Sistemas Operativos 2

Autor: Agustín Fernández Bergé

Indice

FIlesystem

Commodore 64 Filesystem

> **Commodore 64 Filesystem**
> El filesystem de la commodore 64 represento una mejora respecto al TAR Filesystem. En este caso, se reserva una sección del medio para almacenar las cabeceras de todos los archivos y la región restante contiene los datos de los archivos(se concentran los metadatos).
>
> | Sector de cabeceras | Sector de datos |
> | --- | --- |
> | \| Cabecera 1 \| Cabecera 2 \| Cabecera 3 \| | Archivo1.txt \| Archivo2.txt \| Archivo3.txt |
>
> Además de los datos usuales, cada cabecera contiene un apuntador al inicio de su archivo correspondiente. En el ejemplo, *Cabecera 1* contiene la dirección de inicio de *Archivo1.txt*, *Cabecera 2* apunta al *Archivo2.txt* y así.
>
> Además de ser simple, se mitiga la desventaja del acceso secuencial del TAR Filesystem ya que solo hay que revisar en el sector de cabeceras el archivo que hay que buscar.
>
> Sin embargo sigue existiendo la desventaja de la actualización compleja y los archivos en subbloques(es más ahora es peor, ya que como no hay relleno es posible leer datos de un archivo diferente). A estas desventajas se le suma que el tamaño del sector de cabeceras es fijo, por lo que la cantidad máxima de archivos esta determinada por el tamaño del sector de cabeceras.

FAT Filesystem

> **FAT Filesystem**
> A partir de los años 80, Microsoft empezó a usar un sistema de archivos denominado **FAT**. El sistema se llama asi por su estructura de datos más importante, la **F**ile **A**ccess **T**able. FAT se basa en unificar dos aspectos clave a manejar:
>
> - Gestión del espacio libre.
> - Fragmentación arbitraria de archivos.
>
> Como consecuencia este ya no contiene un bitmap ni gestión de fragmentos en las dir entries(como Questar - MFS filesystem). Sino que estos aspectos se gestionan directamente en la FAT.
>
> Hoy en dia FAT fue reemplazado en Windows por NTFS(*New Technology File System*), sin embargo sigue siendo utilizado en medios de almacenamiento pequeños como los Pendrives.

Estructura

En FAT, el medio fisico se divide en clusters. Un cluster es una sección contigua de 4 sectores fisicos(equivalente al sector logico).

El disco se estructura en las siguiente regiones:

Sector de arranque	1era FAT	2da FAT	Directorio raiz	Datos
			Contiene las dir entries	. . .

- Sector de arranque: Contiene información básica del sistema de archivos como que tipo de filesystem es y punteros al resto de secciónes del sistema.
- 1era FAT: Es la estructura principal del sistema. Contiene información sobre el estado de cada cluster en el sector de datos. Puede verse a la FAT como un bitmap extendido.
- 2da FAT: La FAT es la estructura principal del sistema, como consecuencia si la FAT se daña o corrompe el sistema puede verse seriamente comprometido. Para chequeos de redundancia el sistema implementa una segunda FAT la cual es una copia de la primera.
- Directorio raiz: Esta sección contiene las dir entries. Cada dir entry se corresponde con un archivo del sistema y contiene los metadatos del archivo y información para buscar en la FAT.
- Datos: Contiene los datos en si de los archivos, esta sección se divide en los clusters ya mencionados. El acceso al sector de datos se consigue usando la FAT. Al igual que otros sistemas que dividen el disco en sectores, si no se usa todo el cluster para guardar datos el espacio sobrante se pierde.

FAT

La FAT es la estructura cable del sistema, cada entrada de la FAT se corresponde con un cluster y cada entrada contiene un valor entero que indica el estado del cluster.

La FAT indica que clusters estan libres y para cada cluster a que cluster continuar para seguir trabajando con el archivo. Para esto se le asigna a cada cluster del disco un numero entero. De esta manera la FAT se puede ver como una lista enlazada de clusters que indican como leer un archivo.

Los valores que pueden tener cada entrada de la FAT son los siguientes:
- 0: El cluster esta libre y puede ser usado para guardar datos.
- 0xfffe(-2): El cluster esta dañado.
- 0xffff(-1): El cluster esta siendo usado por un archivo pero no corresponde.
Si la entrada de la FAT no contiene ninguno de los valores anteriores, entonces la entrada contiene el numero de cluster siguiente.

Los directorios en FAT son nuevamente un tipo de archivo especial que contiene las dir-entries de otros archivos y subdirectorios.

Tamaño de la FAT

El tamaño maximo del sector de datos depende del numero de clusters direccionables y este numero depende tanto del tamaño de la FAT como del numero de bits que usa cada entrada de la FAT. Por eso versiones posteriores del filesystem mejoraron(entre otras cosas) el numero de bits usado para cada entrada de la FAT. Esto viene determinado por el numero de al lado en el nombre del sistema de archivos. FAT12 usa 12 bits por entrada, FAT16 usa 16 bits y FAT32 usa 32 bits.

Directorio raiz

El area del directorio raiz contiene los metadatos de cada archivo existente en el filesystem asi como a que entrada de la FAT hay que ir para operar con el archivo.

Cada dir entry esta compuesta por 32 bytes y contiene la siguiente información:

- El nombre del archivo.
- Su extensión.
- Atributos(oculto, etc).
- Fecha de creación.
- Fecha de ultima actualización.
- Cluster de inicio(para buscar en la FAT).
- Largo del archivo.

Nombre	Extensión	Atributos	Reservado	Fecha de creación	Fecha de ultimo acceso	Nro Cluster alto	Fecha de modificación	Nro Cluster	Tamaño
8 B	3 B	1 B	1 B	4 B	2 B	2 B	4 B	2 B	4 B

Algunas aclaraciones sobre las dir entries:

- El primer byte en el nombre del archivo puede tener valores especiales, $0x00$ indica que la entrada esta libre y $0xe5$ indica que la entrada esta borrada.
- El byte de atributos utliza algunos bits para indicar ciertas caracteristicas de un archivo:
  - Oculto: Si esta prendido, el archivo no se debe mostrar al usuario al listar un directorio.
  - Solo lectura: Si esta prendido, el sistema debe evitar toda modificación del archivo.
  - Sistema: Dado que FAT tiende a fragmentar la información de un archivo. Si este bit esta prendido, el sistema tiene que cuidar no mover ni fragmentar el archivo.
  - Subdirectorio: Indica que el archivo en cuestión es un subdirectorio y que debe ser implementado como tal.
  - Archivado: Este es el bit de "Archivado" y era usado por las herramientas de backup de MS-DOS. Cuando se realizaba un respaldo el bit quedaba prendido y al realizar una modificación el bit se apagaba. De esta manera la herramienta de respaldo solo tenia que respaldar aquellos archivos que tenian el bit de archivo apagado.
    Dado que no se usan todos los bits de atributos, algunos sistemas(junto al byte de reservado) hacen uso de más bits para especificar mas atributos. Sin embargo los 4 mencionados son la base del sistema.
- El Nro de cluster alto es usado en FAT32 ya que los 2 bytes del nro de cluster no son suficientes para alcanzar los 32 bits de nro de cluster mientras que en FAT12 y FAT16 si.
- Los 2 primeros bytes tanto de las fechas de creación como de actualización se usan para guardar el tiempo mientras que los dos ultimos se usan para el dia. En el caso del ultimo acceso los 2 bits son solo para el dia.

Dado que las dir-entries estan en un sector de tamaño fijo, la cantidad de máxima de archivos en el directorio raiz esta limitada por el tamaño de este sector. FAT12 y FAT16 tenían esta limitación. FAT32 resolvio esto fusionando el sector de directorio raiz con el sector de datos, lo cual le permitia crecer. Tipicamente el sector de directorio raiz empieza en el cluster 3.

Operaciones de archivos en FAT

Veamos como son algunas operaciones con archivos en FAT:

Creación de archivos

Hay que reservar una dir-entry para el archivo y escribir en ella metadatos como nombre, extensión y fecha de creación. A su vez hay que buscar un cluster libre en la FAT, marcar dicho cluster como fin de archivo y poner en la dir-entry que el cluster inicial del archivo es aquel que se reservo.

Abrir

Hay que buscar la dir-entry del archivo en disco y una vez encontrada leer el numero de cluster inicial para saber donde buscarlo en la FAT. Luego hay que crear el file descriptor(aunque esto es por fuera de FAT).

Lectura

Para leer un archivo ya abierto simplemente hay que leer los clusters de la FAT al buffer de usuario. Si se llega a fin de cluster hay que pasar al cluster siguiente que marca la FAT.

Escritura

Si hay que escribir en algún sector ya reservado entonces hay que escribir en los sectores correspondientes. Puede ser necesario extender el archivo, en cuyo caso hay que buscar en la FAT algún cluster libre y una vez encontrado actualizar la FAT para que el viejo ultimo cluster apunte al nuevo cluster y el nuevo cluster este marcado como fin de archivo. Una

vez hecho esto hay que actualizar en la dir-entry la fecha de ultima actualización, fecha de acceso y el tamaño del archivo. Si la FAT fue actualizada también hay que sincronizar la FAT de disco.

Borrado

El borrado en FAT se realiza mediante un mecanismo "flojo". En vez de borrar toda la dir-entry y poner en cero todos los clusters del archivo se realiza lo siguiente:

- Se marca el primer byte de la dir-entry(primer byte del nombre del archivo) con el valor $0xe5$ indicando que la entrada debe ser ignorada.
- Se sigue la cadena de clusters en la FAT marcando **las entradas** en 0 indicando que estan disponibles. Los clusters correspondientes a las entradas quedan sin modificar.
  La dir-entry marcada puede ser reclamada por un nuevo archivo y cada una de las entradas marcadas con 0 pueden ser reclamadas por otro archivo diferente, en cuyo caso los datos guardados en el cluster se sobreescribiran.

Ventajas y desventajas

FAT filesystem es un buen filesystem si se usa en discos chicos. Primero porque las operaciones de creación y extensión de archivos requieren sincronizar la FAT, lo cual ralentiza la escritura. Segundo porque para que el sistema funcione rapidamente la FAT tiene que estar cargada en memoria. Si el disco es muy grande la FAT tambien sera muy grande y puede restar espacio util a procesos del sistema. Si bien es posible guardar la FAT usando memoria virtual esto ralentiza aun más el sistema ya que se incrementan los fallos de pagina y las escrituras al disco.

Operaciones con archivos

Operaciones con archivos

En lineas generales, un archivo posee las siguientes operaciones:

- Borrar: Borra el archivo del directorio y, si es necesario, se libera el archivo del dispositivo.
- Abrir: Se solicita al sistema operativo si el archivo pedido existe o puede ser creado dependiendo de:
  - El modo con el que se abrió el archivo(lectura, escritura, etc).
  - Si se cuenta con los permisos necesarios para abrir el archivo.
  - Si el medio lo soporta(ejemplo, un CD-ROM es un medio de solo lectura por lo que no se pueden abrir archivos para escritura).
    Al abrir un archivo, el SO asigna un *descriptor de archivo*(file descriptor). El descriptor es un identificador de la relación que hay entre el proceso que abrió el archivo y el archivo en cuestión.
- Cerrar: Se notifica al sistema operativo que el proceso ya termino de trabajar con el archivo. Para cerrarlo, el SO debe escribir los buffers del archivo a disco y eliminar el descriptor en la tabla de archivos abiertos del sistema y en la tabla de archivos abiertos del proceso(de esta manera, se *invalida* el file descriptor). Si después de esto el proceso quiere volver a usar el archivo tendrá que volver abrirlo, ya que el file descriptor ya no es valido.
- Leer: Se solicita al sistema leer un pedazo de información del archivo a un Buffer en memoria. Para hacer esto el sistema operativo mantiene un apuntador a la ultima posición leída, de manera que todas las lecturas incrementan el contador una cantidad determinada de posiciones, de manera similar a como lo hace una maquina de escribir.
- Escribir: Guardar información en un archivo abierto. Esta escritura puede hacer desde el inicio del archivo(borrando toda información que ya este en el) o al final del archivo manteniendo la información que tenia de antes. Este procedimiento se realiza con el mismo apuntador que se usa en la lectura.
- Reposicionar: Reposicionar consiste en colocar el apuntador de lectura/escritura en otro punto del archivo. De esta manera se puede saltar a otro punto del archivo y escribir o leer en el.

Puede haber mas operaciones dependiendo del sistema operativo, pero estas son las básicas.

Questar - MFS filesystem

Questar - MFS filesystem

Tanto TAR Filesystem como Commodore 64 Filesystem especificaban en su estructura que los archivos ocupan bloques contiguos en memoria lo cual dificulta enormemente la actualización de archivos. Questar/MFS(MFS probablemente signifique *Macintosh File System* pero no esta chequeado) fue un primer enfoque para permitir que los archivos estén fragmentados en disco.

Tiene la siguiente estructura:

Mapa de bits	Dir entries	Datos
$10111\ldots00110$	Dir entry 1 | Dir entry 2 | Dir entry 3 | . . .	. . .

Cada entrada de directorio(dir-entry) se corresponde con un archivo en el sistema y contiene tanto los metadatos del archivo como información de fragmentación:

Esta es la estructura de una dir entry:

Nombre	en ASCII	Largo	1er sector	nro de sectores	2do sector	3er sector	. . .
7+3 B	clave de acceso						

La sección que corresponde desde el 1er sector hasta el final corresponde a los fragmentos y cada celda contiene la dirección en disco del sector del archivo.

Un sector es equivalente a una región del disco de 512 bytes. A su vez el mapa de bits indica que sectores del disco están libres y cuales están ocupados por datos, cada bit del mapa se corresponde con un sector.

Este filesystem tiene en común con el de commodore que los metadatos están centralizados y ya no tiene la desventaja de la actualización compleja debido a que se puede extender un archivo simplemente añadiendo un sector más al mismo. Sin embargo el numero máximo de fragmentos esta determinado por el tamaño asignado a la dir entry por lo que el nivel de fragmentación es acotado.

Este sistema de archivos también permite la existencia de subdirectorios, un subdirectorio es un archivo el cual sus fragmentos apuntan a dir-entries.

TAR Filesystem

TAR Filesystem
TAR es un acrónimo de **T**ape **AR**chive(archivo de cinta) es un tipo de filesystem sencillo de implementar(y por la misma razón también es muy limitado).

TAR tiene la siguiente estructura en disco:

Cabecera	Datos	(Relleno)	Cabecera	Datos	(Relleno)	
512 B	Archivo1.txt		512 B	Archivo2.txt		. . .

El disco se divide en bloques de 512 bytes.

Los archivos se colocan de manera secuencial en disco, cada archivo contiene:

- Cabecera: Contiene información sobre los metadatos del archivo, esto es, Nombre, permisos, tamaño, creación, ultima actualización, etc.
- Datos: Los datos del archivo en si, cuando se quieren leer datos del archivo se leen de a bloques de 512 bytes. Por ejemplo, si yo quiero leer los caracteres del 600 al 800 del archivo tengo que leer el segundo bloque de datos y de ese bloque leer los caracteres desde el 88 al 288 del bloque que leí($600 - 512 = 88$ y $800 - 512 = 288$). Si quisiera actualizar esa misma sección tendría que leer dicho bloque, cambiar el rango de caracteres que quiero ver y reescribir el bloque entero en disco(ver Transferencias orientadas a bloques).
- Relleno: Las lecturas y escrituras en base a bloques requieren que los datos estén alineados a 512 bytes. Por eso si el final del bloque no termina en una dirección múltiplo de 512 bytes se rellena el espacio restante con 0s. Como estos bytes de relleno no contienen información real dicha información se pierde.

TAR filesystem es simple de implementar y de entender pero posee claras desventajas:

- Acceso secuencial: Si yo quiero buscar un archivo en disco tengo que revisar todo el disco buscando la cabecera del archivo que estoy buscando(terriblemente lento). Lo mismo si yo quisiera leer una parte especifica de un archivo.
- Actualización compleja: Para actualizar datos de un archivo hay que escribir en el bloque de datos la actualización y reescribir los metadatos. Si hay que extender los datos del archivo habría que modificar los datos de relleno. Si estos no son suficientes entonces podría ser necesario borrar el archivo y reescribir una copia actualizada al final del TAR. La cosa se complica más si el medio físico no permite actualización hacia atrás(es el caso de una cinta).
- Archivos en subbloque: Se denomina subbloque a una sección del medio de almacenamiento que tiene un tamaño menor al bloque de división físico. En este caso cuando decimos que un archivo esta en un subbloque nos referimos a que la parte final del archivo no entra en un bloque completo(ya que la parte faltante se rellena con 0s). Debido a esto se desperdicia espacio y se lecturas y escrituras sobre el espacio de relleno(lo cual es más ineficiente).

TAR filesystem es util cuando no es necesario realizar actualizaciones.

Unix Filesystem

Unix Filesystem
El unix filesystem puede verse como una evolución natural del Questar - MFS filesystem. Tiene la siguiente estructura en disco:

Arranque	Superblock	Bitmap de bloques libres	Tabla de i-nodos	Dir-entries directorio raiz	Datos

Arranque: Contiene el codigo ejecutable necesario para montar el sistema de archivos. Una vez montado el filesystem, el bloque de arranque deja de usarse. Si el medio de almacenamiento no es booteable se marca este sector con un numero magico indicando que dicho sector no debe ser usado para bootear.

Superblock: El superblock contiene información para identificar el tipo de sistema de archivos e información sobre el resto de sectores en disco(tamaño y ubicación) junto con otros parámetros administrativos.

Bitmap de bloques libres: Cada bit identifica un sector logico del disco e indica si el sector esta libre o esta ocupado con datos.

Tabla de i-nodos: La tabla de inodos contiene todos los inodos del filesystem alineados de manera contigua. Cada inodo tiene un tamaño fijo(tipicamente 512 bytes) y tienen un indice asociado. Como consecuencia la cantidad maxima de archivos esta determinada por la cantidad maxima de indices disponibles.

Directorio raiz: Las dir entries del directorio raiz indican que archivos y directorios se encuentran en el. En Unix el directorio raiz se identifica como "/".

Datos: Los bloques de datos que pueden contener tanto datos de archivos como dir-entries que apuntan a otros archivos.

Inodos

Un inodo contiene la siguiente información:

Tamaño	Permisos	Fechas	Dueño	Grupo	Link count	Punteros directos	Punteros indirectos	Doble puntero indirecto	Triple puntero indirecto

Cada entrada tiene 4 bytes de tamaño. Se puede ver que los inodos guardan tanto los metadatos del archivo como la ubicación de los datos.

La ventaja de los inodos respecto a la FAT es que para acceder a todos los bloques del archivo solamente hay que tener el inodo cargado en memoria. Generalmente la cantidad de inodos cargados en memoria suele tener menos tamaño que la FAT entera.

Dir entries

Por cada archivo existente en el filesystem hay un direntry asociado al mismo. Cada dir entry contiene el nombre del archivo y el indice de inodo asociado.

Nombre	Indice de Inodo
14 Bytes	4 Bytes

El tamaño del indice de inodo depende, por supuesto, de la cantidad de bytes asociados al indice. En el ejemplo se presupone que se usa un entero de 32 bits.

Punteros directos, indirectos, doble indirectos y triple indirectos

Los punteros directos no es una entrada en si sino un conjunto de entradas, tipicamente son entre 8 y 12 entradas de 4 bytes. Cada puntero directo contiene la dirección de un sector lógico donde estan los datos del archivo.

En caso de que no sean suficientes los punteros directos para almacenar todos los datos del archivo se usa un puntero indirecto. El cual contiene la dirección de un sector lógico que contiene punteros directos. De esta manera si se quiere acceder al puntero directo numero 13 hay que ir primero a la dirección del puntero indirecto y luego acceder al primer puntero directo del bloque(suponiendo que inicialmente hay 12 punteros directos en el inodo).

Si esto no es suficiente se usa un doble puntero indirecto. Este puntero apunta a un bloque en disco que contiene punteros indirectos los cuales apuntan a punteros directos y estos apuntan a bloques de disco. Si es necesario acceder a un bloque por fuera de los punteros directos e indirectos entonces sera necesario hacer 3 accesos al disco.

Si con esto no alcanza se usa un triple puntero indirecto. El triple puntero solo se usa en archivos extremadamente grandes.

Usuario, grupos y permisos

Los permisos se indican con 9 bits los cuales se dividen en 3 grupos, un grupo para el usuario propietario del archivo, otro para el grupo al que pertence el archivo y el ultimo para el resto de usuarios.

- El primer bit indica permisos de lectura(**r**), y decide si el archivo puede ser leido o no.
- El segundo bit indica permisos de escritura(**w**), decide si el archivo puede modificarse o no.
- El ultimo es el bit de ejecucion(**x**), decide si el archivo puede ejecutarse o no.

Tambien hay un bit **d** que indica si el archivo es un directorio o no(recordar que los directorios no son más que archivos que contienen dir-entries). En el caso de directorios los permisos de archivos cambian un poco:

- El bit **r** indica si se pueden listar los archivos en el directorio.
- El bit **w** indica si se pueden crear archivos en el directorio.
- El bit **x** si se puede acceder al directorio o no.

Hay permisos adicionales como el SUID/SGID y los atributos extendidos, pero estos son los básicos.

Links duros y Links simbolicos

Unix permite hacer referencias a otros links mediante dos mecanismos. Los links duros(hard links) y los links simbolicos(symbolic links).

Link duro

Un link duro se representa en el sistema de archivos como una dir-entry que apunta a un inodo. Por lo que puede ocurrir que haya multiples dir-entries apuntando al mismo inodo(todas con el mismo peso, no hay una dir-entry maestro y otra esclavo). Hay 2 restricciones en este sistema:

- Solo puede haber links duros dentro del mismo volumen.
- No puede haber links duros a directorios, solo a archivos.

En base a esto se puede ver que en la estructura de archivos y directorios, los links duros forman un DAG dentro del volumen(grafo dirigido aciclico).

El inodo lleva una cuenta de cuantos dir-entries apuntan a el. Si se tiene que no hay ninguna dir-entry esta apuntando al inodo entonces los bloques de datos que apuntan a el se marcan como libres, por lo que pueden ser usados por otros archivos.

Link simbolicos

Un link simbolico es un archivo especial que contiene la direccion del archivo al que apunta. Quien tiene que obtener dicha direccion y resolverla es el sistema operativo.

No hay restricciones sobre a que tiene que apuntar un link simbolico, por lo que puede apuntar a archivos y directorios tanto del mismo volumen como de volumenes diferentes.

Dado que no hay restricciones, los links simbolicos en conjuncion con los links duros forman en la estructura de archivos y directorios un *grafo dirigido*.

Por lo tanto el link simbolico no apunta a un inodo sino que contiene el *path* o *ruta* del archivo al que apunta, en consecuencia si se borra o se renombra la dir-entry al que se apunta el link simbolico queda roto.

Operaciones con archivos

Veamos como son algunas operaciones con archivos en el filesystem de Unix:

Creación de un archivo

1. Para crear un archivo hay que cargar el inodo correspondiente del directorio padre en memoria.
2. Una vez cargado hay que escribir en los bloques de datos del directorio padre los datos de una nueva direntry(reservando bloques si los ya servados no son suficientes).
3. Una vez hecho esto hay que reservar un nuevo inodo de la tabla de inodos. En dicho inodo hay que escribir metadatos como fecha de creación, permisos, usuario propietario, grupo, tamaño, etc.
4. De ser necesario hay que reservar bloques de datos de acuerdo al tamaño del archivo y hacer que los punteros directos e indirectos apunten a estos nuevos bloques.
5. Finalmente actualizar la direntry del archivo con el nombre del archivo y el indice del inodo reservado.

Abrir un archivo

Para abrir un archivo hay que recorrer el path del archivo buscando las direntries de cada subdirectorio padre, una vez que se consigue la direntry del archivo hay que cargar el inodo en memoria y devolver un descriptor de archivo.

Leer un archivo

Para leer un archivo hay que obtener la cantidad de sectores logicos necesarios que hay que cargar en memoria. Para eso hay que seguir los punteros directos o los punteros indirectos si con los directos no son suficientes. Finalmente hay que actualizar en los metadatos la fecha de ultimo acceso.

Escribir en un archivo

Es similar a la lectura, hay que calcular en que sectores logicos hay que escribir siguiendo los punteros directos e indirectos. Si estos no son suficientes hay que exteneder el archivo reservando bloques en el bitmap de bloques. Puede ser necesario reservar más bloques en el caso de que haya que reservar un bloque de punteros directos/indirectos. Finalmente hay que actualizar en los metadatos la fecha de ultimo acceso, fecha de modificación y el tamaño si es necesario.

Borrar un archivo

Para borrar un archivo primero hay que borrar el dir entry del archivo(tomando nota del inode al que apuntaba). Una vez borrado hay que bajar en 1 la entrada de link-node del inodo. Si la misma llega a 0 hay que liberar los bloques de datos a los que apunta el inodo(poniendo los bits asociados en el bitmap a 0) y finalmente borrar el inodo. Si el archivo se borro el sistema operativo tiene que asegurarse que los demas procesos que tienen abierto el archivo sigan trabajando con el, no permitiendo que otros procesos abran el archivo(porque esta borrado).

Scheduling

First Come First Served

First Come First Served
First Come First Serve(FCFS) es un algoritmo de planificación de procesos en el cual el primer proceso que entra en estado *Ready* es el primero al que se le otorga acceso a la CPU.

FCFS sigue un esquema de multitarea cooperativa. Una vez que un proceso entra a la CPU, el SO no puede sacarle el control hasta que este termine o ceda el uso de la CPU voluntariamente.

FCFS es un algoritmo de planificación bastante pobre para sistemas interactivos(ya que un proceso interactivo que no se este ejecutando tardaría bastante en recibir tiempo de CPU) por lo que solo favorece a procesos largos.

FCFS también puede causar inanición si la cantidad de procesos en estado listo aumenta mas rápido que la capacidad de la CPU para atenderlos.

Aun así, FCFS reduce al mínimo la *sobrecarga administrativa*(es fácil y rápido elegir el siguiente proceso a ejecutar, al igual que cambiar de contexto) y no requiere hardware de apoyo(como un *temporizador*) por lo que sigue siendo ampliamente utilizado.

Highest Penalty Ratio Next

Highest Penalty Ratio Next
Es un algoritmo de planificación no apropiativo que busca un balance entre favorecer a procesos largos y procesos cortos.

Para cada proceso se calcula una puntuación de penalización P(inicialmente $P = 1$). Cuando un proceso es obligado a esperar ω unidades de tiempo se actualiza el valor de P con $P = 1 + \frac{\omega}{t}$. El proceso que se elige para usar la CPU es aquel con mayor valor de P.

Si la cola de procesos no crece mucho, HPRN evitara que los procesos largos sufran inanición.

Es un punto medio entre FCFS y SPN; la principal desventaja es que la elección del próximo proceso es $O(n)$(es necesario calcular el valor de P cada vez que se elige un nuevo proceso) por lo que no es viable si la cantidad de procesos en cola es muy alta.

Round Robin

Round Robin
Round Robin o **Ronda** es un algoritmo de planificación de procesos en el cual el tiempo físico de CPU se divide en unidades llamadas quantums, luego los quantums se dividen equitativamente entre los procesos en estado listo. Al repartir equitativamente el tiempo de CPU, la ronda da la misma calidad de respuesta tanto a procesos cortos como procesos largos.

Un proceso en la ronda solo puede ejecutarse en el quatum que recibió, si este no termino su ejecución entonces es sacado de la CPU para que el siguiente proceso entre en ejecución. Por lo tanto, a diferencia de First Come First Served, Round Robin emplea multitarea apropiativa.

El funcionamiento de la ronda puede regularse modificando el parametro q que indica el tiempo en ticks de un quantum. Una ronda con un quantum muy largo puede degenerar en un First Come First Served y en consecuencia perjudicar a los procesos cortos. Una ronda con un quantum muy corto distribuye mejor el tiempo de computo entre los procesos y da una mayor ilusión de que cada proceso tiene una CPU dedicada para el, en consecuencia también aumenta la cantidad de cambios de contexto y la *sobrecarga administrativa* por lo que el tiempo no útil de la CPU aumenta.

En general, el valor de q debe mantenerse menor al 80% de la duración promedio de los procesos para que la ronda sea eficiente.

Selfish Round Robin

Selfish Round Robin
Es un algoritmo de planificación de procesos que busca favorecer a los procesos que ya están en ejecución por sobre los que no.

Divide los procesos en 2 colas, una de *nuevos* y otra de *aceptados*. Los procesos en la cola de aceptados entran a la CPU de la misma manera que en una ronda normal mientras que los procesos nuevos esperan para entrar a la cola de aceptados.

A todos los procesos se les asigna una *prioridad* que ira cambiando conforme avanza el tiempo. La prioridad de los procesos nuevos incrementa a un ritmo a mientras que la de los procesos aceptados aumenta a un ritmo b.

Cuando un proceso nuevo alcanza la prioridad de un proceso aceptado, este pasa de nuevo a aceptado.

Si $b < a$, los procesos nuevos eventualmente alcanzaran la prioridad de un proceso aceptados y por lo tanto serán candidatos para entrar a la CPU.

Si $a \leq b$ el proceso en ejecución terminara antes de dar paso al siguiente, por lo que el esquema se convierte en un FCFS.

Si $b = 0$, los procesos nuevos serán aceptados inmediatamente y el esquema se convierte en una ronda.

Mientras $0 < b < a$ la ronda sera *relativamente egoísta* y le dará entrada a los procesos nuevos incluso si los aceptados llevan mucho tiempo ejecutándose.

Shortest Process Next

Shortest Process Next
Shortest Process Next(o **SPN**) es un algoritmo de planificación de procesos de multitarea cooperativa que busca favorecer con el uso de la CPU al proceso en estado listo que utilice la menor cantidad de CPU posible.

Naturalmente, saber la duración real del proceso es complicado, por lo que en realidad se trata de realizar un estimado de la duración usando el *promedio exponenicial*. Sea f un parametro que cumple $0 \leq f \leq 1$, si en su ultima ejecución el proceso uso q quantums de ejecución entonces el nuevo promedio se obtiene con la formula:

$$e' = f \cdot e + (1 - f)q$$

Donde e es el promedio original. El valor de f suele ser de 0.9 y el valor inicial de e suele ser un valor representativo de la duración del proceso(como la duración promedio de un proceso general).

SPN favorece a los procesos cortos y desfavorece a los largos, ya que un proceso largo puede esperar mucho tiempo a ser atendido.

SPN apropiativo

Tambien conocido como *PSPN*(*Preemptive Shortest Process Next*) o *Shortest Remaining Time*(*SRT*), combina el algoritmo SPN con multitarea apropiativa. Se favorece al proceso más corto en la lista de procesos listos, si un proceso con menor tiempo de ejecución entra a la cola, se interrumpe al proceso actual y el nuevo proceso entra a la CPU.
