:

> Cola 0: quantum=10 ms, 40% de CPU.
> Cola 1: quantum=20 ms, 30% de CPU.

Cola 2: quantum=35 ms, 20% de CPU.
Cola 3: FCFS, 10% de CPU.

Así los procesos de fases más cortas tienen prioridad. Este algoritmo es uno de los más generales, pero también uno de los más complejos de implementar. También es difícil de afinar, pues hay múltiples parámetros que definir.

Planificación en dos niveles

Hasta ahora de alguna manera hemos supuesto que todos los procesos ejecutables están en memoria. Pero si hay poca memoria disponible y muchos procesos, entonces algunos procesos deben mantenerse en disco, lo que cambia radicalmente la problemática de la planificación, porque el tiempo requerido para hacer un cambio de contexto que involucre traer un proceso del disco es muchísimo mayor que el tiempo de un cambio de contexto entre procesos en memoria.

Las cosas se simplifican si se divide el problema en dos, y se usa un scheduler distinto para cada caso. Un scheduler **de corto plazo** se encarga sólo de decidir a qué proceso se le asigna la CPU, de entre todos los que están en memoria. Periódicamente, otro scheduler **de largo plazo** decide qué procesos que han estado demasiado tiempo en memoria deben ser pasados a disco para dar oportunidad a procesos que han estado mucho rato en el disco. Para tomar esa decisión se pueden usar factores como el tiempo que un proceso lleva en memoria o disco, cantidad de CPU usada hasta el momento, tamaño del proceso, prioridad, etc.

Planificación en multiprocesadores

Cuando hay varias CPUs (y una memoria común), la planificación también se hace más compleja. Podríamos asignar una cola READY a cada procesador, pero se corre el riesgo de que la carga quede desbalanceada: algunos procesadores pueden llegar a tener una cola muy larga de procesos para ejecutar, mientras otros están desocupados (con la cola vacía). Para prevenir esta situación se usa una cola común de procesos listos, para lo cual hay dos opciones:

1. Cada procesador es responsable de su planificación, y saca procesos de la cola READY para ejecutar. ¿El problema? Hay ineficiencias por la necesaria sincronización entre los procesadores para accesar la cola.
2. Dejar que sólo uno de los procesadores planifique y decida qué procesos deben correr los demás: **multiprocesamiento asimétrico**.

Evaluación de los algoritmos

Está claro que hay numerosas alternativas de planificación. ¿cómo seleccionar una? Primero, hay que definir qué métricas son las que nos importan más: p.ej., maximizar la utilización de la CPU pero con la restricción que el tiempo de respuesta no supere 1 segundo. Después hay que escoger un método para evaluar los algoritmos y ver cuál se ajusta mejor al criterio escogido.

Modelación determinista

Tomamos un caso en particular, y determinamos analíticamente la medida de interés. Por ejemplo, (Proceso, fase de CPU): *(P_1, 10), (P_2,30), (P_3, 5)*. Podemos medir el tiempo medio de espera para FCFS,

SJF, y RR con quantum de 8 ms.

Desventajas: es demasiado específico y requiere un conocimiento muy exacto de la situación. Ventajas: es rápido. Analizando muchos casos se puede encontrar una tendencia.

Modelos de colas

A pesar de que cada proceso es único e impredecible, sí podemos determinar experimentalmente una distribución de las duraciones de las fases de CPU y de I/O. Típicamente esta distribución es exponencial. De la misma manera podemos determinar la distribución probabilística de los tiempos de llegada de nuevos procesos. El sistema entonces se modela como una colección de servidores (CPU y dispositivos), que atienden procesos. Conociendo la distribución de los tiempos de llegada y de atención, se pueden determinar métricas como utilización (porcentaje de uso de los servidores), tamaños medios de las colas y tiempos medios de espera.

El problema es que para que el modelo sea matemáticamente tratable, hay que hacer simplificaciones y suposiciones que no siempre son realistas. Siendo así, los resultados son aproximados.

Simulaciones

Para obtener una evaluación más acuciosa, podemos hacer una simulación: en vez de resolver el modelo matemáticamente, lo programamos y vemos como se comporta (Nachos es un ejemplo). El simulador tiene una variable que representa el reloj, y a medida que ésta se incrementa se modifica el estado del simulador para reflejar las actividades de los diversos elementos simulados, a la vez que se van computando las estadísticas del caso.

Los eventos que van a conducir la simulación pueden generarse según su distribución probabilística, o también pueden provenir del monitoreo de un sistema real. Los resultados van a ser muy exactos, pero el problema de las simulaciones es que son caras.

Implementación

En términos de exactitud, lo mejor es programar en el sistema operativo el algoritmo que queremos evaluar, echarlo a andar y ver cómo se comporta. Pero:

- También es caro.
- Los sistemas reales, con usuarios de verdad, no son muy adecuados para hacer experimentos.
- La propia implantación del algoritmo puede modificar el comportamiento de los usuarios, desvirtuando las mediciones.

Go backward to Planificación (*scheduling*) de procesos
Go up to Top
Go forward to Bloqueos mutuos (*deadlocks*)

Sincronización

Muchos problemas se pueden resolver más fácilmente o más eficientemente si usamos procesos (o hebras) cooperativos, que ejecutan concurrentemente, técnica que se conoce como *pogramación concurrente*. La pogramación concurrente es una herramienta poderosa, pero introduce algunos problemas que no existen en la programación secuencial (no concurrente).

Supongamos que usamos dos hebras concurrentes para determinar cuántos números primos hay en un intervalo dado.

```
int numPrimos = 0;

void primos (int ini, int fin)
{
  for (i=ini; i<=fin; i++)
    if (primo(i)) numPrimos=numPrimos+1;
}
```

Siendo `numPrimos` una variable global, podemos ejecutar `primos(1,5000)` en paralelo con `primos (5001,10000)` para saber cuántos números primos hay entre 1 y 10000. ¿El problema? `numPrimos` es una variable compartida que se accesa concurrentemente, y puede quedar mal actualizada si se dan determinadas intercalaciones de las operaciones de las dos hebras.

Para ejecutar una operación como `numPrimos=numPrimos+1` se suelen requerir varias instrucciones de máquina, como p.ej.:

```
LOAD numPrimos     |cargar valor en el acumulador
ADD 1              |sumarle 1 al acumulador
STORE numPrimos    |escribir acumulador en memoria
```

Supongamos que, más o menos simultáneamente, las dos hebras encuentran su primer número primo. Podría darse el siguiente *timing* (inicialmente, `numPrimos==0`).

```
Hebra 1                          Hebra 2
   LOAD numPrimos
   ADD 1
                                    LOAD numPrimos
                                    ADD 1
                                    STORE numPrimos
   STORE numPrimos
```

El resultado es que `numPrimos` queda en 1, pero lo correcto sería 2. El problema se produce porque las dos hebras o procesos tratan de actualizar una variable compartida al mismo tiempo; o mejor dicho, una hebra comienza a actualizarla cuando la otra no ha terminado. Esto se conoce como **competencia por datos** (*data race*, o también, *race condition*): cuando el resultado depende del orden particular en que se

intercalan las operaciones de procesos concurrentes. La cosa se resuelve si garantizamos que sólo una hebra a la vez puede estar actualizando variables compartidas.

El problema de la sección crítica

Modelo: *n* procesos, $\{ P_0, P_1, \ldots , P_{n-1}\}$, cada uno de los cuales tiene un trozo de código llamado **sección crítica**, en el que accesa variables, o en general, recursos compartidos. El problema de la sección crítica consiste en encontrar un mecanismo o protocolo que permita que los procesos cooperen de manera tal que se cumplan las siguientes condiciones:

- **Exclusión mutua.** La ejecución de las respectivas secciones críticas es mutuamente exclusiva, es decir, nunca hay más de un proceso ejecutando su sección crítica.
- **Ausencia de postergación innecesaria.** Si un proceso quiere entrar en su sección crítica, entonces podrá hacerlo si todos los demás procesos están ejecutando sus secciones no-críticas.
- **Entrada garantizada (ausencia de inanición).** Si un proceso quiere entrar a la sección crítica, entonces (algún día) entrará.

Podemos pensar que cada proceso ejecuta:

```
while(TRUE) {
   protocolo de entrada
   sección crítica
   protocolo de salida
   sección no crítica
}
```

Vamos a suponer que:

- Todos los procesos tienen oportunidad de ejecutar.
- Todo proceso que comienza a ejecutar su sección crítica, ejecutará en algún momento el protocolo de salida.
- Las intrucciones básicas de lenguaje de máquina como LOAD, STORE y TEST se ejecutan de forma atómica.

No debemos hacer ninguna suposición respecto de la velocidad relativa de ejecución de un proceso respecto de otro. Por ahora, vamos a restringir el análisis al caso de dos procesos.

Primer intento

Deshabilitar interrupciones. Se descarta, por peligroso: un proceso de usuario con capacidad de deshabilitar interrupciones puede botar el sistema. Además, no funcionaría en multiprocesadores.

Segundo intento

Compartir una variable `turno` que indique quién puede entrar a la sección crítica.

```
while (TRUE) {                    Otro proceso: simétrico.
  while (turno != 0);
  sección crítica
  turno = 1;
  sección no crítica
```

```
}
```

Provee exclusión mutua, pero hay postergación innecesaria (estricta alternancia).

Tercer intento

Para remediar el problema de la alternancia estricta, podríamos manejar más información: `interesado[i]` está en verdadero si proceso *i* quiere entrar a la sección crítica.

```
int interesado[2];

Proceso i (i=0,1)::
   while (TRUE) {
     interesado[i] = TRUE;
     while (interesado[1-i]);
     sección crítica
     interesado[i]=FALSE;
     sección no crítica
   }
```

Tampoco funciona: si ambos procesos ponen `interesado[i]` en TRUE al mismo tiempo, ambos quedan esperando para siempre. Podríamos cambiar el orden: chequear primero y setear despúes la variable `interesado`, pero entonces podríamos terminar con los dos procesos en la sección crítica.

Cuarto intento: algoritmo de Peterson

Combinando las dos ideas anteriores podemos obtener un algoritmo que sí funciona.

```
const otro=1-i;
while (TRUE) {
  interesado[i] = TRUE;
  turno = otro;
  while (interesado[otro] && turno==otro);

  sección crítica

  interesado[i]=FALSE;

  sección no crítica
}
```

Inicialmente `interesado[0]=intersado[1]=FALSE` y `turno` puede ser 1 o 0.

Exclusión mutua.
Supongamos que ambos procesos quieren entrar. Claramente, no pueden entrar al mismo tiempo, porque la condición de espera en el while `(interesado[otro] && turno==otro)` no puede ser negativa para ambos procesos a la vez. Entonces supongamos que P_i entra primero. En ese caso, `turno=i`, y para que el otro entre, `turno` debe cambiar a `1-i`, o bien `interesado[i]` a `FALSE`. El único que hace esos cambios es el propio proceso *i*, y lo hace cuando está fuera de su sección crítica.

Ausencia de postergación innecesaria.
Si proceso *i* quiere entrar y el otro está ejecutando su sección no crítica, entonces `interesado[otro]==FALSE` y P_i no tiene impedimento para entrar.

Entrada garantizada.

Si un proceso quiere entrar, a lo más debe esperar que el otro salga de la sección crítica. Hay que tener cuidado cuando se demuestra esta propiedad. En este caso, una vez que se cumple la condición que permite a un proceso entrar, la condición permanece al menos hasta que le toca la CPU al proceso que quiere entrar. Si la condición fuera intermitente, hay que considerar que el scheduler puede ser nuestro enemigo y tomar decisiones de planificación caprichosas, de manera de darle la CPU al proceso que quiere entrar justo cuando la condición se apaga.

Solución para *n* procesos: en sección 6.2.2 (Multiple-Process Solution, página 170) del Silberschatz (cuarta edición).

Con ayuda del hardware

Muchos sistemas tienen instrucciones especiales que ayudan bastante a la hora de resolver el problema de la sección crítica. En general, son instrucciones que de alguna manera combinan dos o tres operaciones en una sola operación atómica. Por ejemplo, TEST-AND-SET (TS) lee el contenido de una dirección de memoria en un registro, y pone un 1 en la misma dirección, todo en una sola acción atómica o indivisible, en el sentido que ningún otro proceso puede accesar esa dirección de memoria hasta que la instrucción se complete. De cierto modo es equivalente a:

```
TS (registro,variable):
  deshabilitar interrupciones
  LOAD registro,variable
  STORE variable, 1
  rehabilitar interrupciones
```

Con esto, una solución al problema de la sección crítica, en *pseudoasssembler*, sería:

```
Entrada: TS (R, lock)     |Copiar lock a R, poner lock en 1
         cmp R, 0         |Valor de lock era 0?
         jne Entrada      |No era 0, repetir
         ret              |Era 0, retornar: podemos entrar

Salida:  store lock, 0    |Volver lock a 0
         ret
```

Inicialmente, la variable `lock` está en 0. Este protocolo provee exclusión mutua sin postergación innecesaria, y sirve para *n* procesos, pero puede haber inanición ¿por qué?

Solución sin inanición: en sección 6.3 (Synchronization Hardware, página 174) del Silberschatz (cuarta edición).

Semáforos

Inconvenientes de las soluciones anteriores:

1. Es difícil generalizarlas para problemas de sincronización más complejos, o simplemente distintos, de la sección crítica.
2. Derrochan CPU, ya que usan **espera ocupada** (*busy waiting*): cuando un proceso quiere entrar a la sección crítica, está permanentemente chequeando si puede hacerlo.

Si el sistema operativo tuviera un mayor conocimiento de la situación, podría bloquear a un proceso que quiere entrar a la sección crítica, sin concederle oportunidad de ejecutar, mientras no pueda entrar. Una posibilidad es usar semáforos. Un semáforo *S* es un tipo abstracto de dato que puede manipularse mediante dos operaciones: **P** (o *wait* o *down*) y **V** (o *signal* o *up*), y cuyo efecto es equivalente a:

```
P(S): while (S.val<=0);
      S.val=S.val-1;

V(S): S.val=S.val+1;
```

donde `S.val` es un entero que corresponde al *valor* del semáforo y que nunca debe ser negativo. El incrementos a `S.val` deben ejecutarse de manera indivisible o atómica. Asimismo, el chequeo `S.val<=0` en conjunto con el posible decremento deben ejecutarse también de manera atómica.

Usos

El problema de la sección crítica se puede resolver fácilmente con un semáforo S, con valor inicial 1.

```
Semaphore S; // inicialmente en 1

while (TRUE) {
  P(S);
  sección crítica
  V(S);
  sección no crítica
}
```

También, si queremos que un proceso P_2 ejecute un trozo de código C_2 pero sólo después que P_1 haya completado otro trozo C_1, usamos semáforo con valor inicial 0.

```
P1:: C1; V(S);

P2:: P(S); C2;
```

Implementación

Podríamos implementar semáforos usando directamente la definición anterior, usando algún protocolo de exclusión mutua para proteger los accesos a `S.val`. Pero la idea era justamente no usar espera ocupada. Alternativa: ya que los semáforos tienen una semántica precisa, el sistema operativo puede proveer semáforos, y bloquear a un proceso que hace **P**(S) hasta que `S.val` sea positivo.

Para implementarlos en el sistema operativo, además del valor asociado al semáforo, manejamos una lista con los procesos que están bloqueados esperando que el valor se haga positivo.

```
struct Semaforo {
  int val;
  PCB *Bloqueados;  // Puntero a primer proceso de la lista
}

P (Semaforo S)
{
   if (S.val > 0)
      S.val=S.val-1;
```

```
   else {
      Poner proceso en la lista S.Bloqueados
      Ceder CPU
   }
}

V (Semaforo S)
{
   if (S.Bloqueados==NULL)
      S.val=S.val+1;
   else
      Sacar un proceso de S.Bloqueados y ponerlo en READY
}
```

Lo habitual es que la lista sea una cola FIFO, lo que garantiza que los procesos bloqueados no esperarán eternamente, pero la solución sigue siendo correcta si se usa otra estrategia. Lo que falta por resolver es que las ejecuciones de las operaciones P y V tengan la atomicidad requerdida. Dos posibilidades:

1. Deshabilitar interrupciones. Ahora no es peligroso, pues se trata de rutinas del sistema operativo. No sirve para multiprocesadores.
2. Usar algún protocolo de exclusión mutua, como los que hemos visto. ¡Otra vez tendríamos espera ocupada! El punto es que ahora sabemos que las secciones críticas son muy cortas, y en consecuencia es muy poco probable que un proceso gaste mucha CPU esperando entrar a la sección crítica.

Problemas clásicos de sincronización

Productores-consumidores con buffer limitado

`Buffer` es una lista circular de *n* elementos, donde `in` apunta al próximo elemento desocupado, y `out` al elemento más antiguo en el buffer.

```
Productor::                        Consumidor::
  while (TRUE) {                     while (TRUE) {
    nextP=producir();                  nextC=buffer[out];
    buffer[in]=nextP;                  out = out+1 % n;
    in = in+1 % n;                     consumir(nextC);
  }                                  }
```

Pero: (1) `buffer` tiene capacidad limitada, y por ende `Productor` no debe poner elementos cuando `buffer` está lleno, y (2) `Consumidor` no debe consumir cuando `buffer` está vacío. Usamos semáforos `lleno` y `vacio`, cuyos valores corresponden, respectivamente, al número de posiciones ocupadas y desocupadas del buffer. Por lo tanto, `lleno` se inicializa en 0 y `vacío` en *n*.

```
Productor::                        Consumidor::
  while (TRUE) {                     while (TRUE) {
    nextP=producir();                  P(lleno);
    P(vacio);                          nextC=buffer[out];
    buffer[in]=nextP;                  out = out+1 % n;
    in = in+1 % n;                     V(vacio);
    V(lleno);                          consumir(nextC);
  }                                  }
```

Esta solución funciona mientras haya sólo 1 productor y un consumidor. Si hay más, hay que evitar que

dos consumidores consuman el mismo elemento, y que dos productores usen la misma posición del buffer. Solución: secciones críticas, pero diferentes, de modo que un productor pueda operar en paralelo con un consumidor.

```
Productor::                          Consumidor::
  while (TRUE) {                       while (TRUE) {
    nextP=producir();                    P(lleno);
    P(vacio);                            P(mutexC);
    P(mutexP);                           nextC=buffer[out];
    buffer[in]=nextP;                    out = out+1 % n;
    in = in+1 % n;                       P(mutexC);
    V(mutexP);                           V(vacio);
    V(lleno);                            consumir(nextC);
 }                                     }
```

Lectores y escritores

Un objeto, tal como un archivo o un registro de una base de datos, es compartido por varios procesos concurrentes. Hay procesos escritores, que modifican el objeto, y procesos lectores, que sólo lo consultan. Puede haber múltiples procesos leyendo el objeto simultáneamente, pero cuando hay un proceso escribiendo, no puede haber ningún lector ni ningún otro escritor.

```
Escritor::
   P(wrt);
   escribir();
   V(wrt);
```

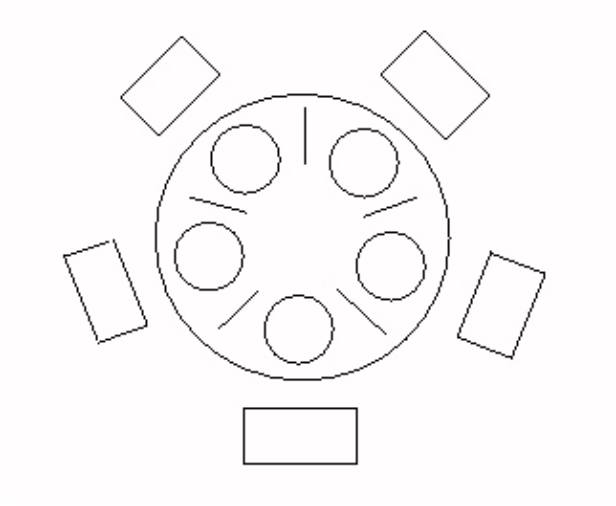
`wrt` es un semáforo inicializado en 1. La idea es que los escritores, como bloque, tmabién hagan `P(wrt)` y `V(wrt)` de manera de impedir accesos concurrentes por parte de lectores y escritores. Solución: primer lector que llega hace `P(wrt)`, y el último que sale hace `V(wrt)`.

```
Lector::
   P(mutex);
   lectores++;
   if (lectores==1) P(wrt);  // Somos los primeros en llegar
   V(mutex);

   leer();

   P(mutex);
   lectores--;
   if (lectores==0) V(wrt);  // Somos los últimos en salir
   V(mutex);
```

Esta solución no es justa, ya que da prioridad a los lectores: si hay un flujo constante de lectores, los escritores nunca podrán escribir. Una alternativa es impedir que entren más lectores si es que hay algún escritor esperando, lo cual sólo invierte el problema: ahora los lectores podrían esperar indefinidamente.

Los filósofos comensales

Cinco filósofos pasan la vida alternando entre comer y pensar, alrededor de una mesa redonda. Como son medio torpes, requieren dos tenedores para comer, pero como son pobres, sólo tienen 5 tenedores. Han acordado que cada uno usará sólo los tenedores a su izq. y a su derecha. Entonces, de los 5 filósofos, sólo 2 pueden comer al mismo tiempo, y no pueden comer dos que son vecinos.



Una posibilidad: representar cada tenedor con un semáforo inicializado en 1 (`Semaforo tenedor[5];`), y cada filósofo *i* hace:

```
while (TRUE) {
   P(Tenedor[i]);
   P(Tenedor[i+1 % 5]);
   comer();
   V(Tenedor[i]);
   V(Tenedor[i+1 % 5]);
   pensar();
}
```

Problema: bloqueo mutuo. Soluciones posibles:

- Usar solución asimétrica: uno de los filósofos toma primero el tenedor `i+1%5` y despúes el `i`.
- Permitir que un filósofo tome sus tenedores sólo si ambos están disponibles (y hacerlo dentro de una sección crítica).
- Permitir que sólo cuatro filósofos se sienten en la mesa.

Estas soluciones evitan bloqueo mutuo, pero también es conveniente asegurar que ningún filósofo muera de hambre.

Monitores

Los semáforos son una herramienta general y eficiente para sincronizar procesos, pero siguen siendo de bajo nivel. Las soluciones mediante semáforos no son ni muy limpias ni muy claras, y siempre están sujetas a errores como la omisión o mala ubicación de una operación **P** o **V**.

Los **monitores** son una herramienta de sincronización más estructurada: un monitor encapsula variables compartidas en conjunto con los procedimientos para accesar esas variables.

```
monitor ejemplo
   // declaración de variables permanentes
   integer i;
   condition c;
```

```
   procedure productor(x);
   :
   end;

   procedure consumidor(x);
   :
   end;

begin
   i=0; // inicialización
end.
```

Un monitor es, en consecuencia, un tipo de dato abstracto, y por lo tanto:

- Procesos pueden llamar a los procedimientos del monitor en cualquier momento, pero **no** pueden accesar directamente las variables encapsuladas por el monitor.
- A la inversa, los procedimientos dentro del monitor no pueden accesar variables externas al monitor: sólo pueden accesar las variables permanentes del monitor (`i` y `c` en el ejemplo), las variables locales al procedimiento, y los argumentos del procedimiento (`x` en el ejemplo).
- Por último, las variables permanentes del monitor deben estar inicializadas antes que ningún procedimiento sea ejecutado.

En un monitor la exclusión mutua es implícita: a lo más un proceso a la vez puede *estar activo* dentro de un monitor. La concurrencia puede darse fuera del monitor o en monitores diferentes. Pero la exclusión mutua no basta para resolver todos los problemas de sincronización. Para esperar eventos (sincronización por condición) se usan **variables de condición**. Una variables de condición `c` tiene asociada una cola de procesos, y soporta dos operaciones:

- `wait(c)`: suspende al proceso invocador, lo pone al final de la cola asociada a `c`, y libera el monitor. O sea, aunque no se haya completado un procedimiento, el proceso deja de estar activo dentro del monitor, y por lo tanto otro proceso puede ejecutar dentro del monitor.
- `signal(c)`: despierta al proceso al frente de la cola asociada a `c`, si es que existe. En monitores con **semántica signal-and-continue**, el proceso que hace `signal` continúa dentro del monitor; el proceso despertado, si existe, debe esperar su oportunidad para volver a entrar al monitor y continuar después del `wait`. En monitores con **semántica signal-and-wait**, si es que el `signal` despierta a un proceso, entonces el proceso señalizador suspende su ejecución en favor del proceso señalizado; el señalizador debe esperar (posiblemente compitiendo con otros procesos) para poder volver al monitor.

A diferencia de los semáforos, los monitores son un *constructo* del lenguaje. Como son de alto nivel y potencialmente igual de eficientes que los semáforos, muchos lenguajes concurrentes proveen monitores (Modula, Concurrent Pascal). En tal caso, el compilador es quien debe traducir los monitores a instrucciones que comprendan herramientas de más bajo nivel (p.ej., semáforos). C++ no tiene monitores.

Ejemplo: productores-consumidores con monitor signal-and-wait.

```
monitor PC
   TipoX buffer[n];
   integer in, out, cta;
   condition no_lleno, no_vacio;

   procedure depositar (TipoX data);
```

```
      if (cta==n) wait(no_lleno);
      bufer[in] = data;
      in = in+1 % n;
      cta = cta+1;
      signal (no_vacio);
   end;

   procedure sacar (var TipoX data);
      if (cta==0) wait(no_vacio);
      data = buffer[out];
      out = out+1 % n;
      cta = cta-1;
      signal (no_lleno);
   end

begin
   in=0; out=0; cta=0;
end.
```

¿Cómo cambia la solución si se dispone de monitores signal-and-continue? Otro ejemplo: filósofos comensales, basándose en el estado de los filósofos (HAMbriento, COMiendo, PENsando).

```
monitor FC
  integer Estado[5];  //HAM,COM,PEN
  condition C[5];

  procedure tomar (integer i);
    Estado[i]=HAM;
    Test(i);
    if (Estado[i]!=COM) wait(C[i]);
  end;

  procedure dejar (integer i);
    Estado[i]=PEN;
    Test(i+4 % 5);
    Test(i+1 % 5);
  end;

  local procedure test (integer k);
    if (Estado[k+4 % 5]!=COM and Estado[k]==HAM
                        and Estado[k+4 % 5]!=COM) begin
      Estado[k]=COM;
      signal(C[k]);
    end;
  end;

begin
  for (i=0 to 4) Estado[i]=PEN;
end.
```

Esta solución está libre de bloqueos mutuos, pero puede haber inanición. Funciona con ambas semánticas de monitores (¿por qué?).

Paso de mensajes

Primitivas que hemos visto hasta ahora sirven para sincronizar procesos, pero si además queremos que los procesos intercambien información tenemos que agregar variables compartidas a nuestra solución.

¿Y que pasa si los procesos no comparten memoria? Podemos intercambiar información mediante paso de mensajes. En tal caso, el sistema operativo ofrece dos primitivas: `send(destino, mensaje)` y `receive(fuente, mensaje)`.

Problemas de diseño

Si los procesos que se comunican están en diferentes máquinas, los mensajes pueden perderse en la red. Para prevenir esta situación, el receptor debe enviar de vuelta un mensaje de acuse de recibo (*acknowledgment* o simplemente ACK). Así, si el emisor no recibe el ACK correspondiente, entonces reenvía el mensaje. Pero, ¿qué pasa ahora si lo que se pierde no es el mensaje, sino el ACK? El receptor recibirá el mensaje duplicado. Entonces, hay que reconocer duplicados, y para eso se usa secuenciación: cada mensaje lleva un número correlativo.

Otro problema: cómo identificar procesos. El sistema operativo asigna un identificador a cada proceso cuando lo crea, pero si la máquina es reinicializada, entonces lo más probable es que el identificador de la nueva encarnación del proceso sea distinto. Típicamente se usan buzones o puertos, que también son identificadores: un proceso siempre va a *escuchar* o recibir por un puerto determinado.

Ejemplo: productor-consumidor

El consumidor envía *N* mensajes vacíos al productor. Cuando el productor produce un item, toma un mensaje vacío y envía de vuelta uno lleno.

```
void productor()
{
   int item;
   message m;

   while (TRUE) {
      producir(&item);
      receive (consumidor, &m);
      poner_item(&m, item);
      send (consumidor, &m);
   }
}


void consumidor()
{
   int item, i;
   message m;

   // Enviar N mensajes vacíos:
   for (i=0; i<N; i++) send(productor, &m);
   while (TRUE) {
      receive (productor, &m);
      extraer_item (&m, &item);
      send (productor, &m);
      consumir(item);
   }
}
```

Si hubiera varios productores y varios consumidores la solución es más compleja. En general, soluciones multiproceso con paso de mensajes son bastante más complejas que con memoria

compartida.

IIC2332: Sistemas Operativos

Ultima modificación: July 01, 1998, por Juan E. Navarro (jnavarro@ing.puc.cl)

Go backward to Sincronización
Go up to Top
Go forward to Administración de memoria

Bloqueos mutuos (*deadlocks*)

Cuando tenemos muchos procesos que compiten por recursos finitos, puede darse una situación en la que un proceso está bloqueado esperando por un recurso que nunca se liberará, porque lo posee otro proceso también bloqueado.

Ley de principios de siglo, en Kansas: "cuando dos trenes se aproximan a un cruce, ambos deben detenerse completamente, y ninguno podrá continuar hasta que el otro se haya ido." Otro ejemplo: solución al problema de los filósofos comensales del capítulo anterior. Otro ejemplo similar: los semáforos `Q` y `S` controlan acceso a recursos compartidos, y:

```
Proceso 0:              Proceso 1:
   P(Q);                   P(S);
   P(S);                   P(Q);
   usar recursos           usar recursos
   V(S);                   V(Q);
   V(Q);                   V(S);
```

Caracterización

Un conjunto de procesos C está bajo bloqueo mutuo si cada proceso en C está esperando por un evento que sólo otro proceso de C puede causar.

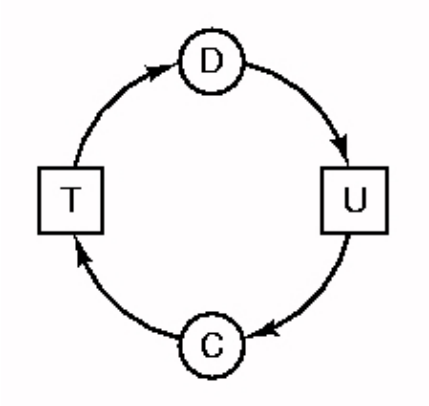
Tal evento es típicamente la liberación de un recurso físico (como impresora) o lógico (como registros o semáforos). Para que haya bloqueo mutuo, deben darse simultáneamente las siguientes condiciones:

1. **Exclusión mutua.** Los recursos no son compartidos. Si un proceso está usando un recurso, otros no pueden hacerlo.
2. **No expropiación.** Los recursos no pueden ser expropiados a los procesos. (Ejemplo: impresora).
3. **Retención y espera.** Hay procesos que tienen recursos asignados y al mismo tiempo están esperando poder adquirir otros recursos.
4. **Espera circular.** Existe un conjunto $\{P_0, P_1, ..., P_n\}$ de procesos tales que el proceso P_i está esperando por un recurso retenido por P_{i+1} para $0 <= i < n$, y P_n esté esperando recurso retenido por P_0.

En rigor, espera circular implica retención y espera, pero es útil separar las condiciones, pues basta con que una no se dé para que no haya bloqueo mutuo.

Una forma de modelar estas condiciones es usando un *grafo de recursos*: círculos representan procesos, los cuadrados recursos. Una arista desde un recurso a un proceso indica que el recurso ha sido asignado al proceso. Una arista desde un proceso a un recurso indica que el proceso ha solicitado el recurso, y está bloqueado esperándolo. Entonces, si hacemos el grafo con todos lo procesos y todos los recursos del

sistema y encontramos un ciclo, los procesos en el ciclo están bajo bloqueo mutuo.



Métodos para manejar bloqueos mutuos

Método del avestruz

Consiste en no hacer absolutamente nada (esconder la cabeza). Los bloqueos mutuos son evidentemente indeseables. Los procesos bajo bloqueo mutuo no pueden avanzar y los recursos que ellos retienen no están disponibles para otros procesos. Pero si un deadlock se produce, en promedio, cada diez años y en cambio el sistema se cae todos los días por fallas en el hardware o en el sistema operativo, entonces el problema de los bloqueos mutuos es irrelevante. Muchos sistema operativo modernos (UNIX entre ellos) no se preocupan de evitar bloqueos mutuos (porque es caro y los procesos deben someterse a ciertas restricciones), pero algunas tendencias hacen que el tema vaya adquiriendo importancia: a medida que progresa la tecnología, tiende a haber más recursos y más procesos en un sistema.

Detección y recuperación

Una posibilidad es mantener el grafo de recursos, actualizándolo cada vez que se solicita o libera un recurso, o bien, cada cierto tiempo construir el grafo. (¿Cada cuánto?) Si se detecta un ciclo, se matan todos los procesos del ciclo (¡se usa!), o se van matando procesos del ciclo hasta que no queden ciclos (¿cuál matar primero? ¿Qué pasa si un proceso está en medio de una transacción?). Es un método drástico, pero sirve.

Prevención

Una tercera estrategia es imponer restricciones a los procesos de manera de hacer estructuralmente imposible la ocurrencia de un bloqueo mutuo. La idea es asegurar que no se dé al menos una de las cuatro condiciones necesarias para que haya bloqueo mutuo.

Exclusión mutua.

Hay recursos que son intrínsecamente no compartibles, de modo que no se puede descartar la exclusión mutua.

No expropiación.

Esta condición se puede eliminar imponiendo expropiación. Si un proceso *P* tiene recursos y solicita otro que está ocupado, se le pueden expropiar a *P* los que ya tiene, o bien expropiarle al otro proceso el recurso que *P* necesita. Es aplicable a cierta clase de recursos (cuyo estado se puede almacenar y luego recuperar), pero no a otros como registros de base de datos o impresoras.

Retención y espera.

Podemos impedir esta condición exigiendo que los procesos soliciten de una vez, al comienzo, todos los recursos que van a necesitar. Si uno o más recursos no están disponibles, el proceso se bloquea hasta que pueda obtenerlos todos. Inconvenientes: muchos procesos no pueden saber de antemano qué y cuántos recursos necesitarán. Subutilización de recursos, ya que quedan retenidos de principio a fin de la ejecución.

Alternativa: Hacer que los procesos liberen todos los recursos que tienen antes de solicitar un nuevo conjunto de recursos. También tiene inconvenientes: por ejemplo, programa que usa archivo temporal durante toda su ejecución. Si lo libera entremedio, otro proceso se lo puede borrar.

Espera circular.

Podemos evitar la espera circular si imponemos un orden total a los recursos (o sea, asignamos a cada recurso *R* un número único *F(R)*), y obligamos a los procesos a que soliciten recursos en orden: un proceso no puede solicitar *Q* y después *R* si *F(Q)>F(R)*. Por ejemplo:

- F(CD-ROM)=1
- F(impresora)=2
- F(plotter)=3
- F(Cinta)=4

De esta manera se garantiza que no se generarán ciclos en el grafo de recursos.

Una mejora inmediata es exigir solamente que ningún proceso solicite un recurso cuyo número es inferior a los recursos que ya tiene. Pero tampoco es la panacea. En general, el número potencial de recursos es tan alto que es difícil dar con tal función *F* para ordenarlos.

Evitación

En vez de restringir la forma u orden en que los procesos deben solicitar recursos, podríamos chequear que sea seguro conceder un recurso, antes de otorgarlo.

Hay diversos algoritmos, que varían en el tipo de información que requieren a priori de cada proceso. En el que vamos a ver, necesitamos que cada proceso declare la cantidad máxima de recursos de cada clase que va a necesitar: Por ejemplo: 2 unidades de cinta, una impresora láser y 200 bloques de disco, como **máximo**.

Estado seguro.

Un **estado de asignación de recursos** es el número de recursos disponibles y asignados, y el máximo declarado por cada proceso. Ejemplo: Sistema con 12 unidades de cinta y 3 procesos.

	Máximo	Actual	Diferencia	Disponible

P	10	5	5	3
Q	4	2	2	
R	9	2	7	

Un **estado seguro** es un estado en el cual el sistema puede asignar recursos a los procesos (hasta su máximo) en alguna secuencia, y evitar bloqueo mutuo. Más formalmente, un estado es seguro sólo si existe una **secuencia segura**, es decir, una secuencia de procesos $<P_1, P_2,...,P_n>$ donde, para cada P_i, los recursos que P_i aún puede solicitar pueden satisfacerse con los recursos disponibles y los retenidos por P_j con $j<i$. Si los recursos que P_i necesita no están disponibles, P_i puede esperar hasta que los P_j terminen. Si no existe tal secuencia, se dice que el sistema está en un estado **inseguro**.

El estado del ejemplo es seguro, ya que la secuencia $<Q,P,R>$ satisface la condición de seguridad. Si en ese estado se le concede una unidad más a *R*, entonces pasamos a un estado inseguro. Un estado inseguro no necesariamente implica que se va a producir bloqueo mutuo, pero se corre el riesgo. En cambio, mientras el sistema esté en un estado seguro, no puede haber bloqueo mutuo. La clave entonces es no pasar a un estado inseguro; en el ejemplo, no se le puede conceder otra unidad a *R*; si la solicita, hay que suspenderlo, aunque el recurso esté disponible. Esto significa que habrá una subutilización de los recursos que no existiría si no se hiciera nada para manejar los bloqueos mutuos.

Algoritmo del banquero

Sistematizando y generalizando para múltiples recursos, obtenemos el *algoritmo del banquero* (se supone que los procesos son clientes que piden crédito, y los recursos corresponden al dinero del que dispone el banco). La idea es básicamente la misma, sólo que debemos manejar vectores en vez de escalares. Por ejemplo, para 5 procesos y 3 recursos, **Máximo**, **Actual** y **Diferencia**, que antes eran columnas, ahora son matrices de 5x3; y **Disponible**, que antes era un escalar, ahora es un vector de 3 elementos.

	Máximo			Actual			Diferencia			Disponible		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
P_0	7	5	3	0	1	0	7	4	3	3	3	2
P_1	3	2	2	2	0	0	1	2	2			
P_2	9	0	2	3	0	2	6	0	0			
P_3	2	2	2	2	1	1	0	1	1			
P_4	4	3	3	0	0	2	4	3	1			

Para chequear si un estado, como el anterior, es seguro:

1. Buscar un proceso (cualquiera) cuyos recursos máximos puedan satisfacerse con los disponibles (más los que ya tiene); o sea, un proceso tal que Diferencia <= Disponible, componente a componente. Si no existe, el estado es inseguro.
2. Si existe, suponer que el proceso termina y libera todos sus recursos, agregándolos a los disponibles.

3. Repetir 1 y 2 hasta haber pasado por todos los procesos, en cuyo caso el estado es seguro, o hasta determinar que el estado es inseguro.

En el ejemplo, podemos escoger primero P_1 o P_3 (da igual); escojamos P_1. Al terminar, los recursos disponibles quedan en $<4,5,4>$. Ahora podemos escoger P_3 o P_4, y así, hasta determinar que el estado es seguro.

Supongamos ahora que, en ese estado, P_1 pide 1 instancia adicional de A y 2 de C. Los recursos están disponibles, pero antes de concedérselos a P_1, debemos chequear si el nuevo estado sería seguro o no. Dicho estado sería:

	Máximo			Actual			Diferencia			Disponible		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
P_0	7	5	3	0	1	0	7	4	3	2	3	0
P_1	3	2	2	3	0	2	0	2	0			
P_2	9	0	2	3	0	2	6	0	0			
P_3	2	2	2	2	1	1	0	1	1			
P_4	4	3	3	0	0	2	4	3	1			

y también es seguro, ya que la secuencia $<P_1, P_3, P_4, P_0, P_2>$ es segura. En consecuencia, podemos otorgar los recursos solicitados por P_1.

Si en ese nuevo estado, P_4 pide ahora 3 unidades de A y 3 de B, entonces P_4 debe suspenderse porque no hay suficientes recursos para satisfacer el pedido. Si P_3 pide 2 unidades adicionales de B, simplemente abortamos el proceso, pues no cumpliría con el máximo declarado. Y si P_0 pide 2 unidades de B, también debe suspenderse, pero en este caso porque si se le otorgan, el estado pasaría a ser inseguro. Cada vez que algún proceso libera recursos, debería chequearse cuáles de las solicitudes pendientes pueden atenderse.

Este algoritmo fue publicado en 1965 y ha sido objeto de innumerables estudios. En teoría funciona muy bien, pero en la práctica, rara vez se usa, puesto que es muy difícil para los procesos saber *a priori* sus necesidades máximas de cada recurso.

En resumen, los algoritmos de prevención son extremadamente restrictivos y los de evitación requieren información que, por lo general, no está disponible. Por eso, los métodos de detección y recuperación no son tan poco usuales ni descabellados. También se usan métodos adecuados a situaciones específicas, pero no existe ningún método *mentholatum* (que sirva para todo).

IIC2332: Sistemas Operativos

Ultima modificación: July 01, 1998, por Juan E. Navarro (jnavarro@ing.puc.cl)

Go backward to Bloqueos mutuos (*deadlocks*)
Go up to Top
Go forward to Memoria Virtual

Administración de memoria

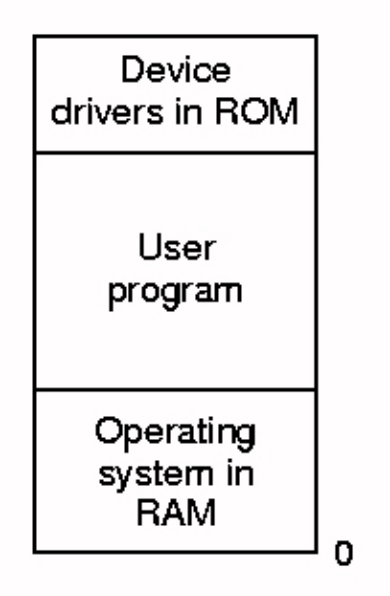
La memoria es un recurso escaso, y para aprovecharla bien hay que administrarla bien. A pesar de que la memoria es más barata cada día, los requerimientos de almacenamiento crecen en proporción similar. Por otra parte, la memoria más rápida es obviamente más cara, por lo que la mayoría de los computadores tiene una *jerarquía de memoria*. Por ejemplo, en un Pentium típico:

1. Caché de nivel 1: 8 KB empaquetados dentro del chip; por lo mismo, la velocidad de acceso es de unos pocos nanosegundos.
2. Caché de nivel 2: 256 a 512 KB, 12-20 ns, U$20/MB
3. Memoria RAM: 8 a 32 MB, 70ns, U$2.5/MB
4. Disco duro. Para almacenamiento estable, y también para extender la RAM de manera virtual. 4GB, 8ms, U$0.08/MB.
5. Cinta. 1 a 40 GB. U$0.01/MB.

Administración básica

Monoprogramación

La forma más simple de administrar memoria es ejecutando sólo un programa a la vez, compartiendo la memoria con el sistema operativo. Por ejemplo, MS-DOS, en direcciones crecientes de memoria: Sistema operativo; programa de usuario; manejadores de dispositivos (en ROM). Cuando usuario digita un comando, el sistema operativo carga el programa correspondiente en la memoria, y lo ejecuta. Cuando el programa termina, el sistema operativo solicita un nuevo comando y carga el nuevo programa en la memoria, sobreescribiendo el anterior.



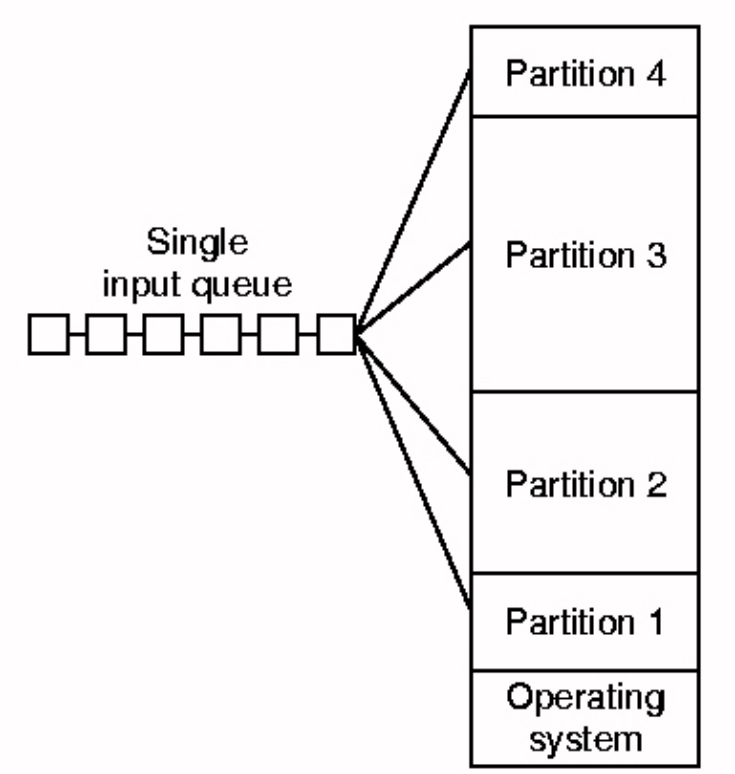
Multiprogramación con particiones fijas

Ya hemos hablado bastante de las ventajas de la multiprogramación, para aumentar la utilización de la CPU. La forma más simple de obtener multiprogramación es dividiendo la memoria en *n* particiones fijas, de tamaños no necesariamente iguales, como lo hacía el IBM 360 en la década del 60.

Puede haber una cola de trabajos por partición, o bien una sola cola general. En el primer caso, cuando llega un trabajo, se pone en la cola de la partición más pequeña en la que todavía quepa el trabajo. Si llegan muchos trabajos pequeños podría pasar que, mientras las colas para las particiones chicas están llenas, las particiones grandes quedan sin uso. En el caso de una sola cola, cada vez que un programa termina y se libera una partición, se escoge un trabajo de la cola general. ¿Cómo escoger?

FIFO:
: podría quedar un trabajo pequeño en una partición grande, desaprovechando memoria.

El más grande que quepa:
: trabajos pequeños son postergados, lo cual no es buena política de programación, si suponemos que los trabajos más chicos son también los más cortos (recordar SJN).



Reubicación y protección

Bajo multiprogramación, si un mismo programa se ejecuta varias veces, no siempre va a quedar en la misma partición. Por ejemplo, si el programa, ejecutando en la partición 1 (que comienza en la dirección 100K) ejecuta un salto a la posición 100K+100, entonces, si el mismo programa ejecuta en la partición 3 (que comienza en 400K), debe ejecutar un salto a la posición 400K+100, porque si salta a la 100K+100 va a caer en la memoria que corresponde a otro proceso. Este es el problema de la *reubicación*, y hay varias formas de resolverlo.

Consideremos un programa en assembler, donde las direcciones se especifican de manera simbólica, como es el caso de la variable `count` y la entrada al ciclo `loop` en el siguiente fragmento de programa:

```
      LOAD R1, count
loop: ADD  R1, -1
      CMP  R1, 0
      JNE  loop
```

La *ligadura* o *binding* de las direcciones `count` y `loop` a direcciones físicas puede realizarse en:

Tiempo de compilación.

Cuando al compilar se conoce la dirección absoluta, el compilador escribe la imagen exacta del código que se cargará en la memoria, por ejemplo:

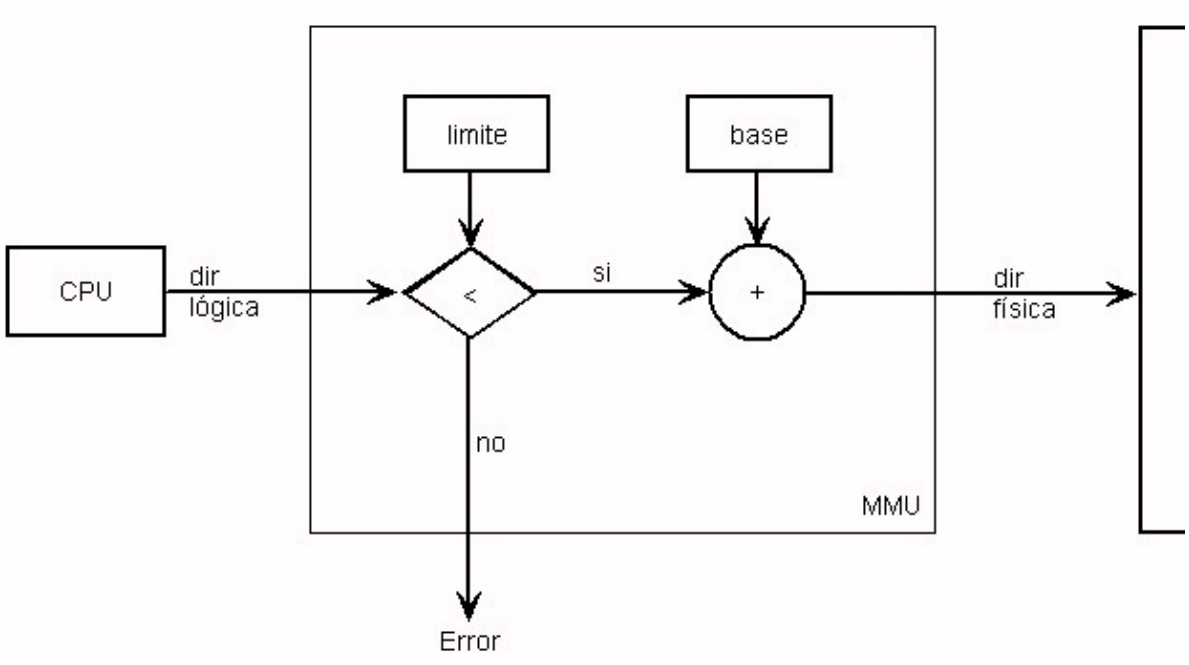
```
100:  LOAD $210, R1
102:  ADD  R1, -1
104:  CMP  R1, 0
106:  JNE  102
```

En este caso el código debe cargarse a partir de la dirección 100.

Tiempo de carga.
Si al compilar no se conocen todavía las direcciones absolutas, hay que generar código reubicable. El *cargador* o *loader*, es responsable de determinar las direcciones absolutas al momento de poner el programa en memoria. Para eso, el compilador genera código como para ejecutar a partir de la dirección 0, y agrega un encabezado con información acerca de cuáles de las posiciones corresponden a direcciones que deben ajustarse según donde se cargue el programa .

Tiempo de ejecución.
Si el sistema operativo usa *swapping* (pasar procesos a disco para hacer espacio en la memoria), puede que un proceso sea cambiado de posición durante su ejecución. En este caso se usa hardware especial.

Pero además de la reubicación tenemos el problema de la protección: queremos que un programa ejecutando en una partición no pueda leer ni escribir la memoria que corresponde a otra partición. Podemos entonces matar dos pájaros de un tiro con una variante de los registro **base** y **límite** que vimos en el capítulo 1. Cuando se carga un programa en una partición, se hace apuntar **base** al comienzo de la partición, y **límite** se fija como la longitud de la partición. Cada vez que la CPU quiere accesar la dirección *d*, el hardware se encarga de que en realidad se accese la dirección física **base**+*d* (y, por supuesto, además se chequea que *d*<**límite**).



Decimos que *d* es la dirección *lógica* o *virtual*, y **base**+*d* la dirección física. El programa de usuario sólo ve direcciones lógicas; es la unidad de administración de memoria (MMU) quien se encarga de traducirlas transparentemente a direcciones físicas. La gracia es que el compilador puede generar código absoluto, pensando que el programa se va a cargar siempre en la posición 0, y en realidad el binding se hace en tiempo de ejecución.

Intercambio (*Swapping*)

Si en un sistema interactivo no es posible mantener a todos los procesos en memoria, se puede usar el disco como apoyo para extender la capacidad de la memoria: pasar procesos temporalmente a disco (recordar scheduler de largo plazo).

El sistema operativo mantiene una tabla que indica qué partes de la memoria están desocupadas, y cuáles en uso. Las partes desocupadas son *hoyos* en la memoria; inicialmente, toda la memoria es un solo gran hoyo. Cuando se crea un proceso o se trae uno del disco, se busca un hoyo capaz de contenerlo, y se pone el proceso allí.

Las particiones ya no son fijas, sino que van cambiando dinámicamente, tanto en cantidad como en ubicación y tamaño. Además, cuando un proceso es pasado a disco, no hay ninguna garantía de que vuelva a quedar en la misma posición de memoria al traerlo de vuelta, de manera que es imprescindible el apoyo del hardware para hacer ligadura en tiempo de ejecución.

Si los procesos pueden crecer, conviene reservar un poco más de memoria que la que estrictamente necesita al momento de ponerlo en memoria. Al hacer swapping, no es necesario guardar todo el espacio que tiene reservado, sino sólo el que está usando. ¿Qué pasa si proceso quiere crecer más allá del espacio que se le había reservado?

Otro punto que hay que tener en cuenta al usar swappping, es el I/O que pudiera estar pendiente. Cuando se hace, por ejemplo, input, se especifica una dirección de memoria donde se va a poner lo que se lea desde el dispositivo. Supongamos que proceso *A* trata de leer del disco hacia la dirección *d*, pero el dispositivo está ocupado: su solicitud, por lo tanto, es encolada. Entretanto, el proceso *A* es intercambiado a disco, y la operación se completa cuando *A* no está en memoria. En esas circunstancias, lo leído se escribe en la dirección *d*, que ahora corresponde a otro proceso. ¿Cómo se puede evitar tal desastre?

Administración de la memoria con mapas de bits

Podemos dividir la memoria en pequeñas unidades, y registrar en un mapa de bits las unidades ocupadas y desocupadas. Las unidades pueden ser de unas pocas palabras cada una, hasta de un par de KB. A mayor tamaño de las unidades, menor espacio ocupa el mapa de bits, pero puede haber mayor fragmentación interna. Desventaja: para encontrar hoyo de *n* unidades hay que recorrer el mapa hasta encontrar *n* ceros seguidos (puede ser caro).

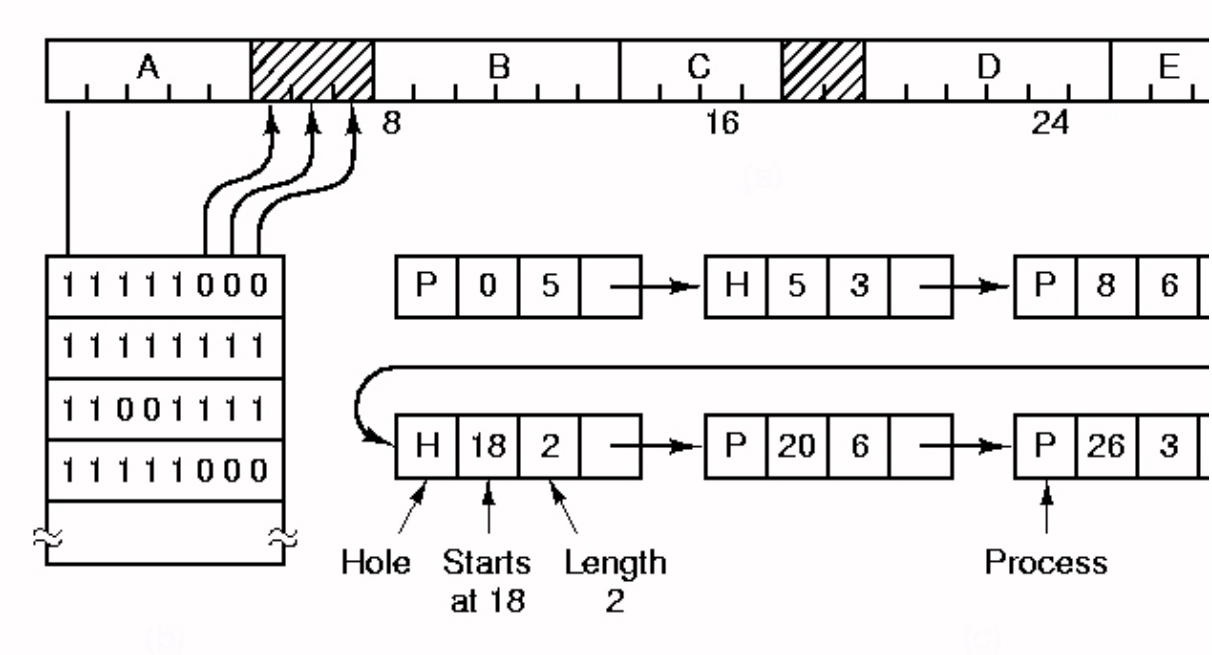
Administración de la memoria con lista ligada

Otra forma es con una lista ligada de segmentos: estado (ocupado o en uso), dirección (de inicio), tamaño. Cuando un proceso termina o se pasa a disco, si quedan dos hoyos juntos, se funden en un solo segmento. Si la lista se mantiene ordenada por dirección, podemos usar uno de los siguientes algoritmos para escoger un hoyo donde poner un nuevo proceso.

- **First-fit.** Asignar el primer hoyo que sea suficientemente grande como para contener al proceso.
- **Best-fit.** Asignar el menor hoyo en el que el proceso quepa.
- **Worst-fit.** Asignar el mayor hoyo.

Cada vez que se asigna un hoyo a un proceso, a menos que quepa exactamente, se convierte en un segmento asignado y un hoyo más pequeño. Best-fit deja hoyos pequeños y worst-fit deja hoyos grandes (¿qué es mejor?). Simulaciones han mostrado que first-fit y best-fit son mejores en términos de utilización de la memoria. First-fit es más rápido (no hay que revisar toda la lista). Se puede pensar en

otras variantes.



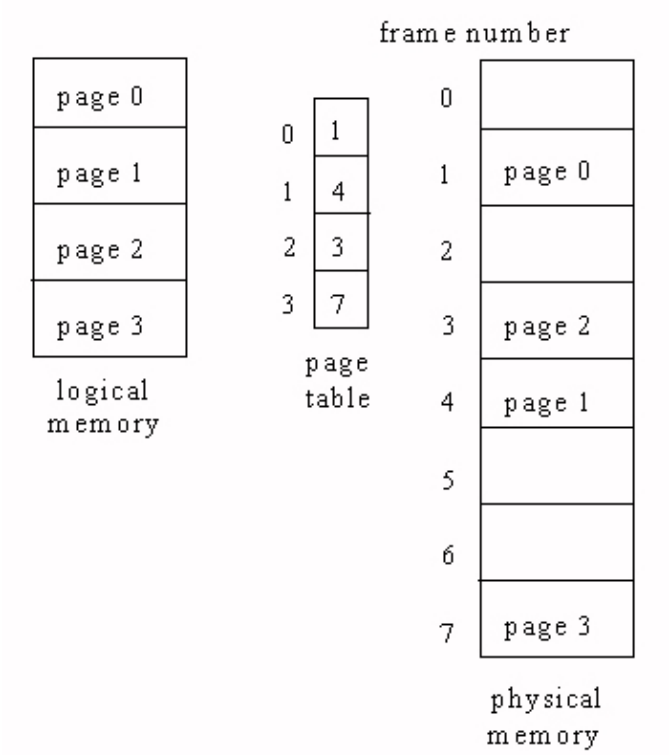
Fragmentación externa e interna

Los métodos anteriores de administración de memoria sufren el problema de la *fragmentación externa*: puede que haya suficiente espacio libre como para agregar un proceso a la memoria, pero no es contiguo. La fragmentación puede ser un problema severo. En algunos sistemas usar first-fit puede ser mejor en términos de fragmentación; en otros puede ser mejor best-fit, pero el problema existe igual. Por ejemplo para first-fit, las estadísticas hablan de un tercio de la memoria inutilizable por estar fragmentada. Una forma de resolver este problema es usando *compactación de la memoria*: mover los procesos de manera que queden contiguos. Generalmente no se hace por su alto costo. Si se usa, no es trivial decidir la mejor forma de hacerlo.

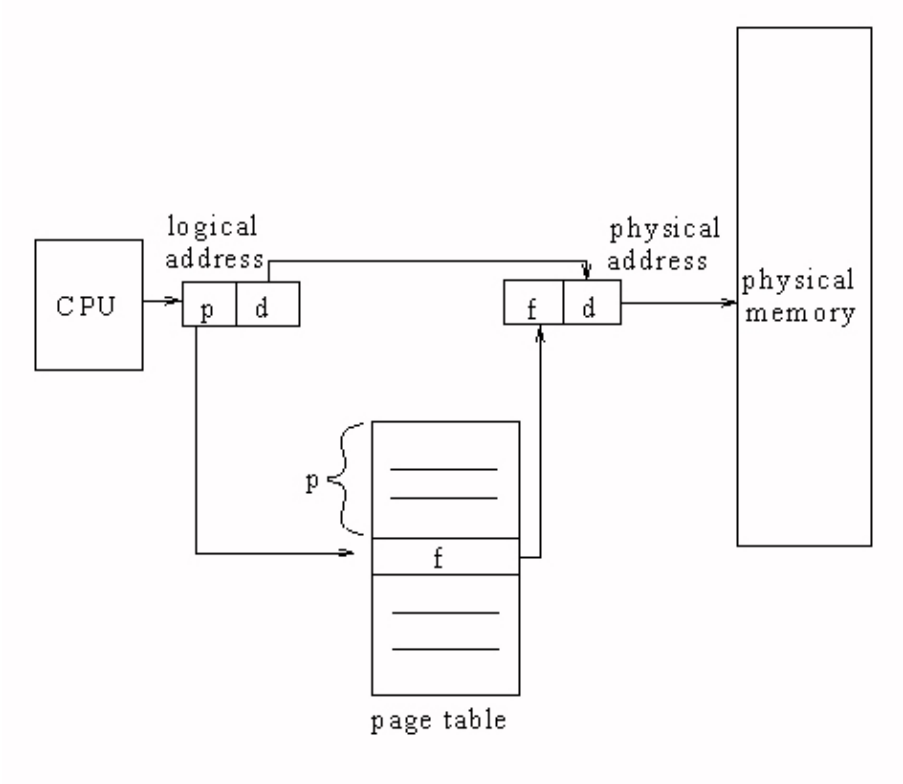
Por otra parte, supongamos que tenemos un segmento desocupado de 1000 bytes, y queremos poner un proceso de 998 bytes de tamaño, lo que dejaría un hoyo de 2 bytes. Difícilmente otro proceso podrá aprovechar esos dos bytes: es más caro mantener un segmento en la lista que "regalarle" esos dos bytes al proceso. La *fragmentación interna* se produce cuando un proceso tiene asignada más memoria de la que necesita.

Paginación

Esto de que los procesos tengan que usar un espacio contiguo de la memoria es un impedimento serio para optimizar el uso de la memoria. Idea: que las direcciones lógicas sean contiguas, pero que no necesariamente correspondan a direcciones físicas contiguas. O sea, dividir la memoria física en bloques de tamaño fijo, llamados **marcos** (*frames*), y dividir la memoria lógica (la que los procesos ven) en bloques del mismo tamaño llamados *páginas*.



Se usa una tabla de páginas para saber en que marco se encuentra cada página. Obviamente, se requiere apoyo del hardware. Cada vez que la CPU intenta accesar una dirección, la dirección se divide en número de página *p* y desplazamiento u *offset d*. El número de página se transforma en el marco correspondiente, antes de accesar físicamente la memoria.



El tamaño de las páginas es una potencia de 2, típicamente entre 0.5 y 8K. Si las direcciones son de *m* bits y el tamaño de página es 2^n, entonces los primeros *m-n* bits de cada dirección forman *p*, y los restantes *n* forman *d*. Este esquema es parecido a tener varios pares de registros *base* y *límite*, uno para cada marco (o sea, ligadura en tiempo de ejecución).

Cada proceso tiene su propia tabla: : cuando la CPU se concede a otro proceso, hay que cambiar la tabla de páginas a la del nuevo proceso. La paginación en general encarece los cambios de contexto. La seguridad sale gratis, ya que cada proceso sólo puede accesar las páginas que están en su tabla de páginas.

¿Fragmentación? Sólo interna, y de media página por proceso, en promedio. Esto sugeriría que conviene usar páginas chicas, pero eso aumentaría el sobrecosto de administrar las páginas. En general, el tamaño de página ha ido aumentando con el tamaño de la memoria física de una máquina típica.

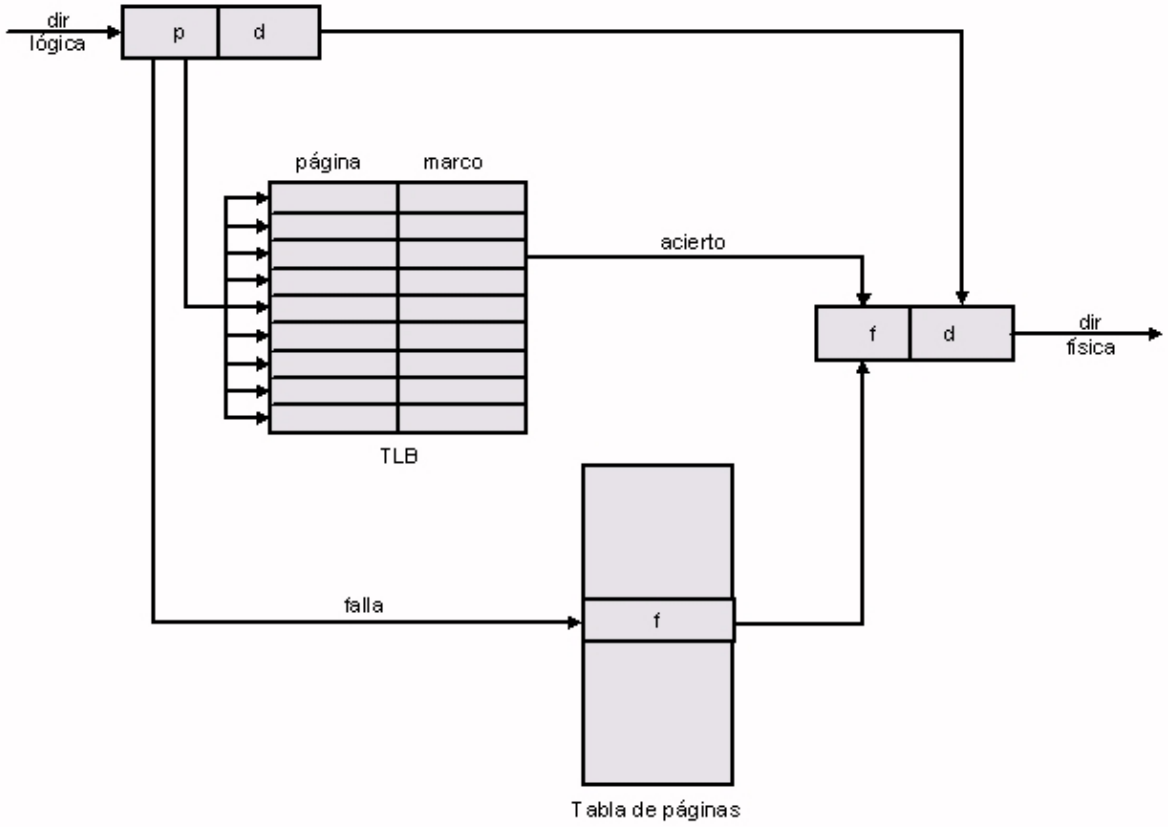
Tablas de páginas

Si las direcciones son de *m* bits y el tamaño de página es 2^n, la tabla de páginas puede llegar a contener 2^{m-n} entradas. En el PDP-11 *m=16* y *n=13*; una relación poco usual, pero basta con 8 entradas para la tabla de páginas, lo que permite tener 8 registros de CPU dedicados especialmente a tal propósito. En una CPU moderna, *m=32* (y ahora, incluso, 64), y *n=12* o 13, lo que significa que se requerirían cientos de miles de entradas: ya no es posible tener la tabla en registros.

Lo que se puede hacer es manejar la tabla de páginas de cada proceso completamente en memoria, y usar sólo un registro que apunte a la ubicación de la tabla. Para cambiar de un proceso a otro, sólo cambiamos ese registro. Ventaja: agregamos sólo un registro al cambio de contexto. Desventaja: costo

de cada acceso a memoria se duplica, porque primero hay que accesar la tabla (indexada por el número de página). O sea, si sin paginación cada acceso costaba 70ns, ¡ahora cuesta 140!

Solución típica (intermedia): usar un pequeño y rápido caché especial de memoria asociativa, llamado **translation look-aside buffer** (TLB). La memoria asociativa guarda pares (clave, valor), y cuando se le presenta la clave, busca simultáneamente en todos sus registros, y retorna, en pocos nanosegundos, el valor correspondiente. Obviamente, la memoria asociativa es cara; por eso, los TLBs rara vez contienen más de 64 registros. El TLB forma parte de la MMU, y contiene los pares (página, marco) de las páginas más recientemente accesadas. Cuando la CPU genera una dirección lógica a la página *p*, la MMU busca una entrada (*p*,*f*) en el TLB. Si está, se usa marco *f*, sin acudir a la memoria. Sólo si no hay una entrada para *p* la MMU debe accesar la tabla de páginas en memoria (e incorporar una entrada para *p* en el TLB, posiblemente eliminando otra).



Por pequeño que sea el TLB, la probabilidad de que la página esté en el TLB (**tasa de aciertos**) es alta, porque los programas suelen hacer muchas referencias a unas pocas páginas. . Si la tasa de aciertos es del 90% y un acceso al TLB cuesta 10ns, entonces, en promedio, cada acceso a memoria costará 87ns (¿por qué?). O sea, sólo un 24% más que un acceso sin paginación. Una Intel 80486 tiene 32 entradas en el TLB; Intel asegura una tasa de aciertos del 98%.

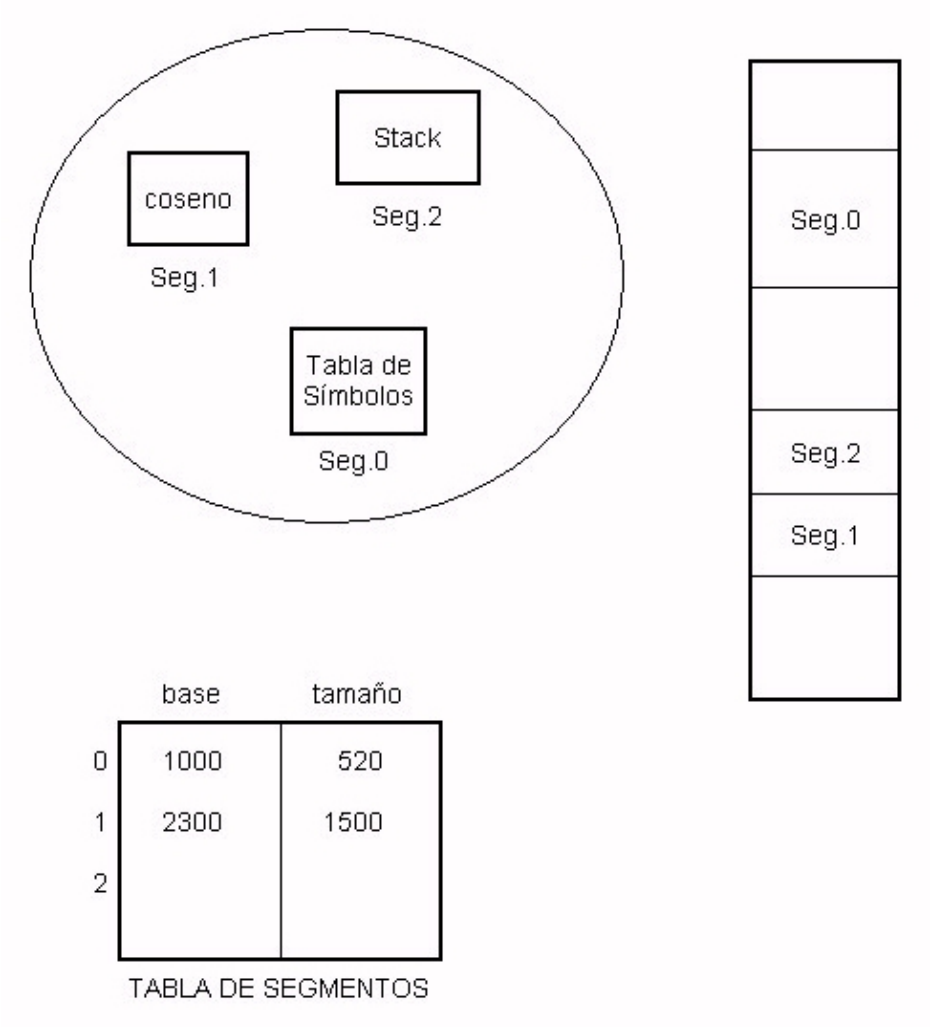
En cada cambio de contexto, hay que limpiar el TLB, lo que puede ser barato, pero hay que considerar un costo indirecto: si los cambios de contexto son muy frecuentes, la tasa de aciertos se puede reducir (¿por qué?).

Otra ventaja de paginación: permite que procesos compartan páginas. Por ejemplo, varios procesos ejecutando el mismo código: las primeras páginas lógicas apuntan a las mismas páginas físicas, que contienen el código. El resto, apunta a datos locales, propios de cada ejecución.

Segmentación

Cuando se usa paginación, el sistema divide la memoria para poder administrarla, no para facilitarle la vida al programador. La vista lógica que el programador tiene de la memoria no tiene nada que ver con la vista física que el sistema tiene de ella. El sistema ve un sólo gran arreglo dividido en páginas, pero el programador piensa en términos de un conjunto de subrutinas y estructuras de datos, a las cuales se refiere por su nombre: la función *coseno*, el *stack*, la *tabla de símbolos*, sin importar la ubicación en memoria, y si acaso una está antes o después que la otra.

La *segmentación* es una forma de administrar la memoria que permite que el usuario vea la memoria como una colección de segmentos, cada uno de los cuales tiene un nombre y un tamaño (que, además, puede variar dinámicamente). Las direcciones lógicas se especifican como un par (segmento, desplazamiento).



Implementación

Similar a paginación: en vez de tabla de páginas, tabla de segmentos; para cada segmento, hay que saber su tamaño y dónde comienza (base). Una dirección lógica *(s,d)*, se traduce a *base(s)+d*. Si *d* es mayor que el tamaño del segmento, entonces ERROR.

Ventajas:

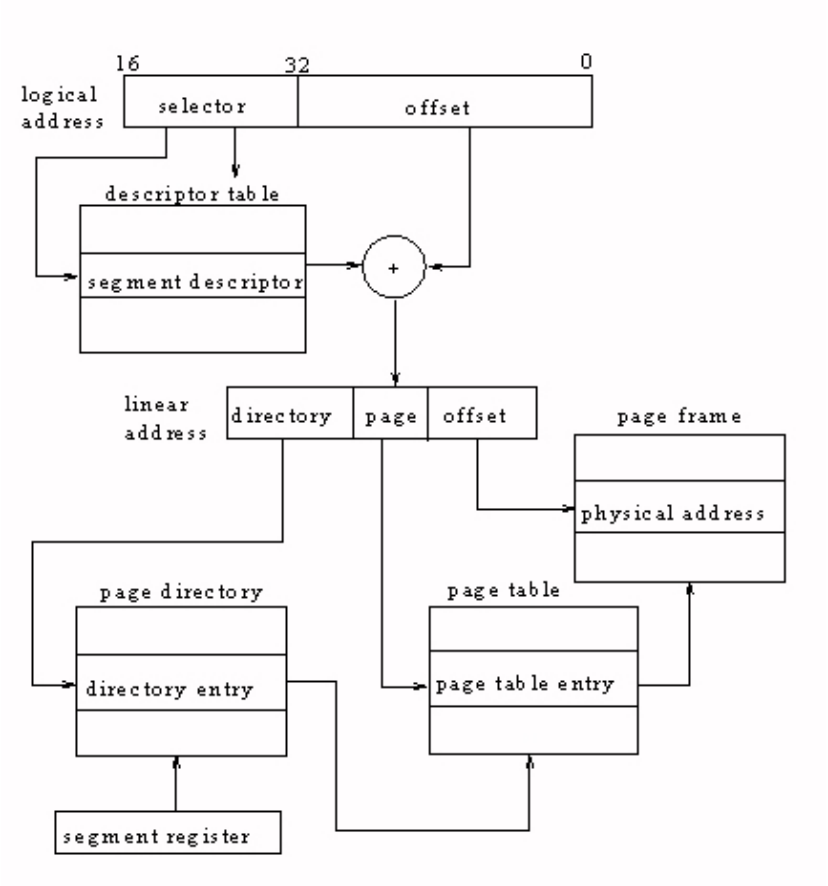
- Al usuario se le simplifica el manejo de estructuras de datos de tamaño dinámico.
- Se facilita el que los procesos compartan memoria.
- Los segmentos pueden estar protegidos según la semántica de su contenido. Por ejemplo, un segmento que contiene código, puede especificarse como sólo para ejecución (y nadie puede copiarlo ni sobreescribirlo); un arreglo puede especificarse como read/write but not execute. Esto facilita enormemente la detección de errores en el código.
- Librerías compartidas de enlace dinámico (DLLs).

Pero la memoria sigue siendo, físicamente, un sólo arreglo de bytes, que debe contener los segmentos de todos los procesos. A medida que se van creando y eliminando procesos, se va a ir produciendo, inevitablemente fragmentación externa.

Si consideramos cada proceso como un sólo gran segmento, tenemos el mismo caso que el de las particiones variables (ver Intercambio). Como cada proceso tiene varios segmentos, puede que el problema de la partición externa se reduzca, pues ya no necesitamos espacio contiguo para todo el proceso, sino que sólo para cada segmento. Para eliminar completamente el problema de la fragmentación interna, se puede usar una combinación de segmentación y paginación, en la que los segmentos se paginan.

Segmentación paginada en los 386

En una 80386 puede haber hasta 8K segmentos privados y 8K segmentos globales, compartidos con todos los otros procesos. Existe una *tabla de descripción local* (LDT) y una *tabla de descripción global* (GDT) con la información de cada grupo de segmentos. Cada entrada en esas tablas tiene 8 bytes con información detallada del segmento, incluyendo su tamaño y dirección. Una dirección lógica es un par (selector, desplazamiento), donde el desplazamiento es de 32 bits y el selector de 16, dividido en: 13 para el segmento, 1 para global/local, y 2 para manejar la protección. La CPU tiene un caché interno para 6 segmentos.



IIC2332: Sistemas Operativos

Ultima modificación: July 01, 1998, por Juan E. Navarro (jnavarro@ing.puc.cl)

Go backward to Administración de memoria
Go up to Top
Go forward to Sistemas de archivos

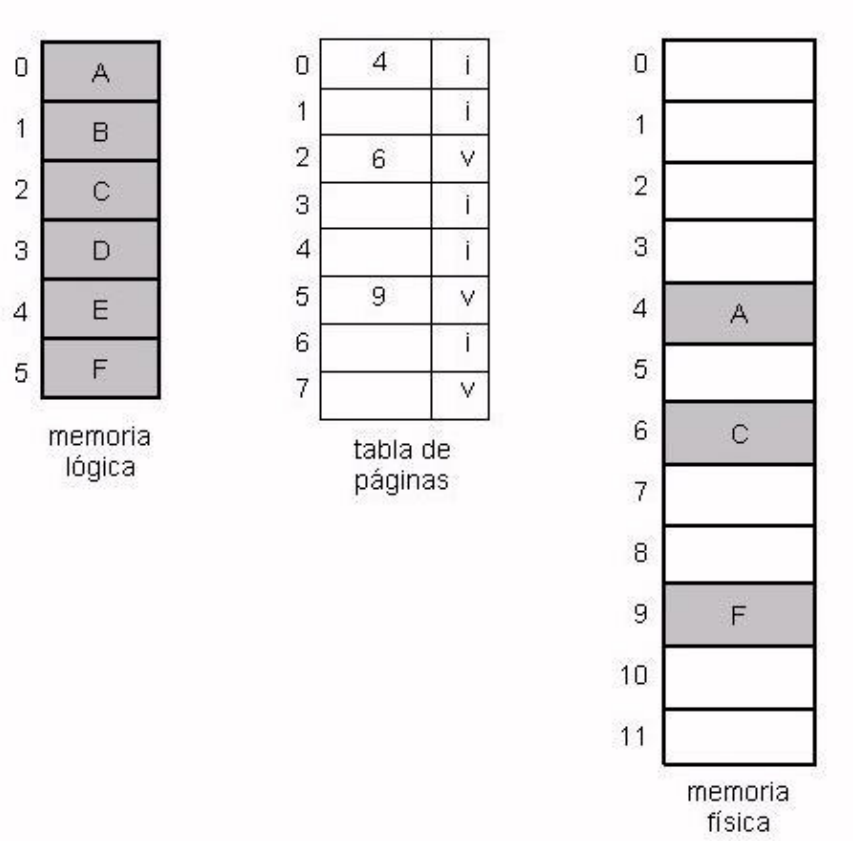
Memoria Virtual

¿Qué podemos hacer si un programa es demasiado grande para caber en la memoria disponible? Una posibilidad es usar **superposiciones** (*overlays*), como en MS-DOS: dividimos el programa en trozos independientes, y los vamos cargando de a uno, a medida que se necesiten. Por ejemplo, compilador de dos pasadas: cargar primero el código de la primera pasada, ejecutarlo, y después descartarlo para cargar el código de la segunda pasada. Las rutinas comunes y estructuras de datos compartidas entre las dos pasadas se mantienen en memoria permanentemente. El problema es que se agrega complejidad a la solución.

Mucho mejor sería poder extender la memoria de manera virtual, es decir, hacer que el proceso tenga la ilusión de que la memoria es mucho más grande que la memoria física (o que el trozo de memoria física que le corresponde, si tenemos multiprogramación). El sistema operativo se encarga de mantener en memoria física los trozos (páginas) que el proceso está usando, y el resto en disco. Ya que el disco es barato, podemos tener espacios de direccionamiento enormes.

Paginación por demanda

Memoria virtual se implementa con un poco de ayuda del hardware. Agregamos un bit a la tabla de páginas, que indica si la página en cuestión es válida o inválida. Válida significa que está en memoria (en el marco que indica la tabla), e inválida significa que no está.



El acceso a las páginas válidas funciona igual que antes, pero si la CPU intenta accesar una página inválida, se produce una **falta de página** que genera un *trap* al sistema operativo, quien debe encargarse de ir a buscar esa página al disco, ponerla en la memoria, y reanudar el proceso. O sea, en detalle, el sistema operativo debe:

- Bloquear al proceso.
- Ubicar un marco libre. Si no hay, tiene que liberar uno.
- Ordenar al controlador de disco que lea la página (de alguna manera, el sistema operativo debe saber dónde la guardó).
- Cuando se completa la lectura, modificar la tabla de páginas para reflejar que ahora la página si es válida.
- Pasar el proceso a la cola READY. Cuando le toque ejecutar, no notará ninguna diferencia entre esta falta de página y una simple expropiación de la CPU. En ambos casos, el efecto neto es que está un rato sin ejecutar.)

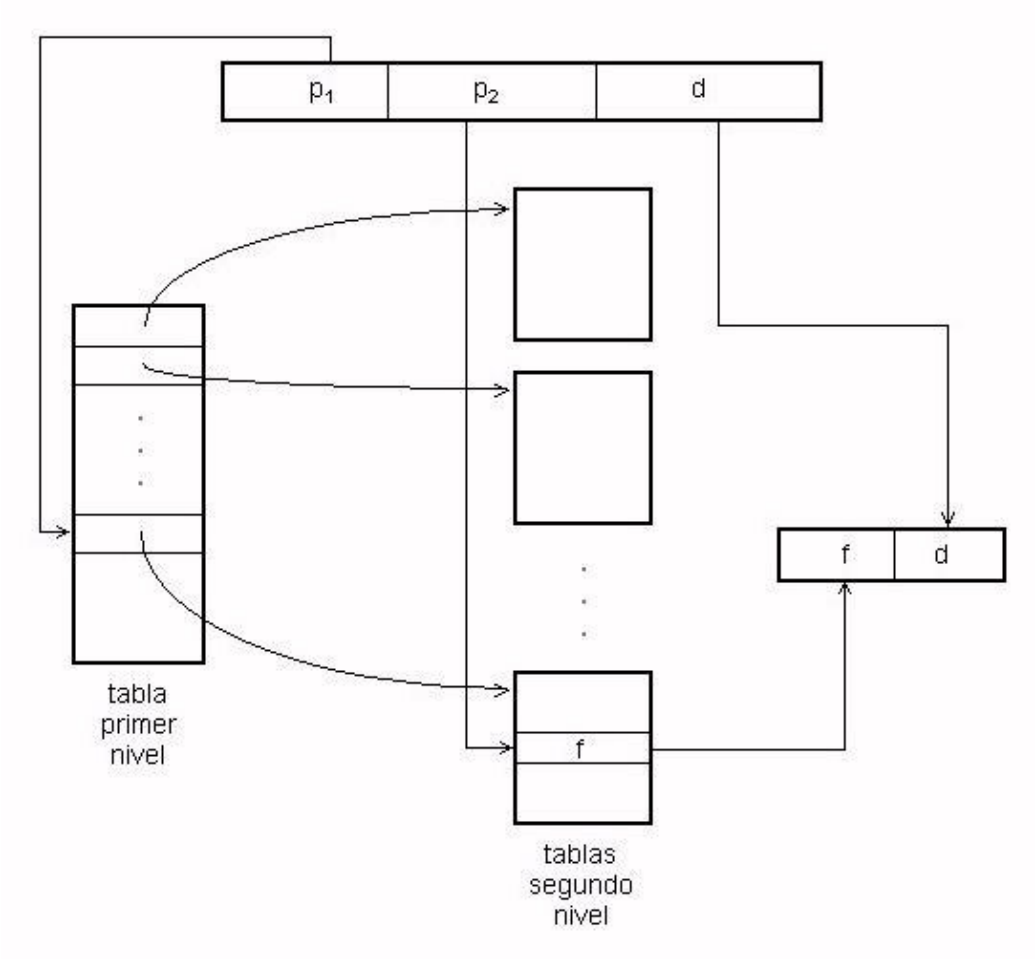
¿Qué hemos ganado?

- Ya no necesitamos superposiciones para ejecutar programas grandes.
- Sólo cargamos en memoria las partes del programa que se usan, reduciendo costos de lectura desde disco.
- Ahora que no es necesario tener el programa completo en memoria para poder ejecutarlo, podemos aumentar el nivel de multiprogramación.

Tablas de páginas

Ahora que la memoria lógica puede ser mucho más grande que la física, se acentúa el problema del tamaño de la tabla de páginas. Con direcciones de 32 bits, la memoria virtual puede alcanzar los 4GB. Con páginas de 4 KB, necesitaríamos hasta 1M entradas. Si cada entrada usa 32 bits, entonces la tabla podría pesar 4 MB. Además, cada proceso tiene su tabla, y esta tabla **debe** estar siempre en memoria física (al menos mientras ejecuta ¿por qué?). O sea, tenemos un problema serio.

Solución: tablas de varios niveles. El 386 usa dos niveles: una dirección lógica de 32 bits se divide en (p_1, p_2, d), de 10, 10 y 12 bits respectivamente. p_1 es un índice a una tabla de tablas, metatabla, o tabla de primer nivel, que contiene $2^{10}=1024$ entradas de 32 bits, o sea, exactamente una página. La entrada correspondiente a p_1 en esta tabla indica el marco donde se encuentra otra tabla (de segundo nivel), donde hay que buscar indexando por p_2, el marco donde está la página que finalmente se necesita.



¿Por qué se ha resuelto el problema? La clave es que sólo es indispensable mantener en memoria la tabla de primer nivel. Las otras , se tratan como páginas *ordinarias*, es decir, pueden estar en disco. Así, al utilizar la propia memoria virtual para estas páginas, se mantienen en memoria física sólo las más usadas. Sin embargo, se está agregando un acceso más a la memoria para convertir cada dirección lógica a una física, pero con un TLB con tasa de aciertos alta el impacto es mínimo.

Si las direcciones fueran de 64 bits, hay que usar más niveles. Otra posibilidad es usar **tablas de página invertidas**, en las que, en vez de haber una entrada por página virtual, hay una entrada por marco en la memoria física. Así, si tenemos 32 MB de memoria real y páginas de 4 KB, una tabla invertida necesita sólo 8K entradas (independientemente de si las direcciones son de 32 o 64 bits). Cada entrada dice a qué página virtual de qué proceso corresponde el marco. Cuando proceso *P* accesa dirección lógica *l*, hay que buscar un par *(P,l)* en esta tabla. La posición de este par es el marco que hay que accesar. Problema obvio: la búsqueda en una tabla de 8192 entradas puede costar muchos accesos a memoria. ¿Soluciones?

Reemplazo de páginas

Hasta ahora hemos soslayado un tema fundamental. Cuando un proceso accesa una página inválida, hay que ir a buscar su contenido a disco y ponerlo en algún marco. El punto es ¿en qué marco, si no hay ninguno libre? Una posibilidad en tal caso es pasar a disco un proceso, con lo cual los marcos que ocupaba quedan libres. Vamos a analizar este caso más adelante.

Lo usual es usar algún **algoritmo de reemplazo de páginas** para seleccionar un marco **víctima** (por ejemplo, al azar). Para poder usar el marco víctima, primero hay que sacar la página contenida en ese marco (pasarla a disco). Sin embargo, si la página original no había sido modificada, la copia en disco es idéntica, así que se puede descartar esta operación de escritura. El sistema operativo puede saber si una página ha sido modificada examinando un bit asociado a la página (el *dirty bit* o bit de modificación) que el hardware se encarga de poner en 1 cada vez que se escribe en ella.

O sea, cada falta de página produce una o dos transferencias de páginas entre la memoria y el disco. Para comparar algoritmos de reemplazo hay que contar cuántas faltas se producen dadas algunas secuencias de acceso a la memoria (¿cuáles?).

Algoritmo óptimo

El algoritmo que evidentemente minimiza el número de faltas consiste en escoger la página para la cual el lapso de tiempo hasta la próxima referencia es el mayor. Si tenemos tres marcos y las páginas son accesadas en la secuencia 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1, tendríamos 9 faltas. Lo "único" malo es que es imposible de implementar, pues para escoger una página hay que conocer cuáles serán los accesos futuros. En todo caso, sirve como referencia.

FIFO

Consiste en reemplazar la página más vieja (la que lleva más tiempo en memoria física). Es simple, pero no es bueno, ya que la página más antigua no necesariamente es la menos usada. En el ejemplo tendríamos 15 faltas. Además sufre de la **Anomalía de Belady**: las faltas pueden aumentar si aumentamos el número de marcos disponibles (probar 1,2,3,4,1,2,5,1,2,3,4,5 con 3 y 4 marcos).

La menos recientemente usada (LRU)

La política óptima usa conocimiento del futuro. Se puede aproximar este conocimiento usando el pasado reciente para predecir el futuro. Se selecciona la página menos recientemente usada con la esperanza de que, si no ha sido accesada en el pasado reciente, no lo será en el futuro cercano. Con el ejemplo, LRU produciría 12 faltas. Es una buena política en la medida que los procesos presenten localidad temporal de referencia.

El problema es que la implementación requiere un apoyo del hardware que es caro de implementar. Por ejemplo, con un contador y un campo adicional (de unos 64 bits) para cada página. Cada vez que se accesa una página, se incrementa el contador y se guarda en el campo de la página accesada. Hay otras posibilidades, pero todas caras, por lo que LRU en general no se usa; más bien se usan aproximaciones más baratas.

Aproximaciones a LRU

Sin soporte de hardware es poco lo que se puede hacer (habría que usar FIFO u otro algoritmo). Pero la mayoría de las máquinas proveen un **bit de referencia** por cada página cuya implementación en hardware es barata, y puede ayudar bastante si el software hace su parte. El sistema operativo pone en 0 el bit de referencia de cada página al inicializar un proceso, y el hardware lo pone en 1 cada vez que se accesa la página. Así, después de un rato, si examinamos los bits de referencia podemos saber qué páginas han sido usadas (aunque desconocemos el *orden* en que han sido usadas).

Más bits de referencia

Idea: mirar los bits de referencia a intervalos regulares, registralos, y resetearlos. Si los registramos en el bit de mayor orden de un byte, previo desplazamiento a la derecha, e interpretamos el byte como un entero sin signo, entonces la página cuyo byte sea el menor es la menos recientemente usada. Por supuesto, puede haber varias páginas con el menor byte; hay que escoger alguna (¿FIFO?) o también todas.

Segunda oportunidad

Es un caso especial del anterior, en el que se usa un *byte de un bit*. O sea, se usa FIFO, pero si la potencial víctima tiene el bit de referencia en 1, se pone en 0 y se le da una segunda oportunidad (se escoge otra). Posible implementación: con lista circular (por eso también se conoce como algoritmo del reloj).

Una optimización se basa en considerar que sale más caro reemplazar una página que ha sido modificada, pues hay que actualizar la copia en disco antes de reemplazarla. Según (bit de referencia, bit de modificación) hay 4 clases: 00, 01, 10, 11. Se escoge FIFO dentro de la clase (no vacía) más baja. (La clase 01 significa no referenciada, pero modificada: ¿pueden existir páginas en esta categoría?). Este algoritmo se usa en los MACs.

Fondos de marcos

Independientemente del algoritmo de reemplazo usado, se puede mantener un *pool* o fondo de marcos libres. Ante una falta de página, se escoge una víctima como siempre, pero la página requerida se pone en un marco del fondo si es que la víctima tiene el bit de modificación encendido. La víctima se puede escribir a disco y pasar al fondo después de reanudar el proceso, con lo que se reduce el tiempo de espera.

Además, si se produce una falta sobre una página que todavía está en el pool, entonces no es necesario ir a buscarla al disco. Las primeras partidas de la CPU VAX no mantenían correctamente el bit de referencia, de manera que el sistema operativo VMS usaba FIFO, pero gracias a un fondo de páginas se mejoraba el rendimiento.

También se pueden aprovechar los momentos en que el disco esté libre para escribir las páginas sucias

(¿qué se gana? ¿por cuáles páginas conviene comenzar?).

Hiperpaginación o *thrashing*

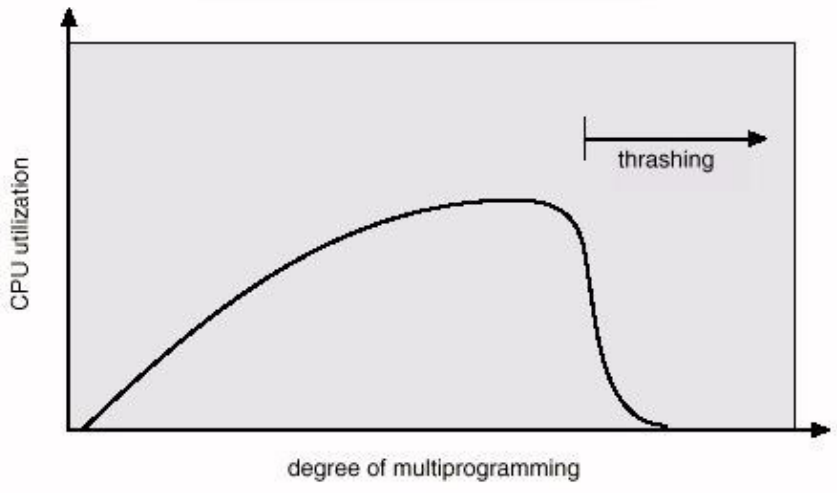
El conjunto de páginas que un proceso está usando se denomina el **conjunto de trabajo**.

Supongamos que un proceso está usando 4 páginas en forma intensiva, (por ejemplo, 1,2,3,4,1,2,3,4,1,2,3,4...). Si las 4 páginas que conforman su conjunto de trabajo están en memoria, entonces el proceso ejecutará fluidamente. Pero si sólo dispone de 3 marcos, cualquiera sea el algoritmo de reemplazo, sólo alcanzará a ejecutar unas pocas instrucciones cada vez, antes de producir una falta de página, o sea, hace **hiperpaginación**, y como consecuencia ¡el proceso andará un millón de veces más lento que con cuatro marcos!

Asignación de marcos

¿Cuántos marcos le asignamos a cada proceso? Debemos considerar que

- No podemos asignar más marcos que los que hay.
- Pasado un límite, si asignamos más marcos a un proceso no vamos a reducir significativametne el número de faltas de páginas (ejemplo del compilador de dos pasadas).
- Si *m* es el máximo número de marcos que requiere una instrucción de la máquina, no podemos asignar menos que *m*. Ejemplo: `ADD A,B` podría requerir 6 marcos.
- Si asignamos pocos, nos caben más procesos en memoria, pero corremos el riesgo de que los procesos pasen a régimen de hiperpaginación.

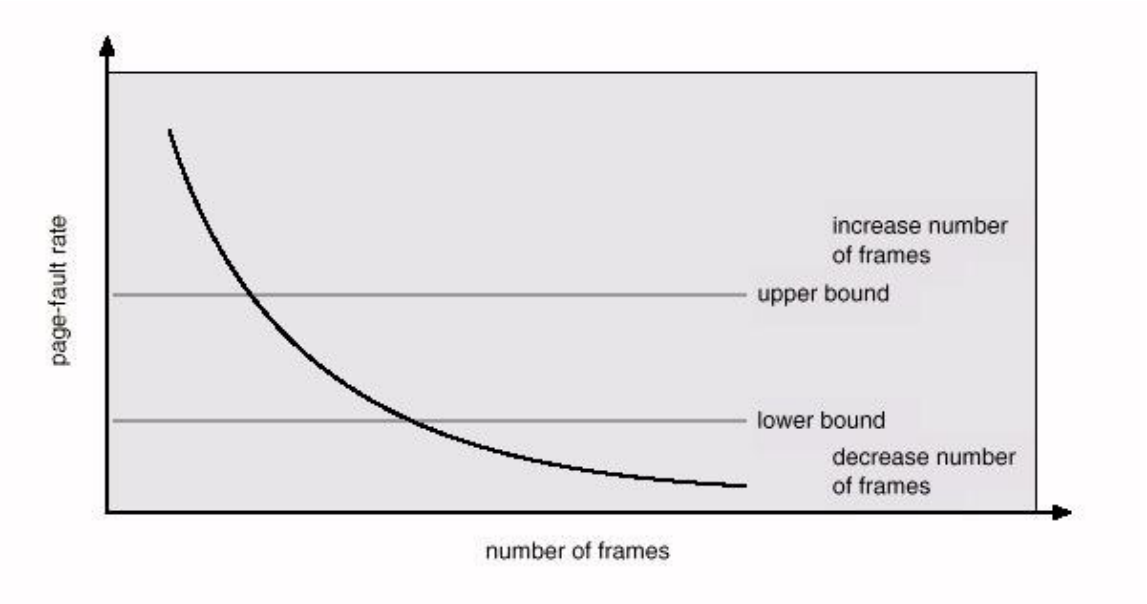


Si hay *N* procesos y *M* marcos, podemos asignar *M/N* marcos a cada uno, pero los procesos chicos podrían quedar con más marcos que los que necesitan. Mejor sería hacer una asignación proporcional al tamaño de los procesos. En cualquiera de los dos casos, hay que considerar la entrada y salida de procesos al sistema. Además, para mantener la proporcionalidad, el reemplazo de páginas debe ser local, es decir, si un proceso produce una falta de página, se escoge una víctima dentro de sus propios marcos.

Otra posibilidad es dejar que la asignación se vaya ajustando sola, por la vía de usar reemplazo global,

es decir, escoger una víctima entre todos los marcos, sin importar a que proceso esté actualmente asignado. En general, se obtienen mejores resultados, pues la asignación se adapta dinámicamente al tamaño de los conjuntos de trabajo.

Una manera más directa de controlar la hiperpaginación es monitoreando la tasa de faltas por segundo de cada proceso. Si supera un límite *MAX* se le asignan más marcos, y si baja de *MIN* se le quitan marcos. Si se llega a una situación en la que no es posible mantener a todos los procesos bajo *MAX*, entonces se pasan procesos a disco.



IIC2332: Sistemas Operativos

Ultima modificación: July 01, 1998, por Juan E. Navarro (jnavarro@ing.puc.cl)

Go backward to Memoria Virtual
Go up to Top
Go forward to Administración de dispositivos

Sistemas de archivos

Utilizando la memoria principal los procesos pueden almacenar y también compartir información. Sin embargo, la cantidad de información que se puede almacenar es limitada, aún con memoria virtual. Además, deja de existir cuando el proceso termina, y --peor aún-- se pierde si se corta la luz.

En síntesis, se requiere otra clase de almacenamiento, que cumpla los siguientes requisitos:

- Que permita guardar grandes volúmenes de información.
- Que la información sobreviva a los procesos que la usan.
- Que múltiples procesos puedan accesar la información de manera concurrente.

La solución usual es guardar la información en disco, organizándola lógicamente en unidades llamadas archivos. Los archivos son administrados por una parte del sistema operativo que se conoce como **sistema de archivos**.

Archivos

Desde el punto de vista del usuario, un archivo suele tener los siguientes atributos:

Nombre.
: Más que nada, para ayudar a los humanos. Las reglas para nombrar archivos varían de un sistema a otro (cuántos caracters, qué caracters, mayúsculas y minúsculas diferentes, extensión, etc.).

Tipo.
: El MacOS clasifica los archivos en distintos tipos (por ejemplo, text, pict, etc.).

Contenido.
: Para el SO, el contenido es una mera secuencia de bytes; el usuario lo interpreta a su manera, otorgándole alguna estrucura adicional.

Tamaño.
: Actual y máximo.

Fecha y hora.
: De creación, de la última modificación, del último acceso.

Usuarios.
: Identificador del creador y del dueño actual.

Atributos de protección.
: Quién tiene derecho a hacer qué con el archivo.

Otros atributos.
: Oculto, sistema, respaldado, etc.

Operaciones sobre archivos

El sistema operativo debe permitir operaciones sobre los archivos, mediante llamadas al sistema.

- Crear y eliminar.
- Leer y escribir (típicamente, a partir de la posición actual).
- Reposicionarse.
- Obtener y cambiar atributos, renombrar.

La información acerca de un archivo, exceptuando su contenido, se guarda en una entrada del **directorio**. Dicha entrada contiene un puntero a la ubicación en disco del archivo.

Una interfaz muy general para la llamada al sistema que lee o escribe en un archivo, podría ser una que especifique

1. Nombre del archivo.
2. Posición desde donde se va a escribir o leer.
3. Información a escribir (p.ej., puntero y tamaño) o buffer para recibir lo que se va a leer.

Sin embargo, ese esquema resulta ineficiente para el caso más que usual en que se hacen varias escrituras al mismo archivo, puesto que cada escritura implica una búsqueda en el directorio para conocer la ubicación del archivo. Por eso, lo típico es usar una tabla de archivos abiertos por proceso, con la información relevante de cada archivo abierto (como ubicación en disco y posición actual). Los procesos, antes de leer o escribir en un archivo, deben hacer una llamada para abrirlo. Esta llamada agrega una entrada en la tabla, y retorna un identificador (por ejemplo, el índice dentro de la tabla). De ahí en adelante, el proceso ya no se refiere al archivo por nombre, sino por el identificador, lo que evita nuevas búsquedas en el directorio. Además, por lo general se escribe a partir de la posición actual. Al terminar de usar un archivo, el programa debe cerrarlo.

En sistemas multiusuario, existe una tabla de archivos abiertos por proceso, y otra global. La tabla local contiene información propia del proceso (por ejemplo, la posición actual), e incluye un puntero a la entrada correspondiente en la tabla global. En ésta última se guarda información que no depende de los procesos, como la ubicación del archivo en disco.

Métodos de acceso

Acceso secuencial.

Es el método más simple y el más común. Los bytes se leen uno a uno, en orden. Las escrituras siempre se hacen agregando datos al final. Es suficiente en algunos casos (compiladores, editores). Suele ser el único método posible para archivos almacenados en cintas magnéticas.

Acceso directo, relativo o aleatorio.

Se puede accesar (leer o escribir) cualquier byte, en cualquier orden, como en un arreglo. Es esencial para aplicaciones como bases de datos.

Directorios

La información acerca de los archivos se almacena en **directorios**, que, en la mayoría de los sistemas operativos, se tratan también como archivos. Un directorio es una tabla que contiene una entrada por archivo. Una posibilidad es tener un sólo directorio, que contenga todos los archivos. Es simple, pero no muy útil para organizar muchos archivos de múltiples usuarios. Otra posibilidad es contar con un directorio para cada usuario, pero lo más flexible y lo más usado es una estructura en forma de árbol,

como en Unix y DOS. Existe un directorio raíz, que contiene archivos. Algunos de esos archivos son directorios, y así sucesivamente. Esto permite organizar los archivos de manera natural. Cada archivo tiene un nombre único, que se compone de los nombres de todos los directorios en el camino desde la raíz hasta el archivo.

Implementación de sistemas de archivos

Podemos modelar el disco duro como una secuencia de bloques, de 0 a N-1, de tamaño fijo (usualmente, de 512 bytes), a los cuales tenemos acceso directo, pero, por el movimiento del brazo es más caro leer dos bloques separados que dos bloques contiguos.

El primer problema consiste en determinar cómo a almacenar los archivos en el disco.

Asignación contigua

Este método requiere que cada archivo se almacene en un conjunto contiguo de bloques.

Ventajas.

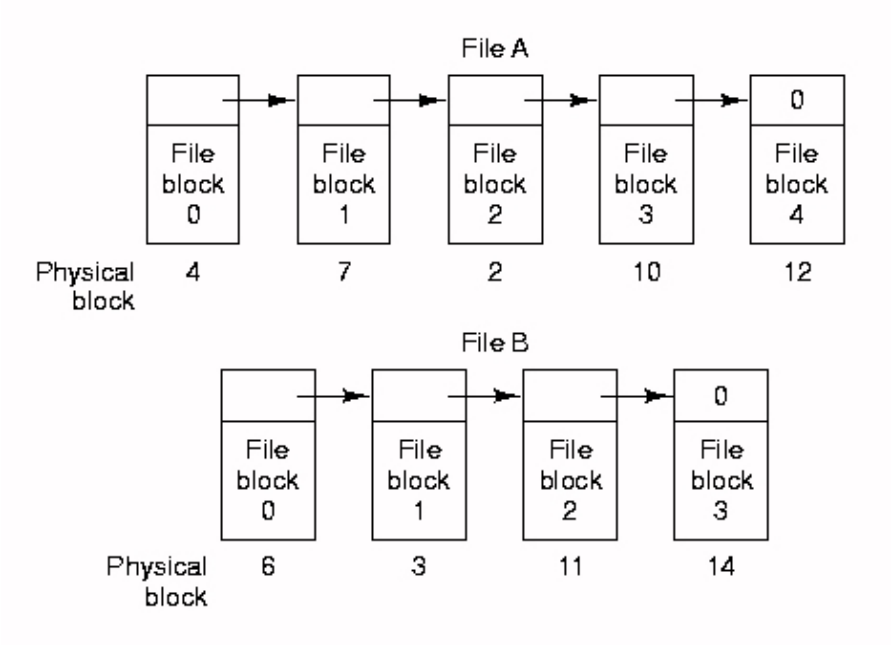
Por una parte es simple, porque para saber qué bloques ocupa un archivo, basta con registrar el primero. También es muy simple implementar acceso directo. Por otra parte, tiene muy buen rendimiento cuando los archivos se leen enteros, en forma secuencial, pues el movimiento del brazo del disco es el mínimo posible.

Desventajas.

¿qué hacer si un archivo crece? Se debería reservar espacio suficiente, pero no necesariamente se conoce de antemano el tamaño máximo. Además, la situación es análoga a la de asignación de particiones de memoria de tamaño variable, y por lo mismo, hay fragmentación externa. Podríamos pensar en compactación, pero, tratándose de disco, no es muy aconsejable hacerlo (salvo de noche, cuando el sistema no esté en uso).

Lista ligada

El segundo método para almacenar archivos consiste en mantener cada uno como una lista ligada de bloques. Los primeros bytes de cada bloque se usan como puntero al siguiente; el resto para los datos.



Ventajas.

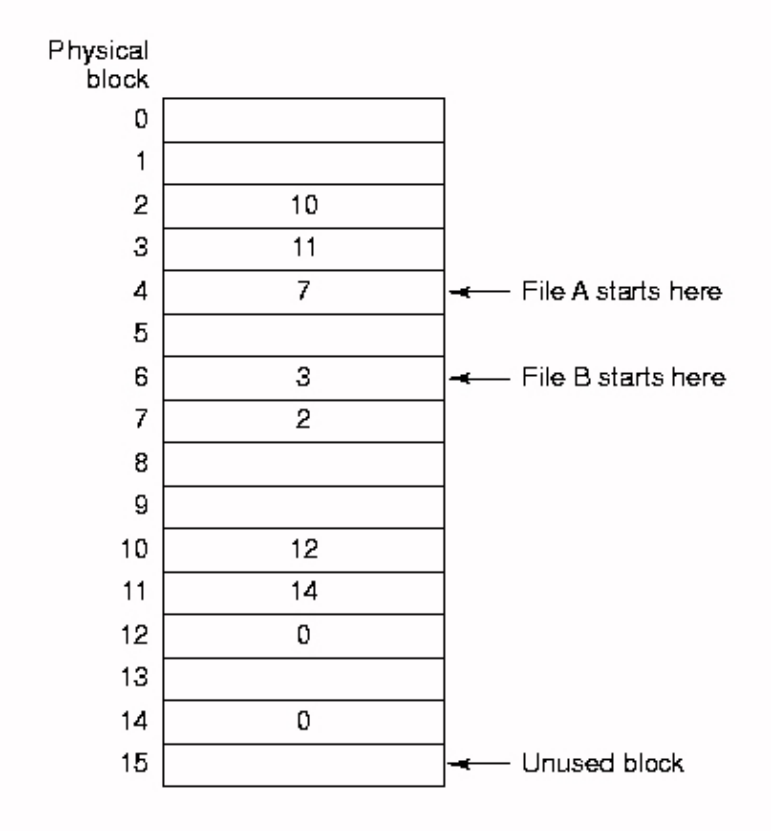
Se resuelven los problemas del método anterior: no hay fragmentación externa (aunque sí un poco de fragmentación interna). El directorio sólo debe registrar el primer bloque de cada archivo. No es necesario declarar el máximo tamaño que puede llegar a tener un archivo, puesto que puede crecer sin problemas (mientras queden bloques libres).

Desventajas.

Acceso secuencial es simple, pero no tan eficiente como con asignación contigua. Acceso aleatorio es extremadamente lento. Punteros ocupan espacio, y además hacen que la información útil de cada bloque deje de ser potencia de 2. Para reducir impacto en espacio, se pueden agrupar varios bloques físicos en uno sólo lógico, llamado unidad de asignación (o *cluster*).

Tabla de asignación de archivos (FAT)

Para eliminar algunas desventajas del método anterior, en vez de tener esparcidos los punteros en cada bloque, se pueden agrupar y poner todos juntos en una tabla o FAT. Es lo que hace MS-DOS, reservando un pedazo al comienzo del disco para la FAT.



Ventajas.

Mismas del anterios, y además: Si mantenemos la FAT en memoria, acceso aleatorio se agiliza enormemente. Información en cada bloque vuelve a ser potencia de 2.

Desventajas.

Se sigue perdiendo espacio para los punteros (ahora en forma de FAT). FAT puede ser grande (y hay que mantenerla en memoria). MS-DOS impone un límite de 2^{16} entradas en la FAT, lo que obliga a usar clusters demasiado grandes en discos de alta capacidad, (¡Para 1 GB, clusters de 16K!) lo que se traduce en una alta fragmentación interna.

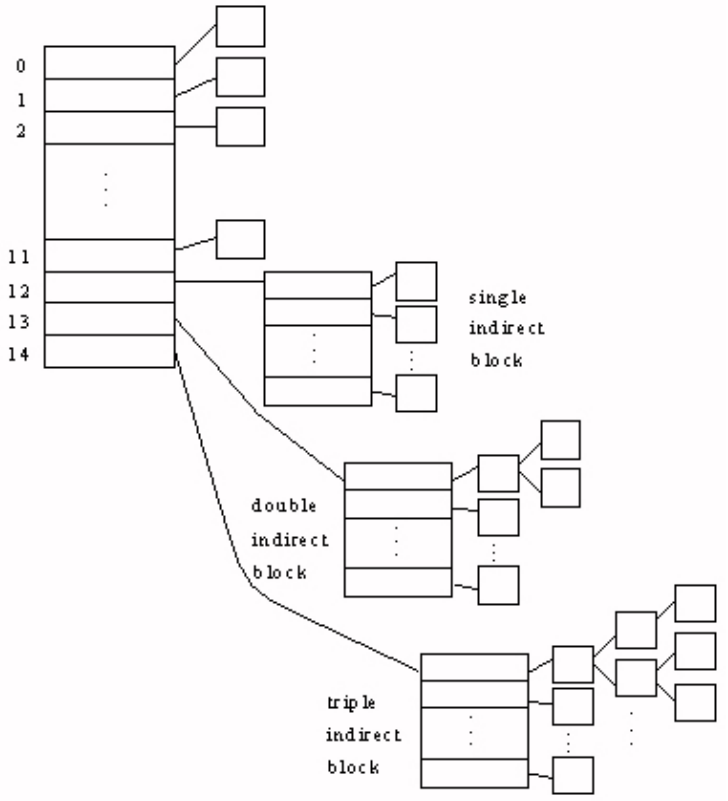
Nodos-I

El último método consiste en mantener jutos los punteros de cada archivo, en una tabla, asociada al archivo, y que se guarda en un bloque como cualquiera (esta tabla es análoga a la tabla de páginas). Además, en ese bloque se puede guardar información de los otros atributos del archivo. El directorio sólo contiene un puntero al bloque que contiene la tabla.

Problema: ¿y si el archivo es grande y su tabla no cabe en un solo bloque? Soluciones: lista ligada de bloques, índice multinivel (como en las tablas de páginas), o un esquema combinado, que es lo que se usa en Unix:

Supongamos que en un bloque, aparte del espacio para los atributos, queda espacio para 32 punteros a otros bloques. Entonces los 29 primeros se usan como se ha dicho, es decir, apuntan a bloques que

contienen datos del archivo. Pero los tres últimos apuntan a **bloques indirectos**. El antepenúltimo apunta a un **bloque indirecto simple**, es decir a un bloque que contiene punteros a bloques con datos. El penúltimo apunta a un **bloque indirecto doble**, es decir a un bloque que contiene punteros a bloques con punteros a bloques con datos. El último apunta a un **bloque indirecto triple**, es decir... adivinen. ¿Hasta qué tamaño de archivo soporta este sistema?



Ventajas.

Prácticamente lo único que hay que registrar en el directorio es un puntero al nodo-i. Acceso aleatorio rápido para archivos pequeños. Bajo sobrecosto fijo (no hay FAT, entradas en directorios pequeñas).

Desventajas.

Relativamente complejo. Mayor sobrecosto variable: cada archivo, por pequeño que sea, necesita al menos dos bloques. Acceso aleatorio más complicado para archivos grandes (en todo caso, los bloques con índices pueden guardarse en memoria para agilizar el acceso).

Cada uno de estos método tiene ventajas y desventajas. Por eso, algunos sistemas (como RDOS de Data General), ofrecen varios (se escoge uno al crear el archivo). Otra posibilidad es usar asignación contigua para archivos pequeños, e indexada para grandes. Lo usual es, en todo caso, uno sólo para todos los archivos.

En general, se usan métodos mucho más complejos que los que hemos visto. La enorme disparidad en velocidad entre la CPU y el disco justifica usar algoritmos complejos (incluso inteligencia artificial), que utilicen mucha CPU para ahorrar un par de movimientos del brazo.

Windows NT administra el contenido del disco a través de una base de datos relacional, y utiliza el modelo de transacciones para asegurarse que las modificaciones son atómicas y no se produzcan inconsistencias.

Administración del espacio libre

Es necesario saber qué bloques están libres. Las opciones son parecidas a las que se pueden usar para administrar espacio en memoria.

Mapa de bits.

Un bit por bloque. Es eficiente si se puede mantener el mapa entero en memoria. Disco de 1 GB, con bloques de 512 KB requiere un mapa de 256 KB. Usado en los Macs.

Lista ligada.

En un bloque reservado (fijo) del disco se registran las direcciones de los bloques desocupados. La última dirección apunta no a un bloque libre, sino a otro bloque con más direcciones de bloques libres...

En MS-DOS se usa la misma FAT para administrar el espacio libre.

Cachés de disco

Ya que el disco es tan lento comparado con la memoria (unas 10000 veces) resulta rentable usar un caché para mantener en memoria física parte de la información que hay en el disco, de manera que, si en el futuro se requiere un bloque que ya está en memoria, se ahorra el acceso al disco.

Igual que en el caso de memoria virtual, hay que tratar de adivinar qué bloques se van a accesar en el futuro cercano, para mantener esos bloques en el caché. Pero al contrario de lo que ocurre con memoria virtual, no se requiere ningún apoyo especial del hardware para implementar LRU, ya que todos los accesos a disco pasan por las manos del sistema operativo. Paradójicamente, LRU no es necesariamente la mejor alternativa tratándose de bloques de disco. ¿Qué pasa, por ejemplo, en el caso del acceso secuencial a un archivo? Por otra parte, algunos de los bloques contienen información crítica respecto del sistema de archivos (por ejemplo, un bloque que contiene información del directorio raíz o de un i-node o de los bloques libres). Si este bloque es modificado y puesto al final de la cola LRU, puede pasar un buen tiempo antes de que llegue a ser el menos recientemente usado, y sea escrito en el disco para ser reemplazado. Si el sistema se cae antes que eso, esa información crítica se perderá, y el sistema de archivos quedará en un estado inconsistente.

Se puede modificar un poco LRU, considerando dos factores:

- Qué tan probable es que el bloque se necesite de nuevo. Bloques de directorios se suelen usar bastante. El último bloque de un archivo que se está escribiendo, también es probable que se vuelva a necesitar.
- Qué tan esencial es el bloque para la consistencia del sistema de archivos. Básicamente todos los bloques, excepto los de datos, que han sido modificados. Estos deben grabarse en disco lo más rápidamente posible.

Según eso, ponemos cada bloque más adelante o más atrás en la lista LRU.

En Unix, hay una llamada al sistema, `sync`, para asegurarse que los bloques modificados se graben en el disco. Por eso, antes de sacar un disquet, se debe hacer `sync`. MS-DOS, en cambio, usa un caché *write-through*, es decir, cada vez que se escribe un bloque en el caché, este se graba inmediatamente en el disco. Por eso no hay que hacer nada especial antes de retirar un disquet, pero es menos eficiente.

IIC2332: Sistemas Operativos

Ultima modificación: July 01, 1998, por Juan E. Navarro (jnavarro@ing.puc.cl)

Go backward to Sistemas de archivos
Go up to Top
Go forward to Seguridad y protección

Administración de dispositivos

El código destinado a manejar dispositivos de entrada y salida representa una fracción significativa de un sistema operativo. Este capítulo trata este tema, centrándose en el caso de los discos.

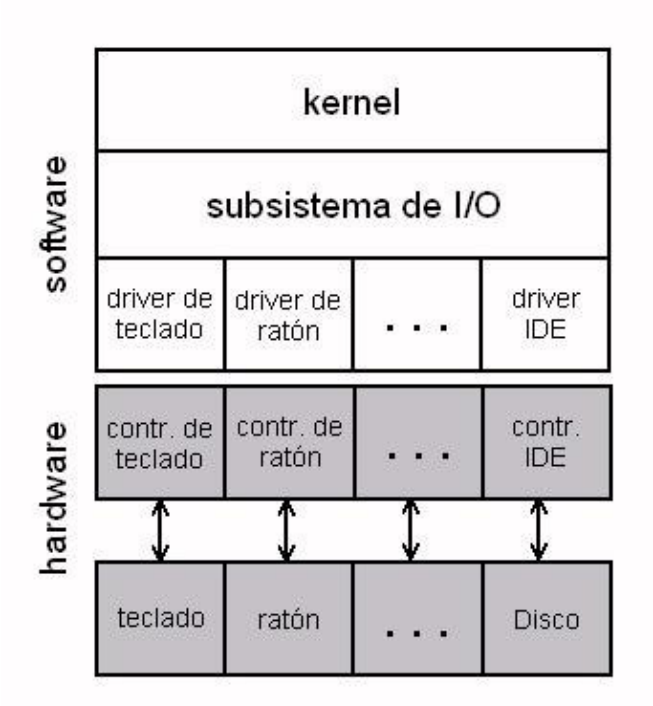
Dispositivos y controladores

Los dispositivos de I/O se pueden clasificar en dispositivos de almacenamiento, dispositivos de comunicación y dispostivos de interfaz con humanos. Ortogonalmente, se pueden clasificar como dispositivos de bloques o de caracteres.

Las unidades de I/O consisten típicamente de una parte mecánica y una parte electrónica, conocida como controlador de dispositivo. El sistema operativo sólo se comunica con el controlador. La parte del sistema operativo encargada de ello es el manejador o *driver* del dispositivo. La interfaz entre el controlador y la parte mecánica es de muy bajo nivel.

I/O estructurado en capas

El I/O se puede estructurar en forma eficiente y modular usando capas. Cada capa se encarga de una tarea específica, y ofrece una interfaz bien definida a la capa superior.



El subsistema de I/O es independiente del dispositivo. Los manejadores de dispositivos contienen todo el código que es dependiente del dispositivo; cada dispositivo diferente requiere un driver. La tarea del driver es la de aceptar órdenes abstractas (independientes del dispositivo) de la capa superior (como por ejemplo, leer el bloque *n*), traducirla a órdenes concretas (dependientes del dispositivo) y dar esas órdenes al controlador correspondiente.

Las tareas del subsistema de I/O son las siguientes.

Planificación.
Ejecutar las solicitudes de I/O en el mismo orden en que el sistema operativo las recibe de los procesos, no siempre es lo más eficiente. Muchas veces conviene usar otros criterios para escoger la siguiente a solicitud a ejecutar, de entre las solicitudes pendientes para un determinado dispositivo.

Buffering.
Un buffer es un espacio de memoria reservado para almacenamiento temporal de datos, para acomodar disparidades enter las velocidades de un dispositivo productor de datos y un dispositivo consumidor, o diferencias en tamaños de bloques.

Caching.
Un caché es una región de la memoria que contiene copias de datos que residen en dispositivos más lentos. Cuando se requiere un dato que está en el caché, el acceso es más rápido, puesto que no es necesario accesar el dispositivo lento.

Spooling.
Un spool es un buffer que contiene la salida para un dispositivo de caracteres, tal como una impresora, en el cual no se pueden mezclar las salidas de varios procesos. Mediante spooling, los procesos tienen la ilusión de estar imprimiendo simultáneamente, pero en realidad el sistema operativo está almacenando la salida de cada proceso para imprimirla de una sola vez cuando el proceso termine.

Manejo de errores.
Parte del manejo de errores la realiza el subsistema de I/O. El manejo de errores se hace principalmente en el driver, pues la mayoría de los errores son dependientes del dispositivo (y por ende, sólo el driver podría saber qué hacer, sobre todo si el error es transitorio). Por ejemplo, si el driver recibe un mensaje de error del controlador al intentar leer un bloque, reintentará varias veces antes de reportar el error a la capa superior. Entonces el subsistema de I/O decide qué hacer, en una forma independiente del dispositivo.

Planificación de disco

Un disco, mirado desde más bajo nivel, no es simplemente una secuencia de bloques. Está compuesto de **platos**, cada uno de los cuales contiene una serie de **pistas** o *tracks* concéntricos. A su vez, las pistas se dividen en sectores. Las pistas exteriores, que son más grandes, pueden contener más sectores que las interiores. (En un CD, en realidad hay una espiral de sectores.) Existe un brazo mecánico con un cabezal lector/escritor para cada plato. El brazo mueve todos los cabezales juntos. Un **cilindro** se conforma por las pistas que los cabezales pueden leer cuando el brazo está en una posición determinada.

Los bloques lógicos (secuenciales) que ve el sistema de archivos deben traducirse a un trío (cilindro, plato, sector). El tiempo requerido para leer un sector depende de:

1. El tiempo de búsqueda (*seek time*), es decir, el tiempo requerido para mover el brazo al cilindro apropiado.
2. El retardo rotacional, o sea, el tiempo que hay que esperar hasta que el sector requerido pase por

debajo del cabezal.
3. El tiempo de transferencia de los datos.

El primero es el que predomina, de manera que conviene reducirlo para aumentar la eficiencia del sistema. El sistema de archivo puede ayudar (por ejemplo, con asignación contigua). Obviamente, bloques en el mismo cilindro deben considerarse contiguos. Pero hay otra cosa que se puede hacer, considerando que en un sistema con muchos procesos la cola de solicitudes pendientes de un dispositivo suele no estar vacía: atenderlas en un orden que reduzca los movimientos del brazo.

Algoritmos de planificación de disco

FIFO.

Es simple, pero no estamos haciendo nada por la eficiencia. Es malo si las solicitudes se alternan entre cilindros exteriores e interiores. Por ejemplo, si, mientras se lee el cilindro 11 llegan solicitudes para los cilindros 1,36,16,34,9,12, y se atienden en ese orden, el brazo recorrerá 111 cilindros.

SSTF (shortest seek-time first).

Se trata de atender primero las solicitudes más cercanas a la posición actual del brazo. La atención sería en el orden 11,12,9,16,1,34,36, para un total de 61 cilindros de desplazamiento. El problema es que, cuando hay muchas solicitudes, es posible que sólo se atiendan las cercanas al centro. Puede haber inanición para los procesos que solicitan cilindros de los extremos.

Algoritmo del ascensor.

Para evitar inanición, se mantiene la dirección de movimiento del brazo hasta que no queden solicitudes pendientes en esa dirección. Es lo mismo que hacen los ascensores. En el ejemplo, suponiendo que el brazo iba hacia las direcciones altas, las solicitudes se atenderían en el orden 11,12,16,34,36,9,1, lo que da un total de 60 cilindros de recorrido del brazo. O sea, en este caso en particular es un poco mejor que SSTF, pero en general es peor. Una propiedad interesante es que para cualquier conjunto de solicitudes, el movimiento del brazo está acotado: 2 veces el ancho del disco. Un pequeño problema es que las solicitudes en los extremos tienen, en promedio, un tiempo de espera mayor. Esto se puede resolver si las solicitudes siempre se atienden en un solo sentido. En el otro sentido, el cabezal se devuelve, pero sin atender solicitudes a su paso.

También podríamos pensar en un algoritmo óptimo, pero su complejidad no justifica usarlo. Si la carga es muy poca (la cola nunca tiene más de una solicitud pendiente) todos los algoritmos tienen el mismo rendimiento. Para cargas pesadas, se usa el del ascensor.

Discos RAM

Gracias a la estructuración en capas, podemos usar el mismo sistema de archivos en cualquier dispositivo de bloques con un driver adecuado, que implemente la interfaz para el software independiente del dispositivo. Por ejemplo, en los primeros computadores personales, que tenían sólo una disquetera como medio de almacenamiento, era habitual crear un **disco RAM**, es decir reservar un trozo de la memoria para usarlo como un disco virtual, para almacenar archivos.

Un driver de disco RAM es extremadamente simple. Dado un tamaño de bloque *B*, leer o escribir el bloque *i* es simplemente accesar *B* bytes a partir de la posición *B**i del área reservada para el disco

RAM.

Bloques dañados

Los discos, en cuanto dispositivos mecánicos, son propensos a fallas. A veces la falla es transitoria: el controlador no puede leer un sector debido a que se interpuso una partícula de polvo entre el cabezal y la superficie del disco. El controlador siempre reintenta varias veces una operación que fracasa por si la falla es transitoria; muchas veces se resuelve, sin que el driver siquiera se entere. En los casos en que el sector está permanentemente dañado, el error se informa al driver, y el driver informa al sistema de archivos, quien registra el bloque como *dañado*, para no volver a usarlo. ¿Cómo se pueden registrar los bloques dañados? Igual hay bloques críticos: en todo sistema de archivo, debe haber al menos un bloque en una dirección fija. Si ese bloque se daña, el disco entero se hace inusable.

Algunos controladores inteligentes reservan de antemano algunas pistas, que no son visibles para el driver. Cuando se daña un sector, el propio controlador lo reemplaza por uno de los reservados. (en forma transparente, si la operación era de escritura, pero no tan transparente si era de lectura). Muchos discos vienen con sectores dañados ya marcados desde la fábrica. Pero ¿dónde se guarda la información de los bloques malos? Así, si el bloque 5 se daña, entonces el controlador usa, digamos, el 999 cada vez que el driver le solicita el 5. Pero ¿que pasaría entonces con los algoritmos de scheduling de disco? Un esquema que a veces se usa para no perjudicarlos, es que el controlador reserva bloques esparcidos en el disco, y cuando se daña un sector, trata de sustituirlo por uno de los de reserva que se encuentre en el mismo cilindro, o por lo menos cerca.

Arreglos de discos

Se puede decir que los discos son la componente menos confiable de un computador, la componente más complicada de sustituir, y la que frena el mejoramiento de la velocidad de procesamiento con los avances tecnológicos. En efecto, la velocidad de los procesadores se duplica más o menos cada 2 años, y la capacidad de los chips de memoria crece a un ritmo parecido. No obstante, el ancho de banda (velocidad de transferencia) del I/O ha variado muy poco.

A este ritmo, en 7 años más los procesadores van a ser 10 veces más rápidos, pero en general las aplicaciones correrán menos de 5 veces más rápido, por las limitaciones de I/O.

Una solución posible: en lugar de uno solo disco grande, usar muchos discos chicos y baratos, en paralelo, para mejorar el ancho de banda. Para garantizar paralelismo, se hace *disk striping* o división en franjas. Cada bloque lógico se compone de varios sectores físicos, cada uno en un disco distinto. Así, cada acceso a un bloque lógico se divide en accesos simultáneos a los discos. En 1991 la situación era la siguiente:

- IBM 3380: 7500 MB, 18 U$/MB, 30000 horas de MTTF (mean time to failure)
- Conner CP3100: 100 MB, 10 U$/MB, 30000 horas de MTTF

El IBM 3380 tiene bastante más ancho de banda que un CP3100, pero si juntamos 75 de estos últimos tenemos la misma capacidad total, con menor costo, menor consumo de electricidad, y potencialmente 12 veces más ancho de banda. El gran problema es la confiabilidad: si antes teníamos 30000 horas de funcionamiento sin fallas, ahora tendríamos 400 (30000/75) horas, o sea, sólo dos semanas. O sea, la tolerancia a fallas es crucial, y para obtenerla se usa redundancia, en una configuración conocida como RAID (Redundant Array of Inexpensive Disks), y que se puede implementar en varios niveles.

RAID 1.

Se usan **discos espejos**, o sea, la información de cada disco se mantiene siempre duplicada en otro idéntico. O sea, MTTF aumenta notoriamente, pero duplicando el costo.

RAID 2.

Se reduce la redundancia usando técnicas de detección y corrección de errores (códigos de Hamming). Por ejemplo, si un bloque se reparte entre 10 discos y suponemos que no va a haber más de una falla simultáneamente, entonces no necesitamos duplicar el bloque entero para reconstituirlo en caso de falla, puesto que ante una falla sólo se perderá un 10% de la información. El problema es que si no sabemos qué 10% se perdió, de todas maneras se necesita bastante redundancia (20 a 40%).

RAID 3.

El punto es que, gracias a que cada controlador usa sumas de chequeo (y suponiendo que además podemos saber cuándo un controlador falla) sí podemos saber qué trozo de la información está errónea. Y sabiendo eso, basta con usar sólo un disco adicional para guardar información de paridad con la cual es posible reconstituir la información original.

Hay otros niveles (RAID 4 y 5). Ahora (1996) la situación es:

- IBM 3390: un disco de 102 GB, 3.9 MB/s, 22.8 ms de latencia.
- IBM RAMDAC 2: 64 discos, 180 GB en total, 12.6 MB/s, 4.2 ms de latencia.

La ganancia en ancho de banda es menor que la teórica, entre otras cosas porque la tolerancia a fallas impone un overhead. Por otra parte, con un RAID de 100 discos para datos y otros 10 para paridad, el MTDL (mean time to data loss) es de 90 años, comparado con 3 años de los discos estándares.

IIC2332: Sistemas Operativos

Ultima modificación: July 01, 1998, por Juan E. Navarro (jnavarro@ing.puc.cl)

Go backward to Administración de dispositivos
Go up to Top
Go forward to Sistemas distribuidos

Seguridad y protección

Vamos a hacer una distinción entre **seguridad** y **protección**. El problema de la seguridad consiste en lograr que los recursos de un sistema sean, bajo toda circunstancia, utilizados para los fines previstos. Para eso se utilizan mecanismos de protección.

Los sistemas operativos proveen algunos mecanismos de protección para poder implementar políticas de seguridad. Las políticas definen qué hay que hacer (qué datos y recursos deben protegerse de quién; es un problema de administración), y los mecanismos determinan cómo hay que hacerlo. Esta separación es importante en términos de flexibilidad, puesto que las políticas pueden variar en el tiempo y de una organización a otra. Los mismos mecanismos, si son flexibles, pueden usarse para implementar distintas políticas.

Los mecanismos que ofrece el sistema operativo necesariamente deben complementarse con otros de carácter externo. Por ejemplo, impedir el acceso físico de personas no autorizadas a los sistemas es un mecanismo de protección cuya implementación no tiene nada que ver con el sistema operativo.

Un aspecto importante de la seguridad es el de impedir la pérdida de información, la cual puede producirse por diversas causas: fenómenos naturales, guerras, errores de hardware o de software, o errores humanos. La solución es una sola: mantener la información respaldada, de preferencia en un lugar lejano.

Otro aspecto importante de la seguridad, es el que tiene que ver con el uso no autorizado de los recursos:

- Lectura de datos.
- Modificación de datos.
- Destrucción de datos.
- Uso de recursos: ciclos de CPU, impresora, almacenamiento.

Aquí el sistema operativo juega un rol fundamental, ofreciendo mecanismos de autorización y autentificación.

Protección absoluta contra uso malicioso de los sistemas es imposible, pero si los costos de violar un sistema son superiores a los potenciales beneficios que se pueden obtener, entonces el sistema puede considerarse seguro. El problema es que esa protección no obstaculice el uso del sistema por parte de usuarios autorizados. Demasiada seguridad podría ser contraproducente si es muy engorrosa para los usuarios, pues estos tenderán a eludir los procedimientos para facilitarse la vida.

Casos famosos

Algunos de los Titanics y Hindenburgs de la seguridad en computadores son los siguientes:

En Unix `lpr -r archivo` imprime `archivo` y después lo elimina. En versiones antiguas de Unix se podía hacer `lpr -r /etc/passwd`, lo que terminaba con la eliminación del archivo donde se registran los usuarios.

El comando `mkdir xx` era un programa que ejecutaba en modo superusuario, creando un nodo-i para el directorio `xx`, y luego cambiando el dueño de `xx` de *root* al del usuario. Con un sistema lento y un poco de suerte, se podía modificar el nodo-i para que apuntara a cualquier archivo (por ejemplo, el `passwd`), justo antes de que `mkdir` fijara el nuevo dueño.

Otro ejemplo ilustrativo de lo fácil que es pensar que un sistema es seguro, cuando en realidad no lo es, es el del sistema operativo TENEX, usado en los DEC-10 de Digital. TENEX usaba memoria virtual, y para permitir al usuario monitorear el comportamiento de sus programas, éste podía especificar una rutina que el sistema ejecutaría cada vez que hay una falta de página. Al mismo tiempo, los archivos en TENEX estaban protegidos por una clave: cada vez que se abría un archivo, debía especificarse su clave. El sistema operativo chequeaba las claves de un caracter a la vez, deteniéndose apenas un caracter difiriera. Gracias a eso y a que era posible saber cuándo había una falta de página, se podía descubrir la clave de cualquier archivo, de la siguiente manera:

Poner primer caracter de posible clave en la última posición de una página *p*, y arreglárselas para que la siguiente (la *p+1*) estuviera inválida (no presente en memoria física). Tratar de abrir el archivo, usando esa posible clave. El resultado esperado es un mensaje diciendo "clave incorrecta", pero dependiendo de si hubo falta de página o no en *p+1*, podemos saber si el primer caracter era correcto o no. Después de unas pocas pruebas, se habrá descubierto el primer caracter, y es cosa de seguir la misma idea para descubrir el resto. Si hay 128 caracteres posibles, necesitaremos $128n$ intentos para descubrir una clave de n caracteres, en lugar de 128^n

Probablemente la violación más famosa de todos los tiempos ocurrió en 1988 cuando un estudiante lanzó un gusano por la internet que botó miles de máquinas en cosa de horas. El gusano tomaba el control de una máquina intentando diversos mecanismos. Uno de ellos era un bugo en el programa `finger`. Una vez obtenido el control, trataba de descubrir las claves de los usuarios de esa máquina intentando palabras comunes. Si descubría una, entonces tenía acceso a todas las máquinas en que ese usuario tuviera cuenta. El gusano no hacía ninguna acción dañina en sí, pero usaba tantos recursos de las máquinas infectadas que las botaba.

Aunque estos defectos han sido corregido, todavía hay más. En Unix, algunos nunca serán corregidos (si se corrigen, dejaría de ser Unix).

Otras amenazas y ataques posibles

- **Virus.** Un virus es parecido a un gusano, en cuanto se reproduce, pero la diferencia es que no es un programa por sí sólo, si no que es un trozo de código que se adosa a un programa legítimo, contaminándolo. Cuando un programa contaminado se ejecuta, ejecutará también el código del virus, lo que permitirá nuevas reproducciones, además de alguna acción (desde un simple mensaje inocuo hasta la destrucción de todos los archivos).
- **Cabayo de troya.** Un cabayo de troya es un programa aparentemente útil que contiene un trozo de código que hace algo no deseado.
- **Puerta trasera.** Una puerta trasera es un punto de entrada secreto, dejado por los implementadores del sistema para saltarse los procedimientos normales de seguridad. La puerta trasera puede haberse dejado con fines maliciosos o como parte del diseño; en cualquier caso, son un riesgo.

- **Caza claves.** Dejar corriendo en un terminal un programa que pida "login:" y luego "password:", para engañar a los usuarios de modo que estos revelen su clave.
- Solicitar recursos como páginas de memoria o bloques de disco, y ver qué información contienen; muchos sistemas no los borran cuando se liberan, de modo que se puede encontrar información "interesante".
- Sobornar o torturar al administrador para que suelte la clave.

Principios básicos para la seguridad

- Suponer que el diseño del sistema es público.
- El defecto debe ser: sin acceso.
- Chequear permanentemente.
- Los mecanismos de protección deben ser simples, uniformes y construidos en las capas más básicas del sistema.
- Los mecanismos deben ser aceptados sicológicamente por los usuarios.

En cualquier caso, hay que tener presente que:

Seguridad = 1/Conveniencia

En otras palabras, mientras más seguro es tu sistema, más desdichado serás.

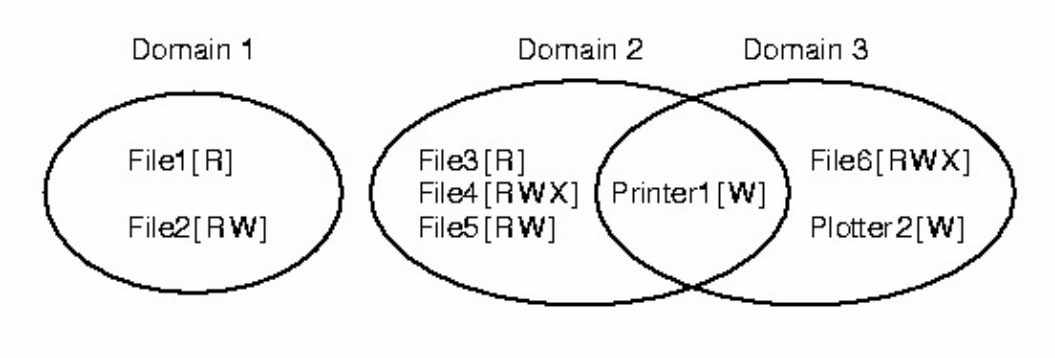
Mecanismos de autorización

Un sistema de computación puede verse como una colección de objetos (procesos, procesadores, segmentos de memoria, discos, impresoras, archivos, semáforos). Cada objeto debe tener un nombre único para poder identificarlo, y un número finito de operaciones que los procesos pueden efectuar sobre él (leer y escribir en archivos, P y V en semáforos). Podemos ver a estos objetos como tipos abstractos de datos.

Obviamente, un proceso no debe poder accesar objetos sobre los que no tenga autorización. También debe ser posible restringir el uso de un objeto por parte de un proceso sólo a ciertas operaciones. Por ejemplo, un proceso podría tener autorización para leer, pero no para escribir un determinado archivo.

Dominios de protección

Un **dominio de protección** es un conjunto de pares (objeto, operaciones); cada par identifica un objeto y las operaciones permitidas sobre él.



En cada instante, cada proceso ejecuta dentro de un dominio de protección. Los procesos pueden cambiar de un dominio a otro en el tiempo; el cómo depende mucho del sistema. En UNIX, se asocia un dominio a cada usuario+grupo; dado un usuario y el grupo al cual pertenece, se puede construir una lista de todos los objetos que puede accesar y con qué operaciones. Cuando un usuario ejecuta un programa alamacenado en un archivo de propiedad de otro usuario B, el proceso puede ejecutar dentro del dominio de protección de A o B, dependiendo del **bit de dominio** o *SETUSERID bit* del archivo. Este mecanismo se usa con algunos utilitarios. Por ejemplo, el programa `passwd` debe tener privilegios que un usuario común no tiene, para poder modificar el archivo donde se guardan las claves. Lo que se hace es que el archivo `/bin/passwd` que contiene el programa es propiedad del superusuario, y tiene el SETUSERID encendido. Este esquema es peligroso: un proceso puede pasar de un estado en que tiene poco poder a otro en que tiene poder absoluto (no hay términos medios). Cualquier error en un programa como `passwd` puede significar un gran hoyo en la seguridad del sistema. Cuando se hace una llamada al sistema también se produce un cambio de dominio, puesto que la llamada se ejecuta en modo protegido.

Matriz de acceso

Ahora bien, ¿cómo se las arregla el sistema para llevar la cuenta de quién puede accesar qué objetos y con qué operaciones? Conceptualmente al menos, podemos ver este modelo de protección como una gran **matriz de acceso**.

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

Los cambios de dominio que un proceso puede hacer también podemos integrarlos a la matriz, tratando a los dominios como otros objetos, con una operación: entrar.

Una política de protección involucra decidir cómo se va a llenar esta matriz. Normalmente el usuario que crea un objeto es quién decide cómo se va a llenar la columna de la matriz correspondiente a ese objeto. La matriz de acceso es suficientemente general como para apoyar diversas políticas. Por

ejemplo:

- La capacidad para copiar o transferir un derecho de un objeto a otro dominio.
- Capacidad de un dominio para modificar los derechos en otros dominios (todos, o para un recurso específico).

El problema es cómo almacenar esta matriz. Como es una matriz poco densa (muchos de los elementos son vacíos), no resulta práctico representarla como matriz propiamente. Podríamos usar una tabla con triples (dominio, objeto, derechos). Si un proceso dentro de un dominio *D* intenta efectuar una operación *M* sobre un objeto *O*, se busca *(D, O, C)*, y se verifica si *M* pertenece a *C*. De todas maneras, la tabla es grande, y el esquema no es muy eficiente. Además, si un objeto puede ser, por ejemplo, leído por todo el mundo, debe tener entradas para cada dominio.

Listas de acceso

Alternativamente, podemos guardar la matriz por columnas (descartando las entradas vacías). Es decir, a cada objeto se le asocia una lista de pares (dominio, derechos). Es lo que se conoce como lista de acceso o ACL. Si pensamos en archivos de Unix, podemos almacenar esta lista en el nodo-i de cada archivo, y sería algo así como

((Juan, *, RW), (Pedro, Profes, RW), (*, Profes, R))

En la práctica, se usa un esquema más simple (y menos poderoso), pero que puede considerarse aún una lista de accesos, reducida a 9 bits. 3 para el dueño (RWX), 3 para el grupo, y 3 para el resto del mundo.

Windows NT usa listas de accesos con todo el nivel de detalle que uno quiera: para cualquier usuario o grupo, se puede especificar cualquier subconjunto de derechos para un archivo, de entre {RWXDPO}. .

Capacidades

La otra posibilidad es almacenar la matriz por filas. En este caso, a cada proceso se le asocia una lista de **capacidades**. Cada capacidad corresponde a un objeto más las operaciones permitidas.

Cuando se usan capacidades, lo usual es que, para efectuar una operación *M* sobre un objeto *O*, el proceso ejecute la operación especificando un puntero a la capacidad correspondiente al objeto, en vez de un puntero al objeto. La sola *posesión* de la capacidad por parte del proceso quiere decir que tiene los derechos que en ella se indican. Por lo tanto, obviamente, se debe evitar que los procesos puedan "falsificar" capacidades.

Una posibilidad es mantener las listas de capacidades dentro del sistema operativo, y que los procesos sólo manejen punteros a las capacidades, no las capacidades propiamente. Otra posibilidad es cifrar las capacidades con una clave conocida por el sistema, pero no por el usuario. Este enfoque es particularmente adecuado para sistemas distribuidos, y es usado en Amoeba.

Un problema de las capacidades es que puede ser difícil revocar derechos ya entregados. En Amoeba, cada objeto tiene asociado un número al azar, grande, que también está presente en la capacidad. Cuando se presenta una capacidad, ambos números deben coincidir. De esta menera, para revocar los derechos ya otorgados, se cambia el número asociado al objeto. Problema: no se puede revocar selectivamente. Las revocaciones con ACL son más simples y más flexibles.

Mecanismos de autentificación

La autentificación, que consiste en identificar a los usuarios que entran al sistema, se puede basar en posesión (llave o tarjeta), conocimiento (clave) o en un atributo del usuario (huella digital).

Claves

El mecanismo de autentificación más ampliamente usado se basa en el uso de claves o *passwords*; es fácil de entender y fácil de implementar. En UNIX, existe un archivo `/etc/passwd` donde se guarda los nombres de usuarios y sus claves, cifradas mediante una función *one-way* F. El programa `login` pide nombre y clave, computa F(clave), y busca el par (nombre, F(clave)) en el archivo.

Con claves de 7 caracteres tomados al azar de entre los 95 caracteres ASCII que se pueden digitar con cualquier teclado, entonces las 95^7 posibles claves deberían desincentivar cualquier intento por adivinarla. Sin embargo, una proporción demasiado grande de las claves escogidas por los usuarios son fáciles de adivinar, pues la idea es que sean también fáciles de recordar. La clave también se puede descubrir mirando (o filmando) cuando el usuario la digita, o, si el usuario hace login remoto, interviniendo la red y observando todos los paquetes que pasan por ella. Por último, además de que las claves se pueden descubrir, éstas también se pueden "compartir", violando las reglas de seguridad. . En definitiva, el sistema no tiene nunguna garantía de que quien hizo login es realmente el usuario que se supone que es.

Identificación física

Un enfoque diferente es usar un elemento físico difícil de copiar, típicamente una tarjeta con una banda magnética. Para mayor seguridad este enfoque se suele combinar con una clave (como es el caso de los cajeros automáticos). Otra posibilidad es medir características físicas particulares del sujeto: huella digital, patrón de vasos sanguíneos de la retina, longitud de los dedos. Incluso la firma sirve.

Algunas medidas básicas

- Demorar la respuesta ante claves erróneas; aumentar la demora cada vez. Alertar si hay demasiados intentos.
- Registrar todas las entradas. Cada vez que un usuario entra, chequear cuándo y desde dónde entró la vez anterior.
- Hacer chequeos periódicos de claves fáciles de adivinar, procesos que llevan demasiado tiempo corriendo, permisos erróneos, actividades extrañas (por ejemplo cuando usuario está de vacaciones).
- Para los más paranoicos: poner trampas para descubrir intentos de uso no autorizado.

Criptografía

Los mecanismos de protección que hemos visto hasta ahora muchas veces no son suficientes para mantener información confidencial adecuadamente resguardada. Con el uso masivo de las redes de computadores, más y más información se transmite por ella, y nadie puede estar seguro de que no hay *mirones* en el alambre. Los métodos criptográficos son los más comúnmente usados para proteger información confidencial. Lo que se envía por la red no es la información original, sino la información codificada, que carece de sentido salvo para el receptor, que puede decodificarla.

Criptografía simétrica

La criptografía simétrica se basa en un algoritmo general de codificación *C*, un algoritmo general de decodificación *D*, y una clave secreta *k*, tales que

1. $D_k\,(C_k(m)) = m$.
2. C_k y D_k son computables eficientemente.
3. La seguridad depende sólo de que la clave --no los algoritmos-- sea secreta.

Un esquema ampliamente usado es el DES (data encryption standard), creado por la NSA.

El inconveniente de la criptografía simétrica es la distribución de la clave. Si quiero enviar un texto confidencial por la red, lo envío cifrado, pero ¿cómo le comunico la clave a mi interlocutor? Por otra parte, se requiere una clave por cada par de usuarios.

Criptografía de clave pública

La criptografía de clave pública es asimétrica. Se basa en métodos que requieren una clave pública para cifrar, y otra, distinta y privada, para descifrar. Supongamos que los procedimentos para cifrar y descifrar de los usuarios *A* y *B*, son, respectivamente C_A, D_A, C_B y D_B.

Para que *B* envíe mensaje *m* a *A*:

- *B* averigua la clave pública de *A*, en un directorio público.
- Envía $C_A(m)$
- *A* descifra el mensaje con su clave: $m=D_A(C_A(m))$. Sólo *A* puede hacerlo, pues es el único que conoce la clave.

Los métodos de criptografía de clave pública tienen la interesante propiedad

$$m=C_A(D_A(m))$$

que permite implementar también firmas digitales. Una firma digital debe ser dependiente del firmador y del mensaje que está firmando.

Un mensaje *m* firmado por *B* es

$$s=(D_B(m))$$

Si se lo quiere mandar privadamente a *B*, además lo cifra, enviando $C_A(s)$, pero esto es sólo para privacidad.

A primero recupera *s*, descifrando el mensaje como antes, si viene cifrado, y luego obtiene el mensaje original con

$$m=C_B(s)$$

Y ahora *A* posee el par *(m,s)* que equivale a un documento firmado por *B*, puesto que:

- *B* no puede negar que envió *m*, pues nadie más que *B* puede haber creado $s=(D_B(m))$.
- *A* puede convencer a un juez que $m=C_B(s)$.
- *A* no puede modificar *m*, pues la firma habría sido otra.

En particular, RSA opera de la siguiente manera:

La clave pública de cifrado es un par *(c,n)*, y la clave privada un par *(d,n)*. Cada mensaje se representa como un número entre 0 y *n-1*.

Los procedimientos para cifrar y descifrar con esas claves son:

$$C(m) = m^c \bmod n = w$$

$$D(w) = w^d \bmod n$$

El problema es escoger las claves. *n* es el producto de dos números primos grandes (100 dígitos) *p* y *q*. *d* se escoge al azar como un número grande relativamente primo con *(p-1)(q-1)*. Finalmente *c* se computa como el inverso de *d* en módulo *(p-1)(q-1)*, o sea:

$$c\ d \bmod (p-1)(q-1) = 1$$

A pesar de que *n* es público, *p* y *q* no lo son, y todo se basa en la suposición de que es difícil factorizar un número grande.

Criptografía híbrida

Los métodos de criptografía de clave pública resuelven el problema del intercambio de claves, pero son bastante más lentos (100 a 1000 veces) que los métodos de criptografía simétrica. Se puede obtener lo mejor de ambos mundos con un esquema híbrido: se usa criptografía de clave pública para acordar una clave privada, y el grueso de la comunicación se cifra con esa clave usando criptografía simétrica.

Ultima modificación: July 01, 1998, por Juan E. Navarro (jnavarro@ing.puc.cl)

Go backward to Seguridad y protección
Go up to Top
Go forward to Estudio de casos

Sistemas distribuidos

Un sistema distribuido es una colección de computadores conectados por una red de comunicaciones, que el usuario percibe como un solo sistema (no necesita saber qué cosas están en qué máquinas). El usuario accesa los recursos remotos de la misma manera en que accesa recursos locales.

En comparación con un sistema centralizado:

- Mejor aprovechamiento de los recursos.
- Mayor poder de cómputo a más bajo costo.
- En teoría, mayor confiablidad, si se maneja suficiente redundancia.
- Crecimiento incremental.

En contraposición:

- El software es mucho más complejo (de hecho, todavía no está del todo claro cómo hacerlo).
- Muchos usuarios desde muchas partes: problemas de seguridad.

Redes de computadores

En una red de computadores, la comunicación y sincronización entre los nodos se basa exclusivamente en mensajes, ya que no hay memoria ni dispositivos compartidos. Un sistema operativo con soporte para redes provee primitivas de comunicación interprocesos (recordar Paso de Mensajes).

send(P, mensaje)
: envía el mensaje al proceso P, y continúa.

receive(Q, mensaje)
: bloquea el proceso hasta recibir un mensaje de Q.

receive(id, mensaje)
: bloquea el proceso hasta recibir un mensaje de cualquier proceso. El identificador del emisor se devuelve en id.

Sistemas operativos de red

Un sistema operativo de red provee un ambiente en el que los usuarios pueden accesar recursos remotos, pero deben estar conscientes de la multiplicidad de máquinas.

Login remoto

En Unix, uno puede hacer `telnet lucifer.cs.uwm.edu` si quiere entrar a esa máquina en forma remota. Cuando se ejecuta este programa se crea un proceso cliente telnet que intenta comunicarse con la máquina remota. Para que esto funcione, en tal máquina debe haber un *servidor telnet*, que, cuando

recibe una solicitud de conexión, crea un proceso que actúa en representación del usuario (previa autentificación). Todo lo que el usuario digita es enviado por el cliente al proceso remoto, quien ejecuta los comandos y envía la salida para que el cliente la despliegue.

Transferencia de archivos

Otra función que los sistemas operativos de red proveen es la de transferencia de archivos. En la Internet hacemos

```
ftp altar.ing.puc.cl
<autentificación...>
get archivo
```

o también

```
rcp maquina1:archivo1 maquina2:archivo2
```

Bajo este esquema no hay transparencia para el usuario, pues éste debe saber exactamente dónde está el archivo que necesita. El acceso a un archivo remoto difiere bastante del acceso a un archivo local. Además, los archivos en realidad no se comparten; más bien, hay muchas copias de un mismo archivo en todos los lugares en los que se necesita; no sólo se gasta espacio, sino que puede haber problemas de consistencia. FTP se implementa de manera similar a telnet, sólo que el servidor FTP responde sólo a un conjunto predefinido de comandos (get, put, ls, cd, etc.).

Un punto importante acerca de telnet y FTP es que el usuario debe cambiar de paradigma. Para usar FTP el usuario debe conocer un conjunto de comandos que difiere bastante de los comandos del sistema operativo. En el caso de telnet la diferencia no es tan drástica, pero existe en la medida que el sistema operativo de la máquina remota difiera del de la máquina local.

Sistemas operativos distribuidos

Según nuestra definición, un sistema operativo distribuido debe hacer que los usuario (procesos) perciban el sistema como un *monoprocesador virtual*. No es necesario que los usuarios estén al tanto de la existencia de múltiples máquinas y múltiples procesadores en el sistema. En la actualidad no hay ningún sistema que cumpla cabalmente con esta definición, pero se están haciendo avances.

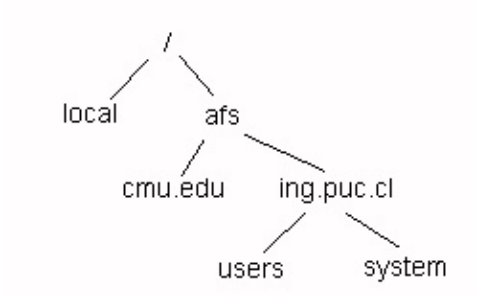
Los aspectos que hay tener en cuenta en el diseño de un sistema operativo distribuido se describen a continuación.

Transparencia

Los usuarios deben poder accesar los objetos remotos de la misma forma que los locales. Es responsabilidad del sistema operativo distribuido localizar el recurso y obtener la interacción adecuada.

La transparencia también tiene que ver con la forma de nombrar los objetos: el nombre de un objeto no debe depender del lugar en que se almacena. Un recurso debe poder migrar de un lugar a otro, sin que esto signifique que haya que cambiar su nombre.

Los usuarios, además, deben tener la misma vista del sistema, independientemente del lugar en que el usuario haga *login*.



Confiabilidad

Supongamos que tenemos un sistema con 5 máquinas, cada una con una probabilidad del 95% de estar funcionando normalmente en cualquier instante. Si el hecho de que una máquina se caiga bota todo el sistema, entonces la probabilidad de que el sistema esté funcionando en un instante dado es del 77%. Si en cambio el sistema está hecho de manera tal que cualquier máquina puede asumir el trabajo de una máquina que se cae, entonces el sistema estará funcionando un 99.9994% del tiempo. La primera opción es definitivamente mala (mucho peor que en un sistema centralizado); la segunda, es poco realista (difícil de implementar eficientemente). Además, la tolerancia a fallas es un tema particularmente complejo, debido a que, en general, no es posible diferenciar entre un enlace de comunicaciones caído, una máquina caída, una máquina sobrecargada, y pérdida de mensajes.

En la práctica: ubicar un punto intermedio razonable. O sea, replicar hasta un punto razonable los elementos claves (datos, servicios, hardware). Obviamente, la confiablidad tiene que ver con la consistencia de los datos. Si un archivo importante se replica, hay que asegurarse que las réplicas se mantengan consistentes; mientras más haya, más caro es mantenerlas, y más probable es que haya inconsistencias.

La seguridad es también un aspecto fundamental de la confiablidad.

Escalabilidad

La escalabilidad de un sistema es la capacidad para responder a cargas de trabajo crecientes. En particular un sistema distribuido escalable debe diseñarse de manera que opere correcta y eficientemente con diez o con millones de máquinas.

Un principio básico en el diseño de sistemas escalables es que la carga de trabajo de cualquier componente del sistema debe estar acotada por una constante independiente del tamaño del sistema. Si no es así, entonces el crecimiento del sistema estará limitado. En consecuencia, todo lo que huela a centralización a la larga constituirá un freno a la escalabilidad. Por ejemplo, servidores de autentificación, de nombres o de archivos: si estos son centralizados (uno para todo el sistema), entonces hay un límite en la cantidad de servicios que pueden atender. Los algoritmos centralizados también deben evitarse. Por ejemplo, dada un topología de red compleja, en teoría la mejor manera de rutear mensajes es recopilando la información acerca del estado de los enlaces (qué conexiones hay, y qué grado de congestión tiene cada una), correr un algoritmo de teoría de grafos que encuentra el camino más corto (barato) entre cada par de nodos, y después divulgar esa información. Pero, otra vez, un mecanismo así no es escalable. En la medida que sea posible, hay que usar algoritmos descentralizados, que tienen las siguientes características:

1. Ninguna máquina tiene el conocimiento completo del estado del sistema.

2. Cada nodo toma decisiones basándose únicamente en la información que tiene disponible.
3. La falla de una máquina no hace fracasar el algoritmo.
4. Los algoritmos no dependen de la existencia de relojes sincronizados. (Por ejemplo, si un algoritmo obliga a que todos los nodos registren alguna información acerca de su estado a las 12:00:00.0...)

Todas estas imposiciones son bastante fuertes. No siempre es posible encontrar algoritmos que cumplan todos estos requisitos. Incluso, algoritmos que los cumplen no siempre escalan bien.

Flexibilidad

Sistemas distribuidos son nuevos. Es importante, por ende, que se puedan adaptar a nuevas tecnologías y a nuevos avances en el tema.

Rendimiento

Ciertamente, es un aspecto que no se puede dejar de lado. ¿Cómo se mide?

Sistemas de archivos distribuidos

Un sistemas de archivos distribuidos o SAD es un sistema de archivos en el que los clientes, servidores y dispositivos de almacenamiento están dispersos en una red, pero la distribución es transparente para los clientes, es decir, el SAD se ve como un sistema de archivos centralizado convencional.

Servicio de archivos:
Es la especificación de lo que ofrece un sistema de archivos a sus clientes.
Servidor de archivos:
Entidad (proceso) que implementa el servicio de archivos.
Clientes:
Procesos que usan los servicios.

Nombres y transparencia

Los sistemas de archivos (distribuidos y de los otros) en general usan simultáneamente dos formas de nombrar archivos: una simbólica, para consumo humano (archivo `prog.cc`), y otra con identificadores binarios, para uso interno.

El sistema de archivos debe relacionar (*mapear*) nombres simbólicos con nombres binarios (y para eso se usan los directorios). En un sistema convencional, el nombre binario podría ser simplemente el nodo-i o su equivalente. En un sistema más general como un SAD se requiere que el nombre binario contenga también el identificador de la máquina y del dispositivo dentro de esa máquina. Si además se usa replicación, entonces dado un nombre simbólico el sistema debe obtener no uno, sino un conjunto de nombres binarios: uno por cada réplica del archivo.

Patrones de uso de archivos

Estudios señalan que:

- La mayoría de los archivos pesa menos de 10KB. No sería descabellado entonces transferir

archivos completos al lugar donde se usan.
- Muchos archivos tienen un tiempo de vida muy corto (típicos archivos temporales). Sería razonable tratar de crear los archivos en el cliente.
- Pocos archivos son compartidos.
- Patrones de uso según tipos de archivos. Por ejemplo, ejecutables se usan masivamente, pero rara vez cambian: .

Pero estos estudios se han realizado en ambientes universitarios; en otros pueden ser bastante diferentes.

Uso de cachés

En un SAD hay cuatro posibles lugares donde poner los archivos que se están usando (o parte de ellos): el disco del servidor, la memoria principal del servidor, el disco del cliente (si tiene), o la memoria principal del cliente.

Lo más directo es manejarlos en el disco del servidor: hay espacio, es accesible por los clientes, y al haber una sola copia no hay problemas de consistencia. Pero hay problemas de rendimiento y potenciales problemas de congestión. Cada acceso por parte del cliente requiere acceso al disco del servidor y transferencia de los datos por la red. Los accesos al disco se pueden reducir sustancialmente si el servidor usa su memoria como caché. La granularidad del caché puede ser un bloque o archivos enteros. Como siempre, también hay que definir una política de reemplazo. Esta opción también es fácil, y transparente para el cliente: el propio servidor se encarga de mantener sincronizada la información de los archivos que está en memoria con la que está en el disco. Sin embargo, esta alternativa no elimina ni reduce la transferencia de información por la red. La única forma de reducir el tráfico por la red es haciendo *caching* en el cliente, pero ahí es donde empiezan los problemas (independientemente de si se usa la memoria o el disco como caché), porque ahora sí que puede haber inconsistencias. ¿Qué pasa si dos clientes leen el mismo archivo y lo modifican? Si un tercer cliente también lo lee, va a obtener la versión antigua, que todavía está en el servidor. O, cuando los archivos sean escritos de vuelta al servidor, ¿qué modificación debe prevalecer? Moraleja: si se hace caching en el cliente, hay que pensar bien el asunto.

Una posibilidad es usar cachés *write-through*: cada vez que se escribe en un archivo, el cliente envía la modificación al servidor para que actualice su copia. Igual hay problemas, porque esto no garantiza que se vayan a actualizar los cachés de otros clientes que contengan el mismo archivo (o bloque). Por ejemplo: proceso en cliente A abre archivo, lo lee, y lo cierra (queda en el caché de A). Otro proceso en cliente B lee el mismo archivo, y lo modifica (modificaciones se envían al servidor). Ahora otro proceso en A abre el mismo archivo, y lo lee: ¿Qué va a leer? La información obsoleta, que todavía está en el caché de A. Una posible solución es que el administrador del caché en el cliente chequee con el servidor antes de cada acceso si es que la copia local todavía es válida (pueden usarse sellos de tiempo). Pero la cosa se complica y las ventajas ya no son tan claras:

1. Las escrituras no ahorran nada: generan el mismo tráfico.
2. Las lecturas igual obligan a comunicarse con el servidor: a lo más nos ahorramos la transferencia de la información.

Solución más radical: informar las modificaciones al servidor solamente cuando se cierra el archivo. Asimismo, sólo al abrir un archivo, si está en el caché, chequear con el servidor si es que esa copia es todavía válida. Es cierto que si dos procesos en distintas máquinas modifican simultáneamente un archivo, los cambios del último que lo cierre prevalecerán, y los del otro se perderán. Simplemente decidimos que ese comportamiento, que parece incorrecto, es ahora correcto, y debemos vivir con eso.

Después de todo, no es tan malo como parece: en sistemas centralizados puede pasar algo similar si dos procesos leen un archivo, lo modifican dentro de sus respectivos espacios de direccionamiento y después lo escriben. Decimos que, en este caso, los archivos compartidos no tienen *semántica Unix*, sino *semántica de sesión*.

Replicación

Los SADs mantienen (o deberían mantener) múltiples copias de cada archivo, para mejorar la confiabilidad del sistema (si se cae un servidor, se usa la copia en otra máquina).

Problema: cómo manejar la modificación de múltiples copias de un mismo archivo. Una posibilidad: hay un servidor principal, que recibe todas las solicitudes, y las propaga (cuando `tiene tiempo') a los otros. Desventajas: centralización en el servidor principal (¿y si se cae?).

Alternativa: usar votación. Supongamos que un archivo se replica en N servidores. Los clientes deben obtener permiso de la mayoría de los servidores (la mitad más uno) antes de escribir el archivo. Una vez que eso ocurre, los servidores contactados actualizan el archivo y le asocian un nuevo número de versión. Para leer el archivo replicado, los clientes también deben contactar al menos a la mitad más uno de los servidores, y usar los datos de aquel o aquellos servidores con el mayor número de versión (que, necesariamente, corresponde a la versión actual). Este esquema se puede generalizar: para escribir, el cliente debe contactar N_w servidores y para leer, N_r. La restricción es que $N_w+N_r > N$.

En la mayoría de las aplicaciones, las lecturas son mucho más frecuentes que las escrituras; en esos casos, N_r debe ser pequeño y N_w cercano a N.

Ultima modificación: July 01, 1998, por Juan E. Navarro (jnavarro@ing.puc.cl)

Go backward to Sistemas distribuidos
Go up to Top

Estudio de casos

Estructura de un sistema operativo

Estructura simple

Una forma de organizar un sistema operativo, es con una estructura simple (o sin estructura). El sistema operativo se escribe como una colección de procedimientos, cada uno de los cuales llama a los otros cuando lo requiere. Lo único que está bien definido es la interfaz de estos procedimientos, en términos de parámetros y resultados.

Para generar el sistema operativo (el binario), se compilan todos los procedimientos y se enlazan en un solo archivo (el kernel), que es el que se carga en la memoria cuando se enciende la máquina. Hay dos extremos.

Sistemas monolíticos.
: El kernel es grande y cumple todas las funciones del sistema operativo. Es el caso de MS-DOS y de las primeras versiones de Unix.

Sistemas de microkernel.
: El kernel es pequeño, y cumple solo funciones básicas de manejo de interrupciones, comunicación interporcesos, hebras y planificación. El resto de las funciones son implemetadas en procesos de usuario.

Estructuración en capas

Otra posibilidad es diseñar el sistema en capas. El primer exponente de esta categoría fue el sistema operativo THE (Holanda, 1968), un sistema de procesamiento por lotes que tenía la siguiente estructura:

5. Programas de usuario
4. Administración de I/O
3. Comunicación entre procesos y consola
2. Administración de memoria virtual
1. Planificación de CPU
0. Hardware

En el caso de THE, la estructuración en capas era más que nada para modularizar el diseño (igual se compilaba y enlazaba todo en un solo gran archivo). Cada capa se puede considerar un TDA, ya que encapsula datos y las operaciones que manipulan esos datos. Una capa puede usar los servicios sólo de capas inferiores. La capa 1 se encarga de asignar el procesador, ensuciándose las manos con el timer y las interrupciones del hardware. Encima de la capa 1 el sistema consistía sólo de procesos secuenciales, cada uno de los cuales se podía programar sin necesidad de preocuparse de que múltiples procesos estuvieran compartiendo la CPU. La capa 2 administraba la memoria. Encima de esa capa los procesos no tenían que preocuparse de si tenían páginas en disco (bueno, tambor, en ese tiempo). Y así

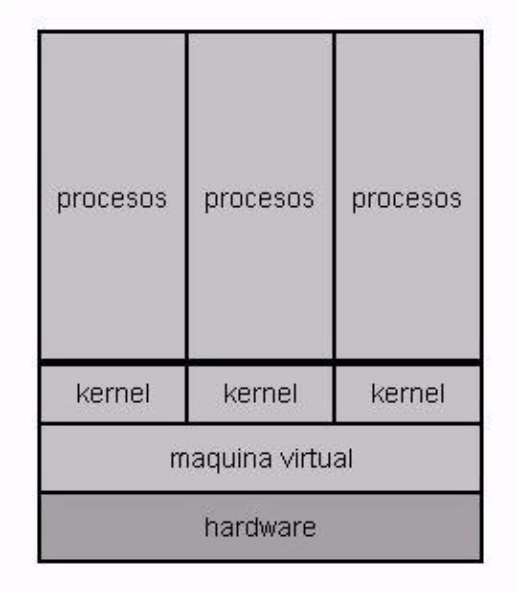
sucesivamente... hasta llegar a la capa 5, correspondiente a los procesos de usuario.

Ventajas: modularización, ocultamiento de información, flexibilidad. Cada capa se desarrolla y depura "sin mirar hacia arriba". Nuevas funciones pueden añadirse, y la implementación de las existentes puede modificarse, mientras no se cambie su interfaz.

Desventajas: requiere cuidadosa planificación, para decidir qué cosa va en qué capa. Por ejemplo, la capa que provee acceso al disco debe estar en una capa más baja que la que administra memoria virtual, pero encima del planificador de CPU, pues el driver probablemente se bloquea para esperar que una operación de I/O se complete. Sin embargo, en un sistema grande, el planificador podría llegar a necesitar manejar más información de los procesos que la que cabe en su memoria, requiriendo apoyarse en el disco. Otro potencial problema: eficiencia, pues algunos servicios pasan por varias capas antes de llevarse a cabo.

Máquinas virtuales

Un sistema operativo de tiempo compartido cumple dos funciones: (1) multiprogramación, y (2) proveer una máquina extendida, con una interfaz más conveniente para los programas de usuario que la máquina pura. En la década del 70, a la IBM se le ocurrió separar completamente estas funciones en el sistema operativo VM/370. El núcleo del sistema, llamado **monitor de la máquina virtual** ejecuta directamente sobre el hardware, y su único objetivo es la multiprogramación, ofreciendo múltiples máquinas virtuales a la capa superior.



Pero estas máquinas virtuales no son máquinas extendidas (con archivos y otras abstracciones), como es lo habitual, sino copias exactas de la máquina original. Gracias a esto, cada máquina virtual puede ejecutar un sistema operativo diferente, construido para operar en la máquina original. La CPU se comparte con planificación de CPU, la memoria con memoria virtual, y el disco se particiona.

No es simple de implementar. La máquina original tiene dos modos: monitor y usuario. El monitor de la máquina virtual opera en modo monitor, y cada máquina virtual puede operar sólo en modo usuario. En consecuencia se debe proveer un modo usuario virtual y un modo monitor virtual. Las acciones que en la máquina real causan el paso de modo usuario a modo monitor deben causar el paso de modo usuario

virtual a modo monitor virtual. ¿Cómo? Cuando un proceso de usuario hace una llamada al sistema, el control se transfiere al monitor de la máquina virtual, quien devuelve el control a la máquina virtual, simulando una llamada al sistema. Si en esas condiciones (operando en modo monitor virtual, que en realidad no es más que modo usuario físico), se intenta accesar el hardware (por ejemplo, una operación de I/O), el control vuelve al monitor de la máquina virtual, quien efectúa, controladamente, la operación).

Ventajas: facilita el desarrollo de sistemas operativos (cuando la máquina es cara y no se puede entregar una a cada programador). Sin este esquema, es necesario bajar la máquina cada vez que quiere probar una modificación al sistema. También permite resolver problemas de compatibilidad. Por ejemplo, si queremos desarrollar un nuevo sistema operativo, pero queremos que los miles de programas DOS que existen sigan corriendo, podemos usar esta solución, asignándo una máquina virtual a cada programa DOS que se ejecute. Se puede ir más lejos, y crear una máquina virtual distinta a la original. Así es como Windows para arquitectura Intel se puede ejecutar, por ejemplo en una Silicon Graphics. El problema es que, en este caso, hay que interpretar cada instrucción (en lugar de ejecutarla directamente).

UNIX

Historia

El primer sistema de tiempo compartido (CTSS) tuvo gran éxito en los años 60. Entusiasmados con ese éxito, los inventores comenzaron a diseñar la segunda generación, que se llamaría MULTICS, pero que nunca vio la luz, en parte por lo ambicioso del proyecto (que pretendía soportar cientos de usuarios en una plataforma no más poderosa que un PC/AT), y en parte por la mala idea de desarrollarlo en PL/I. .

En 1969, Thompson (y después Ritchie), que habían trabajado en MULTICS, comenzaron a desarrollar silenciosamente, y ésta vez en assembler, una versión menos ambiciosa que primero llamaron UNICS y despúes quedó como UNIX. Más tarde, para facilitar la portabilidad de UNIX, se inventó el lenguaje C, se escribió un excelente compilador, y se reescribió el sistema en C, todo esto en los laboratorios Bell.

En 1976, la versión 6 ya era un estándar entre las universidades, a quienes Bell les dio el derecho de usar libremente el código fuente. El problema que tenemos hoy es que hay decenas de versiones distintas, que, aunque tienen un ancestro común, también tienen muchas incompatibilidades.

Principios de diseño

UNIX es un sistema de tiempo compartido, de propósito general, diseñado por programadores para ser usado por programadores. Simpleza, elegancia y consistencia son las características que los programadores desean de un sistema. Por ejemplo, en vez de agregar complejo código para tratar sistuaciones *patológicas*, UNIX se *cae* controladamente (entra en modo llamado *pánico*); más bien trata de prevenir estas situaciones antes que curarlas.

Otras cualidades de UNIX, bien vistas por los programadores, pero no tanto por los no-programadores, son su poder y flexibilidad, gracias a que ofrece una gran cantidad de elementos básicos que se pueden combinar de múltiples maneras para lograr metas más complejas. En UNIX, cada programa hace una sola cosa, y la hace bien (y de todas las formas posibles).

```
cat * | tr -sc A-Za-z '\\012' | sort | uniq -c | sort -n | tail -5
```

Otra característica es el uso de comandos cortos y crípticos. (`cp` en vez de `copy`).

Procesos

Los únicos entes activos en UNIX son los procesos. Cada usuario puede tener varios procesos activos simultáneamente, así que en un sistema UNIX puede haber cientos o miles de procesos simultáneos. Incluso cuando no hay usuarios usando el sistema, hay procesos corriendo, llamados **demonios**, que son creados cuando se inicializa el sistema.

Un ejemplo es el demonio *cron*, que despierta cada minuto para ver si hay algún trabajo que hacer. Gracias a él se pueden fijar actividades periódicas, como por ejempo, hacer respaldos a las 4 AM. Otros demonios manejan el correo entrante, la cola de impresión, monitorean el uso de páginas de memoria, etc.

Para implementar procesos, el kernel mantiene dos estructuras de datos:

Tabla de procesos.
: Residente todo el tiempo en memoria, contiene información para todos los procesos, incluidos los que no están en memoria. Aquí se manejan datos como la prioridad del proceso, mapa de memoria del proceso, dónde se encuentra el proceso cuando está en disco, etc.

Estructura de usuario.
: Se pasa a disco y a memoria junto con el proceso. Contiene registros, tabla de archivos abiertos, stack del kernel, estadísticas de uso de recursos, etc.

Planificación de CPU

El algoritmo de planificación de UNIX está diseñado para favorecer a los procesos interactivos. La planificación se hace en dos niveles: el algortimo de largo plazo determina qué procesos pasan de la memoria al disco, y viceversa. El de corto plazo determina a qué proceso se le entrega la CPU. Para eso usa múltiples colas, una para cada prioridad. Sólo los procesos que están en memoria y listos para ejecutar son puestos en estas colas. El planificador escoge cada vez el primer proceso de la cola de más alta prioridad que no esté vacía, y se le entrega la CPU por un quantum (0.1 seg), o hasta que se bloquee. Para evitar inanición, los procesos van perdiendo prioridad mientras más usan CPU: en cada tick del reloj se incrementa el contador de uso de CPU del proceso ejecutante. Cada segundo se recalculan las prioridades, dividiendo previamente por 2 el uso de CPU contabilizado a cada proceso.

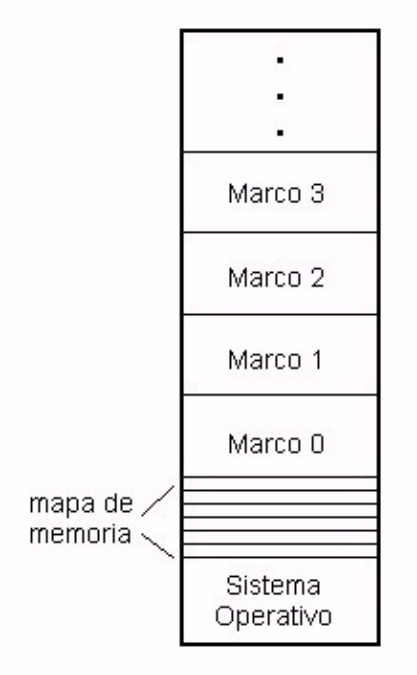
Nueva prioridad = Base + Uso de CPU

A mayor valor, menor prioridad. La base es normalmente 0 para procesos de usuarios. El superusuario puede cambiar la prioridad de cualquier proceso, pero un usuario común sólo puede disminuir la prioridad de sus procesos.

Si el proceso usa todo su quantum, se pone de vuelta al final de su cola. Si un proceso se bloquea (por ejemplo, por I/O), entonces sale de estas colas. Pero cuando vuelve nuevamente a estado READY, se pone, por esa vez en una cola de prioridad alta (negativa). Exactamente a qué cola, depende del evento por el que haya estado esperando.

O sea, los procesos intensivos en I/O (interactivos) tienen mayor prioridad que los intensivos en CPU. La planificación de largo plazo la hace un demonio (el *swapper* o intercambiador), que periódicamente chequea el sistema para decidir qué procesos se pasan de memoria a disco y viceversa.

Memoria virtual



El mapa de la memoria física contiene información acerca del uso de los marcos de páginas (una entrada para cada marco): puntero al próximo (usado cuando el marco está en la lista de marcos libres); dispositivo y bloque donde se almacena la página en disco; flags para indicar si el bloque está libre, **clavado** en la memoria, y otros.

El kernel y el mapa siempre están en memoria física. Cuando un proceso produce una falta de página, se usa un marco de la lista de marcos libres; si no hay, el proceso se suspende hasta que el demonio haga espacio.

Cada 250 mseg, un **demonio de páginas** es despertado para que revise si hay al menos `lotsfree` marcos libres. Si los hay, vuelve a dormir; si no, debe transferir páginas a disco hasta que los haya. Para escoger una víctima, se usa el algoritmo del reloj modificado. La víctima se escoge globalmente (o sea, sin importar a qué proceso pertenece la página).

El problema es que, si la memoria es grande, el reloj se demora mucho en *dar la vuelta*, por lo que la probabilidad de que una página haya sido usada en el intertanto es bastante alta, y el algoritmo degenera en FIFO, que, sabemos, no es muy bueno. La solución es usar un reloj con dos manecillas. Cada vez se pone en cero el bit de referencia de la página apuntada por la manecilla delantera, se chequea el bit de la página apuntada por la manecilla trasera (que es la potencial víctima), y se avanzan ambas manecillas. Si las manecillas están bien juntas, sólo las páginas más usadas tendrán realmente una segunda oportunidad.

Si el sistema nota que hay mucha paginación, se activa el *swapper* para que saque procesos de la memoria: Se saca el proceso que más tiempo ha estado bloqueado, siempre que ese tiempo sea más de 20 seg. Si no hay tal proceso, se escoge, de entre los 4 más grandes, aquél que ha estado más tiempo en la memoria. Esto se repite hasta que se haya recuperado suficiente memoria. Periódicamente, el *swapper* chequea si se debe pasar algún proceso del disco a la memoria. Para eso se considera el tiempo que lleva en disco, su tamaño, su prioridad, y cuánto tiempo llevaba bloqueado cuando se pasó a disco.

Ultima modificación: July 01, 1998, por Juan E. Navarro (jnavarro@ing.puc.cl)