Sistemas Operativos

Resumen de Nico Matovelle

Índice

❸❼❶❸

Introducción

Sistema Operativo: es un modelo de computador fácil de usar y que administra los recursos. Se podría decir que es la capa entre la parte fea del ordenador (hardware) y la parte bonita (aplicaciones de usuario).

Funciones:

- Proporciona a los programadores un conjunto abstracto de recursos, haciendo que los programas de aplicaciones interaccionen con el SO y los usuarios finales solo interaccionen con la interfaz de usuario (shell o GUI, graphical user interface).
- Administra los recursos de hardware, asignándolos ordenadamente entre los programas, resolviendo conflictos entre usuarios y programas y coordinando la compartición de recursos (tiempo de cpu, espacio de memoria...).

Un sistema operativo tiene dos modos de ejecución:

- Modo kernel (o root): el sistema operativo tiene acceso completo a todo el hardware y puede ejecutar cualquier instrucción que la máquina sea capaz.
- Modo usuario: solo permite un subconjunto de instrucciones (las del control de la máquina o las de E/S no están permitidas).

Llamada al sistema: instrucción que permite al programa de usuario obtener los servicios del SO (para cambiar entre modo usuario y modo root se usa la instrucción TRAP. Hay otros traps de hardware que avisan de división por 0, subdesbordamiento de punto flotante...).

Hardware

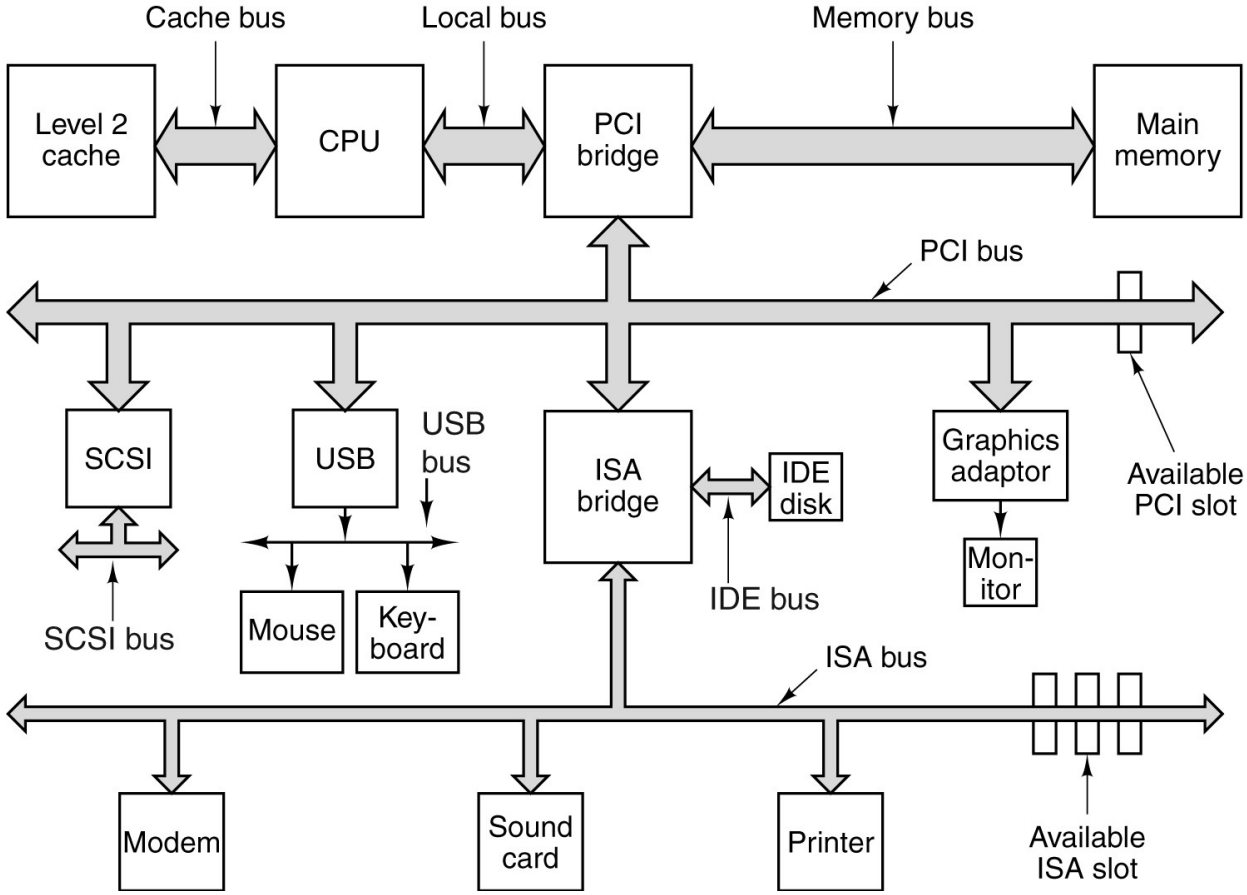
Un sistema operativo debe conocer el hardware del que dispone, ya que está muy ligado a este, pues es el que se encarga de administrar sus recursos. Las principales partes de este hardware son:

- Procesador: es capaz de ejecutar un repertorio de instrucciones ensamblador. El SO debe controlar completamente el estado del procesador y administrar su uso. Hay dos propiedades que pueden tener estos procesadores:
  - Multihilamiento — es la capacidad de alternar en un tiempo de nanosegundos entre procesos para ofrecer la sensación de poder hacer dos cosas a la vez.
  - Multinúcleo — chips de CPU con más de un procesador en su interior, lo cual permite realmente hacer varias cosas a la vez.
- Memoria: el problema de la memoria es que no podemos tener memoria de acceso rápido y gran capacidad en un mismo dispositivo. Por tiempo de acceso creciente (y capacidad creciente también) tendríamos registros, caché (de varios niveles), memoria principal (RAM), disco duro (SSD, HHD). Es en la memoria secundaria (disco duro) donde se almacena el sistema de archivos y la única memoria que permanece al apagar la máquina.
- Dispositivos de Entrada / Salida (E/S): para cada controlador de dispositivo de E/S debe haber un driver (sofware de comunicación con el controlador) que se ejecuta en modo núcleo y puede cargarse de forma dinámica. Para estos drivers pueden utilizarse instrucciones especiales o asignarles un espacio de memoria especial. Para gestionar estos

drivers pueden usarse tres métodos:

- Espera ocupada: fuerza a la CPU a estar sondeando los diversos controladores
- Interrupciones: permite que los controladores manden una señal a la CPU cuando ocurre algún evento en ellos
- DMA (Direct Memory Access): permite que la CPU le mande una orden de movimiento de datos entre memorias y que ésta siga con otros cálculos hasta que el movimiento termine.

- Buses: permiten la comunicación entre dispositivos. Actualmente hay una estructura mucho más mejorada, pero con esta nos podemos hacer una idea.



Arranque del ordenador

- Lo primero en ejecutarse es la BIOS (Basic Input Output System), almacenada en memoria no volátil y que tiene software de E/S de bajo nivel, lo justo para leer el teclado, imprimir en pantalla, explorar la memoria y determinar el dispositivo de arranque.
- La BIOS lee el sector de arranque del dispositivo determinado, lo coloca en memoria y lo ejecuta. Empieza la ejecución del SO.
- El SO carga los drivers, inicia las tablas, crea los procesos y arranca la interfaz de usuario.

Tipos de Sistemas Operativos

- **Mainframe**: procesamiento de muchos trabajos y E/S. Para servidores de alto rendimiento. Ocupan salas enteras (UNIX)
- **Servidores**: dan servicios a varios usuarios y comparten recursos de hardware y software (Linux, Windows, Solaris)
- **Multicomputadoras**: sistemas de varias CPUs (Windows, Linux, Mac OS X)
- **PC** (Personal Computer): buen soporte a un sólo usuario (Windows, Linux, Mac OS)
- **Smartphones**, **Tablets**: telefonía, fotografía digital... (Android, IOS, Windows Phone, Firefox OS)
- **Embedded** (Empotrados): aplicaciones cerradas para microondas, DVDs... (QNX, VxWorks)
- **Tiempo real**: tienen el tiempo como parámetro clave. Control de máquinas de fábrica...

Conceptos

- **Procesos**: programa en ejecución. Tiene toda la información necesaria para ejecución.
  - Espacio de direcciones del proceso: memoria asignada a éste. Guarda el programa, los datos y la pila de ejecución. Dispone de una lista de direcciones desde 0 a un valor máximo.
  - Puede hacer uso de los registros de la CPU (PC, SP...), listar archivos, crear alarmas...
  - Hay la posibilidad de ejecutar varios procesos simultáneamente
    - Los procesos se pueden suspender y reanudar guardando antes toda su información
    - Existe una tabla de procesos con toda la información de éstos
  - Pueden crearse y terminarse, teniendo hijos, cooperando y sincronizándose entre ellos y comunicándose por medio de señales
  - Cada proceso pertenece a un usuario identificado (UID) y a un grupo (GID)
    - El superusuario (root) puede saltarse cualquier regla de protección
  - Se identifican con el PID (Process IDentifier)
- **Espacio de direcciones**: en memoria principal, guarda los programas en ejecución (varios) y evita interferencias.
  - Memoria virtual – técnica de administrar el espacio de direcciones almacenando parte en disco para mayor capacidad y permitir la ejecución de procesos que ocupen más memoria. Esta característica la administra el sistema operativo.
- **Sistema de Archivos**: compuesto por archivos y directorios (que se pueden especificar por ruta relativa, que parte del directorio actual, y ruta absoluta, que parte del directorio raíz, /). Para acceder a un sistema de archivos, es preciso que esté montado.
- **Entrada / Salida**: administración de los dispositivos que permiten controlar el ordenador y que producen una salida.
- **Protección**: también administrada por el SO. Para los archivos por ejemplo, en linux

establece un código de protección de 9 bits, tres bits para cada uno: usuario, grupo, resto; y con formato rwx (read, write, execute). Suele representarse de forma octal.

- **Shell**: interfaz del usuario con el SO que no pertenece a éste. Hay varios (sh, bash, csh...)
  - Contiene comandos para el SO y direcciona la salida estándar a pantalla o a donde le indiquemos mediante tuberías.
  - Permite el uso de comodines para sustituir partes de cadenas de texto
  - `echo $SHELL #Saber que shell usamos`
- **Llamadas al sistema**: son la interfaz entre programas de usuario y SO (varían entre un SO a otro, nosotros usaremos POSIX). Están en bibliotecas para usarlas en programas con C.
  - `cuenta  = read (fd, buffer, nbytes); //devuelve a cuenta los bytes leidos y lo leido en buffer`
  - Lista de llamadas al sistema:

```
//Administracion de procesos
pid_t fork(void); //Create a child process identical to the parent. Returns pid
pid_t waitpid(pid_t pid, int *stat_loc, int options); //wait for a child to terminate
pid_t wait(int *status); //same as waitpid without specifying the child
int execve(const char *path, char *const argv[], char *const envp[]);
//replace a process' core image
void _exit(int status); //terminate process execution and return status

//Administracion de archivos
int open(const char *pathname, int flags, mode_t mode); //open a file for reading, writing, or both
int close(int fd); //close an open file
ssize_t read(int fd, void *buf, size_t count); // read data from a file into a buffer
ssize_t write(int fd, const void *buf, size_t count); //write data from a buffer into a file
off_t lseek(int fd, off_t offset, int whence); //move a file pointer
int stat(const char *path, struct stat *buf); //get a file's status information

//Administracion del sistema de directorios y archivos
int mkdir(const char *pathname, mode_t mode); //create a new directory
int rmdir(const char *pathname); //remove an empty directory
int link(const char *oldpath, const char *newpath); //make a new name for a file
int unlink(const char *pathname); //remove a directory entry
int mount(const char *source, const char *target, const char *filesystemtype, unsigned long mountflags, const void *data); //mount a file system
int umount(const char *target); //unmount a file system

//Llamadas varias
int chdir(const char *path); // change the working directory
int chmod(const char *path, mode_t mode); //change a file's protection bits
int kill(pid_t pid, int sig); //send a signal to a process
time_t time(time_t *t); // get the elapsed time since Jan, 1, 1970
```

Procesos e Hilos

Procesos

Proceso: Instancia de un programa en ejecución

Un proceso no es lo mismo que un programa. El programa es el código, y el proceso es la ejecución en curso de ese código.

Permiten la ejecución concurrente aún con una sola CPU, transformándola en varias unidades virtuales. En prácticamente todos los ordenadores se ejecutan muchos procesos simultáneamente, haciendo a la CPU conmutar de uno a otro y dando apariencia de pseudoparalelismo.

Todo el software ejecutable de una máquina se organiza en procesos secuenciales, conteniendo cada uno de ellos su respectivo código, pila, datos, registros... El concepto de multiprogramación se corresponde a la conmutación entre procesos. Aún así, en cada procesador (o núcleo de la CPU) nunca hay más de un programa activo a la vez.

Clasificación

Un proceso se puede crear en tres situaciones distintas: durante el arranque del sistema (con procesos de 1er y 2^{o} plano, es decir, que interaccionan con el usuario o que no); desde otro proceso que usa llamadas al sistema; por solicitud del usuario mediante comandos.

Como clasificación alternativa, podríamos decir que existen los procesos de modo kernel (realizan muchas llamadas al sistema y son más lentos), de modo usuario y demonios (se ejecutan siempre en segundo plano y no requieren supervisión ninguna del usuario).

Creación

Se realiza mediante la llamada al sistema fork, que crea un proceso hijo igual al proceso padre (en imagen de memoria, variables de retorno, archivos abiertos...). La cabecera de fork es la siguiente:

```
pid_t fork(void);
```

Terminación

Los procesos pueden tener su finalización de 4 formas:

- Salida normal (voluntaria): el proceso termina su trabajo y ejecuta la llamada exit.
  - `void _exit(int status);`
  - El proceso padre puede examinar el status al salir, pero exit nunca produce salida.
  - A su ejecución, el proceso se desconecta del árbol de procesos.
  - La función main de un programa no requiere de la ejecución exit al finalizar.
- Salida por error (voluntaria): el programa descubre un error y tiene como regla terminar.
- Error fatal (involuntaria): se produce por error del programa (división por cero, acceso no permitido a memoria...)
- Eliminado por otro proceso (involuntaria): un proceso externo le envía una señal kill.

- `int kill(pid_t pid, int sig);`

Esperar a la terminación de un proceso

La sincronización con la terminación de un proceso hijo se hace con la llamada wait, que devuelve el pid del hijo que terminó o -1 en caso de error (como que el proceso no tenga hijos), y almacena su variable de retorno en status. En linux también se permite especificar el pid del hijo por el que se está esperando.

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

Invocación de otros programas

Se pueden cargar las regiones de código, pila y datos de un programa en un proceso existente mediante la función execve, que tiene atributos relacionados con el main(argc, argv, envp)

```
int execve(const char *path, char *const argv[], char *const envp[]);
```

Señales

Es el sistema para notificar a los procesos de los eventos que ocurren en el sistema. Suelen identificarse con enteros y constantes simbólicas. También sirven como mecanismo de comunicación y sincronización entre procesos.

El tratamiento de las señales puede permitir finalizar, suspender y reanudar un proceso o ignorar una señal determinada. Para algunas señales se puede definir una acción diferente.

Señales de POSIX

- SIGABRT: Abortar un proceso y forzar un vaciado de núcleo .
- SIGALRM: Se activó el reloj de alarma .
- SIGFPE: Ocurrió un error de punto flotante (división entre 0) .
- SIGHUP: La línea telefónica que utilizaba el proceso se colgó .
- SIGILL: El usuario oprimió SUPR.
- SIGQUIT: El usuario oprimió la tecla que solicita un vaciado de núcleo .
- SIGKILL: Mata el proceso (no se puede ignorar ni tratar de ninguna manera).
- SIGPIPE: El proceso escribió en una tubería que no tiene lectores .
- SIGSEGV: El proceso hizo referencia a una dirección de memoria inválida .
- SIGTERM: Se utiliza para solicitar que un proceso termine en forma correcta .
- SIGUSR1: Disponible para propósitos definidos por la aplicación .
- SIGUSR2: Disponible para propósitos definidos por la aplicación.

Como se puede comprobar, la generación de señales puede venir dada por muchos factores, pero la mayoría de éstas pueden tratarse para adaptar el código de ejecución a la finalidad de la señal.

Envío de señales

Se hace mediante la llamada al sistema kill. Lo único que hace es mandar la señal establecida al proceso con el pid especificado.

```
int kill(pid_t pid, int sig);
```

Manejo de señales

```
sighandler_t signal(int signum, sighandler_t handler); //sets the
disposition of the signal signum to handler (SIG_DFL, SIG_IGN)
int sigaction(int signum, const struct sigaction *act, struct sigaction
*oldact); //examine and change a signal action
int sigreturn(unsigned long __unused); //return from a signal
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset); //examine
and check the signal mask
int sigpending(sigset_t *set); //examine pending signals
int sigsuspend(const sigset_t *mask); //wait for a signal
unsigned int alarm(unsigned int seconds); //set the alarm clock for delivery
of a signal
int pause(void); //suspend de caller until the next signal
```

Jerarquías

Un proceso, sus hijos y posteriores descendientes forman un grupo. Esto es un buen sistema para el envío de señales en grupo, permitiendo que cada proceso trate la señal de forma individual. Formación de la jerarquía:

- Se ejecuta el proceso init con PID 0.
- Init crea un proceso (con fork()) para cada terminal.
- La terminal de inicio de sesión ejecuta un shell y se crean los demás procesos.

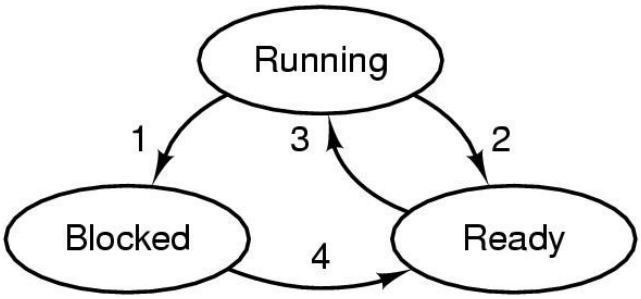
En Windows no existe la jerarquía de procesos.

Estados

Un proceso es una entidad independiente, pero hay iteración entre procesos. Por ejemplo:

```
cat esto se va a redirigir a alguna salida | grep ficheroAlQueSeRedirige.txt
```

Si grep está listo antes de que cat termine, puede bloquearse. Un proceso también puede ser detenido por el SO para cederle la CPU a otro.

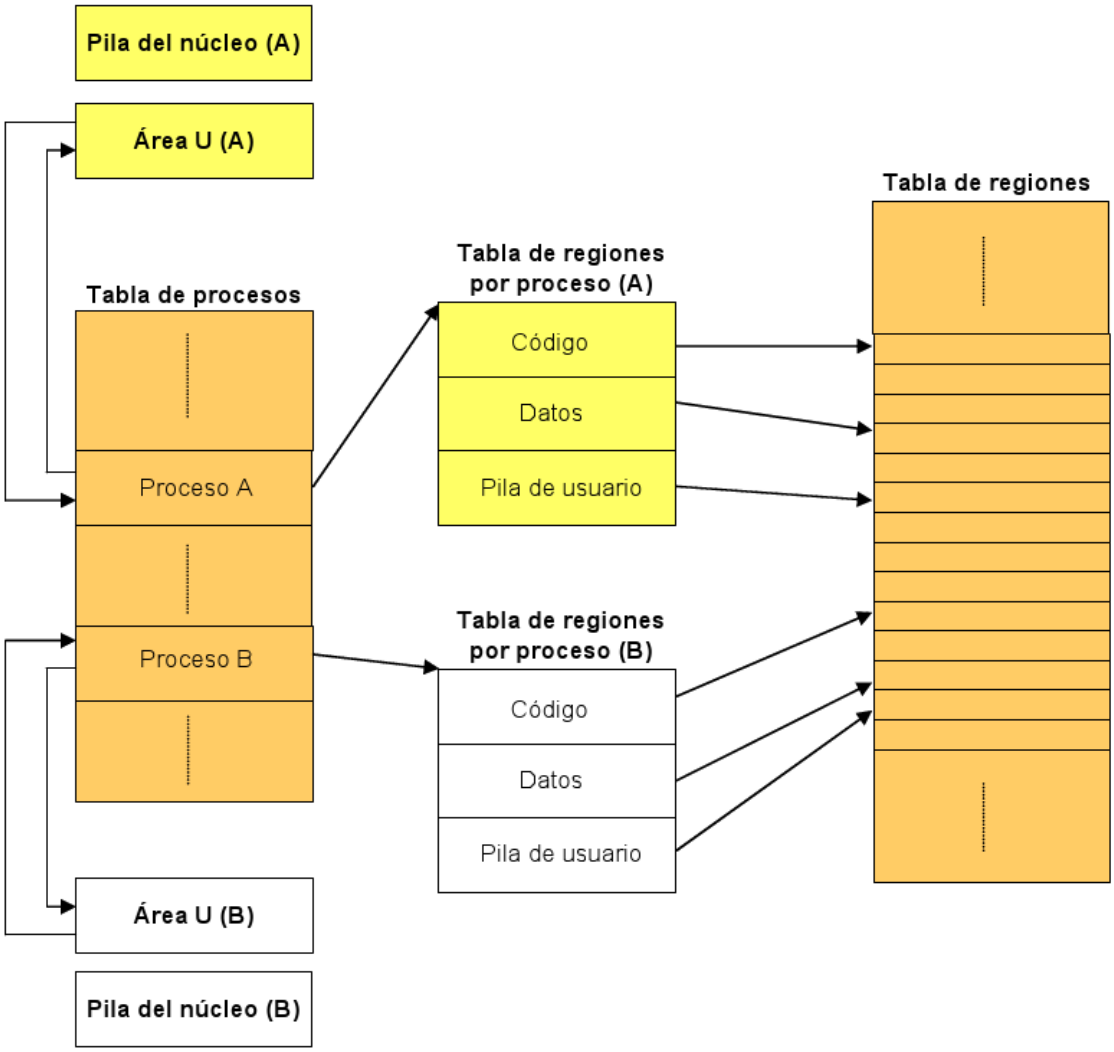


1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Implementación

Se puede realizar mediante una tabla de procesos (TP), con una entrada para cada proceso y con información sobre el estado del proceso.

Tabla de procesos



- En la pila del kernel se almacenan las funciones o rutinas invocadas durante la ejecución del proceso si éste está en modo kernel.
- En el área U se guarda la información de control que el kernel necesita durante la ejecución del proceso.
- En la tabla de regiones se almacena una entrada por cada región de memoria asignada por el kernel.

Interrupciones

Cada clase de E/S tiene un vector de interrupción (normalmente al final de su memoria reservada) gracias al cual se puede pasar de una actividad a otra. En cuanto se produce la interrupción:

- El hardware salva los registros del proceso en ejecución en su pila y salta a la dirección especificada en el vector de interrupciones
- El sofware ejecuta
    - Rutina 1 (igual para todas las interrupciones, está en lenguaje ensamblador): Mete valores en la pila del núcleo y guarda la información en la tabla de procesos
    - Rutina 2 (varía entre interrupciones, está en C): Atiende a la interrupción
    - Se llama al planificador para determinar qué proceso se ejecuta
    - Se restaura la información del proceso actual (en código ensamblador)

Hilos

Hilo: secuencia de instrucciones

En los SO tradicionales cada proceso tiene un espacio de direcciones y un solo hilo. Puede ser conveniente tener varios hilos, ya que comparten el mismo espacio de direcciones y tienen una ejecución pseudoparalela, siendo más rápida su conmutación que entre procesos.

Los procesos comparten la memoria física y otros recursos, pero los hilos comparten el espacio de direcciones, con varias ejecuciones en el mismo entorno del proceso. Exactamente, comparten:

Por cada proceso	Por cada hilo
Espacio de memoria Variables globales Archivos abiertos Procesos hijos Alarmas pendientes Señales y manejadores de señales Información contable	Contador de programa Registros Estado (listo, en ejecución, bloqueado o terminado) Pila (porque cada hilo tiene su historial de ejecución)

Los hilos se turnan para ejecutarse. Algunas CPUs tienen soporte de hardware para conmutación entre hilos. Pero éstos no están pensados para competir, sino para cooperar, ya que el SO no proporciona ninguna protección entre hilos.

Ventajas

- Hay aplicaciones que tienen muchas actividades a la vez, en las que alguna se puede bloquear mientras las otras continúan.
- Comparten el espacio de direcciones, es decir, los datos. Muchas aplicaciones no se pueden organizar como procesos independientes.
- Son procesos ligeros, por lo que son más fáciles y rápidos de crear, destruir...
- Son muy útiles en situaciones de cálculo y E/S (o varias CPUs). En caso de que todos quieran acceder a la misma CPU no se mejora el rendimiento.

Ejemplo: procesador de texto

- Un hilo: rendimiento pobre
  - Cada vez que se hace una copia y se da formato se ignoran las órdenes del teclado
- Tres hilos: solución perfecta
  - Interactuar con el usuario
  - Dar formato en segundo plano
  - Guardar el archivo automáticamente cada cierto tiempo
- Tres procesos: dificulta la compartición del documento
  - Los tres necesitan acceder al mismo archivo y compartir la memoria

POSIX

Un proceso comienza con un único hilo, pero puede crear más. Cada hilo tiene un identificador para permitir operar con ellos. Al crearlos, es imprescindible indicar el procedimiento que se ejecuta.

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void
*(*start_routine) (void *), void *arg); //create a new thread
void pthread_exit(void *retval); //terminate the calling thread
int pthread_join(pthread_t thread, void **retval); //wait for an specific
thread to exit
int pthread_yield(void); //release the CPU to let another thread run
int pthread_attr_init(pthread_attr_t *attr); //create and initialize a
thread's attribute structure
int pthread_attr_destroy(pthread_attr_t *attr); //remove a thread's
atribute structure
```

Implementaciones

En el espacio de usuario

El núcleo no sabe nada sobre los hilos, que se implementan con una biblioteca. Esto permite hilos en SO que no aceptan hilos, y requiere una tabla de hilos privada por cada proceso y administrada en tiempo de ejecución por ese proceso.

Si un hilo debe bloquearse llama a un procedimiento en tiempo de ejecución, en el que se comprueba si el hilo debe bloquearse, almacena los registros del hilo en la tabla de hilos, busca otro que esté listo para ejecutarse y carga en los registros los valores del nuevo hilo para comenzar su ejecución.

Como se puede ver, la conmutación entre hilos es más rápida que la conmutación entre procesos, ya que no necesita trap, vaciar la caché, cambiar el contexto... También permite el uso de algoritmos de planificación personalizados.

Pero este sistema puede tener problemas:

- Escalabilidad: una tabla de hilos central puede ser un problema con muchos hilos.
- Las llamadas al sistema con bloqueo pueden parar todos los hilos en proceso (se soluciona comprobando antes si va a haber bloqueo).

- Los fallos de página de un hilo también bloquean todos los hilos.
- Los hilos deben renunciar voluntariamente a la CPU ya que el SO no conmuta entre ellos.

En el espacio de núcleo

En este caso es el kernel el que administra los hilos, por lo que no hay tabla de hilos en cada proceso ni se necesita un sistema en tiempo de ejecución. Ahora es el kernel el que almacena una tabla de hilos con el estado de cada uno. Para gestionarlos hacen falta llamadas al sistema.

Ahora, cuando un hilo se bloquea, es el kernel el que decide si se ejecuta otro hilo del mismo proceso o se ejecuta un hilo de otro proceso. Ya no se necesitan llamadas al sistema sin bloqueo.

Pero tiene una desventaja: es más lento debido a que las llamadas al sistema son más costosas. Además, se le obliga al SO a resolver algunos problemas, como cuántos hilos tiene un proceso creado con fork por otro proceso con hilos o qué hilo atiende las señales.

Implementaciones híbridas

Es un intento de combinar las ventajas de los hilos en el espacio de usuario con las de los hilos en el espacio de núcleo. Lo que se implementa es un sistema de hilos de núcleo que se multiplexan en varios hilos de usuario. Es decisión del programador el cuántos hilos de cada tipo utiliza. Lógicamente, el SO solo será consciente de los hilos en el espacio de núcleo.

Hilos emergentes (pop-up thread)

Se suelen utilizar frecuentemente en sistemas distribuidos (sistemas con varias computadoras separadas y conectadas entre sí por una red de comunicaciones que permiten que el usuario acceda a los recursos todas las máquinas desde una sola).

El método tradicional es que un proceso esté bloqueado en una llamada al sistema receive esperando un mensaje entrante y cuando este llega lo procesa y lo maneja. Sin embargo, es posible hacer que la llegada del mensaje cree un nuevo hilo para manejar el mensaje. Se trata de un hilo emergente.

Hilos en Linux

Linux es un SO multihilo que es capaz de disolver la distinción entre procesos e hilos con la llamada al sistema clone. Esta llamada permite elegir qué se comparte y qué se mantiene en privado en la creación de cada hilo/proceso mediante el argumento flags. Es capaz de crear un nuevo hilo en este proceso o en uno nuevo, comenzando siempre su ejecución en la función fn y teniendo su propia pila de ejecución (child_stack).

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg, ... );
```

Planificación

Siempre hay varios procesos e hilos (los que están en estado listo) que compiten por la CPU. Pero para decidir cuál se ejecuta es necesario un planificador de procesos que siga algún algoritmo específico. Para los PCs con pocos procesos esta planificación es poco importante, pero en los servidores es imprescindible asegurar un uso eficiente de la CPU, permitiendo que todos los procesos realicen sus cálculos y teniendo en cuenta que la conmutación entre procesos es cara.

Los procesos normalmente alternan ráfagas de cálculo con peticiones de E/S. Aquí encontramos dos tipos de procesos: los limitados a cálculos, con ráfagas de CPU largas y esperas infrecuentes por E/S; y los procesos limitados a E/S, con ráfagas de CPU cortas. En este caso, el factor clave es la duración de la ráfaga de CPU.

Por tanto, si un proceso limitado a E/S quiere ejecutarse, debe obtener rápidamente la CPU. Por ejemplo, si los procesos tiene 80% de E/S, debe haber 10 procesos en memoria para que el desperdicio de la CPU no sea mayor del 10%. En cambio, si tienen el 20%, basta con dos procesos en memoria para superar este uso.

Hay una serie de preguntas que el planificador debe responder en un tiempo mínimo para pasarle el control de la CPU al siguiente hilo:

- Cuando se crea un nuevo proceso, se ejecuta el proceso padre o el hijo?
- Si un proceso termina, qué proceso listo se ejecuta?
    - En caso de que no hay ningún proceso listo, se ejecuta un proceso inactivo.
- Si un proceso se bloquea, se tiene en cuenta la razón?
- Llega una interrupción de E/S, deberíamos ejecutar el proceso que estaba esperando?
- Hay una interrupción de reloj, cambiamos de proceso?

Existen dos categorías de algoritmos de planificación:

- No apropiativo: selecciona un proceso y se ejecuta hasta que se bloquea.
- Apropiativo: selecciona un proceso y ejecuta un tiempo máximo fijado.

Los algoritmos de planificación son los siguientes:

Procesamiento por lotes

No hay procesos que esperen para obtener una respuesta rápida. Pueden ser apropiativos o no apropiativos, con largos períodos para cada proceso. Hay variantes:

- El **primero en entrar** siempre es el primero en ser atendido.
    - Un proceso se ejecuta durante todo el tiempo que necesite o durante períodos largos.
    - Los otros procesos se ponen en la cola, y se ejecuta el primero en cuanto el otro se bloquea.
    - Es muy fácil de implementar, ya que solo es necesario una cola con los procesos listos.
    - Es muy ineficiente si hay procesos con muchas operaciones de E/S y poco cálculo.
- El trabajo **más corto** primero.
    - Los tiempos de ejecución se conocen de antemano.
    - Sólo es óptimo en términos de tiempo de respuesta si todos los trabajos están disponibles al mismo tiempo (si no lo están y se empiezan a ejecutar los de mayor duración antes que los de menor, es muy ineficiente).
- El de **menor tiempo restante** a continuación.
    - Los tiempos de ejecución se conocen de antemano.

- Cuando llega un nuevo proceso, va a tardar menos de lo que le queda al que está en ejecución, entra ese a funcionar (es apropiativo).
- Da muy buen servicio a los procesos cortos.

Interactivos

Evitan que un proceso acapare la CPU. Son siempre apropiativos.

- Planificación por **turno circular** (round-robin).
  - A cada proceso en la cola de listos se le asigna un **quantum** (duración de tiempo).
  - Cuando se termina el quantum o el proceso se bloquea, el proceso vuelve al final de la cola, es decir, hay un **cambio de contexto**, lo cual lleva tiempo y desperdicia tiempo de la CPU.
  - Es un aspecto clave la duración del quantum.
    - Si es demasiado corto, hay demasiados cambios de contexto.
    - Si es demasiado grande, hay mayor tiempo de espera con muchos procesos, éstos se bloquean antes de terminar el quantum y da mala respuesta a peticiones interactivas cortas.
    - Es conveniente fijarlo entre 20 ms y 50 ms.
- El de **mayor prioridad** antes.
  - Se debe evitar que los procesos con alta prioridad copen la CPU (reduciendo prioridades o estableciendo quantums).
  - Las prioridades se pueden asignar dinámicamente.
    - Los procesos limitados a E/S deben recibir la CPU inmediatamente.
    - Dar prioridad = 1/f (f = fracción del último quantum utilizado) es buena opción.
  - Se pueden agrupar los procesos en clases de prioridad (ajustadas dinámicamente), con planificación de turno circular en cada clase.
    - Si hay procesos de máxima prioridad, se van seleccionando por turnos.
    - En cuanto se terminan, se salta a los de prioridad siguiente.
- Planificación por **partes equitativas**: Se asigna una fracción de la CPU a cada usuario, que puede depender del número de procesos que tenga.
- Proceso más corto a continuación: Adaptación de "El más corto primero" del procesamiento por lotes. La estimación del tiempo de ejecución se hace en base a las anteriores →
  - $T = (T_{estimado} + T_{ult.\ Ejec.}) / 2$
- Planificación **garantizada**: con n usuarios, cada usuario recibe 1/n de tiempo de CPU. El SO debe registrar el tiempo de CPU recibido por cada usuario (puede hacerse con procesos).
- Planificación por **sorteo**: se dan boletos y se sortea quién puede ir. Los procesos reciben diferente número de boletos. Los que tienen papás ricos y son caprichosos siempre tienen más boletos ¬¬

Tiempo real

Se bloquean con rapidez. El proceso debe reaccionar ante un estímulo en un tiempo fijo. Pueden ser

- Tiempo real duro: hay tiempos límite que se deben cumplir sí o sí.
- Tiempo real suave: no es conveniente fallar, pero es tolerable hasta un máximo.

Normalmente los programas tienen procesos con comportamiento predecible y tiempo de vida mucho menor a 1 segundo.

Los eventos a los que puede responder un sistema de tiempo real pueden ser aperiódicos (ocurren de manera impredecible) o periódicos.

Estos algoritmos pueden ser dinámicos (hacen sus decisiones en tiempo de ejecución) o estáticos (toman sus decisiones antes de que el sistema empiece a ejecutarse porque tienen información perfecta disponible de antemano).

Gestión de la memoria

Siempre se desea una memoria privada grande, rápida y que no pierda su contenido. La manera de conseguir esto es la jerarquía de memoria.

Administrador de la memoria: parte del SO que administra parte de la jerarquía de memoria, ve qué partes están en uso, asigna memoria a los procesos y libera memoria de los procesos terminados. Sobre todo interesa la parte de memoria principal.

Abstracción de memoria

Sin abstracción de memoria

Con este sistema, el programa ve toda la memoria física y tiene un rango de direcciones de 0 a un valor máximo. Es difícil la ejecución de varios programas ejecutándose, ya que comparten la memoria física.

Es posible ejecutar varios procesos, pero solo puede haber uno en memoria, y el SO, para cambiar entre ellos, intergambia todo el contenido de la memoria en disco.

Sin intercambio, la memoria estaba dividida en bloques de 2 KB y a cada uno se le asignaba una llave de protección de 4 bits, guardada en registros especiales dentro de la CPU. Un equipo con una memoria de 1 MB sólo necesitaba 512 de estos registros de 4 bits para totalizar 256 bytes de almacenamiento de la llave.

Una gran desventaja era el uso de direcciones absolutas y era necesario hacer reubicaciones estáticas (actualizar las referencias a memoria al cargar el proceso), lo cual era muy lento.

Solo los sistemas empotrados siguen haciendo esta gestión de la memoria.

Espacio de Direcciones

Al haber varias aplicaciones en memoria es necesaria la protección de esta y su reubicación. Cada proceso se abstrae y tiene su propio espacio de direcciones.

El espacio de direcciones es el conjunto de direcciones que utiliza un proceso.

Estos espacios de direcciones son independientes. La dirección 28 del proceso A es una ubicación física distinta del proceso B. Para lograr esto se emplea la **reubicación dinámica**.

Con este sistema se establecen unos registros base y límite. El registro base es la dirección donde empieza la memoria asignada al proceso, y el límite es el último. En cada referencia de memoria, el hardware suma el valor base a la dirección para acceder a la memoria, comprobando antes que no se pase del límite.

Técnicas frente a falta de memoria

Cuando la memoria que necesitan los procesos es mayor que la memoria disponible, se usan dos sistemas: el intercambio y la memoria virtual.

Intercambio

Consiste en llevar los procesos completos a memoria, ejecutarlos durante un tiempo y devolverlos a disco (en linux, área de intercambio, Swap). Los procesos inactivos quedan en disco y no consumen memoria. Los procesos que se activan siempre se cargan en zonas vacías de la memoria, que no tienen por qué coincidir con la zona que habían tenido anteriormente (hacen uso de la reubicación dinámica) y puede ser que necesiten compactar la memoria para juntar todos los huecos y hacer uno más grande (operación costosa y poco frecuente).

Una cuestión a resolver es la cantidad de memoria que se debe asignar a un proceso. Los procesos de tamaño fijo implican una asignación sencilla, pero cuando el proceso tiene un tamaño de memoria variable, debe crecer ocupando más memoria:

- Ocupar memoria adyacente. Si no...
- Mover el proceso a otra zona de memoria. Si no...
- Intercambiar procesos para crear un hueco. Si no...
- Suspender el proceso.

Para estas operaciones se pueden usar dos formas de llevar el registro de uso de la memoria: el uso de mapas de bits y el uso de listas ligadas.

Un **mapa de bits** está formando palabras de n bits. Para buscar memoria libre debemos recorrerla en busca de n bits consecutivos con los valores a 0. La lentitud que implica este proceso es un argumento irrebatible contra los mapas de bits. Un sistema mejor que este es el uso de listas ligadas.

Listas ligadas

Consiste en mantener una lista ligada de segmentos de memoria asignados y libres. Cada nodo de la lista especifica si es un proceso o un hueco, dónde empieza, la longitud de ese bloque y un puntero al siguiente nodo. De esta manera podemos tener una lista fácilmente actualizable. Cuando un proceso termina, si tiene procesos como vecinos se forma un hueco, pero si no, se fusionan regiones y se forma un hueco más grande.

Existen diversos algoritmos para elegir los bloques de memoria que asignar a los procesos:

- Primer ajuste: exploración de la lista hasta encontrar un hueco (muy rápido)
- Siguiente ajuste: la exploración comienza desde donde se encontró el hueco anterior (peor que el primer ajuste)
- Mejor ajuste: explora toda la lista y selecciona el hueco más pequeño (es más lento y genera huecos pequeños e inutilizables)
- Peor ajuste: explora toda la lista y selecciona el hueco más grande.
- Listas separadas de procesos y huecos: se busca solo en la lista de huecos, pero hace la liberación de memoria más lenta. Si la lista de huecos está ordenada por tamaño, se acelera el mejor ajuste.
- Ajuste rápido: listas separadas para los tamaños más solicitados (búsqueda muy rápida)

Debido a que el intercambio en sistemas multiprogramados genera problemas (espacios de

direcciones demasiado grandes, necesidad de hacer programas particionados, lentitud al cambiar entre procesos...), se encontró la solución en la memoria virtual. El espacio de direcciones se divide en páginas (no todas las páginas tienen que estar en memoria) y si falta una, ocurre un fallo de página. La memoria virtual es así:

Memoria Virtual

Para la paginación, el programa emplea direcciones virtuales, teniendo un espacio de direcciones virtuales (un espacio de direcciones virtual por proceso). La **MMU (Memory Management Unit)** asocia direcciones virtuales a direcciones físicas antes de ponerlas en el bus. Así, el proceso trabaja con direcciones virtuales y cuando necesita acceder a ellas se traducen.

Página: bloque de memoria de N bytes.

Marco de página: bloque de memoria principal del tamaño de una página.

Fallo de página: intento de acceso a una página cuando esta no se encuentra en memoria. Exige traer la página desde disco hasta memoria principal.

Un ejemplo: para una máquina con 32KB de memoria principal podemos hacer funcionar programas que requieren 64KB. Si dividimos la memoria principal en bloques de 4KB (**marcos de página**) tendremos 8 bloques en total. En teoría no podríamos ejecutar tal programa, pero ahora empieza el juego de la memoria virtual: la MMU nos ofrece 16 **páginas** (bloques de 4KB), y cuando queremos escribir en uno de ellas, la crea en uno de los marcos. En caso de que ya existiese, si estaba en memoria principal simplemente traduciría la dirección, y si no (**fallo de página**), seleccionaría algún marco y copiaría la página desde el disco al marco seleccionado. Así tendríamos una memoria de 32KB pero podríamos usar páginas (o bloques de memoria) que sumasen en total 64KB.

Una dirección virtual por tanto se compondrá de un número de página y el desplazamiento dentro de ésta. La **tabla de páginas (TP)** de la MMU tendrá un listado con el mismo número de nodos que de páginas haya. Cada uno de esos nodos contendrá:

- Obligatoriamente, un bit de si la página está presente en memoria y su número del marco de página en el caso de que esté en memoria.
- Conjunto de bits de permisos concedidos en la página (r,w,x)
- Bit sucio: evita reescribir en disco una página si no ha sido modificada
- Bit de referenciada: ayuda para el reemplazo de páginas (si se han referido a ella)
- Bit de deshabilitación de uso de cache: importante para las páginas asociadas con registros de dispositivos, para no usarlas en caso de que haya la posibilidad de que el registro del dispositivo esté más actualizado.

Mejora de la paginación

- Cómo hacer rápida la traducción de direcciones teniendo en cuenta que por instrucción pueden necesitarse varias traducciones
  - TLB (explicado a continuación)
- Con un espacio de direcciones grande, la tabla de páginas también será grande. Con direcciones de 32/64 bits, 1 tabla por proceso en memoria principal

- Una única tabla de páginas en registros: sistema simple, se carga la tabla de páginas con la copia en memoria
- Tabla de páginas en memoria principal, con un registro que apunte al inicio de la TP, lo que hace fácil cambiar la TP al conmutar el proceso (con varias referencias a memoria se degrada el rendimiento)

Búfer de Traducción Adelantada (TLB)

Cada referencia a la memoria implica un acceso a la TP y vemos que hay un gran número de referencias a un pequeño número de páginas, por lo que solo se lee frecuentemente una pequeña parte de las entradas de la TP. Podemos colocar en la MMU un Búfer de Traducción Adelantada (TLB), que usa hardware para traducir la dirección virtual a dirección física sin usar la tabla de páginas. Tiene una memoria caché de la tabla de páginas con menos de 64 entradas.

Cada vez que se genera una dirección virtual, si está en la TLB se obtiene el marco de página, y si no está, se da un fallo de TLB: se busca en la tabla de páginas (puede darse un fallo de página) y se reemplaza una entrada de la TLB. Esta TLB puede administrarse por software.

Tablas de páginas multinivel

Cuando tenemos espacios de direcciones grandes, para no tardar mucho en buscar las páginas, puede ser mejor tener una tabla de páginas con referencias a bloques de 4MB (por ejemplo) que contengan otras tablas de página (de 2° nivel). Esto hace que no sea necesario mantener todas las tablas en memoria. Además, se puede expandir a más niveles (con más de 3 niveles implica demasiada complejidad adicional).

Así, con estas tablas de páginas, cada dirección virtual de 32 bits tendrá una parte correspondiente a en qué tabla de páginas de 2° nivel está, otra parte correspondiente a qué pagina corresponde y una última parte que especifica qué posición dentro de la página ocupa.

Tabla de Páginas Invertida (TPI)

Con espacios de direcciones de 64 bits las tablas de páginas son demasiado grandes aún con implementación multinivel. Una solución puede ser el uso de una TPI: una tabla con una entrada por cada marco de página de memoria física, y en cada nodo la especificación de qué proceso y qué página virtual está en en el marco. La traducción es más lenta, pues busca en toda la tabla de páginas en cada referencia a memoria, pero se puede mejorar con TLB y una tabla hash.

Reemplazo de páginas

Cuando ocurre un fallo de página, el SO debe elegir una página para desalojarla (interesa que no se use frecuentemente) y actualizar su copia en disco si fue modificada. Se suele dar un problema similar en la memoria cache y en los servidores web. En este caso, la gran cuestión es si la página desalojada tiene que ser del mismo proceso o de otro (si es del mismo, cada proceso debe tener un número fijo de páginas en memoria). Los algoritmos propuestos son:

- **Algoritmo óptimo**: se reemplaza la página que se vaya a referenciar más tarde (para posponer el fallo lo máximo posible). El problema es que no se puede implementar, ya que no se sabe cuándo será la próxima referencia a cada página. Se usa para comparar con otros.

- **No Usadas Recientemente (NRU)**: cada página tiene dos bits de estado, el de Referencia (R) y el de Modificada (M). Estos bits se almacenan en cada referencia a memoria y permanecen en 1 hasta que el SO los borra. El bit R se borra en cada interrupción de reloj.
    - Así, siempre tiene cuatro clases de páginas (RM → 00,01,10,11), y cuando hay un fallo de página, se reemplaza la de menor numeración. Es una implementación sencilla.
- Listas de páginas: algoritmos cuando el SO mantiene una cola de páginas en memoria.
    - **FIFO**: se reemplaza la primera página de la cola (podría ser de uso frecuente)
    - **Segunda oportunidad**: se inspecciona el R de la más antigua. Si R=1, la página va al final de la cola y se borra R (evita el desalojo de páginas usadas recientemente). Si todos R=1, FIFO.
    - **Reloj**: FIFO con lista circular.
- **Menos Recientemente Usada (LRU**): parte de que las páginas usadas en las últimas instrucciones se volverán a utilizar en las siguientes. Si llevan mucho sin usarse, seguirán así.
    - Se implementa con una lista de páginas ordenadas por uso que se actualiza en cada referencia a memoria. Es difícil de implementar.
    - Implementación en hardware: un contador (C) que se incrementa en cada instrucción y que cada vez que se hace una traducción, ya que pasamos por la TP, se almacena C en la entrada correspondiente a la página referenciada. Cuando hay un fallo de página, buscamos la página que tenga el menor valor de C.
    - Otra implementación hardware: para N marcos de página, tenemos una matriz NxN en la que, por cada referencia a la página en el marco k, ponemos la fila k a todo 1 y después la columna k a todo 0. Siempre reemplazamos la página de la fila con menor valor.
- **No Usada Frecuentemente (NFU)**: es una simulación de LRU que tiene un contador asociado (C) a cada página y que en cada interrupción de reloj el SO le suma R (C lleva la cuenta aproximada de la frecuencia). Cuando hay un fallo de página, se reemplaza por la página con menor contador (no tiene en cuenta el tiempo de la página sin usarse)
- **Envejecimiento (NFU modificado)**: cada contador se desplaza un 1 bit a la derecha antes de agregar R y R se agrega en el Most Significant Bit (MSB). Se elimina la página con menor C.
- Otras opciones: **conjuntos de trabajo y WSClock**

Conjunto de trabajo (WS)

Paginación bajo demanda: los procesos se inician sin ninguna de sus páginas en memoria y se van cargando cuando se necesitan.

Localidad de referencia: fenómeno que implica que los programas no hacen referencia a todos los datos a la vez, sino que van utilizando fracciones de sus páginas.

Conjunto de trabajo (WS): es el conjunto de páginas que usa un proceso en un momento dado. Cuando éste está en memoria, se dan muy pocos fallos, pero si no está, se dan muchos fallos, lo cual implica sobrepaginación (demasiadas páginas en memoria) y ejecución lenta. El WS depende

del tamaño de memoria y de la aplicación.

Prepaginación

Cuando el SO trae un proceso a memoria y lo ejecuta, se producen fallos de página hasta que se carga el nuevo WS, por lo que se desperdicia tiempo de CPU dedicándose a procesar fallos de página. La solución es la prepaginación: tener el WS en memoria antes de comienza a ejecutarse. Esto requiere conocer bien el WS de cada proceso e implica una dificultad: el WS cambia con el tiempo.

Las referencias a memoria se agrupan en un número pequeño de páginas, y en cada instante se usa un número reducido de páginas. Además, con paginación bajo demanda el número de fallos de página es grande al principio de la ejecución, pero después se reduce considerablemente.

Dado que el WS varía lentamente con el tiempo, se puede predecir el WS en función del WS cuando se detuvo el proceso, lo que facilita la prepaginación.

Fallos de página

Para decidir el WS de un proceso, podríamos tener un vector (registro de desplazamiento) en el que almacenar las páginas que se van referenciando, y las k primeras serían el WS. Con cada fallo, reemplazaríamos una página que no estuviese en WS.

Actualizar ese registro de desplazamiento con números de página del WS sería muy costoso, por lo que no se utiliza. En cambio, podemos redefinir el WS: serán las páginas usadas durante los últimos T segundos de ejecución del proceso (se le llama tiempo virtual al hecho de contar los segundos de ejecución en lugar de contarlos en tiempo real).

Con cada fallo de página, buscaríamos una página que no esté en WS, examinando su bit R. Si R=1, actualizamos el tiempo de último uso con el tiempo virtual actual (la página estuvo en uso cuando el fallo de página así que no debe reemplazarse). Si R=0 es candidata al reemplazo, y si no está en WS, se reemplaza. En caso de que todas las páginas estén en WS, se desaloja la más antigua.

Algoritmo de reemplazo WSClock

Es más sencillo que el WS, pues tiene una lista circular de marcos de página. Cada nodo tiene un tiempo de último uso y los bits R y M.

- Si R=1, se borra R y se avanza.
- Si R=0, edad>T y página limpia, esa página no está en WS y hay copia en disco, así que se reemplaza.
- Si la página está sucia no se reclama de inmediato (se conmuta el proceso para dar tiempo a que se realice la copia), se planifica la escritura en disco, se avanza y se explora la siguiente entrada.

Diseño de la Memoria Virtual

Asignación Local o Global

Cuando ocurre un fallo de página hay que decidir si se reemplaza una página del mismo proceso que

ha producido e fallo o de otro. Puede usarse un algoritmo local, asignando una fracción fija de la memoria a cada proceso, o un algoritmo global, asignando dinámicamente los marcos de página entre procesos. La gran desventaja del algoritmo local es que puede producirse sobrepaginación si el WS crece o desperdiciarse memoria si el WS disminuye.

Asignación Global

Es un sistema proporcional al tamaño del proceso, que garantiza siempre un mínimo a procesos pequeños y cambia de forma dinámica al ejecutarse procesos. Para este sistema se usa el algoritmo PFF (Page Fault Frecuency), que controla el tamaño del conjunto de marcos de página asignados y en función de si hay muchos o pocos fallos, indica cuánto debe aumentar o disminuir la asignación de tamaño en número de marcos de un proceso.

Algoritmos de reemplazo

Es una cuestión importante, ya que muchos algoritmos sirven para asignación global o local, pero los algoritmos WS y WSClock solo tienen sentido en una estrategia local, ya que no hay un WS para todo el computador.

Control de la carga

Debemos intentar evitar siempre la sobrepaginación. Puede darse que los WS de todos los procesos sean mayores que la memoria principal o que con el algoritmo PFF todos los procesos necesiten más memoria pero ninguno necesite menos. La solución puede ser intercambiar los procesos a memoria y liberar sus marcos de página para asignarlos a procesos que sobrepaginan y repetir esto hasta detener la sobrepaginación.

Tamaño de página

Puede ser elegido por el SO. Intervienen varios factores competitivos que hacen que no haya un tamaño de página óptimo. En caso de una página pequeña hay menor fragmentación dentro de cada página, pero con un tamaño más grande conseguimos una tabla de páginas más pequeña. Hay que tener en cuenta que el tiempo de transferencia de una página grande es similar al de una pequeña.

Espacios de instrucciones y datos separados

Un espacio de direcciones, si es pequeño no caben instrucciones y datos. Si usamos espacios separados, ambos espacios paginan de forma independiente, duplican el espacio de direcciones y crean tablas de páginas con traducción de direcciones independientes. Normalmente, cuando se separan espacios, se crea uno para la implementación del programa (espacio I) y otro para los datos (espacio D). Además, como el espacio I es de solo lectura, nunca se modifica, por lo que puede ser una buena página para ser sustituida.

Páginas compartidas

Si varios usuarios ejecutan el mismo programa puede ser útil compartir páginas en vez de tener varias copias de lo mismo en memoria. Es fácil compartir las páginas de espacio I, que son readonly. Así, cada proceso, en su tabla de páginas tendrá un puntero al espacio I del programa que ejecuta (que no tiene que ser solo para él) y un puntero al espacio D, con sus datos de ejecución.

Bibliotecas y archivos compartidos

Hay bibliotecas extensas utilizadas por muchos procesos y que ocupan varias páginas. Para su uso podemos tener enlaces estáticos, haciendo que las funciones se incorporen al espacio de direcciones de cada proceso, evitando cargar las páginas con funciones que no se utilizan pero haciendo que si muchos programas usan las mismas funciones se desperdicie memoria copiándolas; o podemos tener enlaces dinámicos (DLL), es decir, bibliotecas compartidas, haciendo que el programa incluya una rutina que enlaza la función en tiempo de ejecución y que la biblioteca se cargue cuando se le llama por primera vez y evitando tener que copiarse a memoria varias veces si la necesitan más procesos.

Un proceso puede asociar también un archivo a una porción de su espacio de direcciones virtuales, que se carga a medida que se demandan sus páginas. O también, otro modelo de E/S sería acceder al archivo como un gran array de datos en memoria. Si varios procesos asocian el mismo archivo estarían accediendo a una memoria compartida, y éste podría servir de canal de comunicación entre procesos.

Política de limpieza

La paginación siempre funciona mejor si hay marcos de página libres y listos para usarse. Podemos tener un mecanismo, demonio de paginación, que nos asegura un conjunto abundante de marcos de páginas libres, desalojando páginas (grabándolas en disco si es necesario) si hay pocos marcos libres e intentando asegurar que no se necesite escribir en disco cuando requiere un marco de página.

Implementación de la Memoria Virtual

Participación del SO en la paginación

- Creación del proceso
  - Determinar el tamaño del espacio de direcciones
  - Crear la TP
  - Asignación e iniciación del área de intercambio
  - Actualización de la TP
- Planificación del proceso
  - Actualización de TP y TLB al nuevo proceso
  - Traer páginas a memoria para evitar fallos de página (sobre todo cuando el PC apunte a una página, al reanudar la ejecución, no debería haber fallo de página)
- Fallo de página
  - Determinar qué dirección virtual originó el fallo
  - Determinar qué página se necesita y colocarla en memoria
  - Ejecutar la instrucción de nuevo
- Terminación del proceso: liberar la memoria y la TP

Gestión de fallos de página

Cuando ocurre, se le hace un trap al núcleo y se guarda el PC en la pila y el estado de la instrucción. Se usa una rutina para guardar los registros y luego se determina qué página se necesita (análisis de la instrucción apuntada por PC). Si la dirección analizada es válida, se comprueba si hay marcos de página (si no hay, suspendemos nuestro proceso, se ejecuta el algoritmo de reemplazo, copiamos a disco si está sucio y marcamos el marco como ocupado). Cuando tenemos un marco disponible, traemos la página a memoria, con el proceso aún suspendido, actualizamos la TP y planificamos la vuelta del proceso a ejecución.

Respaldo de instrucción

Cuando volvemos de un cambio de contexto, debemos reiniciar la instrucción que originó el fallo, siendo necesario saber dónde empieza este (puede que una instrucción haga varios accesos a memoria y cualquiera pueda fallar y el SO no sabe si la dirección que origina el fallo es una instrucción o no).

Algunos de los modos de direccionamiento modifican los registros antes de hacer la referencia a memoria, lo cual puede ser un problema para determinar el estado anterior. Para eso, la mayoría de CPUs guardan el PC de la instrucción en un registro especial e indican en otro qué registros han sido modificados.

Bloqueo de páginas en memoria

La memoria virtual y la E/S interaccionan. Con operaciones de E/S se utiliza un búfer en el espacio de direcciones del proceso en el que se coloca lo que se lee o se recibe. Cuando el proceso se suspende para que se complete la E/S puede ocurrir que se planifique otro proceso con fallos de página y que se reemplace la página con el búfer de E/S. La única solución es bloquear las páginas que tienen búfers para dispositivos de E/S.

Almacén de respaldo

Las páginas, cuando se guardan en disco, hay que meterlas en algún lado. Para eso existe la partición de intercambio del disco: con una organización diferente al sistema de archivos y con un acceso más rápido que el sistema de archivos convencional.

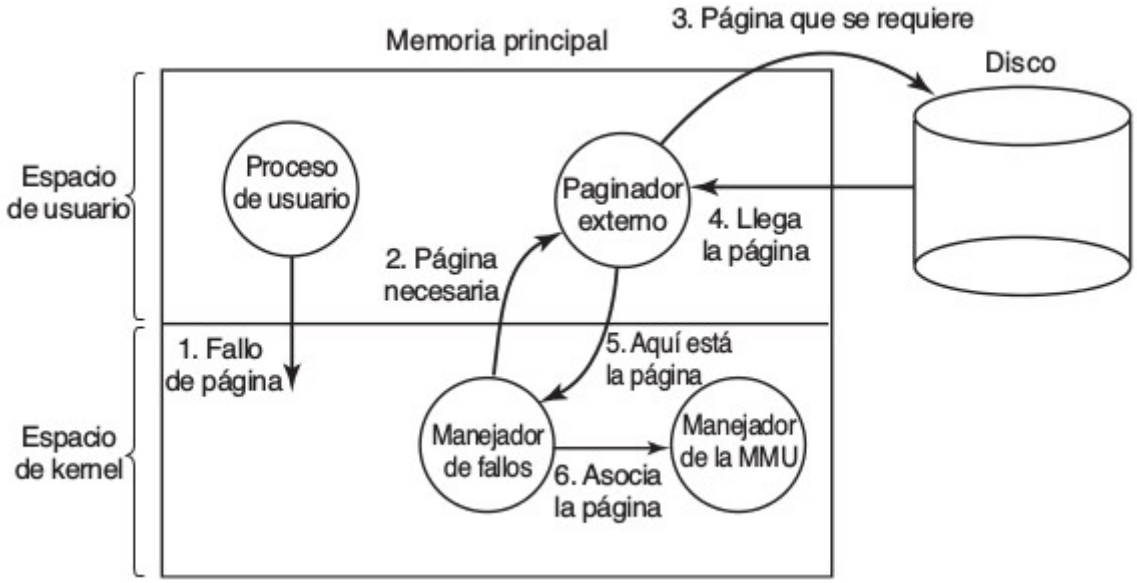
Inicialmente, el área de intercambio (en linux, partición swap) está vacía y cuando se carga un proceso se reserva un trozo de partición con un tamaño equivalente al del proceso. La dirección de este trozo reservado quedará grabada en su TP, y para acceder a una página en concreto será tan sencillo como sumar el desplazamiento de la página a la dirección.

Para asignar el espacio en disco hay que tener en cuenta que los procesos pueden cambiar de tamaño. Existen dos métodos:

- Estático: copia la imagen del proceso (espacio de direcciones entero) en el disco, haciendo que la página en memoria siempre tenga una copia sombra en disco (puede estar obsoleta)
- Dinámico: solo se copian en disco las páginas que se desalojan de la memoria principal, por lo que no tienen direcciones fijas en disco y las páginas en memoria pueden no tener copia.

Separación de política y mecanismo

Esta separación permite gestionar sistemas complejos. Se aplica a la administración de la memoria. Parte de la gestión de memoria se ejecuta como usuario. Este paginador externo tiene un mapa de páginas del proceso, hace el almacenamiento de respaldo y lee las páginas del disco.



Segmentación

Cuando tenemos un espacio de direcciones único para un proceso estamos usando un modelo unidimensional. El espacio asignado a una parte del proceso puede crecer y llenarse, por lo que puede ser mejor tener varios espacios de direcciones separados (segmentos) en los que guardamos distintas cosas (texto, tablas de símbolos, pila...). Así se libera al programador de la administración de la expansión de zonas en memoria.

Los segmentos son espacios de direcciones independientes con una secuencia lineal de direcciones. Distintos segmentos pueden tener diferentes longitudes, que pueden cambiar durante la ejecución. Éstos crecen o se reducen sin afectar a los otros. La dirección se expresa especificando el segmento y la dirección dentro de éste.

Facilitan el compartir bibliotecas, procedimientos... Cada segmento tiene su tipo de protección y el usuario es consciente de lo que hay en cada segmento.

Diferencias con paginación

- El programador necesita ser consciente de que se usa esta técnica.
- Hay muchos muchos espacios de direcciones lineales
- Los procedimientos y los datos pueden protegerse por separado
- Las tablas pueden acomodarse con facilidad a un tamaño que cambio
- Se facilita la compartición de procedimientos entre procesos

las dos técnicas pueden hacer que el espacio de direcciones total exceda de la memoria física.

Esta técnica se inventó porque la paginación sirve para obtener un espacio de direcciones lineal grande sin tener que aumentar la memoria física y la segmentación para permitir que los programas

y los datos se dividan en espacios de direcciones lógicamente independientes, ayudando a la compartición y protección.

Implementación: los segmentos no tienen tamaño fijo, por lo que son fáciles de implementar. Volvemos al problema de la necesidad de la compactación.

Uso de segmentos con páginas

Como los segmentos extensos no es posible mantenerlos en memoria principal, podemos paginar los segmentos.

Ejemplo: MULTICS, tiene una memoria virtual de 2^{18} segmentos de 65536 palabras (de 16K x 36 bits cada una) y en cada segmento, una memoria virtual paginada. Necesita una tabla de segmentos por proceso, con un descriptor por segmento. Si cualquier página del segmento está en memoria, su tabla de páginas también, y se considera que el segmento está en memoria.

Cuando el segmento está en memoria, el descriptor contiene un puntero a su tabla de páginas, el tamaño del segmento y los bits especiales (protección...)

Por tanto, ahora las direcciones de memoria contienen: segmento, página y dirección dentro de la página. En una referencia a memoria necesitaremos el número de segmento para encontrar el descriptor. Si la TP del segmento está en memoria, se localiza, y si no, fallo de segmento. Cuando se examina la TP, si la página está en memoria se extrae el marco de la página y si no, fallo de página. Después se le añade el desplazamiento al marco de página y se realiza el acceso.

Sistemas de Archivos

Normalmente necesitamos almacenar una cantidad muy grande de información que debe permanecer una vez finalizado el proceso. Además, varios procesos deben poder acceder concurrentemente a la información (que es independiente del proceso). Los discos y otros dispositivos son difíciles de gestionar, por lo que empleamos archivos. Un **archivo** es la abstracción de los dispositivos físicos de almacenamiento. Son unidades lógicas de información creadas por los procesos.

Los procesos pueden leer y crear archivos. Esta información debe ser persistente (no esté afectada por la creación o terminación de un proceso) y el archivo solo desaparece si su propietario lo borra.

Estos archivos estarán administrados por una parte del SO, el sistema de archivos.

Consideraciones generales

Un **disco físico** es un dispositivo de almacenamiento permanente, normalmente con direccionamiento en bloques (con un conjunto fijo de bloques de tamaño fijo) y con un identificador para cada bloque.

El **disco lógico** es la abstracción de cómo el SO ve una secuencia de bloques accesibles aleatoriamente. El **driver** traduce estos números de bloque lógico a números de bloque físico.

Los discos físicos suelen dividirse en **particiones** físicas contiguas, cada una asociada a un disco lógico. El administrador del sistema es el que decide qué contienen las particiones. Estas particiones son las que permiten que coexistan varios SO. La partición activa es la que contiene el SO en el momento de arranque.

Un disco lógico contiene un solo sistema de archivos entero (o el área de intercambio). Éste puede estar formado por varios discos físicos (para soportar sistemas de archivos grandes). La jerarquía de archivos puede tener varios subárboles independientes y cada uno tener un sistema de archivos (como es el caso de las imágenes de sistemas de archivos iso o img).

Montaje

Cada sistema de archivos necesita ser montado en un punto de montaje (en linux, una carpeta que no sea la raíz, /). Esto permite ocultar al usuario los detalles de la organización. El espacio de nombres debe ser homogéneo.

El SO tiene una **Tabla de Montaje** (en linux, en el fichero /etc/fstab), la cual identifica los sistemas de archivos que se deben montar al arrancar. Estas operaciones de montaje se realizan con llamadas al sistema

```
int mount(const char *source, const char *target, unsigned long mountflags);
int umount(const char *target);
```

Punto de vista del Usuario

Todos los archivos tienen los siguientes factores:

- Nombre: compuesto por caracteres, se permiten algunos especiales, tienen extensiones...

- Estructura: son secuencias de bytes sin una estructura fija, por lo que se permite que el usuario se la dé
- Tipos: pueden ser regulares (ASCII o binarios), directorios y archivos especiales (los asignados a E/S y a discos)
- Acceso: puede ser secuencial y aleatorio (seek se usa para establecer la posición de acceso)
- Atributos (metadatos): tienen información adicional sobre el archivo, la protección, su creación, su propietario...

Máscara

Cada archivo tiene asociada una máscara de modo (de 16 bits). Esta indica el tipo de archivo con 4 bits, 3 bits con el UGS, y 9 bits para los permisos en formato (propietario, grupo, otros) y cada uno de estos, con tres bits de (read, write, execution)

El UGS son 3 bits: S_ISUID (si está a 1, cambia el Id del usuario al ejecutar), S_ISGID (si 1, cambia el Id del grupo al ejecutar) y S_ISVTX (sticky bit, significa que solo el propietario puede cambiarle el nombre)

Operaciones

```
int open(const char *pathname, int flags, mode_t mode); //open a file for
reading, writing, or both
int close(int fd); //close an open file
ssize_t read(int fd, void *buf, size_t count); // read data from a file into
a buffer
ssize_t write(int fd, const void *buf, size_t count); //write data from a
buffer into a file
off_t lseek(int fd, off_t offset, int whence); //move a file pointer
int stat(const char *path, struct stat *buf); //get a file's status
information
int chmod(const char *path, mode_t mode); //change mode of a file
int rename(const char *oldpath, const char *newpath); //rename a file
//can also set other atributes, create, delete...
```

Directorios

Son los contenedores de archivos. Se organizan de forma jerárquica formando árboles y agrupando archivos relacionados. La especificación de un archivo se puede hacer desde el directorio raíz (ruta absoluta) o desde el directorio de trabajo (ruta relativa, también con .. y .).

Identificadores de Usuario y Grupo

Cada proceso tiene asociados dos identificadores de usuario y dos de grupo (además del pid):

- UID, EUID (identificador de usuario real e identificador de usuario efectivo)
- GID, EGID (identificador de grupo real e identificador de grupo efectivo)
- Son enteros positivos
- El usuario real es el responsable de la ejecución del proceso, mientras que el efectivo es el que determina el propietario de los ficheros, el acceso a éstos y la comprobación de permisos para el envío de señales. Suelen coincidir.

- El EUID y el UID no coinciden si un usuario ejecuta un programa de otro que tiene el bit S_ISUID=1. Esto hace que en la ejecución del programa, el usuario que ejecuta tenga los mismos permisos que el propietario.
- Existen llamadas al sistema para comprobar los identificadores y para cambiarlos
  - `uid_t getuid(void);`
  - `uid_t geteuid(void);`
  - `uid_t getgid(void);`
  - `uid_t getegid(void);`
  - `int setuid(uid_t uid);`
  - `int setgid(uid_t uid);`
- `salida = setuid(par)`
  - Si no lo ejecuta el superusuario, euid==par si par==(uid del proceso) o el programa tiene su bit S_ISUID==1 y par==(uid del propietario del programa).
  - En caso de error, salida = -1

Implementación

El diseñador debe almacenar archivos y directorios administrando el espacio en disco de forma eficiente y fiable.

Distribución y estructura del disco

El disco se divide en particiones con un sistema de archivos en cada una. El sector 0 del disco o MRB (Master Boot Record) se usa para el arranque de la computadora. Contiene la tabla de particiones, con las direcciones de inicio y fin de cada una, y una de esas particiones se marca como activa.

En cada arranque, la BIOS lee y ejecuta el MBR, el cual localiza la participación activa, lee su primer bloque (bloque de arranque) y lo ejecuta. El programa en el bloque de arranque carga el SO contenido en esa partición.

Todas las particiones tienen un bloque de arranque. Además, suelen tener un superbloque con parámetros del sistema de archivos (magic number, número de bloques...) que se carga en memoria. También suelen tener información sobre bloques libres, ya sea en un mapa de bits, con una lista de apuntadores o con nodos-i (uno por archivo, con información de cada uno).

Implementación de archivos

Asignación contigua

El archivo se almacena con una serie de bloques de disco contiguos. Es muy fácil de implementar (la ubicación del archivo es la dirección del primer bloque y el número de bloques), tiene buen rendimiento de lectura (sólo se necesita una búsqueda, rotación del disco, para el primer bloque), pero produce fragmentación del disco (o se compacta, muy costoso, o se reutilizan los espacios, difícil de implementar puesto que se debe conocer el tamaño del archivo de antemano).

Este sistema se suele usar para CD-ROMs, en los que se conoce el tamaño de los archivos.

Asignación de lista enlazada

Consta de una lista enlazada de bloques de disco, en la que la primera palabra del bloque es un puntero al siguiente bloque. Tiene la ventaja de que no hay fragmentación y solo hay que buscar el primer bloque para localizarlo, pero el acceso es muy lento (para llegar al bloque n, hay que leer n-1 bloques antes) y la cantidad de almacenamiento por bloque no es potencia de 2.

Las desventajas se pueden eliminar con una tabla de apuntadores (sistema FAT, File Allocation Table). Esto hace que el bloque completo esté disponible para datos y permite un acceso aleatorio más sencillo, ya que la cadena de apuntadores está en memoria y no en disco. Pero tiene muy mala escalabilidad, ya que el tamaño de la tabla es proporcional al del disco, y con un disco de 200GB, bloques de 1KB y 4 bytes/entrada, tendríamos una tabla de 800MB (inmanejable).

Nodos–i

Consiste en un nodo-i asociado a cada archivo y con una lista de atributos y direcciones de disco de los bloques. Sólo tiene ventajas: el nodo-i está en memoria solo cuando el archivo está abierto y ocupa menos que la tabla de asignación de archivos (tamaño proporcional al número máximo de archivos abiertos). Es el sistema utilizado en Unix (ext).

Implementación de directorios

Antes de abrir un archivo el SO localiza la entrada de directorio (usa la ruta suministrada). Esta es una información necesaria para encontrar los bloques de disco (número del primer bloque, número de nodo-i...). Windows almacena los atributos de los archivos en la entrada del directorio, mientras que Unix los almacena en el nodo-i, y el número del nodo en la entrada del directorio.

Los nombres de archivo tienen longitud variable. Para implementarlos, hay dos maneras: reservar 255 caracteres para el nombre en la entrada del directorio (ineficiente), o hacer que no todas las entradas del directorio sean del mismo tamaño. Para esto último, también hay dos posibilidades: poner el nombre junto con sus atributos y compactar cuando queden huecos (es fácil porque está en MP) o poner los nombres todos juntos al final en un montón, con punteros hacia ellos.

Archivos compartidos

Cuando un archivo está compartido y a la vez en dos carpetas, si en una de las dos carpetas se agregan nuevos bloques, los cambios serán visibles para ambos. No es posible tener una copia de las direcciones de disco en cada directorio, ya que si uno de los dos añade datos al archivo, solo estarían visibles en uno de los directorios. Para esto hay dos alternativas: el sistema de nodos-i, haciendo que los dos directorios apunten al mismo nodo-i y sea este el que mantiene la información, o crear un vínculo simbólico, haciendo que uno de los directorios solo tenga la ruta del archivo vinculado.

El nodo-i tiene el problema de que solo existe un directorio propietario del archivo, y que si se aumenta la cuenta de vínculos (referencias al nodo) y después se elimina desde el directorio propietario, no se elimina el nodo-i y éste sigue figurando con el anterior propietario.

El vínculo simbólico tiene el problema de que solo el directorio propietario tiene un apuntador al nodo-i, y cuando el propietario lo elimina, el vínculo falla. Además, supone un gasto adicional de procesamiento para llegar al nodo-i, ya que se necesita un nodo-i adicional para cada vínculo simbólico.

Cuello de botella en tiempo de ejecución

Los cambios en la tecnología han hecho que el cuello de botella ahora sea el tiempo de acceso al sistema de archivos: leer un bloque en disco requiere ms y en memoria principal, μs. Con CPUs más rápidas, memorias RAM y de disco más grandes, el tiempo de búsqueda en disco sigue siendo el mismo. Para lidiar con esto, tenemos caches de disco (también cada vez más grandes).

La caché de disco almacena búferes de memoria del tamaño de un bloque de disco, en los que se guardan los bloques accedidos recientemente. Si se requiere un bloque, primero se guarda en la caché de disco. Si no está, se busca en el sistema de archivos y se reemplaza un bloque de la cache.

Con la cache de disco, la mayoría de los accesos a disco serán para escrituras, por lo que habrá que actualizar el disco con la copia de la cache. De todas formas, no siempre es eficiente hacer la escritura en el disco en cuanto se requiere, ya que escrituras en trozos muy pequeños puede hacer que sea más costoso buscar que escribir. Estas escrituras se pueden retrasar, pero se expone al sistema a inconsistencias graves si falla algo (las escrituras de nodos-i se realizan de inmediato).

Estructura

Por registros

LFS, Log-Structured File System: intenta alcanzar el ancho de banda completo del disco

Estructura el disco como un registro, haciendo que las escrituras pendientes se almacenan en un búfer en memoria y se escriban en disco como un sólo segmento (con nodos-i, bloques de datos, bloques de directorio) al final del registro. Si el segmento tiene el tamaño adecuado, se alcanza el ancho de banda del disco. Cada segmento incluye información sobre lo que contiene y los nodos-i siguen existiendo, pero no están en posiciones fijas del disco, lo que hace la búsqueda más compleja, requiriendo un mapa de nodos-i. Es necesario realizar periódicamente la compactación del disco.

Por bitácora

JFS, Journaling File Systems; NFTS de Microsoft y ext4 de Linux.

Ofrece robustez: permite mantener un registro con los pasos que debe seguir el sistema de archivos para una cierta operación antes de hacerla. Si falla, acude al registro para saber los pasos para terminar. En cada operación, hay que escribir antes en disco un registro con las operaciones, en el cual, a cada paso, se marca como completado. Este registro se borra al terminar la operación.

Virtuales (VFS)

Dentro del ordenador puede haber sistemas de archivos diferentes en uso, incluso para un mismo SO. Windows identifica cada uno con una letra diferente y no se integran de modo unificado, usándose la letra para indicar a qué sistema se accede. En Unix se integran varios sistemas en una sola estructura de directorios, pudiendo tener diferentes sistemas de archivos montados con total transparencia.

VFS es la técnica de integrar sistemas de archivos diferentes en una estructura ordenada. Es la forma usada en Unix. Esta técnica permite soportar diferentes tipos de sistemas de archivos locales simultáneamente, tanto de Unix como de no Unix, soportar sistemas de archivos distribuidos y

presentar al usuario una imagen homogénea: una única jerarquía de sistemas de archivos, facilitando la incorporación de nuevos sistemas de archivos.

La idea clave de esta técnica es la abstracción de la parte común de todos los sistemas de archivos. Crea una capa separada que llama a los sistemas de archivos concretos para la administración de los datos.

Para registrar (montar) sistemas de archivos en el VFS simplemente es necesario llevar a memoria la lista de direcciones con las funciones específicas del sistema de archivos concreto. Hay tres objetos clave en el VFS con operaciones asociadas a los sistemas de archivos concretos:

- Directorio
- Superbloque: da información sobre el sistema de archivos montado, incluyendo el nodo raíz.
- Nodo-v: abstracción del nodo virtual, da información sobre el nodo-i y tiene un puntero la tabla de funciones con las operaciones dependientes del sistema de archivos. Para crear un archivo, se crea una entrada en la tabla de descriptores de archivos del proceso que apunta al nuevo nodo-v.

En resumen, el VFS necesita de una tabla de procesos que apunta a descriptores de archivos, que a su vez apuntan a sus nodos-v y que a su vez apuntan a una tabla de punteros a las funciones de sus respectivos sistemas de archivos.

Administración y Optimización

Administración del espacio de disco

Para almacenar un archivo existen dos alternativas: escribirlo en bytes consecutivos en disco (poco eficiente) o dividirlo en bloques no necesariamente contiguos.

Aquí es crucial el tamaño de bloque: si es demasiado grande se desperdicia espacio (los archivos ocupan bloques completos) y si es demasiado pequeño un archivo quedará en más bloques, por lo que el acceso será más lento (requerirá varias búsquedas y rotaciones). Se necesita buscar una solución de compromiso. Se pueden hacer estadísticas del tamaño de los archivos para determinar el tamaño de bloque.

Registro de bloques libres: se puede puede implementar con una lista enlazada (usando bloques para mantenerla) o un mapa de bits (para n bloques en total en el disco, n bits necesarios). En el caso del disco lleno, ocupa mucho menos la lista enlazada.

Si los bloques libres se presentan en series de bloques consecutivos, la lista se simplifica, llevando la cuenta de series de bloques libres en lugar de bloques individuales. Contendría por cada nodo el número del primer bloque de la serie y el número de bloques libres consecutivos, por lo que para un disco vacío solo necesitaría dos números. En caso de un disco fragmentado, es muy poco eficiente el registro de la serie de bloques individuales. Tampoco es necesario que esté toda la lista en memoria, llega con un bloque.

Cuotas de disco: evitan que los usuarios ocupen mucho espacio en disco. Cada usuario tiene asignado un máximo de archivos y bloques. Para gestionar las cuotas se necesita una tabla de archivos abiertos (en memoria principal), con atributos (propietario), direcciones a disco y un apuntador a la siguiente tabla; y una tabla de registro de cuotas de cada usuario con un archivo

abierto, referenciada por este archivo y con información de los límites y las estadísticas actuales. Si aumenta el tamaño de un archivo o se crea otro, aumentan los números contenidos en esta tabla.

Respaldo del sistema de archivos

Un respaldo requiere tiempo y ocupa mucho espacio, por lo que hay que seleccionar lo que se respalda: hay archivos temporales, ejecutables, archivos especiales (E/S)... que no es necesario respaldar. Una solución pueden ser los respaldos incrementales, intercalados con respaldos completos periódicos. Estos respaldos solo incluyen los archivos modificados desde el último respaldo, lo cual reduce el tiempo de respaldo pero complica la recuperación. Además podemos comprimir los datos antes de hacer el respaldo. De todas formas, es muy difícil hacer un respaldo de un sistema activo.

Pueden realizarse con dos estrategias:

- Respaldo físico: del bloque 0 al último bloque. Es un respaldo simple y libre de errores, aunque luego dificulta el respaldo incremental. Para saber qué bloques están también sin utilizar y cuáles están defectuosos, accede a los registros de estos bloques.
- Respaldo lógico: comienza en uno o varios directorios especificados. Respalda de forma recursiva. Puede respaldar solo los archivos y directorios modificados desde el último respaldo. Es la estrategia más utilizada.
    - Facilita mucho la restauración incremental
    - Mantiene un mapa de bits indexado por el nodo-i
    - Tiene cuatro fases: marcar archivos modificados y todos los directorios, desmarca los directorios con archivos sin modificar, respalda los directorios y respalda los archivos.

Consistencia del sistema de archivos

Las modificaciones del sistema de archivos no se escriben en disco inmediatamente. Si el sistema falla antes de escribir todos los bloques se puede dar una inconsistencia que puede llegar a ser crítica si los bloques que no se han escrito son nodos-i, directorios o listas de bloques libres. La verificación de esta consistencia la hace tanto Windows (scandisk) como Unix (fsck).

Hay dos tipos de verificaciones:

- Verificación por bloques: hay dos tablas con un contador para cada bloque
    - Una tabla tiene cuántas veces está presente el bloque un bloque en un archivo
    - La otra tiene cuántas veces está un bloque en la lista (o mapa de bits) de bloques libres
    - El verificador lee los nodos-i, construye una lista de bloques utilizados por cada archivo y actualiza los contadores
- Verificación por directorios:
    - Forma una tabla de con un contador por archivo, con la que recorre el sistema de archivos y para cada nodo-i en cada directorio incrementa el contador de ese archivo
    - Después forma una lista indexada por número de nodo-i que indica cuántos directorios contienen a cada archivo (ED).

- Compara la cuenta de vínculos que tiene cada nodo-i (CV) con el ED de la lista. El sistema es consistente si ED == CV

Rendimiento

Cache de bloques o de disco

Son bloques que pertenecen al disco pero que están en memoria para optimizar el rendimiento. Satisface los accesos a disco desde memoria y puede almacenar miles de bloques, permitiendo un acceso rápido a base de codificar en tabla hash (con encadenamiento de lista enlazada para las colisiones) la dirección de dispositivo y de disco. Cuando la cache está llena se usa un algoritmo de reemplazo.

En caso de que se modifique un bloque en la cache y haya un fallo antes de actualizar el disco se produce un estado inconsistente, ya que puede pasar un tiempo antes de que el bloque se copie a disco (siempre se copia antes el LRU (less recently used), que es el que está al principio de la cadena de una entrada hash). Es posible también dividir los bloques en categorías, y los bloques que no se necesiten pronto pasan al frente de la lista. Si un bloque es esencial para la consistencia (nodo-i) se guarda en una cache de escritura inmediata con sincronizaciones periódicas (sync).

Lectura adelantada de bloque

Los bloques siempre se intentan colocar en la cache antes de ser necesitados para que aumente el índice de aciertos. Si se pide el bloque k, se puede traer también el bloque k+1. Esto puede ser muy útil en accesos secuenciales a los archivos. Es conveniente incluir información en los bloques sobre el tipo de acceso (secuencial o aleatorio) para mejorar esto.

Desfragmentación de los discos

Cuando se instala el SO los archivos y los programas se instalan de forma consecutiva, dejando contiguos los bloques libres. Con el tiempo de uso, el disco se desfragmenta, teniendo los bloques asignados y los libres esparcidos por el disco, lo cual reduce el rendimiento, haciendo que el acceso a un archivo implique buscar varios bloques. El rendimiento puede mejorarse juntando los bloques de los archivos.

Entrada / Salida

En esta parte, el SO se espera que sea capaz de emitir comandos, captar interrupciones y manejar errores, proporcionando una interfaz simple entre los dispositivos y el sistema igual para todos los dispositivos. Para la administración de la E/S usa un software estructurado en niveles con taraeas bien definidas. De la parte del hardware nos centraremos en la interfaz con el software (comandos, funciones y reporte de errores).

Hardware

Estos dispositivos, según el tipo pueden soportar o necesitar unas velocidades u otras. Van desde los 10 bytes/seg (teclados) hasta velocidades de 6GB/s (HDD sata).

Tipos de dispositivos

- Dispositivos de bloque: almacenan la información en bloques de tamaño fijo (entre 512B y 32KB). Cada bloque tiene su dirección, permiten la transferencia de uno o más bloques consecutivos, los cuales se leen o escriben independientemente del resto. Suelen ser los dispositivos de almacenamiento masivo.
- Dispositivos de carácter: envían o aceptan un flujo de caracteres sin estructuras de bloque. No son direccionables y no tienen operación de búsqueda. Son dispositivos como impresoras, ratones, interfaces de red...
- Otros dispositivos: hay dispositivos que no entran en ninguno de los otros dos grupos (reloj)

Controladores de dispositivos

Es el componente electrónico de una unidad de E/S (también tiene un componente mecánico). Es un chip en la tarjeta principal o tarjeta PC con conectores hacia el dispositivo. Este controlador puede manejar varios dispositivos distintos. Si el interfaz controlador-dispositivo es estándar, habrá controladores para ese interfaz.

Comunicación controlador–CPU

El controlador tiene una serie de registros para comunicarse con la CPU. Si el SO escribe o lee sobre ellos puede enviar órdenes o conocer el estado del dispositivo. Además, puede disponer de un búfer para la transferencia de datos entre la máquina y el dispositivo.

La CPU puede comunicarse con los registros de control y los búferes de datos mediante un puerto de E/S (y un espacio en memoria para la transferencia de datos) o mediante una dirección de memoria (E/S con asignación de memoria).

Puerto de E/S

Se asigna un número de puerto de E/S a cada registro de control. Éste será un entero de 8 ó 16 bits. El espacio de puertos de E/S será el conjunto de todos estos puertos.

Estos puertos solo puede utilizarlos el SO con operaciones especiales. Además, los espacios de direcciones y de puertos de E/S son diferentes (lo que hace que las instrucciones de acceso a memoria también sean diferentes).

Funcionamiento de estos esquemas

Cuando la CPU quiere leer, coloca la dirección en el bus de direcciones y activa la señal de read en el bus de control. Un bit adicional en estas direcciones especifica que se trata de un espacio de E/S o de memoria. En caso de que solo haya espacio de memoria, los módulos de memoria y los dispositivos de E/S comprueban si la dirección está en su rango.

E/S con asignación de memoria

Se asignan los registros de control al espacio de memoria, teniendo cada registro de control una dirección en la que no hay ninguna memoria asignada.

Es posible la idea de un esquema híbrido en el que entran los búferes de datos de E/S por asignación de memoria y los puertos de E/S para los registros de control.

Ventajas

- Las instrucciones especiales de E/S con puertos se hacen con código ensamblador (porque no es posible hacerlas en C). Con la asignación de memoria, en cambio, los registros de control son variables con una dirección de memoria asignada, por lo que pueden escribirse en C.
- La E/S por asignación de memoria no requiere protecciones especiales, simplemente basta con no darle ese espacio de direcciones al espacio de direcciones virtuales de ningún proceso. Además, cabe la posibilidad de que si esos espacios de memoria están separados en páginas distintas y el SO quiere darle acceso a un proceso para operar con un solo dispositivo, puede seleccionar qué página le asigna.

Desventajas

- Llevar a caché un registro de control puede ser desastroso, ya que las comprobaciones que se hagan de éste pueden no estar actualizadas y producirse un cambio en algún registro y no darse cuenta el SO. La única solución es que se seleccionen páginas que no pueden ser almacenadas en cache.
- Todos los controladores deben examinar las direcciones que lleva el bus para comprobar si tienen que responder a alguna. Esto es fácil de implementar con un solo bus, pero con buses separados los dispositivos de E/S no ven las direcciones de memoria, por lo que es necesario enviar todas las referencias a memoria y, si esta no responde, enviarlas a los otros buses.
- Otra solución es filtrar las direcciones que salen de la CPU con un PCI bridge (puente de PCI o husmeador de bus), el cual contiene registros de rango para diferenciar y cuando detecta que las direcciones no están en el rango de memoria principal, las envía al bus PCI (conecta todos los dispositivos de E/S).

Acceso Directo a Memoria (DMA)

La CPU necesita direccionar los controladores de dispositivos para intercambiar datos, sin importar de si se usa E/S por asignación de memoria o puertos E/S. Esto produce que se desperdicie tiempo de CPU al comunicarse directamente con el controlador.

La DMA libera a la CPU del intercambio de datos, creando un controlador de DMA hardware que puede controlar varios dispositivos y que puede acceder al bus PCI de forma independiente de la CPU.

Este DMA contiene varios registros en los que la CPU puede leer y escribir:

- Registro de dirección de memoria
- Registro contador de bytes
- Registros de control (indican el puerto de E/S, la dirección de transferencia (R/W), la unidad de transferencia y el número de bytes en una ráfaga)

Ejemplo de uso:

- Sin DMA, para copiar datos de disco a memoria, es necesario leer el bloque y colocarlo en un búfer verificando que no haya errores. Después se produce la interrupción y el SO lee el bloque del búfer palabra a palabra almacenándolo en memoria (con varios ciclos)
- Con DMA, el SO programa el DMA (establece qué transferir, a dónde y de qué disco leer los datos). Cuando hay datos en el búfer del controlador, el DMA inicia la lectura y escribe las palabras que lee en memoria. Al completar la escritura, el controlador manda un ACK (ACKnolegdment, confirmación), el DMA genera direcciones de memoria y disminuye la cuenta de bytes. Cuando termina, el DMA interrumpe a la CPU.

Los DMA son capaces de gestionar varias transferencias al mismo tiempo, teniendo varios canales con sus propios registros y usando para cada transferencia un controlador de dispositivo distinto. Pueden operar en modo palabra (compite con la CPU por el acceso al bus, Robo de ciclo) o en modo bloque (adquiere el bus, emite la transferencia y libera el bus, Modo de ráfaga, lo cual es más eficiente pero puede bloquear la CPU). La mayoría de los DMA usan direcciones físicas.

De todas formas, si la CPU no tiene otras tareas que hacer, es más rápido no usar DMA.

Interrupciones

El dispositivo de E/S (o DMA) utiliza las interrupciones para indicar que ha terminado el trabajo. El controlador de estas interrupciones (en la CPU) procesa la interrupción, interrumpiendo a la CPU y haciéndole leer de un vector de interrupciones el nuevo contador de programa, almacenando también la información de retorno de la rutina de interrupción (los datos adjuntos a la interrupción).

Interrupciones precisas e imprecisas

En los procesadores súper-escalares, cuando ocurre una interrupción después de una instrucción, es posible que las instrucciones no hayan terminado.

- Una interrupción **precisa** es en la que el PC apunta a la instrucción que generó la interrupción y que puede reiniciarse más tarde.
- La **imprecisa** ocurre cuando hay distintas instrucciones con distintos estados de ejecución

(instrucciones no ejecutadas al 100%), provocando que haya mucha información de estado de las ejecuciones en la pila, por lo que el reinicio es más complicado y lento, lo cual causa que sólo haya interrupciones precisas para unas cosas y no para otras.

Fundamentos del software

Conceptos

- Independencia de dispositivos: se requieren programas que puedan acceder a cualquier dispositivo de E/S sin tener que especificarlo, funcionando con cualquier dispositivo de entrada y/o salida.
- Denominación uniforme: el nombre del archivo no debe depender del dispositivo (Ejemplo: montaje del dispositivo en el árbol de directorios)
- Manejo de errores: siempre deben intentar resolverse lo más cerca posible del hardware.
- Transferencias síncronas (de bloqueo): son las que se usan en los programas de usuario, después de un read el proceso se para hasta que los datos estén en el búfer
- Transferencias asíncronas (controladas por interrupciones): se usan en la mayoría de la E/S, la CPU inicia la transferencia y atiende a otro proceso hasta que llega la interrupción. El SO debe hacer que las operaciones asíncronas parezcan de bloqueo de cara al usuario.
- Utilización del búfer: sirve para almacenar los datos temporalmente hasta que estos se llevan a su destino final.

Hay tres formas distintas de llevar a cabo la E/S: programada, controlada por interrupciones o mediante DMA.

E/S programada

Consiste en que la CPU tiene que sondear el dispositivo para ver si está listo. La desventaja es que ocupa la CPU hasta completar la E/S.

Por ejemplo, para imprimir una cadena de texto, la CPU escribiría los caracteres uno a uno en el búfer, esperando después de cada uno hasta que la impresora estuviese lista.

E/S controlada por interrupciones

La CPU realiza la transferencia y se planifica algún otro proceso. Esto permite que, para el ejemplo de la impresora, entre carácter y carácter la CPU pueda hacer un cambio de contexto y cuando el dispositivo E/S esté listo, genera una interrupción para detener el proceso actual. La desventaja de esto es que genera muchas interrupciones.

E/S con DMA

El DMA realiza la E/S sin molestar a la CPU. Es como una E/S programada, pero el trabajo lo realiza el DMA en lugar de la CPU, lo cual reduce el número de interrupciones (para el ejemplo de la impresora, las interrupciones se reducirían de una por carácter a una por búfer).

De todas formas, cuando la CPU no tiene nada que hacer, es más rápido que se encargue ella.

Capas del software de E/S

El software de E/S se organiza en cuatro capas, las cuales tienen, cada una, una función y una interfaz con los niveles adyacentes bien definida (como la estructura por capas de TCP/IP). La funcionalidad e interfaces pueden variar de un sistema a otro.

Estas capas son: Hardware → Manejador de interrupciones → Drivers de dispositivos → Software de E/S independiente del dispositivo → Software de E/S en el espacio de usuario

Manejador de Interrupciones

Trata de ocultar las interrupciones: el driver que inicia la E/S se bloquea hasta que se complete la E/S y ocurra la interrupción. Una vez recibida la interrupción, el manejador:

- Guarda los registros actuales de la CPU.
- Establece el contexto y la pila para el procedimiento de servicio de interrupciones (TP, TLB)
- Copia los registros a la tabla de procesos.
- Ejecuta el procedimiento de servicio de interrupciones y extrae la información de los registros del controlador del dispositivo.
- Elige el proceso que se va a ejecutar a causa de la interrupción (puede ser el driver que había iniciado la E/S)
- Establece el contexto, carga los registros del proceso elegido y lo ejecuta.

Drivers de dispositivos

Cada dispositivo de E/S tiene asociado un controlador con registros de datos y control para comunicarse con la CPU. El número de estos puede variar.

Cada controlador de E/S necesita un código específico de control (driver) que proporciona el fabricante y maneja un tipo o clase de dispositivo. Este driver forma parte del SO y debe acceder a los registros del controlador. Es posible construir drivers que se ejecuten en el espacio de usuario.

Existen interfaces estándares del SO con el controlador (funciones básicas) tanto para dispositivos de E/S de bloque o de carácter, con varios procedimientos: leer bloque, escribir bloque, leer carácter... Son las que usan los drivers para comunicarse con sus dispositivos. Éstos funcionan así:

- Se cargan dinámicamente en tiempo de ejecución
- Traducen los términos abstractos a concretos (un número de bloque de disco lo convierten a cabeza, pista y sector)
- Comprueban si el dispositivo está en uso
- Envían comandos al controlador (escribiendo en los registros)
- Comprueban si se ha aceptado el comando y comprueban errores

Una vez emitidos los comandos, puede darse que el controlador deba realizar un trabajo (bloqueando el driver hasta recibir una interrupción) o que la operación termine sin retraso, permitiendo que el proceso no tenga que pasar a estado inactivo. Si no hay errores, pueden pasar los datos a la capa superior y atender otras peticiones.

Software de E/S independiente del dispositivo

Para facilitar el control de los dispositivos, debe haber partes de software que sean independientes de éstos. Este límite entre los controladores y el software independiente del dispositivo depende del sistema. La función básica del software independiente del dispositivo es realizar las funciones de E/S que son comunes para todos los dispositivos y proveer una interfaz uniforme para el software a nivel de usuario. Esta interfaz uniforme consta de:

Interfaz uniforme para drivers

Facilita la conexión de dispositivos y escritura de drivers. Es el conjunto de funciones que el driver debe proporcionar y la protección y el modo de nombrar a los dispositivos. Esto último se hace mediante archivos especiales, con nodos-i especiales también, que constan de un número mayor del dispositivo (que localiza el controlador) y el número menor del dispositivo (que especifica la unidad)

Para cada clase de dispositivo el SO define un conjunto de funciones que el driver debe proporcionar (leer, escribir, encender, apagar...). Para esto, el driver tiene una tabla de punteros a estas funciones.

Uso de búfer

Permite un uso más eficiente por parte de los procesos. Lo explicaremos con el ejemplo de lectura de datos en un módem.

- Sin búfer cada carácter produce una interrupción y un reinicio del proceso
- Con un búfer en el espacio del usuario el proceso debe reiniciarse al llenarlo. Si el búfer es una página, se podría bloquear en memoria y se reduciría el número de páginas disponibles.
- Con un búfer en el kernel con copia en el espacio de usuario, el búfer solo se cargaría en memoria cuando estuviese lleno. El problema es que los caracteres que llegan cuando se está trayendo del disco la página del búfer no tienen dónde almacenarse.
- La mejor solución es un doble búfer en el núcleo con copia de un búfer en el espacio de usuario, la cual se puede implementar como un búfer circular con punteros separados de lectura y escritura.

La desventaja del búfer es que si los datos se colocan en búfer demasiadas veces, el rendimiento se reduce. Por ejemplo, en una red una llamada al sistema para escribir puede necesitar 6 búferes. Y si todos los pasos se realizan de forma secuencial, se reduce la velocidad de transmisión.

Reporte de errores

Sirve para el reporte de errores de E/S (como tratar de leer un bloque de disco dañado), no incluye errores de programación. Sólo se reportan si el driver no sabe tratar ese error, pasándose así el problema al software independiente del dispositivo. La solución puede ser reintentar, ignorar el error, eliminar el proceso...

Asignar y liberar dispositivos dedicados

Algunos dispositivos solo pueden ser usados por un proceso en un momento dado (grabadora de CD). El SO debe rechazar o aceptar peticiones de acceso dependiendo de si el dispositivo está

disponible (en caso de rechazo, por ejemplo, se da fallo en la llamada open). Al cerrar un dispositivo dedicado, este se libera.

Proporcionar un tamaño de bloque independiente del dispositivo

Entre la variedad de discos, también la habrá de tamaños de sector. El software independiente del dispositivo debe ocultar esto y proporcionar un tamaño de bloque uniforme al nivel superior. Estos son los bloques lógicos, propios de dispositivos abstractos con el mismo tamaño de bloque lógico sin importar el sector físico.

Software de E/S en el espacio de usuario

Es la parte del software de E/S en la que se incluyen las bibliotecas vinculadas con programas de usuario y los programas que se ejecutan fuera del núcleo (por ejemplo, write se vincula al programa, incluyéndose en el binario, y cada vez que se llama a printf, se construye una cadena ascii y se llama a write). Usan también sistemas de colas (para impresoras, redes...) para gestionar dispositivos en sistemas multiprogramados. Para ello existen procesos especiales (demonios) que acceden a estas colas (archivos especiales de dispositivos).

De esta manera, cada petición de E/S sale de la capa de software de E/S en el espacio de usuario y pasa por todas las capas hasta llegar al hardware, en el que se realiza la operación y la respuesta de esta pasa otra vez por todas las capas activando sus mecanismos hasta dar la respuesta al usuario.

Discos

Son dispositivos de E/S para almacenamiento masivo de datos de gran tamaño. Están organizados en cilindros pistas y sectores con entre 1 y 16 cabezas (una para cada superficie de disco, dentro de cada dispositivo de disco puede haber varios discos). Además, tienen la capacidad de realizar búsquedas traslapadas, permitiendo que el controlador realice búsquedas en dos o más unidades a la vez.

En los últimos años la capacidad de los discos y la rapidez de éstos ha ido aumentando de forma considerable. De todas formas, la brecha de rendimiento entre CPU y disco es cada vez mayor: el tiempo de búsqueda en disco se ha reducido entre 10 en 40 años mientras, mucho menos que el tiempo de ciclo de la CPU. Por lo tanto, para mejorar el rendimiento, debemos hacer paralelismo en la E/S. Para esto existe el RAID.

El RAID (Redundant Array of Independent Disks) es el conjunto de unidades de disco con un único controlador. Distribuye los datos entre las unidades para hacer operaciones en paralelo y tiene tolerancia de fallos: almacena de forma redundante los datos para poderlos recuperar incluso si falla un disco. Existen varios niveles de RAIS (de 0 a 6).

Organización en RAID

Los objetivos del paralelismo son equilibrar la carga en múltiples accesos pequeños, incrementando la productividad, y paralelizar los accesos grandes para reducir el tiempo de respuesta. Al distribuir los datos en múltiples discos se mejora la velocidad de transferencia. La distribución puede ser:

- A **nivel de bit**: la distribución de cada byte se hace en varios discos: con 8 discos, el bit i de

cada byte se escribe en el disco i, haciendo que cada acceso pueda leer datos a 8 veces la velocidad de un solo disco. Sin embargo, el tiempo de búsqueda/acceso es peor.

- A **nivel de bloque**: la distribución de los bloques de un archivo se hace en varios discos, de forma que con n discos, el bloque i de un archivo va al disco (i mod n) + 1. Esto permite que se puedan ejecutar en paralelo peticiones para diferentes bloques en caso de que residan en diferentes discos. Una petición para una secuencia grande de bloques puede utilizar todos los bloques a la vez en paralelo.

Organización de los niveles de RAID (no hay jerarquía, son maneras de organizar el RAID):

- **RAID 0**: distribución de **bloques** (hay paralelismo en acceso a bloques consecutivos y sin redundancia).
- **RAID 1**: **imágenes de disco** (tiene tolerancia a fallos, con discos de respaldo, y mejora la lectura).
- **RAID 2**: distribución **a nivel de bit con códigos ECC** (aplicaba código Hamming por seguridad).
- **RAID 3**: **paridad con bits enlazados** (solo aplica un bit de paridad para la corrección de errores porque ya sabe que un disco ha fallado. Es de acceso rápido pero no puede atender a varias peticiones simultáneas porque el acceso a un bloque obliga a acceder a todos los discos).
- **RAID 4**: distribución **a nivel de bloques** (establece una paridad por bloque con un bloque de paridad en un disco independiente para los bloques de los otros n discos. Este bloque es el cuello de botella).
- **RAID 5**: **paridad distribuida** con bloques entrelazados (los datos y la paridad se dividen en N+1 discos, permitiendo una velocidad de E/S mayor que en RAID 4).
- **RAID 6**: **redundancia contra fallos** en varios discos (con más bloques de paridad).

Hay que destacar que los RAID 3 y 5 incluyen los RAID 2 y 4. El RAID 5 es el que mayor coste conlleva. Los más utilizados son el RAID 1 y el 5.

Formato de Disco

Los discos están compuestos por pistas concéntricas (cilindros) con un cierto número de sectores. Cada uno tiene un patrón de inicio del sector (preámbulo, con el número de cilindro y sector por ejemplo), un segmento de datos y una parte para corrección de errores (ECC). A mayores, tienen un cierto número de sectores reservados para utilizarlos como reemplazo de sectores defectuosos.

Además, estos cilindros están **desajustados**: el sector 0 de cada pista está desfasado de la pista anterior con una desviación que depende de la geometría, lo cual permite mejorar el rendimiento.

La **capacidad** del disco se reduce en función del preámbulo, ECC y los sectores reservados. La capacidad con formato suele ser un 20% menor que la capacidad antes del formato.

Si los sectores no se entrelazan cuando se lee un lector y los datos se transfieren a memoria, al terminar la transferencia el siguiente sector ya ha pasado por debajo de la cabeza. Una velocidad de acceso constante requiere un búfer extenso y **entrelazado** de sectores.

Algoritmos de programación del brazo del disco

Tiempo de acceso = tiempo de búsqueda (desplazamiento del brazo) + retraso rotacional (colocación del sector bajo la cabeza) + tiempo de transferencia de datos

El dominante es el **tiempo de búsqueda**, el cual debe reducirse en su promedio. Para esto, no es conveniente una política de primero en llegar, primero en ser atendido (FCFS). La alternativa es una tabla indexada por el número de cilindro con las peticiones pendientes para cada cilindro (lista enlazada). Con esta tabla, es posible manejar la **petición más cercana primero** (búsqueda más corta primero, SSF).

El problema del SSF es que si siguen llegando peticiones, es probable que el brazo se mantenga en la parte media del disco y dé mal servicio a los cilindros extremos. Esto se soluciona con el algoritmo del ascensor: no se cambia la dirección de búsqueda mientras haya peticiones pendientes en esa dirección.

Otras mejoras

- Cache del controlador: consiste en que cuando se solicita un sector en la cache, se lleva junto con varios sectores de la misma pista (es diferente a la cache en bloques del sistema de archivos)
- Cuando hay varias unidades en el mismo controlador y cada una tiene su tabla de peticiones, cuando éstas están inactivas es conveniente desplazar su brazo al cilindro que se va a necesitar.
- Todos los algoritmos suponen que la geometría física es igual a la geometría virtual, pero el SO no puede saber qué cilindros están más cercanos al que está siendo accedido.

Relojes

Están formados por un oscilador de cristal de cuarzo y un contador que se decrementa en cada pulso. Cuando éste llega a 0, se produce una interrupción en la CPU, pulso de reloj. El contador puede volver a reiniciarse automáticamente (modo onda cuadrada) o de forma explícita (modo de un solo disparo). Puede haber varios relojes que se programan de forma independiente.

El reloj genera instrucciones a intervalos conocidos. Es muy útil para:

- Mantener la hora del día
- Evitar que los procesos se ejecuten por más tiempo del que tienen asignado (quantum)
- Contabilizar el uso de la CPU (cada proceso inicia un temporizador diferente del principal)
- Manejar alarmas
- Proveer temporizadores watchdog para ciertas partes del sistema (es un temporizador que sirve para que, en caso de que el sistema no lo reinicie, el watchdog pueda reiniciar todo el sistema entendiendo que éste ha fallado)
- Realizar estadísticas

Podemos usar un segundo reloj que se establece como alternativa a las interrupciones. Las interrupciones tienen una sobrecarga considerable para algunas aplicaciones (red Gigabit Ethernet). La aplicación sondea el evento por el que espera y antes de volver al modo usuario comprueba si el

temporizador ha llegado a 0. En caso de que haya llegado, hace el evento programado sin salir del modo kernel.

Teclado

El controlador extrae la información cada vez que se pulsa una tecla. Se usa un código de exploración con 7 bits. Cuando se pulsa una tecla, el bit extra se pone a 0 y vuelve a 1 cuando se suelta. Lo que se pasa después al proceso es una secuencia de códigos ASCII. La transferencia puede ser:

- Carácter a carácter (modo no canónico): pasa absolutamente todos los caracteres
- Línea a línea (modo canónico): genera ya la línea y cuando hay retorno de carro ya la envía

Ratón

Puede ser mecánico (con bolita) u óptico. Cada vez que se desplaza o se pulsa/suelta un botón, se envía un mensaje al ordenador que consta de 3 bytes, incluyendo los incrementos de x e y (solo cambios, no posición absoluta) y los botones (incluye el doble click).

El sistema X windows

Es la base del interfaz de usuario de los sistemas UNIX. Se trata de un software portátil (se ejecuta en el espacio del usuario) y sobre éste se ejecutan los entornos de escritorio (GUI).

Suele dividirse en software de cliente y de servidor:

- El servidor X recolecta la entrada del teclado y el ratón, escribe en pantalla, determina la ventana activa y se comunica con los clientes
- Los clientes X son los programas en ejecución

El servidor X debe estar en el ordenador del usuario, pero el cliente puede estar en un ordenador remoto o en el mismo. En caso de que estén en el mismo ordenador, el cliente es un programa de aplicación que se conecta al servidor con conexión TCP sobre sockets. X windows es el que define el protocolo entre cliente y servidor.

Para obtener una GUI se necesitan otras capas de software:

- Xlib: procedimientos de biblioteca para funcionalidades de X
- Intrinsics: para programación con X, incluye botones, barras de desplazamiento...
- Motif (Qt, GTK): apariencia visual uniforme

Interfaces gráficas de usuario (GUI)

Constan de cuatro elementos esenciales: ventanas, iconos, menús y un dispositivo señalador. Tienen código a nivel de usuario (UNIX) o en el SO (Windows). Emplean el adaptador de gráficos (tarjeta de vídeo, con un controlador que incluye una CPU y una RAM de vídeo que contiene imágenes que deben aparecer en la pantalla). La acción de dibujar en pantalla se gestiona con el GDI (Graphics device interface), que tiene procedimientos para manejar texto y gráficos y que utilizan el contexto de dispositivo (propiedades de la ventana).