Sistemas Operativos

¿Qué incluye un sistema operativo?

Componentes

Sistema operativo

Kernel + Programas del sistema

kernel

Es un programa que **siempre** se esta ejecutando. Provee funcionalidad mínima, como acceso a CPU, Memoria, dispositivos.

Programa del sistema

Extienden las funciones del **kernel**.

¿Dónde vive el so?

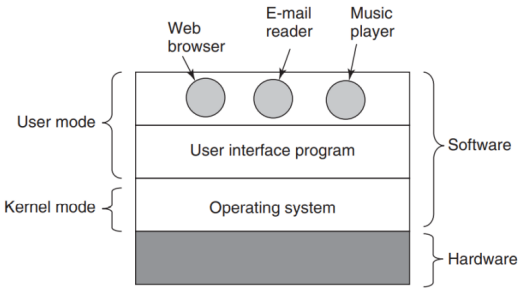


Figure 1-1. Where the operating system fits in.

¿Qué hay dentro de un sistema operativo?

- Administración de procesos
- Manejo de memoria
- Sistemas de archivos
- Protección y seguridad
- Control de I/O
- Manejo de errores

• Comunicación inter-procesos
• Contabilidad / logging
• Controladores (drivers) • etc . .

¿Cómo se invocan los servicios del sistema operativo?

Syscalls

Sistema operativo provee una interfaz de llamadas al sistema (syscalls)

- Cada vez que un programa desea solicitar algo al sistema operativo, debe invocar una syscall
- "Por favor, sistema operativo, haz esto por mi"
- Cada sistema operativo puede proveer un conjunto distinto de syscalls
- Sistemas operativos proveen bibliotecas (libraries) en algún lenguaje de programación (C) para invocar syscalls.

Sistema operativo provee interfaces de usuario (user interface) que enmascara syscalls

Kernel monolítico: Todo dentro en un único programa
• Servicios para el sistema y para el usuario en el mismo espacio
• Todos los servicios se ejecutan en modo kernel
• La falla de un servicio compromete al kernel
• Kernel complejo y de mayor tamaño Ejemplos
• Unix (BSD, System V)
• Linux
• MS-DOS, Windows 95/98/Me

Componentes pueden ser incluidos al momento de compilar, o bien durante la ejecución como módulos.Módulos extienden al kernel. Módulos se ejecutan en el espacio del kernel. Ayuda a controlar el tamaño del kernel (footprint)

Desventajas Kernel monolítico
Como todos los archivos se ejecutan en una sola imagen, si falla algún servicio, falla todo el sistema. Si el usuario necesita agregar cualquier servicio nuevo, entonces el usuario necesita modificar todo el sistema. No es fácil de mantener y depurar.

Microkernel lo más pequeño posible
Mantener el kernel del menor tamaño posible
• Solo servicios básicos en el kernel
• Funcionalidades (servers) se ejecutan en user space
• Errores en los servicios no afectan al kernel
• Kernel más sencillo, pequeño y fácil de portar
• Ejecución con overhead por comunicaciónç
• Procesos se comunican (IPC) mediante paso de mensajes
Ejemplos
• Mach, implementación de UNIX con microkernel (CMU)
• GNU/Hurd, inspirado en Mach
• MINIX 3, desarrollado por Andrew S. Tanenbaum

Desventajas Microkernel La ejecución en microkernel es lenta ya que la comunicación entre la aplicación y el hardware del sistema se establece a través del paso de mensajes.

Aislamiento de fallas
• el diseño del micronúcleo ayuda a aislar las fallas y evitar que afecten a todo el sistema. Si un servidor u otro componente falla, se puede reiniciar o reemplazar sin causar interrupciones al resto del sistema. Modularidad
• debido a que el núcleo y los servidores se pueden desarrollar y mantener de forma independiente, el diseño del micronúcleo permite una mayor modularidad. Esto puede facilitar la adición y eliminación de funciones y servicios del sistema

Modos de protección
Al iniciar el computador se ejecuta código de la tarjeta madre
• BIOS: Basic Input/Output System, o bien
• UEFI: Unified Extensible Firmware Interface
Este código tiene dos misiones fundamentales
• Inicializar/identificar el hardware
• Ubicar y arrancar mediante un bootloader el kernel en modo privilegiado El kernel es el único software que funciona en modo privilegiado

El hardware posee dos modos de protección

- **Kernel Mode**: acceso completo al hardware
- **User Mode**: acceso a un set restringido de instrucciones

Si se está en user mode el hardware genera una interrupción (protection fault), o bien ignora la instrucción

Modos de protección

Instrucciones privilegiadas (ring 0)

- Modificar vector de interrupciones (CLI, IRET)
- Acceder a dispositivos de I/O
- Modificar timer del computador
- Detener el computador (halt, HLT)
- Cambiar el modo de protección (user a kernel, o viceversa)
- Modificar tablas de acceso a memoria (pagetable, INVLPG)
- Almacenar estado de la máquina (SMSW)

Estas instrucciones son un ejemplo de las instrucciones privilegiadas del set de instrucciones Intel 64.

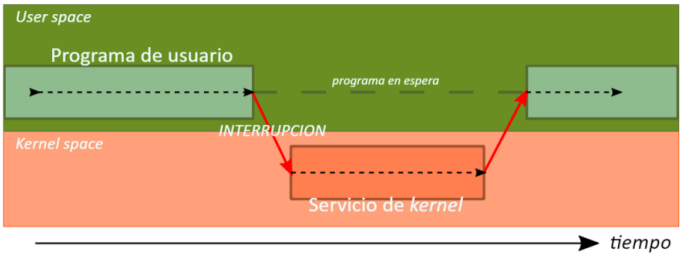
Hardware moderno posee 4 o más modos de protección

- Se les llama rings (anillos) de protección
- Permite implementar virtualización
- Intel 64 posee 4 niveles y una lista de instrucciones privilegiadas

Mientras nadie lo llame, el sistema operativo no hace nada.
Como es el flujo entre el usuario que intenta ejecutar algo.

- Primero: El programa de usuario genera un **trap**: interrupción generada por software.
- Segundo: Se pasará el control al sistema operativo: kernel space
- Tercero: Sistema operativo ejecuta servicio solicitado por el usuario
- Por útlimo: Una vez terminado el servicio, sistema operativo regresa el control al programa de usuario.

Como es el flujo entre el usuario que intenta ejecutar algo.



En la imagen se puede ver que cuando el programa queda en espera el sistema operativo ejecutará el servicio solicitad

Syscalls: Llamadas al sistema Interfaz para solicitar servicios al sistema operativo
• Las syscalls son la manera precisa de llamar al sistema operativo.
• Sistemas proveen bibliotecas que implementan syscalls.

¿Qué pasa cuando copio un archivo?

copy source.txt dest.txt

- Primero abre source.txt (stat, open)
- El SO va chequear: ¿si existe?, ¿tiene permiso de lectura?, . . .
- Luego va a leer source.txt hacia alguna región de memoria (mmap, read)
- Después va escribir desde la memoria hacia dest.txt (write)
- Por último liberará la memoria y guardar los archivos (munmap, close)

Todas estas acciones requieren llamadas al sistema

Pruebe con: strace cp source.txt dest.txt

Procesos

PROCESOS

Un abstracción para un *programa en ejecución*

Proceso = Código (programa) + Recursos

¿Qué recursos?

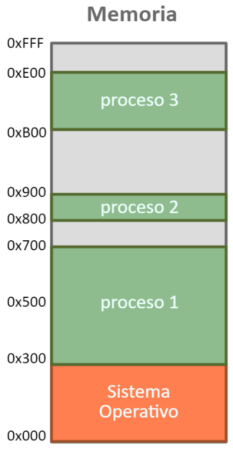
- Memoria, archivos
- Sockets, dispositivos de I/O, bibliotecas
- etc …

Multiprogramación permite mantener **múltiples procesos** en memoria.

- Un proceso en memoria está preparado para ser ejecutado

CPU puede "atenderlos" en algún orden (*scheduling*)

- Cuando los atiende "simultáneamente" tenemos **multitasking**



Composición: ¿Qué hay en un Proceso?

- **Código** (.text, información estática)
- **Datos** (.data): variables globales
- **Stack**. Cada item del stack representa un llamado a función (*call frame*), y contiene:
    - Parámetros
    - Variables locales
    - Lugar de retorno (donde estaba la ejecución anterior, PC)
- **Heap**. Memoria asignada dinámicamente (durante la ejecución)

Sistema Operativo mantiene **Tabla de Procesos**.
Información de cada procesos almacenada en su PCB

- Estado
- Identificador (PID, *Process ID*)
- Program Counter (PC)
- Registros de CPU: *estado de ejecución*
- Información de *scheduling*: prioridades, tipo de cola, …
- Información de memoria: límites, tabla de páginas/segmentos, …
- Contabilidad (*accounting*)
- Información de I/O: archivos y dispositivos abiertos, …

CREANDO PROCESOS

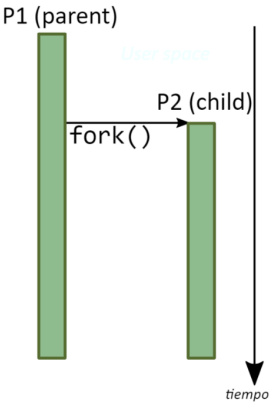
Un proceso (*parent*) crea a otro proceso hijo (*child*)

¿Quién sigue ejecutando?

- *Parent* e *hijo* continúan ejecutando concurrentemente.
- Ambos existen en la memoria del computador.
- Cuál proceso continúa en estado *running* depende de la implementación.

¿Qué "personalidad" tiene el hijo? (código, espacio de direcciones)

- El hijo es un duplicado casi exacto del padre
- El hijo es un **nuevo proceso** (otro espacio de direcciones)
- El hijo **copia** la memoria del padre a su nuevo espacio
- El hijo continúa ejecutando en **la misma instrucción** que el padre: el retorno de `fork()`



Syscall fork()

- Crea un nuevo proceso como **copia** del padre.
- Ambos continúan ejecutando desde la instrucción de retorno de fork()
- fork() retorna PID del hijo al padre, y retorna 0 al hijo

Syscall exec()

- Carga un binario en memoria **reemplazando** el código de quien lo llamó, e inicia su ejecución.
- El programa nuevo se "roba" el proceso (la memoria se sobreescribe)

Syscall wait()

Espera el término de un proceso hijo

Syscall exit()

Termina el proceso con un código de retorno dado, y lo entrega al padre

Un proceso puede enviar *señales* (**signal**s) a otros procesos

Syscall `kill()`

- Envía una **señal** a otro proceso (por defecto *SIGTERM*)
- `kill -l` permite ver las señales disponibles
- `SIGTERM` indica al proceso que debe terminar
- `SIGKILL` elimina al proceso de la tabla de procesos (sin piedad)

Linux Signals

Signal	Value	Description
1	SIGHUP	Hang up the process.
2	SIGINT	Interrupt the process.
3	SIGQUIT	Stop the process.
9	SIGKILL	Unconditionally terminate the process.
15	SIGTERM	Terminate the process if possible.
17	SIGSTOP	Unconditionally stop, but don't terminate the process.
18	SIGTSTP	Stop or pause the process, but don't terminate.
19	SIGCONT	Continue a stopped process.

Si el padre muere, ¿deben morir los hijos?

- Sistema Operativo podría permitir (o no) que un hijo exista si el padre ha terminado.
- Linux permite que procesos hijos sigan viviendo, aún cuando el padre haya terminado.
- Sin embargo, podría parecer que los hijos sí mueren.

*Procesos **huérfanos***

- Linux: Cuando un padre termina (`exit`) o muere (`kill`), sus hijos quedan **huérfanos** y pasan a ser hijos de `init`
- Sin embargo, podría parecer que los hijos sí mueren (las condiciones pueden ser complicadas).
- `init` hace `wait()` periódicamente por sus hijos.

*Procesos **zombies***

- Linux: Cuando un proceso termina y su padre no hace `wait()`.
- Proceso terminado no se borra inmediatamente de la tabla de procesos (pero tampoco ejecuta).
- Proceso queda en estado **zombie** hasta que el padre hace `wait()`.

Scheduling

SCHEDULING

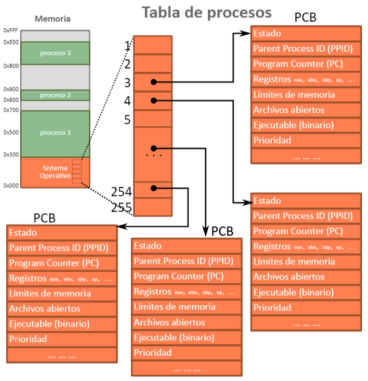
CPU Scheduling a.k.a. Planificación de CPU

Tenemos:

- Múltiples procesos en memoria (*multiprogramación*), ordenados en una tabla de PCBs
- Algunos procesos en estado *ready* (listos para ejecutar)
- CPU que puede atender **solo un proceso** a la vez

Y queremos:

- ***Multitasking***: asignar tiempo a múltiples procesos



*Sistema Operativo debe elegir cuál de todos los procesos en estado **ready** debe ejecutar a continuación: **scheduling***

Long-term Scheduler

- Admite procesos en la *cola ready*.
- Determina el **grado de multiprogramación** (cantidad de procesos en memoria).

Short-term Scheduler (a.k.a. **dispatcher**)

- Selecciona un proceso de la cola *ready* para ejecutar.
- Ejecuta el cambio de contexto

Medium-term Scheduler

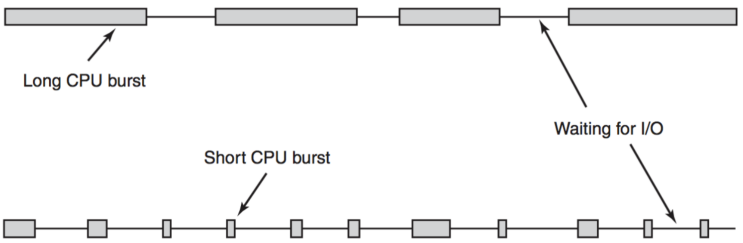
- Modifica temporalmente el grado de multiprogramación
- Ejecuta **swapping** copiando memoria RAM a disco, y de disco a RAM.

MODELO DE EJECUCIÓN DE UN PROCESO

No todos los procesos se comportan igual. Alternan entre dos fases:

- Uso de CPU (**CPU-burst**)
- Espera por I/O (**I/O-burst**)

Procesos suelen estar dominados por una u otra etapa:



Por tipo de interrupción

***Preemptive** (expropiativo)*

Utiliza interrupciones (ej: de reloj (*timer*)) para decidir cuando sacar a un proceso de ejecución.

***Non-Preemptive** (colaborativo o no-expropiativo)*

Permite que un proceso ejecute hasta que:

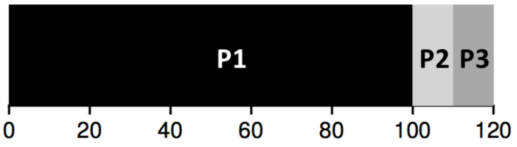
- El proceso deja voluntariamente la CPU, ó
- El proceso se bloquea en I/O, ó
- El proceso termina

ALGORITMO DE SCHEDULING (BATCH)

FIRST-COME, FIRST-SERVED (FCFS)

Orden de llegada. Cola FIFO.

	Llegada	CPU-burst	Término	Turnaround
P1	0	100	100	100
P2	1	10	110	109
P3	2	10	120	118



Turnaround time promedio: 109

Si P2 hubiese llegado en $t = 0$, y P1 hubiese llegado en $t = 1$, entonces *turnaround time promedio* $\rightarrow 79$

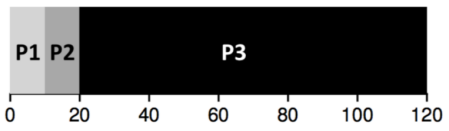
- Non-Preemptive
- 👍 Simple
- 👎 Poco predecible. *Convoy effect*

ALGORITMO DE SCHEDULING (BATCH)

SHORTEST JOB FIRST (SJF)

El más corto primero

	Llegada	CPU-burst	Término	Turnaround
P1	0	10	10	10
P2	2	10	20	18
P3	1	100	120	119



Turnaround time promedio: 49

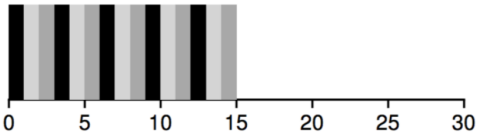
- 🎯 **Óptimo**
- 😭 No sabemos cuánto demora cada *CPU-burst*
- Versión preemptive: **Shortest Remaining Time Next**. Elige al que le queda menos tiempo.
- 💀 Potencial **inanición** (*starvation*) de procesos largos.

ALGORITMO DE SCHEDULING (INTERACTIVE)

ROUND-ROBIN (RR)

Un turno (*quantum*, *time slice*) para cada uno.
Ejemplo con $q = 1$

	Llegada	CPU-burst	Término	Turnaround	Inicio ejec.	T.Respuesta
P1	0	5	13	13	0	0
P2	0.1	5	14	13.9	1	0.9
P3	0.2	5	15	14.8	2	1.8



- Turnaround time promedio: 13.9
- Response time promedio: 0.9

- Con n procesos, cada uno recibe $1/n$ de CPU. **Fair!** ⚖️
- Ningún proceso espera más de $(n-1) \times q$ para ejecutar
- Altamente dependiente de la elección de q (¿qué pasa si q es muy pequeño o muy grande?)

PRIORITY SCHEDULING

Cada proceso tiene asociada una prioridad

- Se atienden por orden de prioridad
- Prioridades iguales: FCFS ó RR
- SJF es un *caso particular* de este algoritmo (¿por qué?)
- Prioridades pueden ser estáticas o dinámicas
- 💀 Starvation de procesos con baja prioridad
- 👴 Aging: incrementar prioridad de procesos que llevan más tiempo

ALGORITMO DE SCHEDULING (INTERACTIVE)
MULTILEVEL FEEDBACK QUEUE (MLFQ)

Fernando J. Corbató (Turing Award 1990) et al. ***"An Experimental Time-Sharing System"****, IFIPS 1962.*

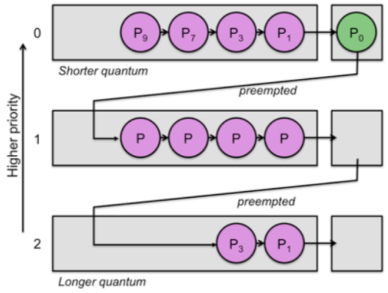
Usado en CTSS (1962) y Multics (1965)

Scheduler que optimize turnaround time (como SJF), pero minimizando response time (como RR)

Múltiples colas con distinta prioridad

- R1. Si $\text{priority}(A) > \text{priority}(B)$), ejecutar A
- R2. Si $\text{priority}(A) = \text{priority}(B)$), ejecutar A y B con RR
- R3. Procesos entran en la cola con **mayor** prioridad
- R4. Si un proceso usa su q (acumulado en todos sus turnos), su prioridad se reduce
- R5. Después de un tiempo S, todos los procesos se mueven a la cola con mayor prioridad

ALGORITMO DE SCHEDULING (INTERACTIVE)
MULTILEVEL FEEDBACK QUEUE (MLFQ)



Base para schedulers de Windows y MacOSX

- R1. Si $\text{priority}(A) > \text{priority}(B)$), ejecutar A
- R2. Si $\text{priority}(A) = \text{priority}(B)$), ejecutar A y B con RR
- R3. Procesos entran en la cola con **mayor** prioridad
- R4. Si un proceso usa su q (acumulado en todos sus turnos), su prioridad se reduce
- R5. Después de un tiempo S, todos los procesos se mueven a la cola con mayor prioridad

Threads

Thread: Como un proceso, pero más liviano.
Unidad básica de uso de CPU

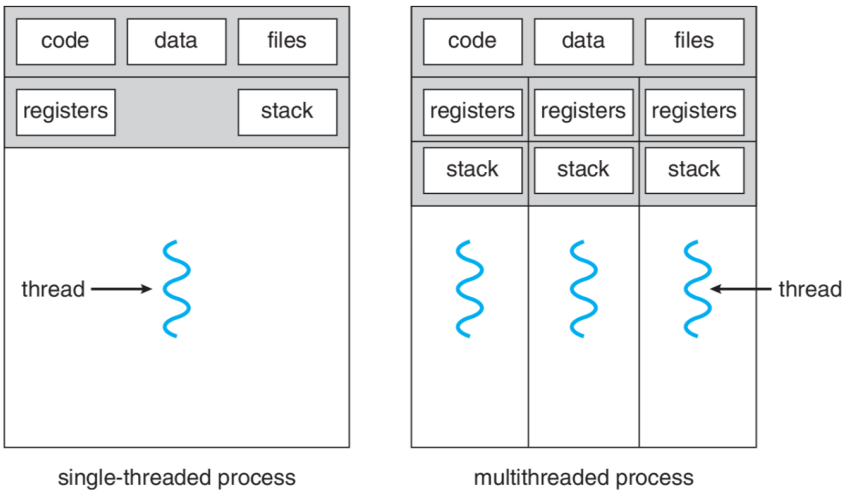
- Thread ID (tid)
- Program Counter
- Registros
- Stack

El resto está compartido con otros *thread*s del mismo proceso.

THREADS

Single-threading vs *Multi-threading*

Procesos pueden tener uno o más *threads*



single-threaded process multithreaded process

Apuntes clases

Cuando tenemos múltiples ejecutando tendremos compartido : código, datos globales, los archivos.

Beneficios de los threads:

- Los threads nos ayudan a organizar mejor nuestro código.
- Threads pueden ejecutar distintas partes del código de un proceso. Ayudan a establecer de manera lógica el paralelismo.

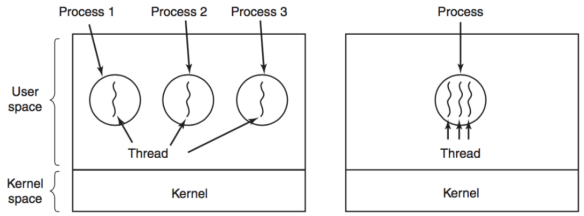
La cpu atiende a un threads simultáneamente . Si tenemos una cpu no ganamos nada usando múltiples threads

¿Cómo se implementan los threads?
Modelo de threads surgen de separar los recursos de la ejecución.
Los threads comparten el mismo espacio de memoria.

MODELOS DE THREADS

Threads surgen de separar el concepto de recursos y ejecución

- **Procesos** agrupan recursos (y ejecución)
- **Threads** son solamente unidades de ejecución: **lightweight processes**



MODELOS DE THREADS

Threads comparten

- Espacio de memoria
- Variables globales (sección *data*)
- Archivos abiertos
- Procesos Hijo
- Alarmas pendiente
- Señales y manejadores de señales
- Información de *accountability*

*Threads **NO** comparten*

- *Stack*
- *Registros* (incluyendo PC)
- Estado

Información privada tiene que ver con *ejecución*.
Información compartida tiene que ver con *recursos*

Stack manejo de múltiples stacks se crean stacks de 132 kb cada uno y hay suficiente espacio para todos son espacios distintos se puede alocar usando mmap

Funciones típicas

- exit() para terminarlos
- create() para crearlos
- join() para esperar otro threads

- yield() permite entregar el turno a otro threads

POSIX threads biblioteca para threads

Implementación de threads

Algunas decisiones de diseño de la implementación de threads no son tan obvias Por ejemplo, ¿ cómo funciona el Fork, quien se bloquea ? Todos o uno no es obvio por eso existe más de un espacio donde los threads se pueden implementar .

Dos opciones donde crear los threads

- **User** space
- **Kernel** space
- **Hybrid** (ok, 3 opciones)

Opción 1:el espacio del usuario, implementar los threads acá hace que hay que usarlos como si fueran una biblioteca.
Kernel no ve threads. Solo ve procesos single threaded.

- 😃 Portable. Implementable en cualquier S.O.
- 😃 Context switch y scheduling más rápido.
- 😃 Mejor escalabilidad (usa memoria del proceso)
- 😞 Llamadas a syscalls pueden bloquear el proceso
- 😞 Requiere scheduling cooperativo (no hay timers)
- Piensen en su uso en multithreaded web server
- Implementaciones usan variante no-bloqueantes de algunas syscalls

El cambio de contexto y el scheduling es más rápido porque no hay intervención del so no hay syscalls. Pero si es que llama por ejemplo a la syscall read va a bloquear al proceso completo incluyendo los otros thread. No hay forma de interrumpir solo a un threads el so va a matar al proceso completo.

La opción dos es el kernel space usualmente hay una cierta cantidad por defecto de threads a nivel de kernel.
No necesita bibliotecas de usuario.*Syscall* manejados por el S.O.

- 😃 No requiere bibliotecas. Soporte nativo.
- 😃 *Syscall*s no bloquean el proceso.
- 😞 *Syscall*s más costosas. Requiere llamadas al kernel.
- 😞 Escalabilidad limitada por el S.O.
- 😞 Semántica de fork() no es clara.
- 😞 Semántica de signals no es clara.

Para hacer cualquier cosa tengo que trabajar como un proceso regular.
Escalabilidad limitada por el so no dejan más de x cantidad de threads.
La implementación del fork no es tan clara el SO como está implementado ahí me dice cómo funciona. Para no tener que crear y destruir threads a nivel de kernel se usa un poll de threads previamente creados. Límite por ejemplo 1024 threads

La alternativa que se usa es tener threads a nivel de kernel y permitir threads a nivel de usuario.

Es bueno porque los threads de kernel ya vienen creados y los threads de usuario son livianos de crear.
Cómo posix threads es una Api no tiene una implementación fija es decir puede tener por debajo algo híbrido como en Linux o una de solo user space o Kernel space.

Sincronización de Threads

Race condition o condición de carrera. Dos threads quieren usar la misma sección de código pero ¿qué pasa si uno llega primero que el otro?¿cambia el resultado? Sí.

El problema de la **sección Crítica** es la parte del código donde solo un thread puede estar corriendo al mismo tiempo.

Recordemos que los threads son la unidad mínima de procesamiento y que comparten memoria y compiten por el uso de la cpu .

Race condition la operation depende cómo se entremezclaron los threads para su resultado, deberíamos programar para que nunca existan estas situaciones de competencia

Hay soluciones que me ayudan a solucionar el problema de la sección crítica.

1. exclusión mutua sólo un threads a la vez en la SC
2. Progreso (al menos 1 puede entrar) para que se avance. No se pueden bloquear los threads uno a otros
3. Espera acotada (ausencia de inanición) entrará en una cantidad acotada de tiempo

EL PROBLEMA DE LA SECCIÓN CRÍTICA

Una solución al problema de la sección crítica (SC) debe cumplir:

Exclusión Mutua

A lo más un *thread* está en su SC.

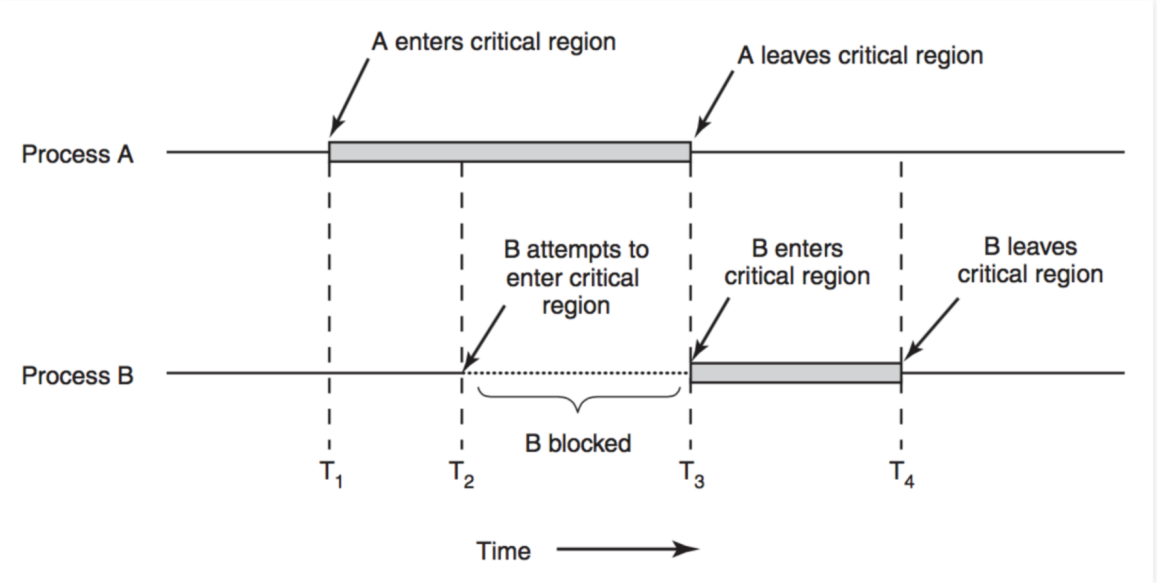
Progreso

Al menos un *thread* puede entrar a su SC.
Si ningún *thread* está en su SC, y hay *thread*s que desean entrar, entonces los que quieren entrar deben decidir quién entra, y decidirlo en un tiempo acotado.

Espera acotada *(ausencia de inanición)*

Si un proceso quiere entrar a su SC, podrá hacerlo luego de una cantidad **finita** de tiempo.

Ejecución deseada:



Algunas soluciones al problema de la SC

Una solución es deshabilitar las interrupciones: si lo soluciona pero esta ejecución es secuencial. Esto lo hacen los módulos del SO porque nos interesa que se sirvan.

Pero para el usuario -> que tiene que vivir con interrupciones

ALGUNAS SOLUCIONES

En el mundo del usuario estamos a merced de las interrupciones

Solución con variable compartida `lock`

```
int lock = 0;  // 0==free, 1==busy
```

Código de cada *thread*:

```
while(TRUE) {
    while(lock);     // busy waiting (spinlock)
    lock = 1;
    /* ... SC ... */
    lock = 0;
    /* ... out of SC ... */
}
```

¿Problemas?

Una solución es usar la variable compartida Lock que valdrá 0 cuando nadie la ha tomado y 1 si alguien la tomo.

Si los dos ven el lock =0 al mismo tiempo ?
No cumple con la exclusión mutua.

Solución con `turn`, para alternar entre dos *threads*

Variable compartida `turn` determina quién puede entrar.

```
int turn = 1;  // o 0 ...
```

Código para *thread* 0 y *thread* 1:

```
/* Thread 0 */
while(TRUE) {
    while(turn != 0);
    /* ... SC ... */
    turn = 1;
    /* ... out of SC ... */
}
```

```
/* Thread 1 */
while(TRUE) {
    while(turn != 1);
    /* ... SC ... */
    turn = 0;
    /* ... out of SC ... */
}
```

¿Cumple todas las condiciones?

Otra solución variable comparativa Turn
Mientras el turno no sea el mío me quedaré en el while
Cumple con exclusión mutua
Pero no cumple con progreso
¿Por qué? que pasa si el thread 1 quiere entrar 2 veces y el 0 no quiere entrar van a quedar atrapados en el turno del 0.

Solución de G.L.Peterson, 1981 (no la primera, pero la más simple)
Variable compartida `turn` determina quién puede entrar.

```
int turn = 1;                // o 0 ...
int flag[2] = {false, false}; // indica t_i está interesado en entrar
```

Variable local `me` puede ser 0 ó 1

```
int me = // ... 0 ó 1;    /* Peterson es para dos threads */
int other = 1-me;
while('o') {
    flag[me] = true;  /* wants */
    turn = other;
    while(flag[other] && turn == other); /* wait */
    /* ... SC ... */
    flag[me] = false;
    /* ... out of SC .../ */
}
```

Otra solución Peterson incorpora dos car compartidas turn y flag
Flag indica si el thread quiere entrar
Entonces while el otro quiere entrar y es su turno yo espero

El turno va se 1 o 0 entonces al menos un threads va poder pasar
La línea de turn=other me permite que halla espera acotada
Si bien cumple no es muy eficiente

Sincronización por hardware

Los procesadores modernos me permiten instrucciones atómicas que por ejemplo la línea del while lock la vea solo uno a la vez.

`test_and_set()` (TSL)

Algunas arquitecturas proveen instrucciones atómicas del tipo `test_and_set`

La instrucción, que se ejecuta atómicamente, es equivalente a este código:

```
bool SOFTWARE_test_and_set (bool *target) {
    bool old_value = *target;
    *target = true;

    return old_value;
}
```

Test_and_set TSL (bool)ejecuta algo equivalente a este código
Almacena un bool y lo cambia
Toma el old value lo cambia a true y retorna su valor anterior
Sirve para el caso del lock

¿Qué significa atómica? -> que no puede ser interrumpida

Compare and swap XCHG

Código equivalente a `compare_and_swap()`

```
int SOFTWARE_compare_and_swap (int *value, int expected, int new_value) {
  int previous = *value;

  if(*value == expected)
    *value = new_value;

  return previous;
}
```

Otra implementación simple, gracias a una instrucción atómica

```
int lock = 0;  // 0==free, 1==busy
```

```
while(true) {
    /* ... ... ... */
    while (compare_and_swap(&lock, 0, 1));
    /* ... SC ... */
    lock = 0;
    /* ... Out of SC ... */
}
```

Toma un valor lo compara con el esperado y si son iguales retorna un nuevo valor
Lock 0 1 compara si el look está en 0 si lo está dale al primer thread que llega la sc y cámbialo a 1

Busy waiting while true espera ocupada esperando sin hacer nada. Sólo cuando el tiempo de espera es muy corto se justifica tener busy waiting.

MUTEX LOCKS

La más simple: **MUTual EXclusion locks** (*mutex locks*)

- `Lock::acquire();` toma el *lock*
- `Lock::release();` libera el *lock* (error si no está tomado)

```
struct Lock {
  bool value = false; // false==free, true==busy
}
```

Implementación sin (mucho) *busy waiting*

```
void acquire() {
  while(test_and_set(&value))
    thread_yield();           // otras implementaciones usan sleep() y colas de espera
}
void release() {
  value = false;
}
```

SEMÁFOROS

E.W.Dijkstra (1965) propone una nueva primitiva de sincronización: **semáforos**, y las introduce en Algol 68.

Semáforos permiten un número limitado de *thread*s en una sección crítica.

Semáforo

Un semáforo S incluye un contador y dos operaciones:

- *proberen* `P()` ó `wait()` ó `down()`. Intenta decrementar el valor.
- *verhogen* `V()` ó `signal()` ó `up()`. Incrementa el valor.

E.W.Dijkstra era holandés, pero si no saben holandés es más fácil recordar `Pedir`/`Votar`

Bounded-buffer con semáforos

```
struct lock l; int stock = 0; T storage[N];
struct semaphore full;  full.init(0);
struct semaphore empty; empty.init(N);
```

```
int in = 0;
while('p') {
    item = produce();
    empty.P();            // Wait one empty slot. P=='wait'=='down'
    l.acquire();
    storage[in] = item;
    in = ++in%N;               // rotate index
    stock++;
    lock.release();
    full.V();             // One more full slot. V=='signal'=='up'
}
```

```
int out = 0;
while ('c') {
    full.P()             // Wait one more item. P=='wait'=='down'
    lock.acquire();
    item = storage[out];
    out = ++out%N;             // rotate index
    stock--;
    lock.release();
    empty.V();           // One more empty slot. V=='signal'=='up'
    consume(item);
}
```

VARIABLES DE CONDICIÓN (CONDITION VARIABLES)

Semáforos sirven para controlar cantidad de accesos a una región protegida
Mecanismo consisten en bloquearse (`P()`) bajo una condición (contador negativo) y avisar o señalar (`V()`) que una condición se ha cumplido (contador mayor a 0)

Variables de condición permitan bloquearse bajo condiciones arbitrarias.

- `Condition::wait()` bloquea **siempre** el *thread*
- `Condition::signal()` despierta a un *thread* bloqueado, si lo hay

Bounded-buffer con *condition variables*

```
struct lock l;
struct condition condCons, condProd;
int stock = 0; T storage[N];
```

```
int in = 0;
while('p') {
    item = produce();
    l.acquire();                        // with mutex
    while(stock == N) condProd.wait(l); // wait if full
    storage[in] = item;  in = ++in%N;   // fill and rotate index
    stock++;
    condCons.signal();                  // signal new item for consumer
    l.release();
}
```

```
int out = 0;
while ('c') {
    l.acquire();
    while(!stock) condCons.wait(l);     // wait if empty
    item = storage[out]; out = ++out%N; // extract and rotate index
    stock--;
    condProd.signal();                  // signal new slot for producer
    l.release();
    consume(item);
}
```

Condition variables utilizan un *lock* y lo liberan antes de bloquearse.

PROBLEMA DE LOS LECTORES Y ESCRITORES
READERS-WRITERS

Planteado por Courtois et al. (1971). Modela acceso a una base datos.

- Permitir acceso de n lectores simultáneos, pero solamente 1 escritor

```
struct semaphore rw; rw.init(1);
struct semaphore mutex; mutex.init(1);
int readers = 0;        // leen o quieren leer
```

Writers

```
while('w') {
    rw.P();
    /* ... WRITE ... */
    rw.V();
}
```

Readers

```
while ('r') {
    mutex.P();
    readers += 1;
    if(readers == 1)
        rw.P();
    mutex.V();
    /* ... READ ... */
    mutex.P();
    readers -= 1;
    if(readers == 0)
        rw.V();
    mutex.V();
}
```

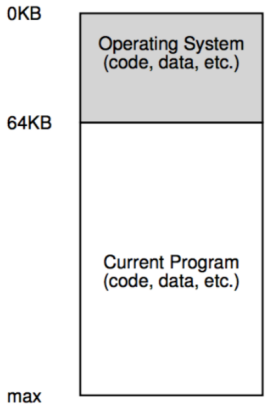
Memoria

g

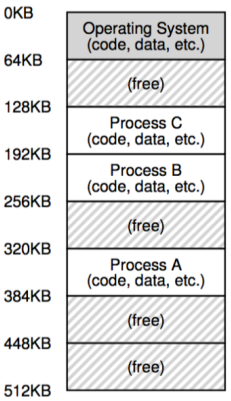
DIRECCIONES DE MEMORIA

- Un gran arreglo de *bytes*
- Cada uno con su propia **dirección**

Sin multiprogramación …



Con multiprogramación …



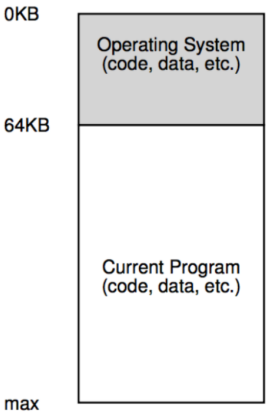
DIRECCIONES DE MEMORIA

Con solamente un proceso en memoria …

```
mov %eax, 0x10001       ; a += 3
add %eax, $0x3
mov 0x10001, %eax
```

La vida del compilador es (más) fácil

- Programa siempre se carga en la misma dirección
- Cada variable tiene la misma dirección en memoria en cada ejecución



El primer problema que surgió al crear la multiprogramación fue que los procesos no intentarán usar la misma memoria que otro proceso.

La dirección de memoria en el espacio virtual es un espacio de memoria asignado para el proceso.

La memoria del computador está organizada como una arreglo muy grande de bytes.

¿Cómo re-localizamos variables ?
La memoria que se le asigna a un proceso es variable es 128 +x

Abstracción: espacio de direcciones
El so le entrega a cada proceso un espacio virtual de direcciones.

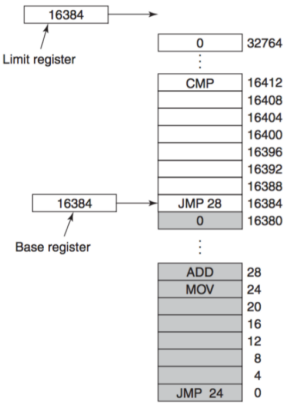
ESPACIOS DE DIRECCIONES

Dos problemas: (1)**relocalización** y (2)**protección**

Abstracción: ***espacios de direcciones***

CPUs desde CDC6600 hasta Intel 8088 integran dos registros:
`base` y `limit`

- Al cargar cada procesos se cargan estos registros
- Proceso quiere ejecutar `JMP 28`
- Pero *hardware* ejecuta `JMP 28+16384`
- Si la dirección es mayor que `limit`, se genera un *trap*
- MMU: **Memory Manager Unit**



Antes como se solucionaba habían dos registros el base y el limit.

Base primera dirección limit última y la mmu la memory manager unit al final era una función que veía que la ejecución de algo fuera mayor a base y menor a limit.
Eso se usaba hasta el intel 8088.
Funciona bien pero que pasa si un proceso no me cae en memoria ? Es difícil anteponerse al tamaño de los procesos.
Mejor voy a cargar ese proceso en partes distintas.

ESPACIOS DE DIRECCIONES

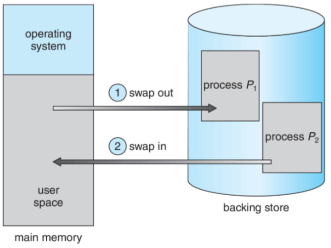
SOBRECARGA DE MEMORIA

¿Cuántos procesos caben en memoria? ¿Qué pasa si la memoria se llena?

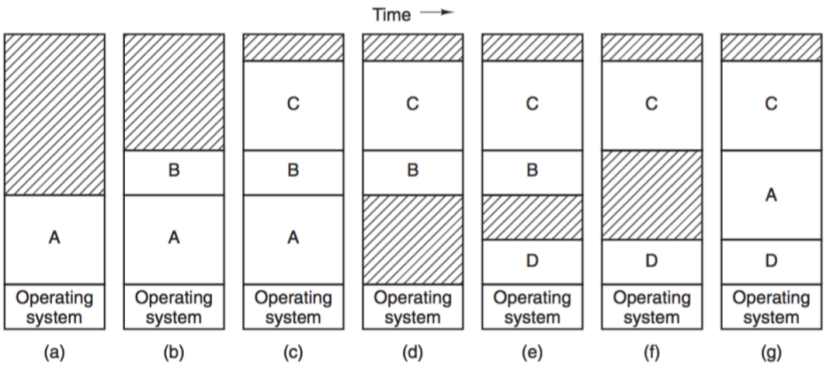
*Operaciones de **swapping***

- **Swap-out**. Memoria del proceso se copia a disco.
- **Swap-in**. Memoria del proceso se copia de disco a memoria.

¿Qué pasa con la relocalización?



Procesos pueden ser cargados y descargados de la memoria



Memoria del sistema queda con huecos

SEGMENTACIÓN

Problema: un espacio de direcciones muy grande.

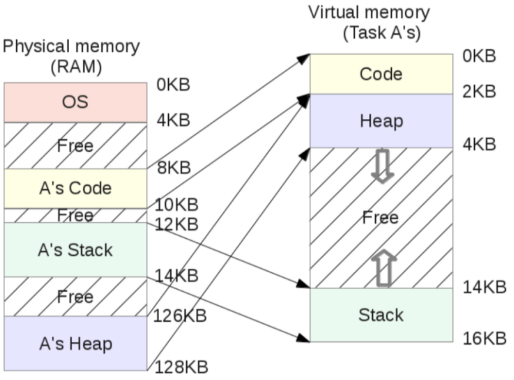
Segmentación

Se divide el espacio de direcciones en varios espacios más pequeños

- Espacios más pequeños pueden ser asignados más fácilmente

Segmento: espacio de direcciones contiguo

- Ejemplo: código, datos, heap, stack, …



IMPLEMENTANDO SEGMENTACIÓN

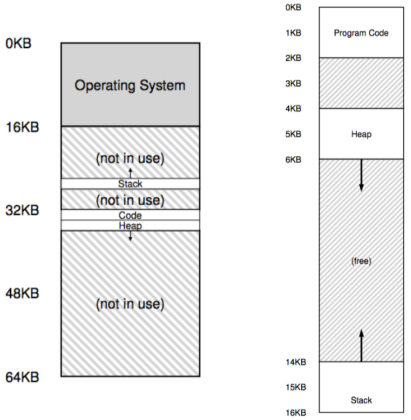
Ahora hay varios espacios lineales de direcciones.

MMU debe ser capaz de hacer la traducción

Segment	Base	Size
`code`	32768 (32KB)	2048 (2KB)
`heap`	34816 (34KB)	2048 (2KB)
`stack`	28672 (28KB)	2048 (2KB)

Tabla de Segmentos

- Dirección virtual (lógica): 100
  - Está en segmento `code`
  - Base + dirección $= 32768 + 100 = 32868$
- Dirección física: 32868



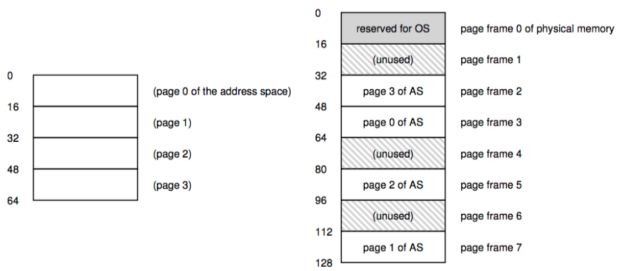
PAGINACIÓN

Problema: un espacio de direcciones muy grande.

- Idea 1: Dividir el espacio de direcciones en **segmentos**
- Idea 2: Que los segmentos sean del mismo tamaño: **páginas de memoria**

Paginación

- Espacio virtual: **páginas**
- Espacio físico: ***frames*** (marcos)
- Páginas y *frames* del **mismo** tamaño

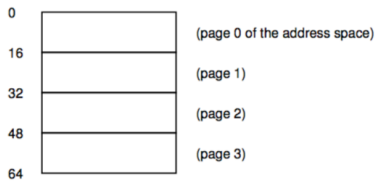


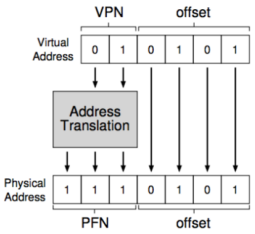
Sistema con 128B RAM

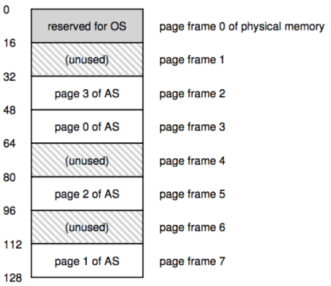
- Espacio virtual (lógico) de 64B.
- Espacio físico de 128B.
- Páginas (y *frames*) de 16B.

PAGINACIÓN
TABLA DE PÁGINAS

Páginas de 16B, espacio virtual 64B, espacio físico 128B.







Page (VPN)	Frame (PFN)
0 (0b00)	3 (0b011)
1 (0b01)	7 (0b111)
2 (0b10)	5 (0b101)
3 (0b11)	2 (0b010)

Dirección virtual: 21 $(0b010101)$
- #Página: 1 $(0b01)$
- *Offset*: 5 $(0b0101)$
- Tabla: $1 \rightarrow 7$

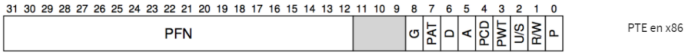
PAGINACIÓN

¿De qué tamaño es una tabla de páginas?

- Arquitectura con espacio de direcciones virtuales de 32 bit
- Páginas de 4KB. 20 bit para #página, 12 bit para *offset*
- ¿Cuántas páginas? 2^{20}. Tabla de páginas necesita 2^{20} entradas (filas, o PTE, *Page Table Entry*)

¿De qué tamaño es una entrada en la tabla de páginas?

Al menos necesita los bit para el #*frame*. Si el espacio físico es de 32 bit, necesita 20 bit.



PTE en x86

- V, **valid bit** (no está en la imagen), indica si la página ha sido asignada
- P, **present bit**, indica si la página está en memoria o en disco.
- RW, **read/write bit**, indica si se puede escribir
- U/S, user-supervisor bit, indica si se puede acceder en *user-mode*
- PWT, PCD, PAT, G, controlan el *caching*
- A, **accessed bit**, o **reference bit** indica si la página ha sido leída
- D, **dirty bit**, indica si la página ha sido modificada

Para hacer la traducción de página a frame conservamos el offsets (el número de direcciones posibles son los bits que ocupa)

En la arquitectura x86 tenia 2 a la 20 bit.
Hay 20 bits para tabla de páginas y 12 para indicar características.
Ejemplo:
1 bit de validez
1 bit que indica si la página está en modo usuario o modo kernel

A accedida
D dirty sí ha sido modificada.

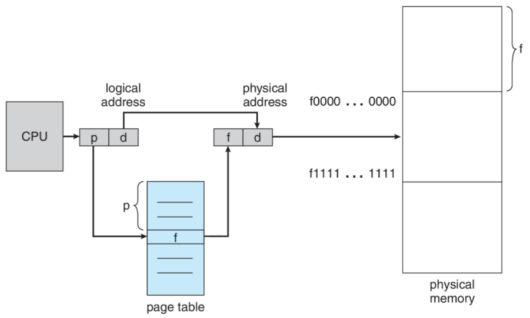
¿Cómo funciona el acceso a memoria ?

¿Cuántos accesos a memoria?

Dirección de tabla de páginas se guarda en PCB, como **PageTableBaseRegister** (PTBR)

```
VPN = (virtualAddress & VPN_MASK) >> VPN_SHIFT;
PTEAddress = PageTableBaseRegister + (VPN * sizeof(PTE));
PTE = ReadMemory(PTEAddress);

// está en memoria
if(!PTE.valid)
  raise(SEG_FAULT);
// verificación de permisos
else if(PTE.protected)
  raise(PROTECTION_FAULT);
// Acceso OK
else {
  offset = virtualAddress & OFFSET_MASK;
  physicalAddress = (PTE.PFN << PFN_SHIFT) | offset;
  register = ReadMemory(physicalAddress);
```



Cada acceso a memoria se convierte en DOS accesos
Problema: accesos en el doble de tiempo 😔

En el dibujo el offset va directo a la dirección física mientras que la página va a la tabla de páginas para que le indique su frame.
Acceso dos veces una para la memoria y otra para la tabla de páginas.

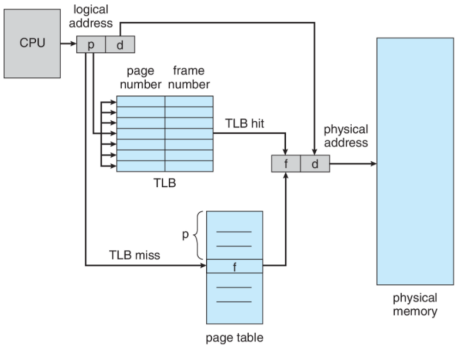
Solución: Un caché de direcciones en la memoria

Solución para acelerar acceso a memoria: *fully associative caché* de direcciones en MMU

Translation Look-aside Buffer (TLB) /

- Si la dirección está en el caché (*TLB hit*), se responde directamente
- Si la dirección no está en el caché (*TLB miss*), se lee desde la memoria y se actualiza TLB

```
VPN = (virtualAddress & VPN_MASK) >> VPN_SHIFT;
(Success, TLBEntry) = TLB_Lookup(VPN);
if(Success) {          //TLB Hit: UN acceso a memoria
  if(!TLBEntry.protected) {
    offset = virtualAddress & OFFSET_MASK;
    physicalAddress = (TLBEntry.PFN << PFN_SHIFT) | offset;
    register = ReadMemory(physicalAddress);
  }
  else raise(PROTECTION_FAULT);
}
else {                 //TLB Miss: DOS accesos a memoria
  PTEAddress = PageTableBaseRegister + (VPN * sizeof(PTE));
  PTE = ReadMemory(PTEAddress);
  if(!PTE.valid)          raise(SEG_FAULT);
  else if (PTE.protected) raise(PROTECTION_FAULT);
  else {
    TLB_Insert(VPN, PTE);
    Retry();
  }
}
```



Si una dirección está en el cache tenemos un hit podemos ir directamente a la memoria porque ya estaba cacheada esa dirección de memoria.

TLB miss no estaba el número de página que estoy buscando camino largo, voy encuentro la dirección física y actualizo el TLB y después de nuevo consulto a la TLB si tiene esa dirección.

Un acceso a memoria física es más lento que al cache

PAGINACIÓN CON TLB
EFECTIVIDAD DE TLB

TLB Hit Rate reales $\approx 99\%$

¿Por qué funciona?

Localidad Espacial

```
int sum = 0;
for(i=0; i<1000; i++)
  sum += a[i];
```

Después de acceder a una dirección x, el programa probablemente accederá a direcciones cercanas a x

- Luego de un *miss*, y guardar un PTE en TLB, vendrán muchos *hit*

Localidad Temporal

```
int sum = 0;
for(k=0; k<10; k++)
  for(i=0; i<1000; i++)
    sum += a[i];
```

Una dirección x que acaba de ser accedida, probablemente será accedida de nuevo dentro de poco tiempo

- Una PTE recién guardada, volverá a ser usada pronto

El comportamiento real depende del programa, pero la mayor parte del tiempo los programas manifiestan localidad de referencia (espacial o temporal).

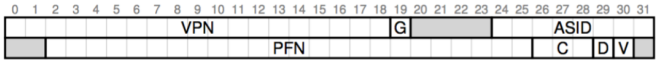
Localidad espacial la mayoría de los programas funcionan que primero van a la dirección 0 y luego 1 2 y así Acceso secuencial

Temporal es que recorre la misma dirección que recorrió antes

PAGINACIÓN CON TLB
DETALLES DE TLB

¿Dónde se guardan los datos de un TLB ante un cambio de contexto?

- TLB no se almacena en memoria del S.O. (como la Tabla de Páginas), ni en el PCB
- PTBR se actualiza en cada cambio de contexto
- PTEs de un proceso no sirven al siguiente proceso
- Se puede borrar completamente el TLB (*flush*), marcando todas los **valid bit** en 0.



TLB Entry de un MIPS R4000

VARIANTES DE PAGINACIÓN
FRAGMENTACIÓN INTERNA

***Fragmentación Externa**. Espacios libres no contiguos.*

Paginación elimina la fragmentación externa (¿por qué?), pero …

***Fragmentación Interna**. Espacios sobreasignado.*

Si las páginas son de 1MB …

- Procesos de $x\text{MB}$, con x entero reciben la cantidad exacta de espacio.
- Un proceso de 16KB recibe 1MB (1008KB desperdiciados)
- Un proceso de 1025KB recibe 2MB

Espacio se asigna a los procesos pero no se utiliza

Paginas más grandes aumentan la fragmentación interna. 😔

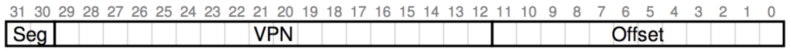
¿De qué tamaño son las páginas de su sistema? (Tip: `getconf PAGE_SIZE`)

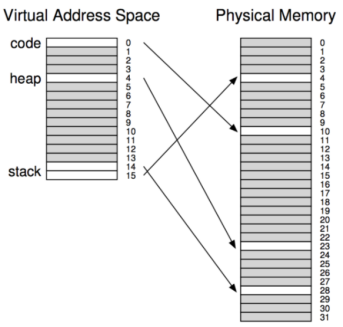
SEGMENTOS PAGINADOS

Procesos no suelen ocupar todas sus páginas

Espacio virtual con 16 páginas.

- Proceso utiliza efectivamente: 4 páginas
- 12 entradas vacías (75%) en tabla de páginas





Idea: una tabla de páginas por segmento

- Tablas de páginas más pequeña 😃
- `base` (direcc. de tabla de páginas)
- `limit` (#paginas)
- Vuelve la fragmentación externa 😕

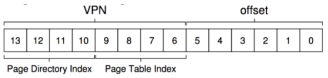
Solucion 1 tabla de páginas para cada segmento las páginas ya no están secuencialmente almacenadas.

Paginación multinivel es dividir una tabla de páginas en tablas de página mas pequeñas.

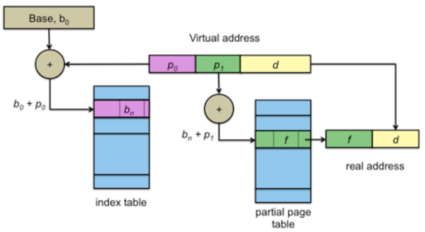
VARIANTES DE PAGINACIÓN

TABLAS DE PÁGINAS MULTINIVEL

En lugar de segmentar, mejor **paginar** la tabla de páginas



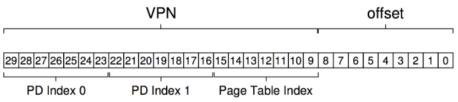
- Dirección virtual de 14-bit, páginas de 64B.
- Esquema de 1 nivel: 256 PTEs.
  - Memoria: $256 \times 4\text{B} = 1\text{KB}$
- Esquema de 2 niveles: 16 PTEs en tabla principal, 16 PTEs en 16 tablas de 2do nivel
  - Memoria: Al menos tabla principal + 1 tabla secundaria
  - Memoria: $16 \times 4\text{B} + 16 \times 4\text{B} = 128B$



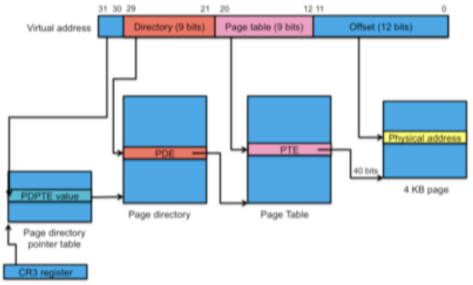
Dividimos los primeros 8 bits los primeros 4 son 1 número de página y los otros 4 otro número.

TABLAS DE PÁGINAS MULTINIVEL

Esquema se puede extender a más niveles



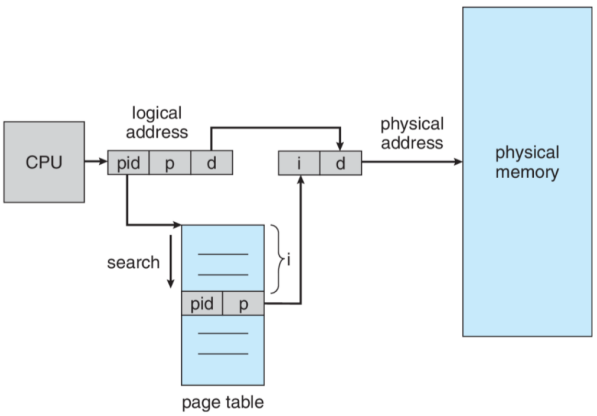
- En memoria se necesita la tabla de páginas del primer nivel (completa), y una tabla de cada nivel para poder resolver una dirección física
- Cada división puede tener distintos tamaños



TABLAS DE PAGINAS INVERTIDAS

En lugar de guardar una tabla por proceso, mantener una tabla **única para todo el sistema**

- Indice es el *frame*. Una PTE por *frame*.
- Clave de búsqueda es $\langle$ pid, #page $\rangle$
- Consume menos memoria 😃
- Pero búsqueda es lenta (secuencial) 😔
- Alternativa: usar *hashing*



PowerPC usaba este esquema

REEMPLAZO DE PÁGINAS

Supuesto a eliminar: todo el espacio de direcciones cabe en memoria física
Procesos pueden tener parte de su memoria no cargada aún (en disco), o en *swap* (en disco).

*Desafío: usar efectivamente el disco (grande, pero lento) para proveer la ilusión que **todo** el espacio virtual está en memoria*

Procesos deben seguir creyendo que pueden acceder a todo el rango de direcciones virtuales

- ¿Por qué? Es más fácil para el proceso (separa la *ilusión* de la realidad)
- Sistema Operativo se encarga de cargar las páginas *on-demand*

Si no caben más páginas en memoria física, alguna debe salir para hacer más espacio

¿Qué página eliminar?, y ¿a dónde moverla?

SWAP SPACE

¿A dónde mover las páginas de memoria?

Páginas que van a ser borradas de memoria van a *espacio de **swap***

- Representan parte del espacio de memoria de un proceso **en ejecución**
- Tabla de páginas utiliza **present bit** para saber si la página está en un *frame* o no

*Si la página no está presente, se genera un **page fault***

Page fault activa el mecanismo de recuperación de una página desde disco a memoria física

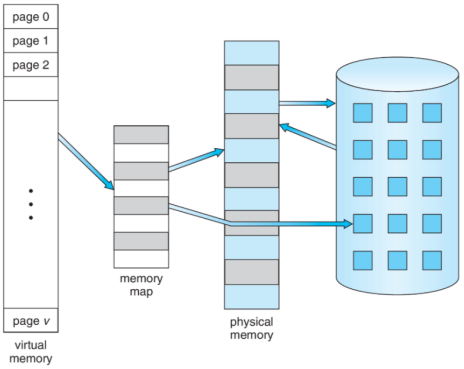
- Sistema Operativo atiende el *page fault*
- Cuando el *page fault* ha sido resuelto, el proceso puede continuar

PAGE FAULT

Ante un *page fault*, el sistema operativo va a buscar la página

Algoritmo para paginación con TLB, y *page faults*

```
VPN = (virtualAddress & VPN_MASK) >> VPN_SHIFT;
(Success, TLBEntry) = TLB_Lookup(VPN);
if(Success) {          //TLB Hit
  if(!TLBEntry.protected()) {
    offset = virtualAddress & OFFSET_MASK;
    physicalAddress = (TLBEntry.PFN << PFN_SHIFT) | offset;
    register = ReadMemory(physicalAddress);
  }
  else raise(PROTECTION_FAULT);
}
else {                 //TLB Miss
  PTEAddress = PageTableBaseRegister + (VPN * sizeof(PTE));
  PTE = ReadMemory(PTEAddress);    // lectura tabla de paginas
  if(!PTE.valid)          raise(SEG_FAULT);
  else if (PTE.protected) raise(PROTECTION_FAULT);
  else if (!PTE.present)  raise(PAGE_FAULT);
  else {                  // pagina en memoria
    TLB_Insert(VPN, PTE);
    Retry();
  }
}
```



¿Qué página reemplazar?

La ocurrencia de un *page-fault* es algo malo (significa acceder al disco)
Cualquier algoritmo de decisión debe intentar minimizar la ocurrencia de *page-faults*

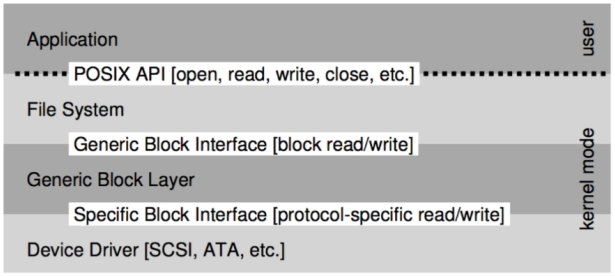
Ejemplo: direcciones virtuales de 16 bit, páginas de 4KB

- Ocurre un *page-fault* para acceder a la dirección
- 0x6300
- 0x6300 (página 6)
- Se escoge la página 9 y se hace *swap*
- Proceso continúa y solicita dirección 0x9204
- Ocurre otro *page-fault*
- No fue tan buena idea haber sacado la página 😕

Los algoritmos de reemplazo de páginas buscan minimizar el page Fault y aumentar el hit rate.

Archivos

DISPOSITIVOS DE I/O
ACCESO A DISPOSITIVOS DE I/O: DRIVERS

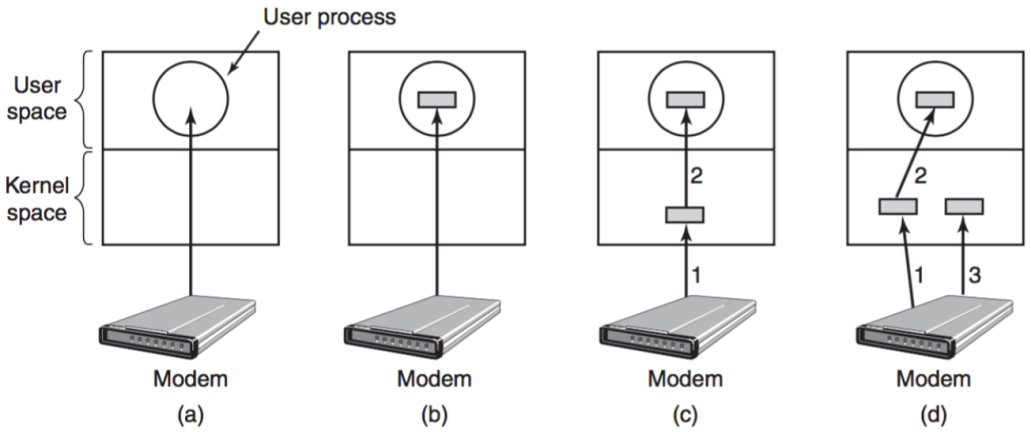


Stack de abstracciones para un sistema de archivos

- **Device Driver**. Código específico para interacción con un dispositivo
- **Block Layer**. Capa intermedia. Abstracción de bloques de disco
- **Sistema de archivos**. Capa visible al programador mediante API

70% código de Linux es para *drivers* (fuente de *bugs*)
Abstracciones persiguen un objetivo de diseño: **device independence**

DISPOSITIVOS DE I/O
TRANSFERENCIA DE DATOS: *BUFFERS*



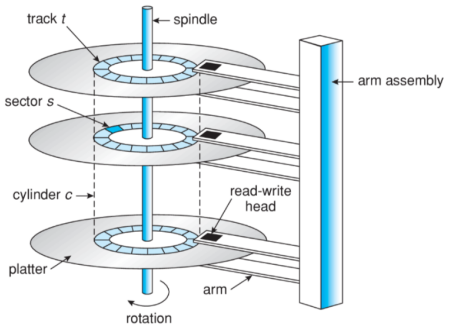
- Lectura directa: *unbuffered*. Lento.
- *Buffer* en *user space*. Página puede ser *swapped out*.
- *Buffer* en *kernel space*. Se transfiere al llenarse.
- *Double-buffer*. Mientras uno se transfiere, se escribe en el otro.

MEMORIA SECUNDARIA

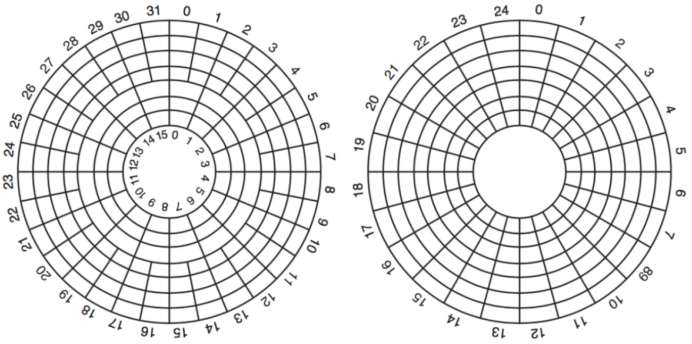
Almacenamiento de alta capacidad
Más lenta que almacenamiento principal
Memoria no-volátil

DISPOSITIVO: DISCOS MAGNÉTICOS

- **Brazos** (*arms*) se mueven juntos.
- **Brazos** poseen cabezas lectoras (**heads**)
- **Platos** (*platter*) divididos en **tracks** circulares
- **Tracks** divididos en **sectores**: bloques de 512 byte
- Conjunto de **tracks** entre varios platos forman un **cilindro**
- Velocidades de rotación: 5400, 7200, 10000, 15000 RPM (rotaciones por minuto)

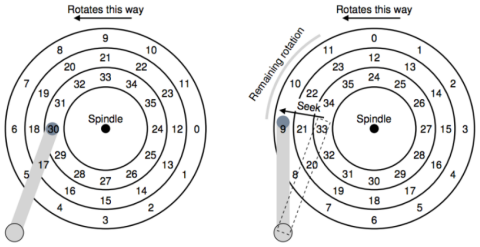


SISTEMAS DE DISCO
GEOMETRÍA DE DISCOS MAGNÉTICOS



Geometría física Geometría virtual

Cada sector está representado por ⟨cilindro, head, sector ⟩: **CHS addressing**
Sistemas modernos asignan número secuencial a los sectores: **LBA, Logical Based Addressing**
Driver de disco traduce la dirección solicitada a una posición física



Tiempo de posicionamiento depende de:

- **Latencia rotacional**. Búsqueda en un *track*. Tiempo para que el sector quede debajo del brazo.
- **Seek time**. Cambio de *track*. Tiempo para mover el brazo hasta el cilindro buscado.

Al **tiempo de posicionamiento** se debe sumar el **tiempo de transferencia**

- Leer sector consecutivos es más rápido
- *Tip*: es bueno leer sectores contiguos de disco

DISCOS DE ESTADO SÓLIDO

Solid-State Drive (SSD)

- Sin *seek time* ni *latencia rotacional*
- Más rápidos
- Más costoso por MB
- Menor consumo eléctrico
- Celdas poseen cantidad limitada de reescrituras

FORMATEO DE DISCO

Formateo de bajo nivel o formateo físico

- Crea estructura de sectores: header + data + trailer (incluyendo ECC)
- Ejecutado en fábrica
- *Low-level Formatting* equivale a regresar a valores de fábrica

Particionamiento

- Agrupación de sectores que son tratados como discos separados
- Tabla de particiones accesible en **Master Boot Record** (MBR).
- MBR: 32-bit para direccionar bloques de 512 byte.
- Sistemas modernos usan GPT: **GUID Partition Table**

Formateo lógico

- Estructuras para *sistema de archivos* en una partición.

SCHEDULING DE ACCESOS

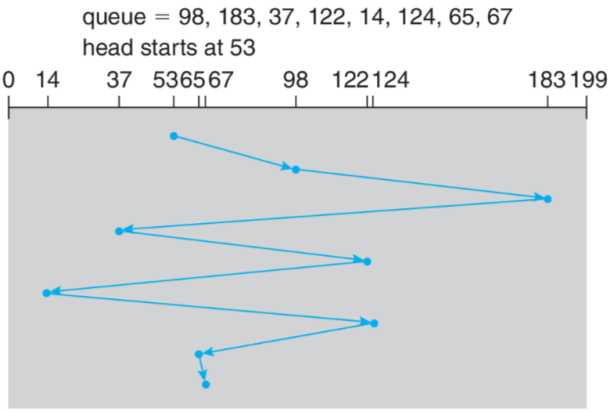
Sistema Operativo solicita datos del disco especificando:

- Lectura o escritura
- Dirección de disco (CHS ó LBA)
- Dirección de memoria física (*buffer*)
- Número de sectores a transferir

Sistema Operativo puede encolar y ordenar solicitudes. El orden de atención determina el seek time de transferencia.

SCHEDULING FCFS

Justo, pero no necesariamente el de mejor servicio

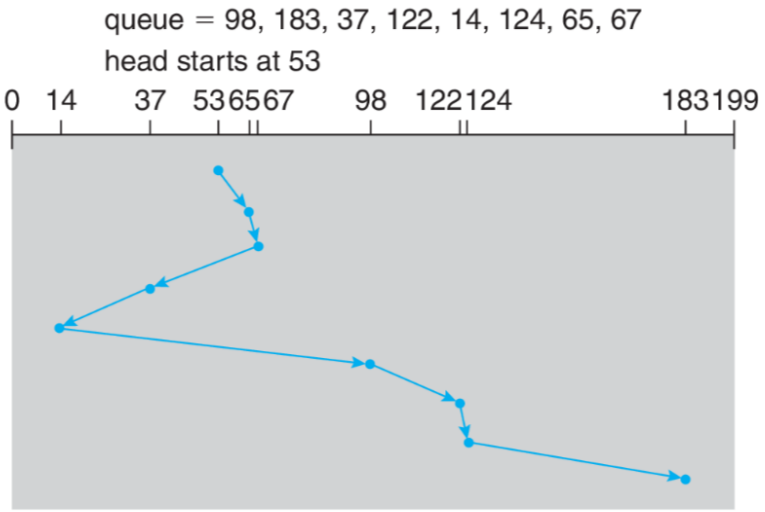


Medida de latencia: desplazamiento que debe hacer el brazo $= 640$

SCHEDULING SSTF

Shortest Seek Time First

Busca el que esté más cerca de la posición actual



Desplazamiento $= 208$

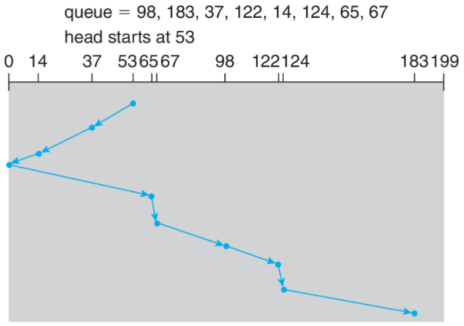
No es óptimo ¿por qué?

Posible inanición de sectores alejados

SCHEDULING SCAN

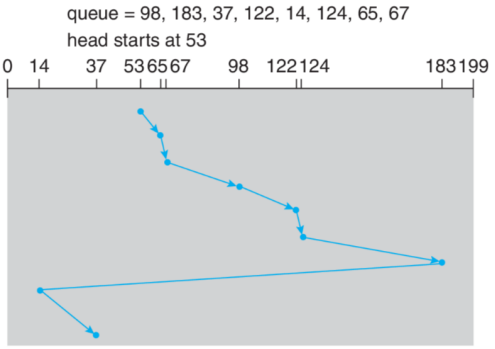
El elevador

Brazo se mueve de un extremo a otro y sirve todo lo que encuentre en el camino ...



SCHEDULING LOOK Y C-LOOK

Similares a SCAN y C-SCAN, pero no se mueven hasta los extremos si no es necesario



SISTEMAS DE ARCHIVOS

Colección de información en almacenamiento secundario

- Unidad mínima de almacenamiento para el usuario
- Agrupación lógica de *bytes*
- Abstracción

¿Qué contiene? -> cualquier cosa

- Significado está determinado por la forma en que se utiliza su contenido
- Texto, imagen, código fuente, código binario (ejecutable), .

Caracterizando un archivo

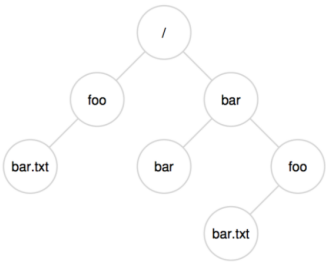
- Nombre (low-level). También conocido como *inode number*
- Nombre (simbólico). Información *human-readable*
- Tipo. Determinado por contenido. Extensión es solo convención.
- Ubicación. Puntero a dirección de disco.
- Tamaño. Tamaño actual en bytes, ó *blocks*. Puede incluir un tamaño máximo.
- Protección. Datos de control de acceso. Lecturas, escritura, ejecución.
- Fecha. Creación, acceso, modificación.
- Usuario. Propietario.
- Atributos extendidos (metadata): Codificación, checksum, programas asociados

DIRECTORIOS

Archivos se organizan en una estructura de **directorios** (otra abstracción)

- Nombre (low-level name): *inode number*
- Contiene lista de pares: (*readable name*, *low-level name*)
- Puede contener otros directorios: **jerarquía** o **árbol de directorios**
  - Jerarquía empieza en una raíz: **root**, o `/`
  - Usa un separador: `/`
  - Unidades son **subdirectorios**
- Nombres con subdirectorios: **pathname** o **ruta**
  - Ruta absoluta: **Absolute path**
    - `/foo`, `/bar/bar`, `/bar/foo/bar.txt`
  - Ruta relativa: **Relative path**
    - `foo`, `foo/bar.txt`, `foo/bar.txt`

    Ruta absoluta no puede repetirse

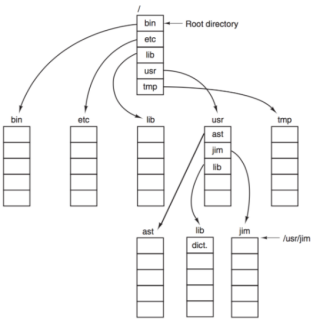


DIRECTORIOS

Archivos se organizan en una estructura de **directorios** (otra abstracción)

- Nombre (low-level name): *inode number*
- Contiene lista de pares: (*readable name*, *low-level name*)
- Puede contener otros directorios: **jerarquía** o **árbol de directorios**
  - Jerarquía empieza en una raíz: **root**, o `/`
  - Usa un separador: `/`
  - Unidades son **subdirectorios**
- Nombres con subdirectorios: **pathname** o **ruta**
  - Ruta absoluta: **Absolute path**
    - `/foo`, `/bar/bar`, `/bar/foo/bar.txt`
  - Ruta relativa: **Relative path**
    - `foo`, `foo/bar.txt`, `foo/bar.txt`

    Ruta absoluta no puede repetirse



API DE SISTEMA DE ARCHIVOS

Archivos

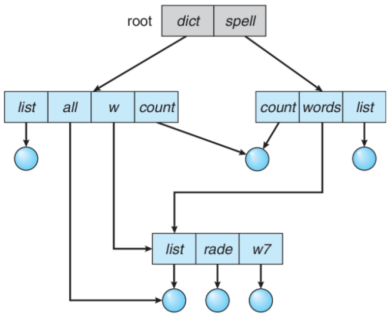
- Creación: `open`
- Lectura/Escritura: `read, write`
- Lectura/Escritura (no secuencial): `lseek`
- Escritura inmediata: `fsync`
- Renaming: `mv`
- Estado: `stat`
- Eliminación: `rm`

Directorios

- Creación: `mkdir`
- Lectura: `opendir`, `readdir`, `closedir`
- Eliminación: `rmdir`

Links

Una manera sencilla de compartir archivos: que el mismo archivo pertenezca a dos directorios



- Implementación usando **link**s
  - **Symbolic links**: *link* a nombre de archivo (`ln -s`)
  - **Hard link**: *link* a archivo (`ln`)

¿Qué hacer al borrar un archivo?

- **Symbolic links.** Si se borra el archivo original, el *link* sigue existiendo, y debe ser eliminado manualmente



- **Hard links.** *Links* aumentan un contador de referencias el archivo. El archivo solo es eliminado cuando no quedan más referencias a él.

Montando sistemas de archivos

Sistema puede manejar múltiples **sistema de archivos** en **discos** (dispositivos de bloques)
Para acceder a sistemas de archivos se necesita un *punto de acceso*, o *mountpoint*. Sistema operativo *integra* el sistema de archivos a la jerarquía actual usando `mount`

```
mount /dev/sda1 /home/users
```



IMPLEMENTACIÓN DE SISTEMAS DE ARCHIVOS

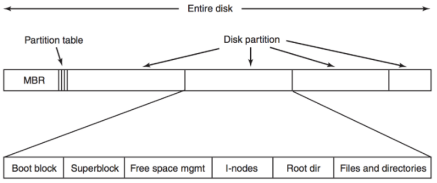
El sistema de archivos permite obtener una dirección de disco (un *bloque*) a partir de un conjunto de datos simbólicos: nombre de archivo, directorio, rutas, links, ...

- ISO9660 (CD-ROMs)
- UFS: Unix File System, basado Berkeley Fast File System (FFS)
- FAT, FAT32, NTFS: Windows

- ext3, ext4: Linux
- HFS+: MacOS X
- VFS, FUSE

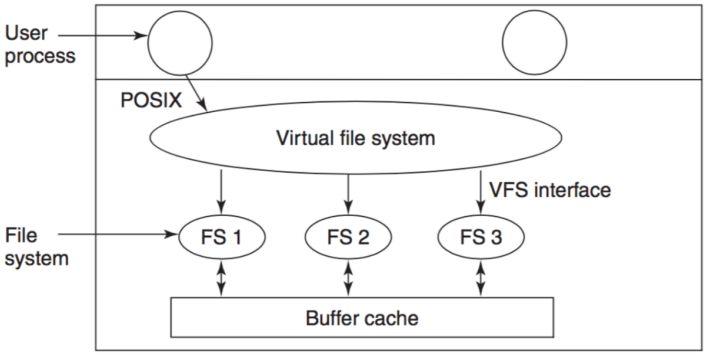
Master Boot Record (MBR). Sector 0. Indica *Partition Table*

- **Boot Control Block**. Uno por cada partición. Permite cargar el sistema operativo.
  - Unix: Boot block
  - Windows: Partition Boot Sector
- **Volume Control Block**. Datos de formato del sistema de archivos.
  - Tipo de sistema de archivos, número de bloques, …
  - Unix: Superblock
  - Windows: Master File Table
- Estructura de directorio. Archivos.



VIRTUAL FILE SYSTEM (VFS)

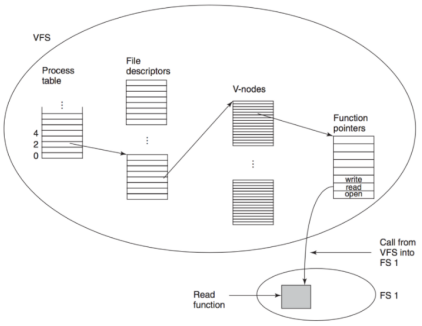
Interfaz común para distintos sistemas de archivos



Cualquier sistema de archivos que implementa la interfaz VFS puede ser incorporado al sistema

VIRTUAL FILE SYSTEM (VFS)

- **Tabla de archivo abiertos** mantiene *file descriptors*. Estructuras en memoria con información de archivo
- **v-node** representa el punto de acceso al archivo. Cada **v-node** apunta a funciones del sistema de archivos que ejecutan las llamadas concretas

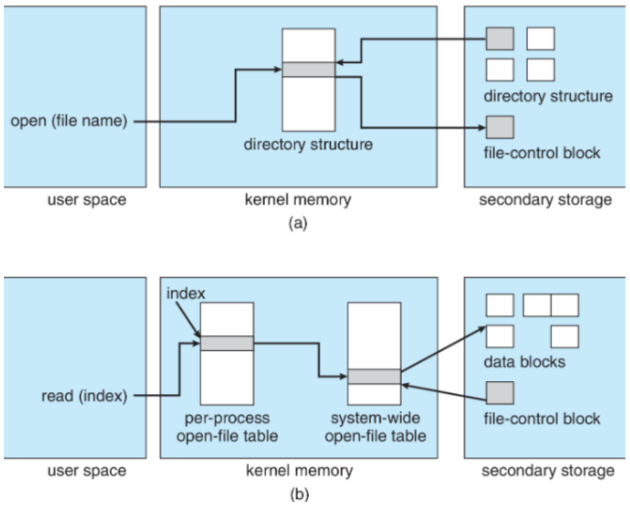


Facilita la incorporación de distintos sistemas de archivos

Estructuras en memoria principal

- **Mount table**. Tabla de sistema de archivos montados.
- Directorios recientes

- Tabla de archivos abiertos en el sistema
- Tabla de archivos abiertos por proceso
- Buffers de bloques de disco



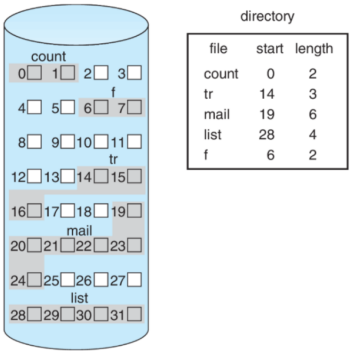
ASIGNACIÓN DE BLOQUES

Cada archivo requiere utilizar bloques de disco

- Bloques deben ser asignados de manera eficiente
- Mala asignación provocará demoras en tiempos de acceso
- Estrategias de asignación de bloques

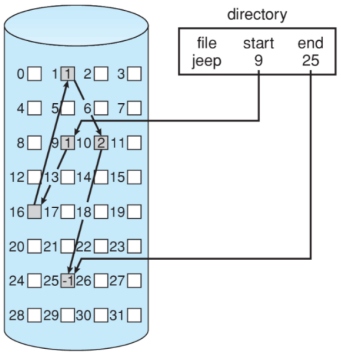
ASIGNACIÓN CONTIGUA

- Archivo de n bloques: $b, b+1, \ldots b+n-1$
- Fácil de acceder a siguiente bloque
- -Problema: fragmentación externa
- Require operaciones de **compactación** (defragmentación)



ASIGNACIÓN ENLAZADA (*LINKED*)

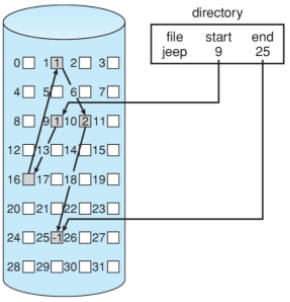
- Almacena punteros a bloque siguiente
- Ej: bloque de 512 bytes, solo puede guardar 508 bytes. Se pierde espacion.
- No requiere tamaño inicial de archivo. Puede crecer.
- -Acceso solo secuencial.
- Alternativa: almacenar *clusters* de bloques en lugar de bloques individuales (frag. interna)
- -Vulnerable a falla de un bloque



Implementación Sistemas de archivos

Asignación Enlazada

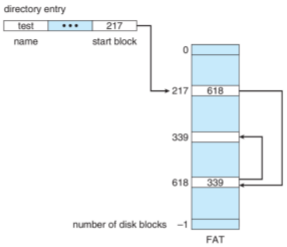
- Almacena punteros a bloque siguiente
- Ej: bloque de 512 bytes, solo puede guardar 508 bytes. Se pierde espacion.
- No requiere tamaño inicial de archivo. Puede crecer.
- -Acceso solo secuencial.
- Alternativa: almacenar clusters de bloques en lugar de bloques individuales (frag. interna)
- -Vulnerable a falla de un bloque



Implementación Sistemas de archivos

Asignación Enlazada Linked FAT

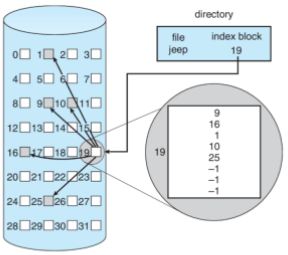
- FAT: File Allocation Table
- Tabla al inicio del disco con una entrada por archivo
- Último cluster almacena EOF (end-of-file)
- +Mejor acceso aleatorio
- -Limitado por tamaño de tabla (para FAT32, 4GB por archivo)



Implementación Sistemas de archivos

Asignación Indexada

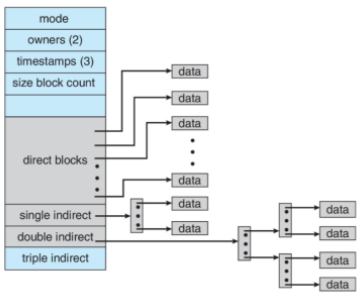
- Archivos contienen index block
- Index Block contiene bloques del archivo
- Último cluster almacena EOF (end-of-file)
- +Acceso aleatorio sin frag. externa
- Se pierde espacio por tamaño de index bloc



Implementación Sistemas de archivos

Asignación Indexada:Tamaño del index block

- **Esquema enlazado:** última entrada de index block apunta a otro index block
- **Índice multinivel:** similar a tablas de página multinivel
  - Ej: blocks de 4KB, permiten 1024 punteros (de 32-bit)
  - Dos niveles permiten direccionar 1048576 bloques $\rightarrow$ 4 GB
- **Esquema combinado:** primeros p punteros son bloques directos, los siguientes apuntan a bloque de índice simple, los siguientes a bloque de índice doble, etc



Implementación Sistemas de archivos

Espacio Libre

¿Cómo encontrar bloques libres?

Bit vector ó **bitmap** de bloques libres

- +Sencillo, eficiente, e implementable en hardware
- -No escala tan bien
  - Disco de 1.3GB, con bloques de 512 byte→ 332 KB
  - Disco de 1TB, con bloques de 4KB→ 256 MB

Implementación Sistemas de archivos

Espacio Libre*ListaEnlazada*

Lista Enlazada

- Punteros a bloques libres
- Bloques nuevos se agregan a la lista
- -Poco eficiente
- Mejora: agrupar espacios contiguos
- Mantener punteros a espacios libres + número de espacios contiguos

