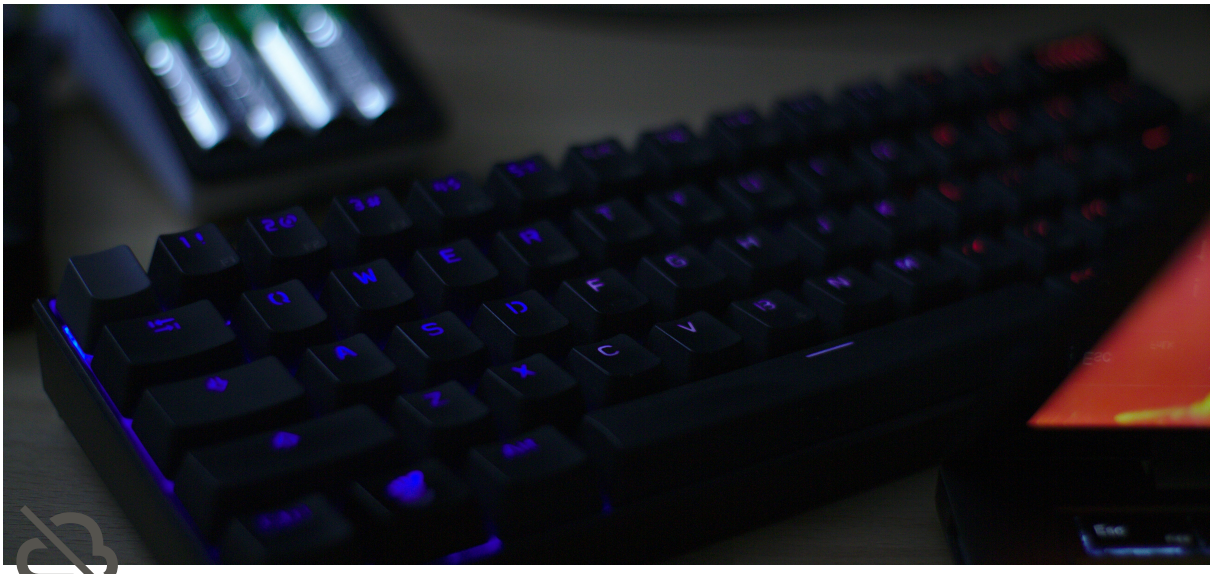


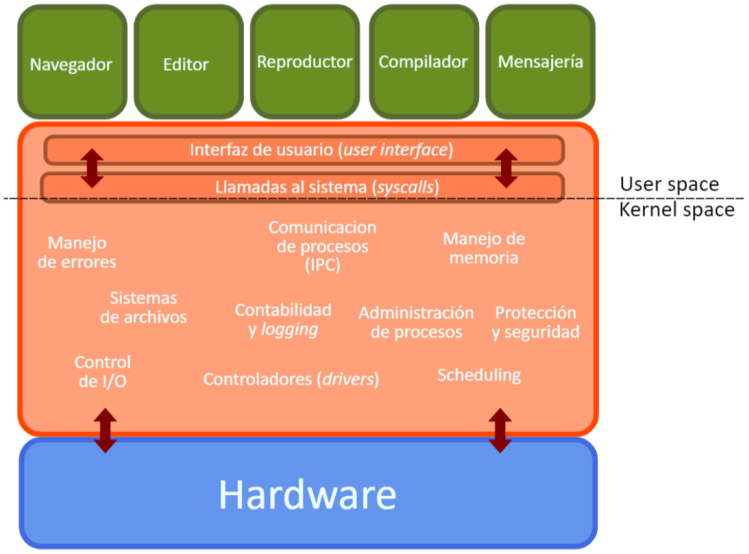
Apuntes - SSOO IIC2333

CLASE 1

12 / 8 / 2024

Sistema Operativo:

- tiene como trabajo proporcionan los programas del usuario un modelo de uso de recursos mas simple y ademas de administrar los recursos
  - Es mas facil que nos proporcione, a que tengamos que ir a buscarlo
- Nace de la necesidad de Abstraerse del Hardware
- El Programa 1 que empieza en el computador → Es un Daemon → Va corriendo todo el tiempo, todo falla si el falla (SystemD = System Daemon) → probee funcionalidades basicas
  - Prender Cpu, verificar que este bien, etc etc



- **Schedulling**: Eligir quien va a estar en la **RAM**
- **Manejo de memoria**: Donde se guardan las descargas
- Control I/O : Maneja teclados, mouse, etc

KERNELL

Programa que siempre se ejecuta, funciones mínimas necesarias de un computador

El típico pantallazo azul, viene de que el kernell detecto un error en memoria o disco

ROL

1. Para el Usuario: Les permite usar recursos de hardware (Memoria/Cpu)
2. Sistema: ADMIN: Selecciona y Provee **Recursos** (Ningun juego o programa nos pregunta a nosotros, por debajo le pide recursos al administrador del sistema)

SO

1. Nos abstrae del hardware

2. Nos permite compartir Recursos
3. Puede Contener ≠ Arquitecturas

DONDE VIVE?

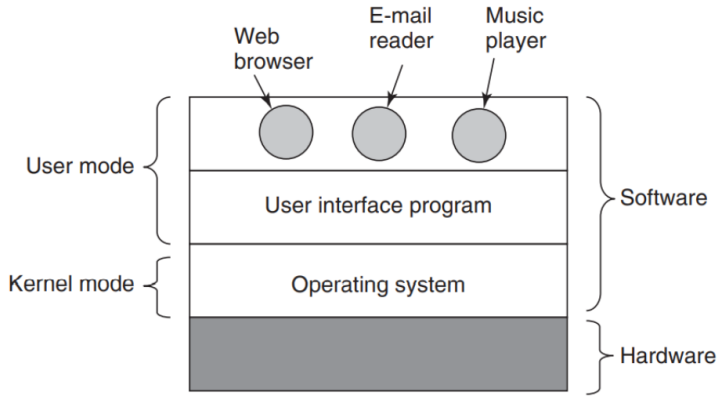


Figure 1-1. Where the operating system fits in.

+Arriba : - Privilegios, menos poder → Programas muy arriba tendran que pedir permiso a abajo → **OVERHEAD**

+Abajo : + Poder → Son los mas **Rápidos**, no tienen que pedir permiso (Permiso Admin)

Ej: Programa de copiar un cd, poco importante → Poco privilegio:

No tiene permisos para hacer todo (ej: Pedir permiso para lectura de I/O del disco)

COMUNICACION S.O y HARDWARE

El Programa se comunica primero con el S.O antes que con el hardware, no se puede sobreescribir memoria directamente

El el S.O el que maneja los permisos directos al hardware, no se puede si no atravez de el

Para esto se usan las **SYSCALL**

SYSCALL

- Read
- Write
- Copiar

Tabla definida de opciones de operaciones para el S.O → guias de llamadas al systema

Son llamadas que se hacen al sistema

****— IMPORTANTE PARA LA TAREA 1 —*****

Se pueden llamar desde linea comandos, interfaz, etc

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file

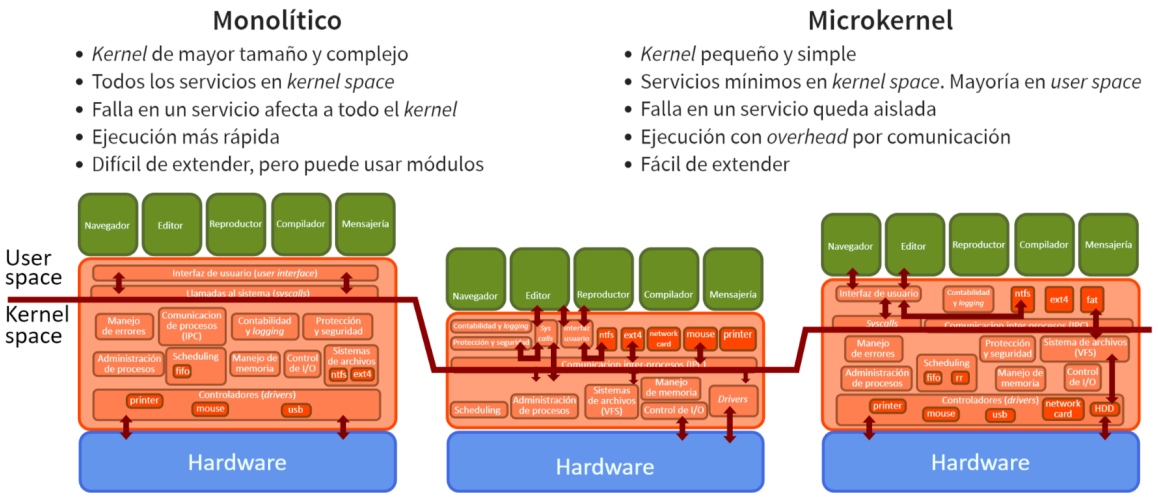
TIPOS DE ARQUITECTURAS

1. **MONOLITICO:** Falta todo → Linux (Mas rápido)
   - **Kernell Gigante** → Mas espacio para **mas poder** (La mayoria de programas en el Kernell)
   - Si falla una cosa, se cae todo el programa → **Mas peligroso**

2. **MICROKERNELL:** (Liviano, Overhead
   - Menos funcionalidades, menos privilegios → **Mas Lento**

- Programas no considerados minimos tienen **OVERHEAD → Son mas Lentos**
- Pero son mas **SEGUROS,** no se va a caer por un error

RESUMEN



CLASE 2

14 / 08 / 2024

SYSCALLS

Interfaz de **Llamadas** al sistema

→ Cada **S.O** tiene su **LIBRERIA** para invocar Syscalls

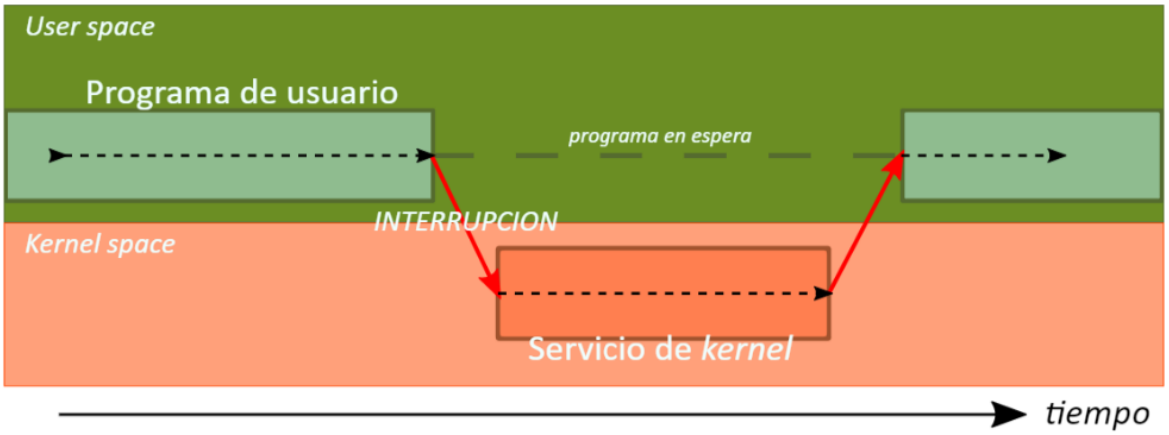
Ejemplo

Al copiar un archivo → Pasan cosas por detras

1. Verificar si existe el archivo
2. Si tengo permisos para leerlo
3. Si puede escribir en memoria

Estos son **INTERRUPCIONES,** el Kernell es quien le responde al usuario si cumple con los permisos anteriores,

INTERRUPCIONES



- El programa le pregunta al S.O si tenemos permisos → Esto quita tiempo (**OVERHEAD**)
- El sistema operativo no hace nada por si solo, porque brinda servicios
- Si yo no hago nada, no genero interrupciones, esperará a que yo le pida algo
- Solo correra lo basico → bateria, etc
- Si nosotros no hacemos nada, solo dara servicios basicos

ARQUITECTURAS

→ Monoliticos:

- Todo dentro de un programa
- Todos los programas adentro se ejecutan en **MODO KERNELL (+Rapido)**
- Poca tolerancia a fallos (un programa se cae → se bloquea todo)

→ Microkernelll:

- Solo servicios basicos en el kernell
- Mejor tolerancia a fallos (No bloqueante)
- Desventaja: **(+LENTO)**

Conclusion: Hoy en día **CASI TODOS** los sistemas son **HIBRIDOS**

MODO DE EJECUCION

1. MODO USUARIO (Overhead → Mayor tiempo de espera en pedir permisos)
2. MODO KERNELL (Todos los permisos → Sin tiempo de espera)

- El modo usuario → No puede leer ni escribir directamente, tiene que pedir permisos
- Si el programa de usuario falla → No pasa nada, porque como no tiene acceso al hardware, podemos cerrarlo y abrirlo sin problemas

ANILLOS DE PRIVILEGIOS

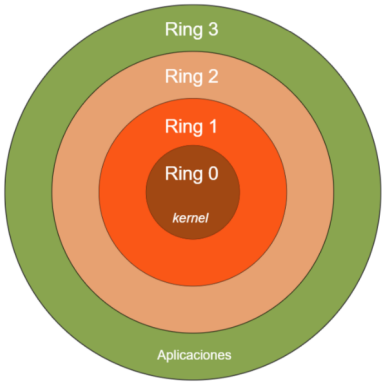


Figure: Anillo de privilegios

PROCESOS → Tarea 0

(Codigo) + (Recursos)

python.py → CPU

Se van poniendo en una cola

Recursos: CPU, Memoria, Archivos

DISTINTAS CPU´S

- **1 CPU: 1** Proceso en el mismo instante
- **2 CPU: 2** Procesos en el mismo instante

PREGUNTA DE PRUEBA: Lo que pasa es que estan todos en una cola, nunca se ejecutan mas de una a la vez, esto es **MULTITASKING:**

MULTITASKING

Los procesos entran y salen de la CPU, haciendo creer al usuario que **ejecutan varios procesos a la vez**

BOOTEO DEL COMPUTADOR

Al iniciar un computador (f11)

- **BIOS:** Basic Input/Output system
- **UEFI:** Unified Extensible Firmware Interface

UEFI es la nueva → Bios es la vieja

- Aqui podemos elegir que particion iniciar → Podemos elegir el sistema operativo que queremos → Linux, Windows, etc...

Este codio tiene dos misions fundamentales

- Inicializar/Identificar el hardware
- Ubicar y arrancar mediante un bootloader el kernell en **modo Privilegiado (kernell)**

→ El **Kernell** es el unico software que funciona en modo **Privilegiado**

SYSCALL

Es una librería, serie de instrucciones para llamar al sistema

Cada sistema tiene la suya

→ LISTA

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory- and file-system management

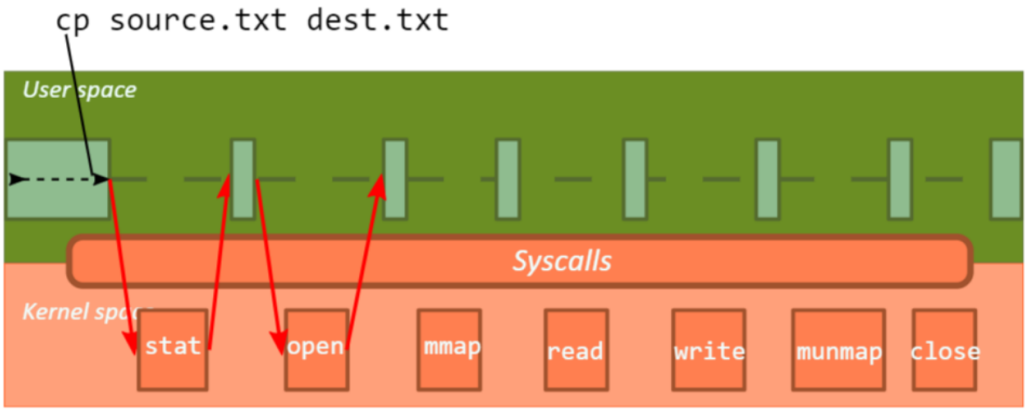
Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

- Necesitamos de las syscalls para por ej: *copiar un archivo*

EJEMPLO DE SYSCALL (Copiar)



Para escribir una syscall, necesitamos permisos de kernell, y esto es peligroso, podria romper nuestro sistema

CLASE 3 (PROCESOS)

19/08/2024

Proceso

Un proceso es un **Programa de ejecucion**

- Proceso = Codigo(programa) + Recursos

¿Que recursos?

Multiprogramacion pemite mantener **multiples procesos** en la memoria (mediante schedulers)

"En el mismo instante, un core solo atiende un proceso"

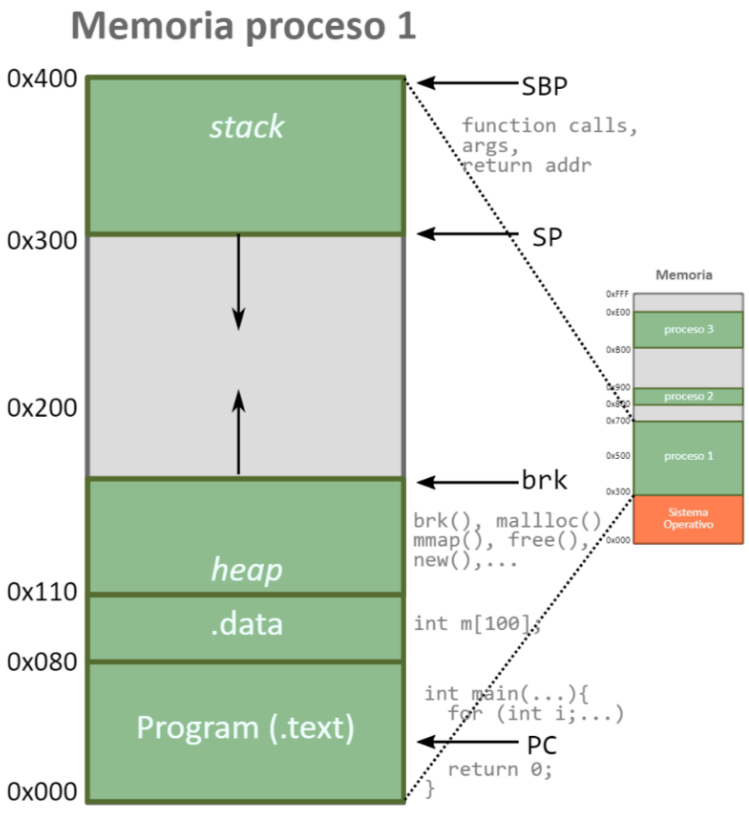
Otros

- CODIGO
- DATOS
- HEAP → Memoria dinamica (durante la ejecucion)
- STACK → Cada item del stack representa un llamado a funcion (call frame) y contiene:
  - Parametros
  - Variables locales
  - Lugar de retorno (PC (donde estaba antes))

- Stack base pointer (SBP) punto de inicio del stack
- Stack pointer limite inferior del stacj, de donde se sacan las direcciones

Pregunta Prueba: El proceso esta compuesto por codigo heap y stack → Es verdadero aunque falte una ..... Pero si inventan algo → es falso

ESQUEMA



PCB → Guarda informacion del proceso

Process Control Block

- Almacena toda la **informacion** de un **Proceso**
  - Donde esta el codigo
  - Program Counter
  - Estados y Anchos

Puede cambiar

Es como la informacion tuya en el registro civil

Si un programa esta finished, luego de un tiempo se va limpiando por temas de espacio

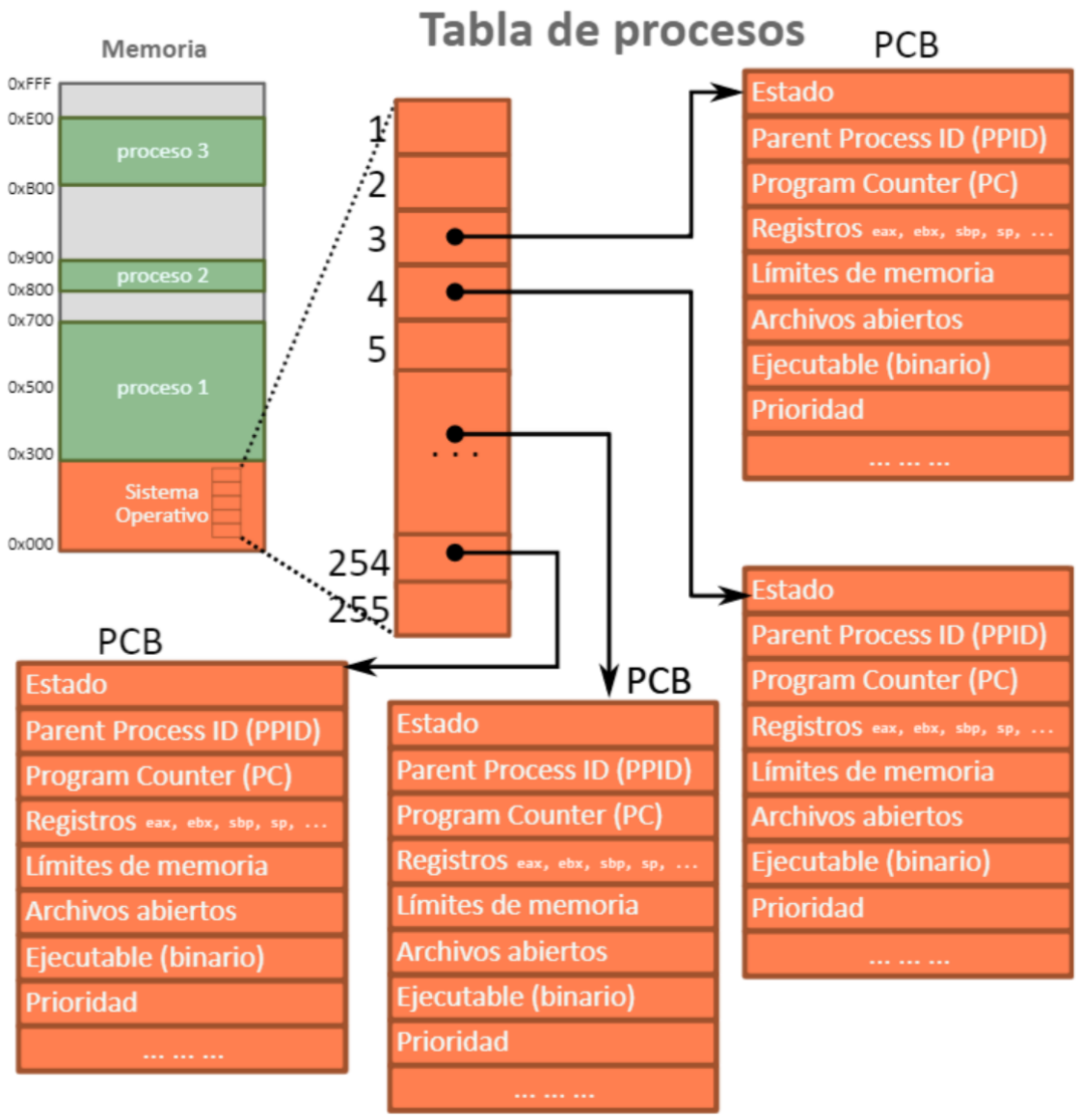
Guarda Id del hijo y de si misma

Cuando se **crea un proceso se crea una PCB**

Hay una PCB por cada hijo (cada proceso que existe)

Todo esta en la misma tabla → El SO tiene solo 1 tabla para todos los procesos

ESQUEMA



Pregunta → El pcb controla los procesos → verdadero

Es una tabla que maneja los stacks → Falso!

Si saco un proceso de la CPU

Tengo que guardar su estado donde estaba

Pero quiero reanudarlo y quiero cargar en la Cpu los datos del procesos

Context Switch

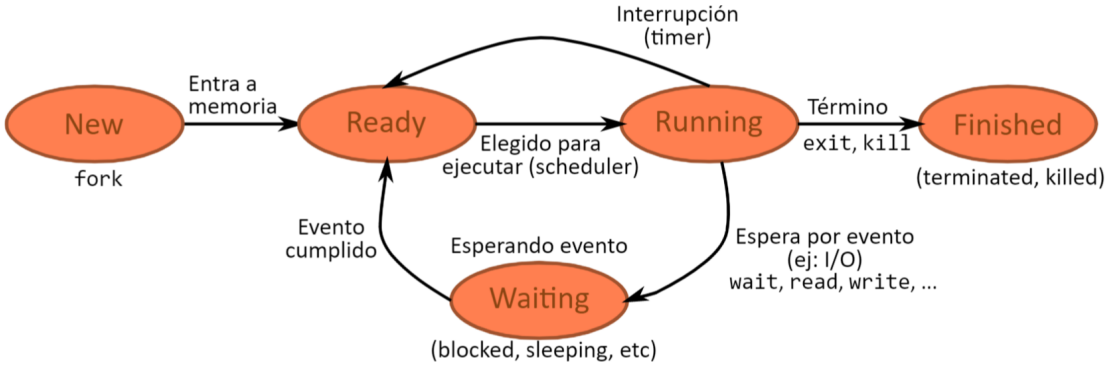
- Cambio de **Proceso Corriendo**
- Siempre tiene un poco de **OVERHEAD** → Tenemos que posicionar todo para correr de nuevo
- Se usa mucho en computacion → Cambio de Contexto
- Cargar datos y variables que habian antes
- Se realiza con un tiempo extra para realizarlo

Estados Proceso

1. **RUNNING:** Corriendo en CPU → **CPU Y CORE ES LO MISMO**
2. **READY:** listo para entrar en la CPU → **LISTO PARA EJECUTAR, tiene TODO lo que necesita**
3. **WAITING:** Esperando I/O o **bloqueado(esperando) → LE FALTA ALGO PARA CORRER**
    a. ej: Proceso de copiar archivos, en que carpeta? → necesito que el usuario ingrese un input para saber donde buscar
    b. Leer archivo en memoria → Cuando reciba lo que falta
4. **FINISHED: Terminado**

Ready → Running → Finished ⟶ Waiting..... → Ready → .....

Un proceso en ejecución puede cambiar de estado



Quien eleije que entra en la CPU

Es una **SCHEDULING**

- Ready es una cola de procesos
- Existen de varios tipos
- Algunos ekigen
- Otro sacan de la cola Ready

Schedule **FIFO: Primero que entra primero en ejecutarse**

Schedule FCFS: Primero que entra → Primero que sale

Etc...

Se asignan tiempo → Ej: Se ejecutan en 5 segundos, si no sale y espera volver a ejecutarse, se pone en la cola READY

Tipos de Procesos

1. **CPU Burst →** Alto encomputo → Necesita muchos recursos
    a. Es altamente computacional
    b. Ej: Calculos matematicos → caja obscura
    c. Codigo que corre solo y no requiere interaccion Humana

2. **CPU I/O** → Recibe Input/ Output
    a. Ej: Recien entrenar una Machine Learning
    b. Ej: Whatsapp → si no hay input , no hay mensaje
    c. EJ: Un Juego → Alto de input output → Constantemente esta esperando recibir y entregar
    d. Necesita de Input Output

Pregunta prueba: Verdadero falso

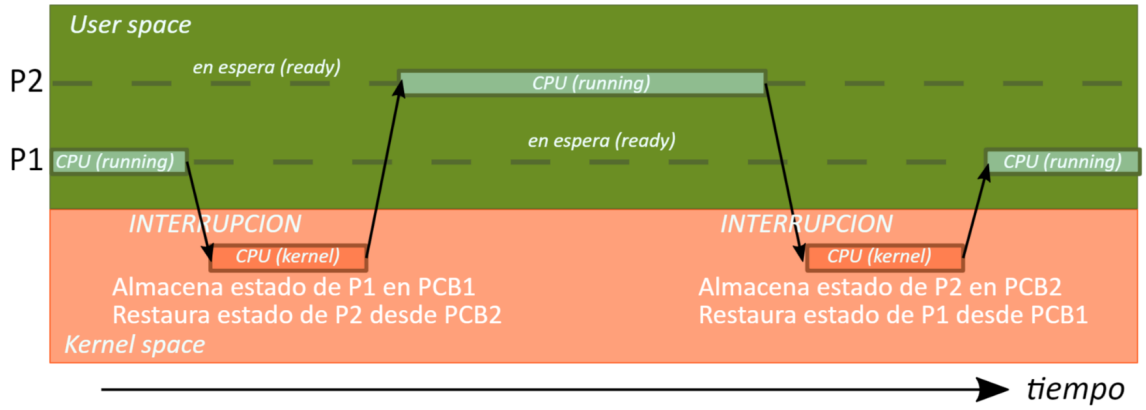
Un proceso de input/output es alto en señales de entrada y salida

Este proceso es alto en input/output

CONTEXT SWITCH EJEMPLO

Multitasking

- Sistema Operativo actúa luego de una interrupción (syscall, timer, evento, . . .).
- Sistema Operativo almacena estado de registros de P1 en PCB1. P1 queda "en pausa".
- Sistema Operativo restaura estado de registros desde PCB2.
- Proceso P2 continúa su ejecución.



En la pcb se esta guardando todo el estado de un Juego por ejemplo

El SO por detras guarda la info que minimizaste y probablemente lo cierra

Cuando tenga que abirlo denuevo, probablemente va a recargar toda la info

Signals

Señales

Sigterm

Sigkill

SigStopy

SigInt

CREACION DE PROCESOS

Exiten Daemons → procesos que correr

Creación de procesos: ¿el huevo o la gallina?

- ¿Quién crea un proceso?... otro proceso
- ¿Quién crea ese otro proceso? otro proceso
- ¿Quién crea ese otro proceso? otro proceso. . .

  Durante la inicialización del kernel se crea un **proceso "raíz"**, con **PID=1**

Proceso Papa de todos los otros → Echa a correr todos los demas procesos

ej: Prender el monitor

ej: Prender algo

ej: Tenia abierto google antes de cerrar

En los S.O

- Linux: **systemd, init, upstart**
- macOS: **launch-d →**
- Windows: **InitialSystemProcess, System**

Todos estos son DAEMONS con el ID: 1

SYSCALLS

- **FORK: Clona el procesos**
- **WAIT**: Espera que el proceso termine

- **EXECV**: Reemplaza el proceso (**Ejecucion**)
- **KILL: Termina** el Proceso

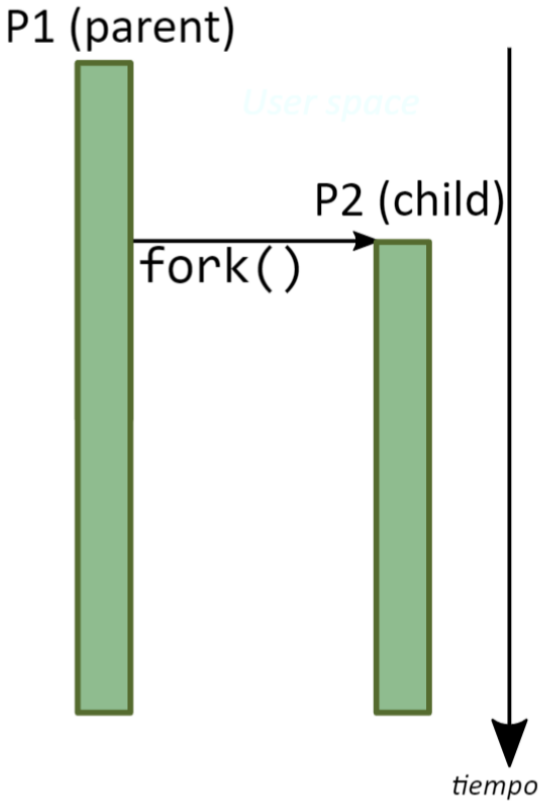
NAS: Tecnica para usar mas espacio → Mas discos que están conectados directos a la red

---

→ Investigar sobre servidores y estas cosas

COMO CREARLOS

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



CLASE 4

21/08/2024

IMPORTANTE

FORK: Crea un **proceso hijo** (copia del padre)

PREGUNTA: Empieza a ejecutar desde la linea en que se hace fork

WAIT: Espera que el proceso hijo termine

Cuando un proceso termina, le retorna al padre

Si no se le retorna al padre → Se genera un **Proceso Zombie**

Si retorna **0** → Termina **bien**

Si retorna otro → **Error**

SEÑALES → Siempre se pregunta

1. **Sigterm:** le avisa que debe **terminar**
    a. Aca le preguntar a terminar
2. **SIGKILL:** Mata/Termina el proceso
    a. El proceso lo terminan sin preguntar
3. **SIGINT:** Interrumpe → Se interrumpe el proceso (Le puedo mandar cosas) → Es como **Esperame un poco**
4. **SIGSTP: S**top → Detenerlo implica guardar lo que tiene para usarlo después
5. **SIGCOM:** Continua

SCHEDULERS

Regals para elegir que proceso **entra en la CPU** (Nucleo)

Depende del tipo de schedule → Es como se elige quien llega a la cola de Ready

→ Quien es el mejor dato para entrar en la CPU

Existen 3 que siempre funcionan dentro de un computador → corto, mediano y largo plazo

SHORT-Term

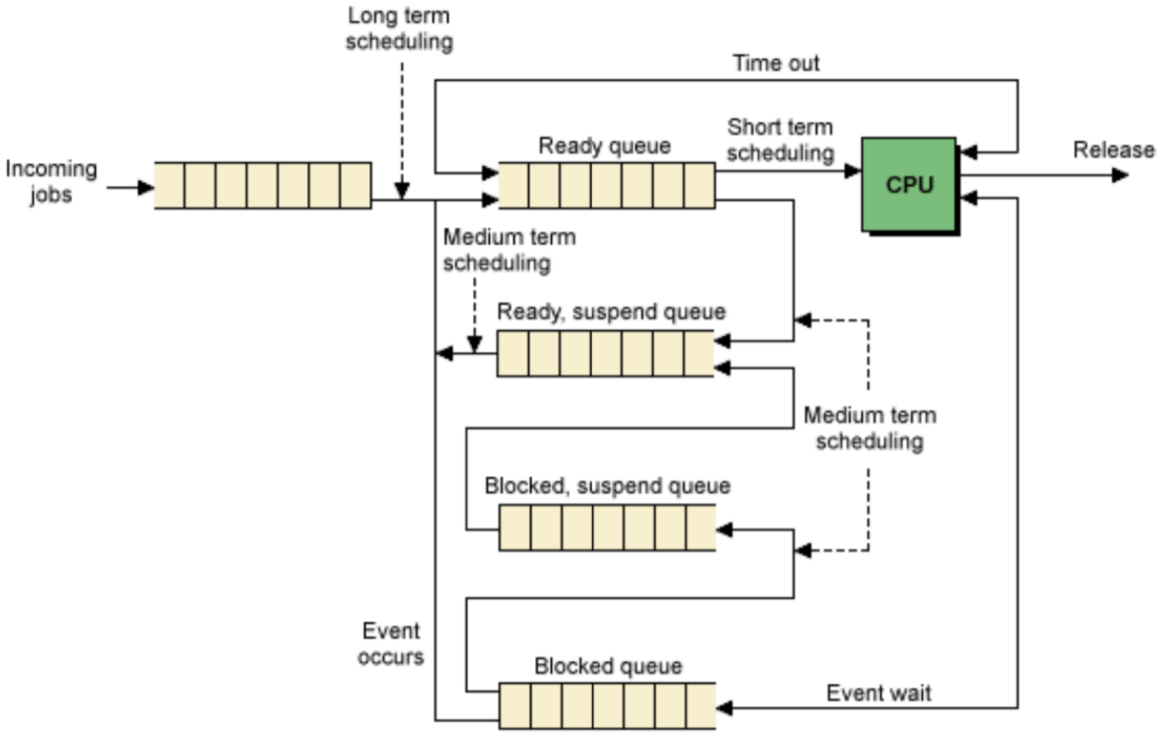
- **ELIGE** quien se ejecuta en la cola READY
- Ejecuta el **CAMBIO de Contexto**

Medium-Term

- Modifica temporalmente el grado de multiprogramacion
- **(Hace SWAP) →** Copiando memoria RAM a disco y de Disco a Ram
- **SACA** los procesos

Long-Term

- **Admite** procesos en la cola READY
- Determina el grado de multiprogramacion (ctd de procesos en memoria)
- **PONE** procesos en la cola **READY**



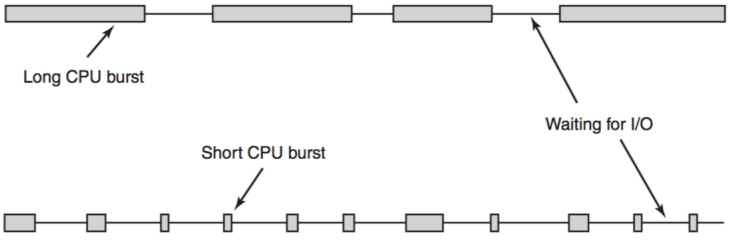
RAM → LA memoria **principal** → Ahi esta la **cola READY**

Disco: Memoria secundaria

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



Al final el que saca lo que no se usa, es el de **mediano plazo**

EXISTEN MUCHOS SCHEDULLERS

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

Pregunta tipica: los mas usados som los **expropiativos (Preemptive)**

→ Dado que los colaborativos son casi inutiles, mejor expropiar que esperar a que un proceso termine por si solo

Otros

Por Objetivo
Batch: Trabajo por lotes. Sin interacción.

- Mantener la CPU lo más ocupada posible
- Minimizar turnaround time: tiempo desde envío hasta término.
- Maximizar throughput: número de trabajos por hora

Interactive : Permite que un proceso ejecute hasta que:

- Minimizar tiempo de respuesta
- Satisfacer usuarios

Real Time:

- Tiempo de respuesta debe ser predecible
- Alcanzar deadlines

Real tiem: en tiempo real se va produciendo

Ej: Apenas pase una llamda, quiero transcribirla ahora

Toma el que va llegando

Usualmente se usa para cosas de urgencias → Como hospitales

O Compras → Eligo el celular → nos llevan a webpay → esa solicitud tiene que ser en el momento

Batch: En un momento mas tardio

Ej: Esperar a la noche en cierto horario → Eligo el momento (no en tiempo real)

Toma por ejemplo, un lote de 40 y transcribo

Ej: Compro un celular → Se procesa mas tarde, y te lo envian a los dias

Interactivo: Minimizar el tiempo de respuesta

Es una mezcla de ambos

Se trata de interacutar con el usuario

CONCEPTOS IMPORTANTEs

- **BATCH** = un proceso batch es sin ninguna interacción
- **Turnaround** = tiempo que el proceso permanece en el sistema
- **Throughput** = cantidad de procesos terminados por unidad de tiempo

Throughput es importante → Es la : Cantidad de procesos / Tiempo (seg generalmente)

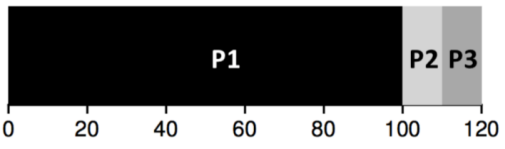
ORDEN DE LLEGADA

FIFO

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 → 79

- Non-Preemptive
- Simple
- Poco predecible. Convoy effect

El segundo en el que sali - El segundo en el que llegue

100 + 10 - 1 → Llegue en 1 = 109

110+ 10 - 2 → Llegue en 2 = 118

0 10 → 10

1 10 → 19

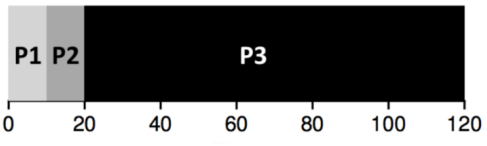
2 100 → 118

Siempre generalmente se pone el mas corto primero

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**

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

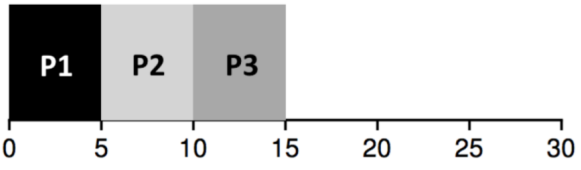
PROBLEMA : No atender un proceso por mucho tiempo → (Pasa hambre)

Puede haber uno que es largo, pero es importante → Y no lo estariamos ejecutando todavia

ALGORITMOS DE SCHEDULING

Métrica: tiempo de respuesta (response time).
Tiempo desde llegada (a la cola) hasta primera ejecución.

	Llegada	CPU-burst	Término	Turnaround	Inicio ejec.	T.Respuesta
P1	0	5	5	5	0	0
P2	1	5	10	9	5	4
P3	2	5	15	13	10	8



- Turnaround time promedio: 9
- Response time promedio: 4

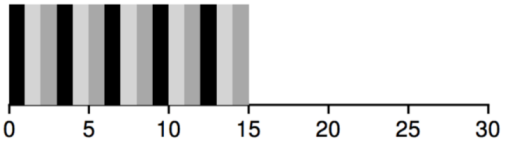
Scheduling interactivo → Cree que los programas le dan mucho atencion a cada programa

Cada cierto tiempo, para que el tiempo de respuesta sea bajo

Esta avanzando, pero significa siempre que van a terminar

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 $(n-1)$xq para ejecutar
  - Altamente dependiente de la elección de q(¿qué pasa si q es muy pequeño o muy grande?)

- Asi se ve interactivo, cada ciertos instantes les toca
- Afecta el turnaround time → Hubieran salido antes con otro FIFO
- Cambio el tiempo que pasan en el sistema (turnaround) → A cambio de tener un tiempo de respuesta mas bajo

PRORITY 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

Permite reducir la prioridad de un proceso (incrementa su niceness)

Existen priorida + tiempo de vida → AGING

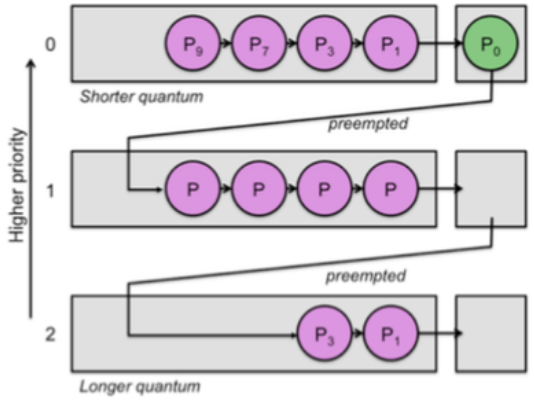
STARVATION: Sufren de baja prioridad → Nadie los tomo en cuenta

- Para eso existe aging, para que cuando pase el tiempo, sumen aging y les de mas prioridad

PRIORIDADES DE LA COLA

Colas de multinivel → Se van cambiando de colas con nivels de prioridad

Me van a atender cuando sea el mas importante de una Cola (A, B o C)



Observacion → No faltar a la clase de paginacion → dado que los videos en youtube son demasiado largos y no sirve tanto verlos → Mejor venir a la clase que es mas densa

CLASE 5: THREADS y SINCRONIZACIÓN

26/08/24

*Threads nacen para ayudar a **paralelizar** tareas, o Distribuir*

- Los procesos adentro tienen threads
- Un proceso, tiene archivos codigos estados, para parelizarlo antes que acabe, existen los threads, porque pueden haber tareas diferentes, y no necesariamente queremos ejecutarlos de manera lineal
- Si es que hay tareas que se pueden hacer en paralelo → Usar Thread

Pregunta de prueba: ¿Que comparten los threads?

NO COMPARTEN	**COMPARTEN**
(Estado)	Recursos
Thread ID	Archivos
Program Counter	Codigo
Registros	Procesos hijos
Stack	**SEÑALES**

Pregunta: Los threads no comparten id, pero si comparten recursos como archivos? → VERDADERO

Al igual que los procesos, los threads tienen id, no lo pueden compartir

Tampoco pueden compartir PC (program counter) igual, dado que cada uno vive en una linea diferente, ya que cada uno hace cosas distintas

Como los threads comparten archivos, si por ejemplo qusiera cargar un excel, dos threads van a intentar abrir primero el thread, y eso dará problemas → **PROBLEMA DE SINCRONIZACION**

Puede que uno escriba la mitad, el otro la otra, y puede quedar incompleto

Funciones (Mas Usadas en Threads)

- **pthread_create:** Crea un nuevo Thread
- **pthread_join:** Espera un Thread
- **pthread_yield:** Le pasa a la CPU o otro Thread → Pasarle el turno a otro Thread

THREADS SEGUN SU LOCALIZACION ¿Donde viven?

Existen Threads que están precreados, esperando a ser usados → Pueden estar en espacio de Usuario o Kernel

- **KERNELL THREAD**: Si tengo un espacio chico en el Kernell → Tendria un numero limitado de Threads
  - No se pueden crear mas → No escalan
  - No se bloquean, porque pasan directo por abajo, y tienen soporte nativo
  - Son mucho mas **RAPIDOS**
- **USER SPACE THREAD** → Vamos a poder crear mas threads → Si escalan porque hay espacio para poner mas
  - Los threads podrían no funcionar tan bien, porque estan mas lejos del kernell, podrian necesitar permisos
  - Los procesos que neceisten ser escalables, tienen que vivir en el espacio del Usuario

No necesariamente tener mas threads, realiza mas rapido las tareas, si divido por ejemplo en muchos threads

Crear y eliminar Threads → Es mucho mas **FACIL** de Crear/Eliminar un proceso (Por el tema de los hijos → HUERFANOS/ZOMBIES)

ELIMINACION

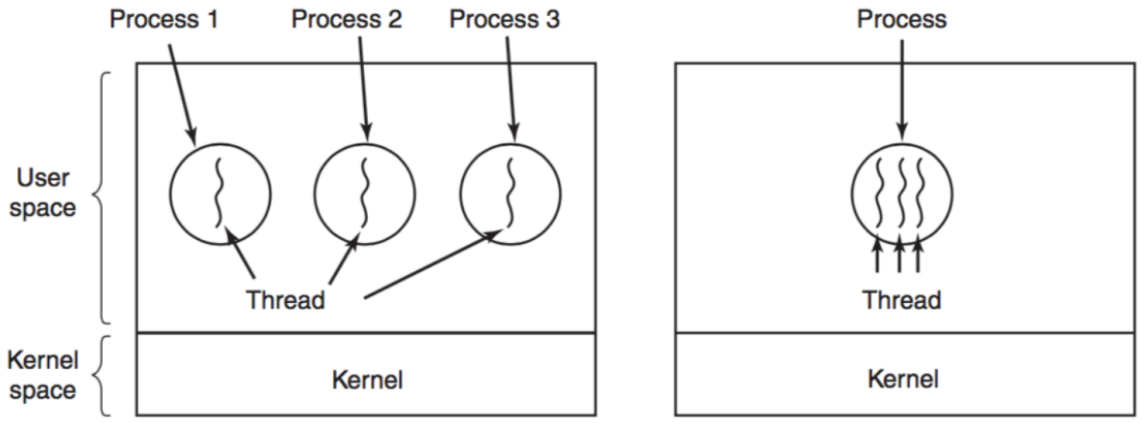
Eliminarmos Threads → Eliminamos su :

- mini Stack,
- su Program Counter y
- ID

v/s

Eliminar procesos → Tiene mucho mas por eliminar

EJEMPLO DE THREAD IN USER SPACE



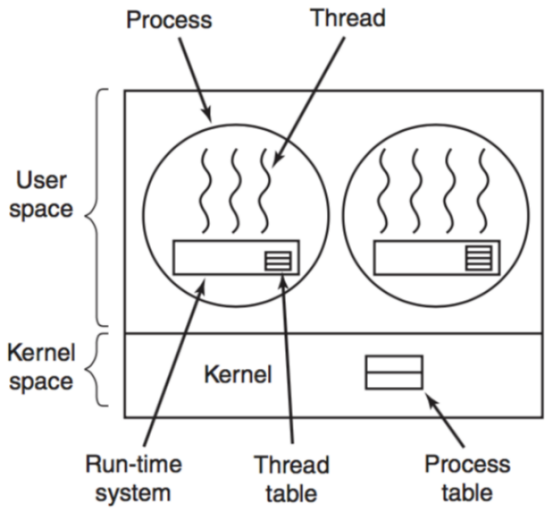
Pregunta → Describa los beneficios de implementar Threads en el espacio del Usuario

- Son mas Escalables (en el kernell son limitados)

- Portable. Implementable en cualquier S.O
- Context Switch y Scheduling mas rapido

MALO:

- Llamadas a syscalls pueden **BLOQUEAR el PROCESO**
- Requiere Scheduling cooperativos



OPERACIONES → FUNCIONES

POSIX THREADS

Estándar IEEE 1003.1c. Librería pthreads

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

CONTENIDO 5: SINCRONIZACIÓN

Los **Threads compiten** por el uso de la **CPU/ARCHIVOS → ¿Si se comparten archivos? A quien leemos primero?**

- Ejemplo: Si tuviera que escribir muchas cosas en un archivo, y cada thread escribiera, no tendría sentido → Race Condition (Respetar su orden)
  - En computacion hay un monton en condiciones de Carrera
- **CONDICION DE CARRERA:** Situacion donde el valor de salida, depende del **ORDEN**, de los Threads

Es muy común, en el mundo empresarial

- Amazon → Tuvo problemas por paralelización (hacian muchos threads) y cada uno tomaba los datos y lo bajaba
  - Cuando el archivo era mas grande,
  - Tenemos un **bien preciado → SECCION CRITICA** como variable, se va sobreescribiendo, segun lo que traiga cada Thread, y gana el que cierra el archivo al final, por lo que los valores no se mantienen, y varian segun el orden de sus Threads

SECCION CRITICA: Seccion critica donde solo 1 Thread accede a la vez

```
void mess_with_shared_things(){
    entry();
    // Seccion critica
    leave();
}
```

Archivos de una **variable Global →** Solo debe leerse 1 a la vez

IMPORTANTE

¿Como nos aseguramos de solucionar secciones criticas?

Si o si esto es pregunta de prueba

3 CRITERIOS (Condiciones)

1. **Exclusion Mutua: A** lo mas uno este en la seccion
2. **Progreso:** Al menos un thread puede entrar, si ninguno entra, y hay threads esperando
    - No habria progreso por ejemplo, si un Thread se queda pegado sin liberar el acceso a la SC (Seccion Critica)
3. **Espera Agotada:** (Ausencia de inanicion) → si un proceso quiere entrar, podrá hacerlo (luego de un tiempo **finito**)

SOLUCION: Flag → Que indique si la seccion esta siendo usada

Sirve para que el Thread no entré si esta siendo usada

Una variable sola no arregla el problema → Podria ser que ya no necesite entrar, deberia haber otra flag para saber si se quiere entrar (ademas si esta disponible)

Puede ser que entre, cuando no quería entrar, y otra que queria entrar no pueda entre → ese thread sufrira de Inanicion

SOLUCION CON TURNOS

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 código para 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 con las condiciones?

- Exclusión mutua: Sí
- Progreso: No
- Porque hay un thread que puede entrar y no quiere. Y otro que quiere entrar y no puede.

Tiene problema? → SI

- Esto genera **inanicion** de threads, que **quieran entrar y no puedan**, porque alguno que **no necesitaba entrar, entro** porque tenia turno

SOLUCION SIMPLE (PETERSON)

Variable turn → Para saber si puedo entrar

Variable extra → Saber si quiero entrar

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 .../ */
}
```

¿Cumple los requisitos? ¿Por qué? ¿Se puede demostrar?

- Exclusión mutua: Sí
- Progreso: Sí
- Espera Acotada: Si un thread 1 quiere entrar va estar bloqueado, y solo va a poder entrar solo cuando el thread 0 salga de SC. Si el thread 0 es muy rápido va esperar infinito.
- Turn == other hace la magia

¿Que pasa si entre, y soy tan rapido que me pongo denuevo en la cola, y entro denuevo, mas rapido que otro que estaba esperando? → UNICO **CASO BORDE**

Para sincronizar Threads por Software → Necesito 2 Variable

Para sincronizar Threads por Hardware → Necesito **Instrucciones Atomicas**

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;
}
```

¿Cómo usarlo?

CLASE 5

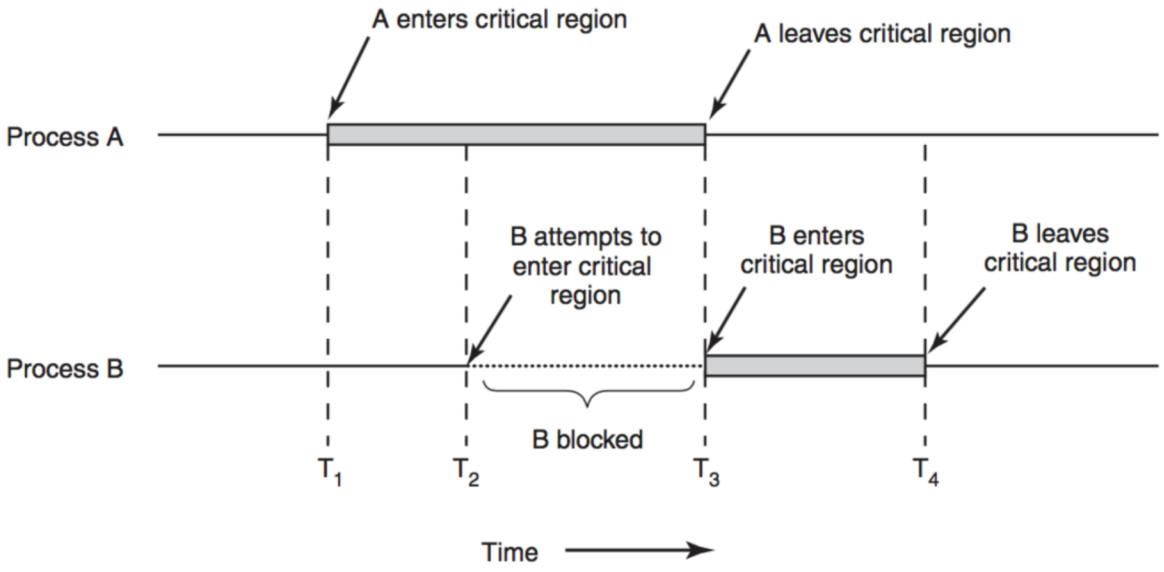
28/08/2024

Cerrando lo de la clase pasada

3 Condiciones de Seccion Critica → Importantes de aprenderselas

- Exclusion Mutua
- Progreso
- Espera Actodada

SITUACION IDEAL



Tenemos que mediante software o hardware, que los Threads entren separados, y no se afecten los resultados

Solucion inicial:

> Turno → Exclusion mutua si → Pero no cumple con progreso → Y por lo tanto con cumple con inanicion

Código para thread 0 y código para 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 con las condiciones?

- Exclusión mutua: Sí
- Progreso: No
- Porque hay un thread que puede entrar y no quiere. Y otro que quiere entrar y no puede.

Solucion Final: Peterson

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 .../ */
}
```

¿Cumple los requisitos? ¿Por qué? ¿Se puede demostrar?

- Exclusión mutua: Sí
- Progreso: Sí
- Espera Acotada: Si un thread 1 quiere entrar va estar bloqueado, y solo va a poder entrar solo cuando el thread 0 salga de SC. Si el thread 0 es muy rápido va esperar infinito.
- Turn == other hace la magia

Sincronizaciones por Hardware

Dependen de que existan **Instrucciones Atomicas**

- Cada computador trae estas, que se ejcutan en el hardware, como Locks que se ejecutan en hardware
- Guardo por ejemplo una varibale que esta bloqueada
- Y que otro thread no puede acceder

Hay varias Sincronizaciones por hardware → Solo se van a usar si se trabaja con sistemas operativos (ej: Microsoft)

- **test_and_set()** : Ejecucion que tiene un booleano, que lo que hace es cambiar su valor, toma su valor lo cambia y lo retorna

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

    return old_value;
}
```

Esto es basicamente lo que hace

Existen muchas formas de implementarla

IMPLEMENTACION INGENUA

```
bool lock = false;  // false==free, true==busy
```

```
while(true) {
    /* ... ... ... */
    while (test_and_set(&lock));
    /* ... SC ... */
    lock = false;
    /* ... Out of SC ... */
}
```

Ingenua porque es la mas facil

Un Thread a la vez (Por hardware) a diferencia de Por software que pueden entrar mas de uno a la vez

- **Compare_and_swap()**

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

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

  return previous;
}
```

Recordar que siempre que esta vacio es falso

No usamos directamente, estas instrucciones atomicas → Si no que usamos **Instrucciones Primitivas (Por Software)**

EJEMPLO

Código equivalente a *compare_and_swap*()(XCHG, CAS)

```
bool 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 ... */
}
```

MUTEX LOCKS

Solucion por Software

Lock::**aquite**() : Toma el lock

Lock::**realease**() : Libera el lock (error si no está tomado)

Pregunta prueba (V/F) → Los mutex son implementaciones por **software**

Hay 2 tipos de espera:

Bussy Waiting → Thread constantemente revisa si puede entrar a la seccion critica → Constantemente revisando un WHILE por su turno

Irse a Dormir → Madar a dormir el Thread mientras espera

EJEMPLOS DE USO

```
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;
}
```

Solución a Sección Crítica

```
while(true) {
    /* ... ... ... */
    lock.acquire();
    /* ... SC ... */
    lock.release();
    /* ... Out of SC ... */
}
```

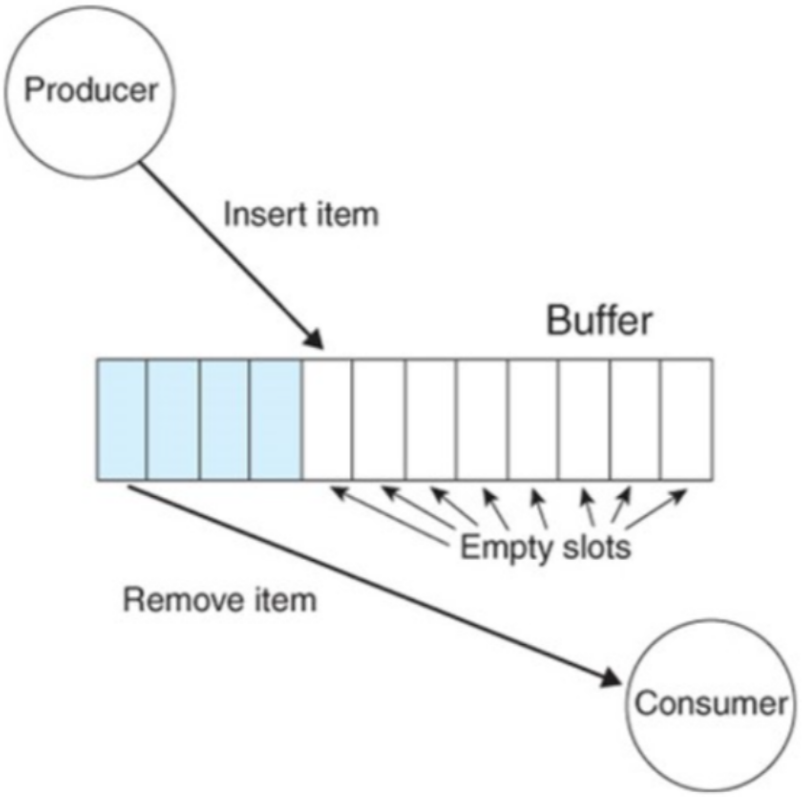
Contador con acceso exclusivo

```
struct Lock l; // compartido
```

```
/* ... ... ... */
for(int i=0; i<max; i++) {
  l.acquire();
  counter = counter + 1;
  l.release();
}
```

Problema concurrente: Si se esta vendiendo y comprando en el momento

Este problema ese llama de **PRODUCTO-CONSUMIDOR**



- El buffer es la lista donde se va guardando cosas

- El espacio se va guardando por el que va produciendo, y sacando el que va consumiendo

- Un buffer circular de **tamaño limitado N**
- Dos threads: productor (agrega) y consumidor (consume)
- Todos los elementos producidos deben ser consumidos en orden

EJEMPLO DE USO

```
int stock = 0;
T storage[N];
```

```
int in = 0;
while('p') {
    item = produce();
    while (stock == N);  // Storage full
    storage[in] = item;
    in = ++in%N;         // rotate index
    stock++;
}
```

```
int out = 0;
while ('c') {
    while(!stock);       // Wait for new items
    item = storage[out];
    out = ++out%N;       // rotate index
    stock--;
    consume(item);
} /* ' */
```

Y ahora agregar LOCKS al stock, porque esa es nuestra seccion critica

Podemos agregar locks en stock

```
int in = 0;
while('p') {
    item = produce();
    while (stock == N);  // Storage full
    storage[in] = item;
    in = ++in%N;         // rotate index
    lock.acquire();
    stock++;
    lock.release();
}
```

```
int out = 0;
while ('c') {
    while(!stock);       // Wait for new items
    item = storage[out];
    out = ++out%N;       // rotate index
    lock.acquire();
    stock--;
    lock.release();
    consume(item);
} /* ' */
```

¿Cómo eliminamos busy waiting?

Como se elimina el busy waiting → Irse a dormir

- sleep() se bloquea en cola waiting
- wakeup(t) saca a t de la cola waiting (si está)

```
int in = 0;
while('p') {
    item = produce();
    while (stock == N) sleep(); // Storage full
    storage[in] = item;
    in = ++in%N;                // rotate index
    lock.acquire();
    stock++;
    lock.release();
    if(stock == 1) wakeup(consumer);
}
```

```
int out = 0;
while ('c') {
    while(!stock) sleep();     // Wait for new items
    item = storage[out];
    out = ++out%N;             // rotate index
    lock.acquire();
    stock--;
    lock.release();
    if(stock == N-1) wakeup(producer);
    consume(item);
} /* ' */
```

Si no hay comida, mandame a dormir, si hay comida, despiertame

Cuando se genere una comida → Le avisa al Consumidor que hay comida

Cuando el consumidor consume → Le avisa al Productor

SEMAFOROS

Primitiva de sincronización que invento Dijkstra → De P dercrementa y V Aumenta

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.

```
struct semaphore{
  int count;
  struct Lock l;
  struct process* slept = NULL;
}
```

```
void init(c) {
    count = c;
}

void P() {      /* wait: put into queue and sleep */
  l.acquire();
  while(count <= 0) {
      /* code to add itself to 'slept' */ // /
      l.release();
      sleep();
      l.acquire();
  }
  count--;
  l.release();
}
```

```
void V(S) {    /* signal */
  l.acquire();
  count++;
  wakeup(/* first from 'slept' */);
  // plus, remove from queue
  l.release();
}
```

Implementan una version sin Bussy Waiting

Los semaforos se pueden arreglar mucho haciendo usando sleep

Pregunta prueba → Semaforos tiene una implementacion sin Bussy Waiting

Variables de Condición

Variables de condición permitan bloquearse bajo condiciones arbitrarias.

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

IMPLEMENTACION MAS SENCILLA → Con 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);
}
```

No hay nada que consumir → andate a dormir

Consumi → Oye despierta → Anda a producir

Consumidor → Existe algo para consumir → Anda a consumir

Implementación bastante bella

- [ ] Estudiar

PROBLEMA LECTOR-ESCRITOR

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

→ Muchos quieren saber el valor de una variable → Pero solo 1 puede cambiarla

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

Escritor (fácil) un semáforo **rw** compartido entre escritores y lectores

```
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();
}
```

Lectores un semáforo mutex compartido

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

Pueden haber muchos readers, pero leer no necesariamente bloquea, pero **NO pueden ESCRIBIR**

Esto es comun en la base de datos

Strongly consistent → Solo uno escribe a la vez

No strongly consistent → Pueden tener errores

PROBLEMA FILOSOFICO

Menos palillos que personas → Y hay que compartir

5 filósofos se pasan la vida en dos actividades: comer y pensar
- Para comer deben tomar ambos palillos
- Sólo pueden tomar un palillo a la vez

ej 4 palillos 4 comensales:



Ilustración del problema de la cena de los filósofos.

Se tienen que poner de acuerdo para sincronizarse

Código para filósofo **i**

```
int left =  i;
int right = (i+1)%N;            // N==5
struct semaphore chopstick[N]; // inicializados en 1

do {
  chopstick[left].P();
  chopstick[right].P();

  /* eat */

  chopstick[left].V();
  chopstick[right].V();

  /* philosophise */

} while ('p');
```

Otra implementacion

```
int N = 5;
enum {PHILOSOPHISE, HUNGRY, EATING} state[N];   /* free / waiting / using */
struct lock l;
struct semaphore philosopher[N]; // inicializados en 0
```

```
int left(i)  { return (i+N-1)%N; }
int right(i) { return (i+1)%N; }

void philosoph(int i) {
    while(TRUE) {
        think();
        take_forks();
        eat();
        release_forks();
    }
}

void take_forks(int i) {
    l.acquire();
    state[i] = HUNGRY;
    try_forks(i);
    l.release();
    philosopher[i].P();
}
```

```
void release_forks(int i) {
    l.acquire();
    state[i] = PHILOSOPHISE;
    try_forks(left(i));
    try_forks(right(i));
    l.release();
}

void try_forks(int i) {
    if(state[i] == HUNGRY && state[left(i)] != EATING
       && state[right] != EATING) {
        state[i] = EATING;
        philosopher[i].V();
    }
}
```

Clase 6 (Online) → Mauro Quintana

MEMORIA

2/09/24

Resumen:

- **Primitivas de Software**
  - Mutex Lock: Aquire() - Realease()
  - Semaforos: V() - P()
  - Varaibles de condicion: singal() - wait()

- Propiedades de la Sincronizacion (Seccion Critica

- Exclusion mutua
- Progreso
- Espera Acotada

Problema visto en clases → Problema de los 5 filósofos

Cada filosofo tendre un semaforo que va a empezar en 1

Los filosofos van a tener 3 estados posibles

Estado filosofal → Tiene palillo

Estado hungry → Tiene intenciones de utilizar el palillo

Estado eating → Esta usando el

```
int N = 5;
enum {PHILOSOPHISE, HUNGRY, EATING} state[N];   /* free / waiting / using */
struct lock l;
struct semaphore philosopher[N]; // inicializados en 0
```

```
int left(i)  { return (i+N-1)%N; }
int right(i) { return (i+1)%N; }

void philosoph(int i) {
    while(TRUE) {
        think();
        take_forks();
        eat();
        release_forks();
    }
}

void take_forks(int i) {
    l.acquire();
    state[i] = HUNGRY;
    try_forks(i);
    l.release();
    philosopher[i].P();
}
```

```
void release_forks(int i) {
    l.acquire();
    state[i] = PHILOSOPHISE;
    try_forks(left(i));
    try_forks(right(i));
    l.release();
}

void try_forks(int i) {
    if(state[i] == HUNGRY && state[left(i)] != EATING
       && state[right] != EATING) {
        state[i] = EATING;
        philosopher[i].V();
    }
}
```

POSIBLES CASOS DE LOS THREADS

1. **Happy path** → irse a take_forks : Y se ejecuta a la primera, aquire el lock

   Intento tomar los palillos

   Entro a try forks

   Compruebo el estado de los comesales a la derecha y izquierda

2. **Unhappy path** → Alguno esta usando los palillos

Dentro de try forks

Nunca accedi a philoshoppher[i}.V() → Por lo tanto el contador sigue en 0

Y se va a bloquear en la instruccion philosopher[i].P() → Porque el contador sigue en 0

No pude decrementar en el contador

El de la izquierda o la derecha esta comiendo

Por lo tanto ese ejecuto take_force

va a comer eat()

y va a soltar los palillos

en realease forks → Cambia de estado a filosofando

Y le avisa al de la izquierda → Intenta tomar los palillos

Ahora en nuestro principal si cumple, aumenta el contador → Vuelve a estar en 1

Dentro de la funcion, desperte al proceso de la cola de dormidos

El que estaba durmiendo era philosopher[i].P() para decrementar el contador

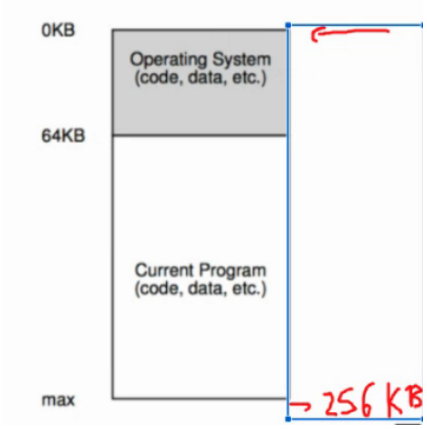
Se levanta y hace lo que debia

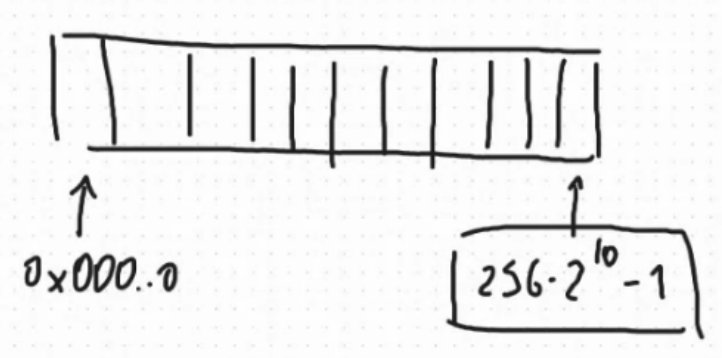
Finalmente en este caso, es otro thread el que nos avisa que podemos continuar (realese_forks)

CONTENIDO 6: MEMORIA

Veremos como el sistema operativo es capaz de adminisitrar la memoria principal del proceso que esta ejecución

La memoria es un arreglo de bytes



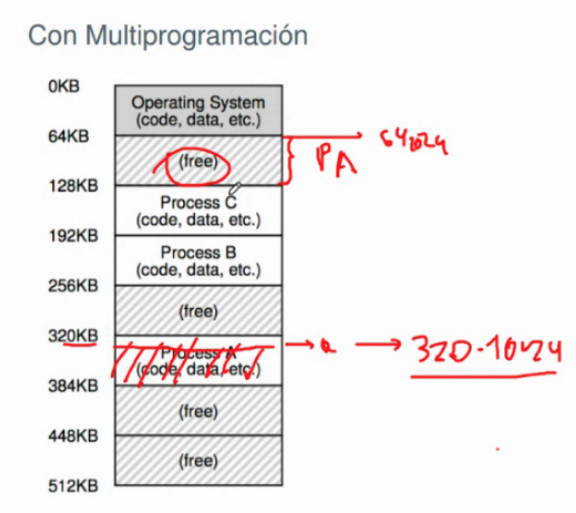


1 KB	1024 B	2**10 B
1 MB	2**20 B	
1 GB	2**30 B	

Como guardamos los procesos? → Variables codigos etc

1. Sin Multiprogramacion → Muy facil
    - Solo tenemos que cargar el espacio completo

2. Con Multiprogramacion se complica mas
    - Si el SO, no se encarga de avisarme donde esta
    - Voy a entrar a una direccion de memoria que no es propia
    - Por lo tanto: **DIRECCIONES ABSOLUTAS EN MULTIPGROMACION → NO SIRVE**



Compilador + Sistema Operativo

Debe encargarse de utilizar **no direccions absolutas**, si no relativas dentro de la memoria principal

No buscar la direccion absoluta, si no la actual → (Se va actualiuzando)

PROBLEMA 1

Relocalización de memoria

→ Problema con intentar acceder a memoria realocalizada, **las variables** no siempre estan en la **misma ubicacion**

PROBLEMA 2

Proteccion de memoria → Segmentation Fault

Por ejemplo leer/escribir memoria que no correspondia al programa

En lugar de usar direcciones absolutas, usaremos Virtuales

DIRECCIONES

- Direcciones **Absoultas**: Relacionadas a la memoria Ram/principal
- Direcciones **Virtuales:** Las que “ve” el proceso, y entiende en el universo del proceso

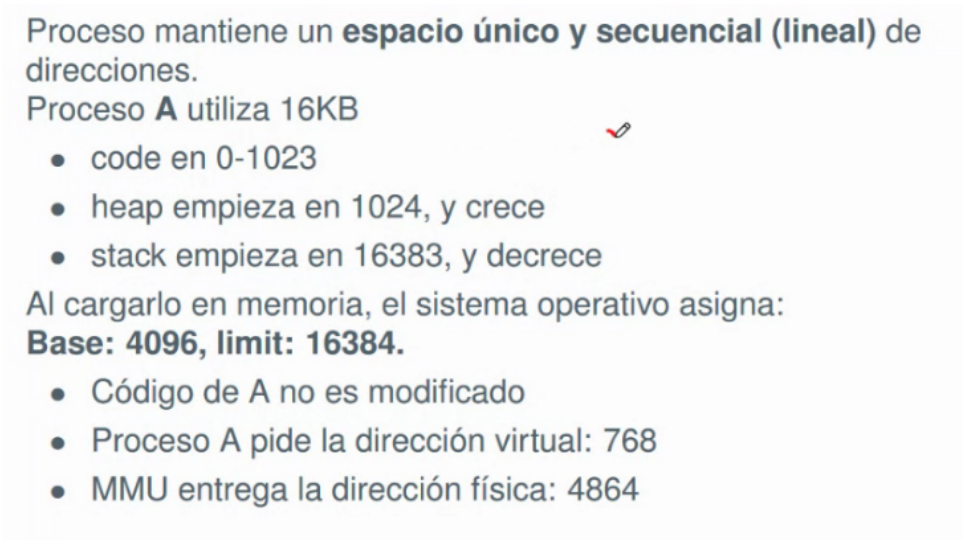
- Espacio de direccion para los procesos

Lo que haremos al final: Usaremos Direcciones virtuales en vez de absolutas

Instrucciones codigo variables se van a encontrar en la misma lugar desde el punto de vista del proceso

TRADUCCION VIRTUAL → ABSOLUTA

Antiguamente: 2 registros, uno base y uno limit, para saber donde empieza y termina el espacio



El espacio de memoria de un proceso se **DIVIDE en SEGMENTOS**

Tienen cierta interpretacion:

→ Code Heap y stack, memoria dinamica va creciendo en funcion del programa

Heap Crece

Stack Decrece

Al codigo base de la direccion, se le suma la direccion virtual, y esa se entrega como direccion Fisica

La **MMU: Memory Managment Unit → Es quien realiza esos calculos**

Los tipicos problemas como **segmentation fault**, son que nos salimos de la dirección que tenemos para el programa

Ahora bien, ese calculo no es tan eficiente, existen mejores estrategias

SWAPPING

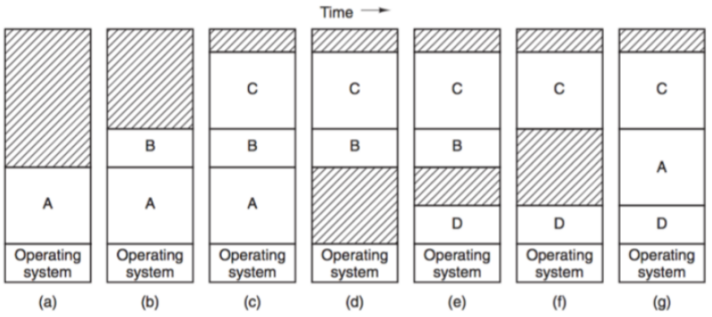
1. Espacio Swap: memoria secundaria que usaremos como memoria secundaria, si se acaba la principal usaremos la swap
2. Tendremos 2 operaciones

- **swap-out**: Llevar la memoria de un proceso a la memoria secundaria (disco (memoria swap))
- **swap-in**: Llevarle memoria del programa de la memoria secundaria a la principal

El scheduler encargado de esto → **MEDIUM-TERM Scheduler**

SOLUCIONES A **PROBLEMAS DE ESPACIOS LIBRES** → **Fragmentación Externa**

Procesos pueden ser cargados y descargados de la memoria

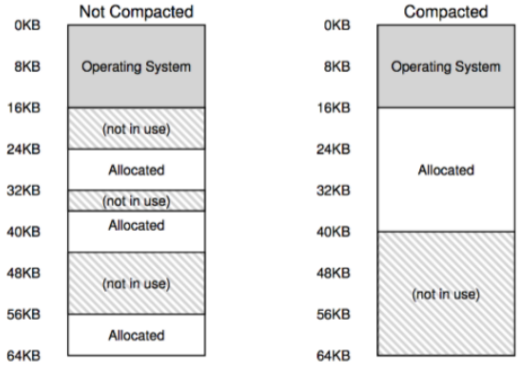


Memoria del sistema queda con huecos
¿Qué hacer con los huecos?

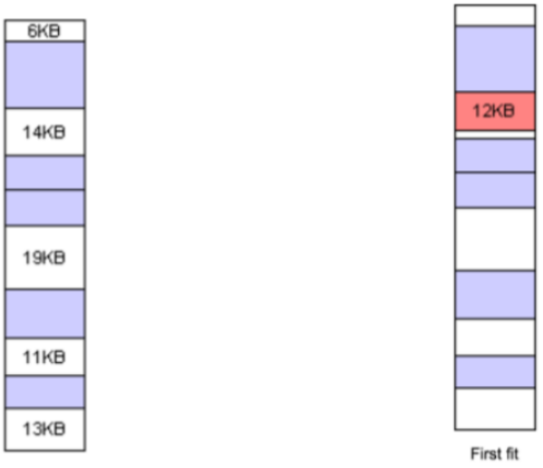
Quedan espacios!!!

Soluciones

1. **COMPACTACION**: Juntar toda la memoria (**Fusionar los huecos)** → Es muy costoso, no es practico (necesito cambiar registros base y limit, por cada uno y no queremos hacer eso)
   - Requiere copiar memoria a otras direcciones



2. **First Fit**: Encontrar el primer espacio que este disponible (donde quepa la memoria)
   - Ventajas:
   - Ninguna, es aleatorio



3. **Best Fit**: El lugar donde cabe y que deja menos espacio (mas pequeño posible)
    - Intentar desperdiciar la menor cantidad de memoria posible



4. **Worst Fit**: El lugar que deja mas espacio libre (mas grande)
    - La gracia es que donde lo deje, haya mas probabilidad que caiga espacio para otro programa

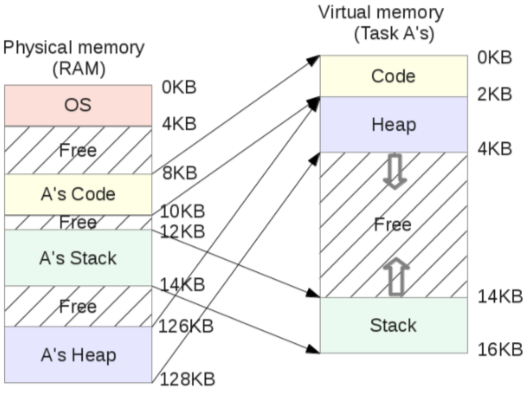
ESTRATEGIAS → ELIMINAR POR COMPLETO PROBLEMA DE FRAGMENTACION

1. **SEGMENTACION →**

Ya mencionamos que a ciertas partes del codigo le damos interpretaciones:

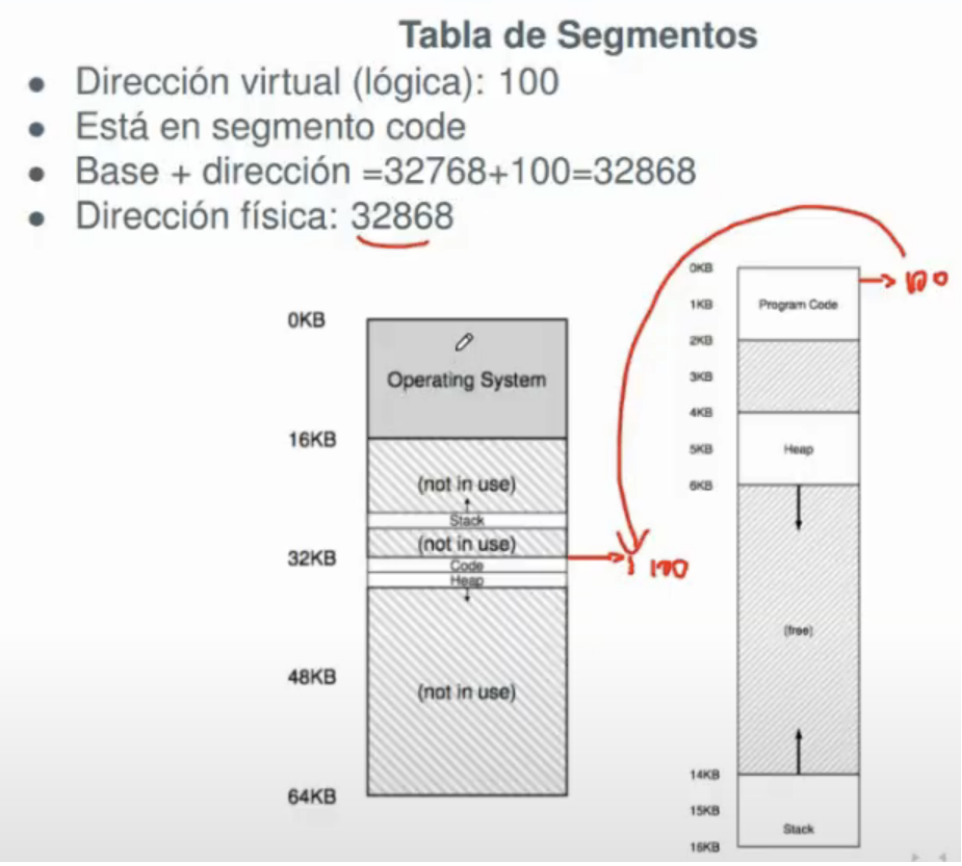
La idea es ya no guardar todo el bloque de codigo unico de un programa dentro de la memoria fisica → si no que guardar segmentos de manera **INDEPENDIENTE**

→ al ser mas pequeños, mas probabilidad que caigan en un lugar mas pequeño



Para ello, la **MMU debe ser capaz de TRADUCIR**

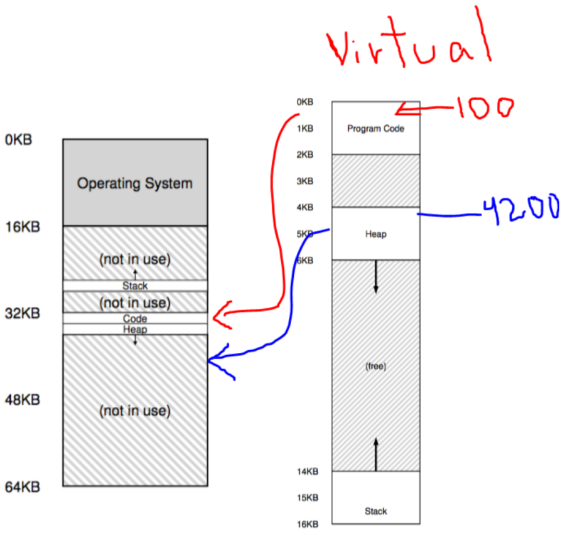
Segment	Base	Size
code	32768 (32KB)	2048 (2KB)
heap	34816 (34KB)	2048 (2KB)
stack	28672 (28KB)	2048 (2KB)



PROBLEMA → SEGMENTION FAULT

Tabla de Segmentos

- Dirección virtual (lógica): 4200 (offset)
- Está en segmento code
- Base + dirección =32768+4200=39016
- Dirección física: 39016

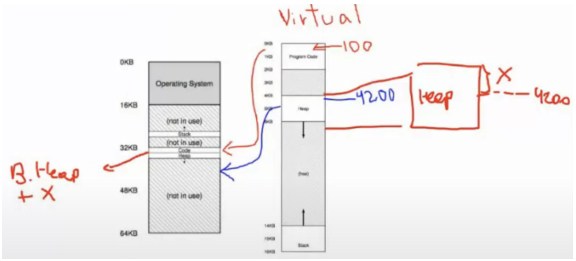


Espacio Libre: Espacio no asignado a este proceso produce segmentation fault

En este caso claro, la direccion virtual del heap por ej buscamos en 4.2kb, eso si se lo sumamos a la base de la direccion fisica, ira a buscar memoria que no pertenece al programa → **SEGMENTION FAULT**

SOL:

- La manera mas simple de corregir esto, es teniendo la direccion virtual de donde empieza el heap, restarle la direccion (offset: 4200), y esto nos da el desplazamiento (virtual)
- Este mismo, se le suma a la direccion del Heap (de nuestro programa en la memoria real), y asi no nos salimos.



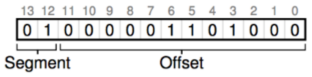
LOGICA DE BITS

Para hacer lo que hicimos con aritmetrica en decimales nuestros, podemos hacerlo con logica de **BITS**

¿Cómo conocer segmento y offset?

- Se pueden hacer restas . . .
- Pero la lógica de bit es más rápida

Ejemplo: ¿cuántos bit necesitamos para identificar 3 segmentos? →2



Dirección virtual: 4200=0b01000001101000=0x1068

- Bits de segmento: 0b01=1=0x1
- Bits de offset: 0b000001101000=104=0x068

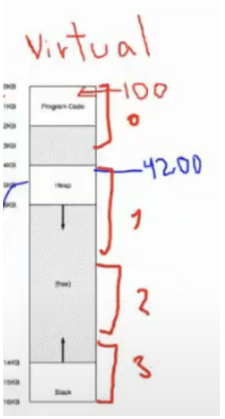
Los **bits de segmento**: Nos dice en **que segmento** nos encontramos

Los **bits de offset**: Nos dice **que tan desplazado** estoy desde el INICIO del segmento

En este caso, tengo 12 bits para el offset, por lo tanto, lo maximo que me puedo desplazar es 2**12 bytes

Si tengo 2 bits de segmento, puedo tener

- 2**2 posibles segmentos: 4
- 00
- 01
- 10
- 11



Notar que

Bits de offset: 0000 0110 1000: 104

Y la direccion del segmento 1 aprox (heap): 4kb → 4096

4096 + 104 = 4200 !!! → Magia? No → LOGICA!

☐ Investigar → Mascara de segmento

Clase 7

4/09/2024

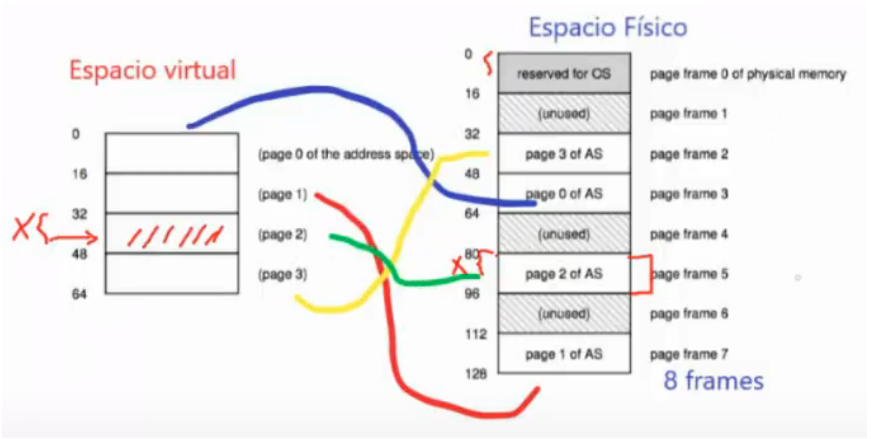
PAGINACION

Otra estrategia para localizacion de memoria

Lo que hace es dividir la memoria fisica → en framse

Y la virtual en → Paginas

|Paginas| = |Frame|

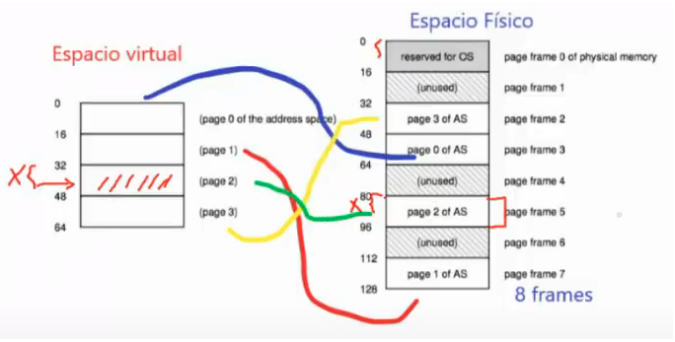


Para hacer esto, necesitamos informacion extra → Necesitamos una **TABLA DE PAGINAS**

TABLA DE PAGINAS:

- Guarda las traducciones de memoria de **pagina a frame**
- Basicamente es en que frame se encuentra guardada cada **pagina de un proceso**

Ejemplo



Para este caso, notemos que tenemos 64b de direcciones de espacio virtual

direcciones = 64 = 2**6 = 6 bits necesarios

Entre cada pagina hay 16 bits de desplazamiento

offset = 16 = 2**4 = 4 bits

y 64/16 = 4 = 2**2 = 2 bits de **numero de pagina**

Bits direccion virtual = b5 b4 b3 b2 b1 b0

- b5 b4 = VPN (numero pagina) → virtual page number
- b3 b2 b1 b0 = OFFSET

En el espacio fisico: 128b = 2**7 = 7 bits necesarios

offset = 16 → 4 bits

Numero frame = 8 = 2**3 = 3 bits (PFN) → physical frame number

Bits direccion fisica = b6 b5 b4 b3 b2 b1 b0

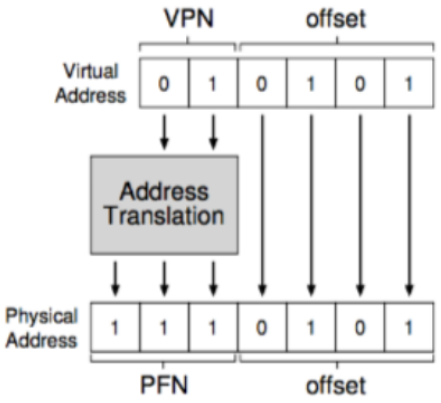
- b6 b5 b4 = PFN (numero frame)
- b3 b2 b1 b0 = OFFSET

Ahora → Como traducir una direccion a la otra?

TABLA DE PAGINA

Almacenara la traduccion de un numero de pagina a un numero de frame

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



CONCLUSION (y pregunta de prueba probablemente):

TRADUCCION: conservar **OFFSET** y cambiar VPN por PFN

La tabla ademas de las traducciones tambien almacena otros valores: (**Metadatos**)

- **Bit de validez**: indica si la entrada es valida (si la pagina ha sido **asignada**) → Importante para la tarea
- Bit de presencia: si esta en la memoria primaria o secundaria
- Bit de escritura: indica si se puede escribir
- etc.... → Depende de cada OS

PTE (page table entry) = PFN + Metadatos

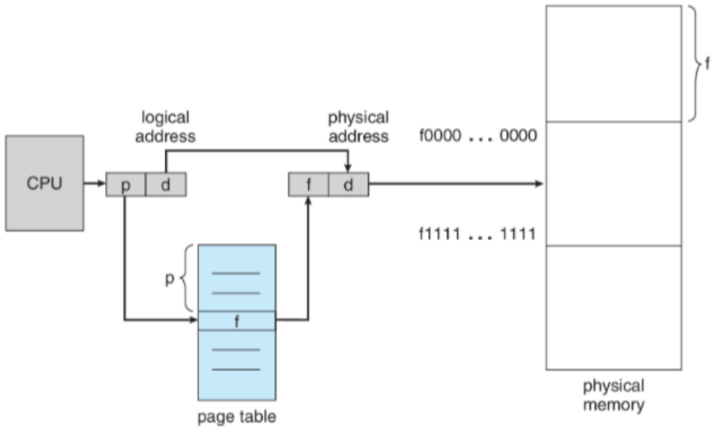
|Tabla de pagina| = | PTE | * Numero de paginas

Esto porque la tabla de paginas se debe almacenar en la memoria principal, y esa cantidad de memoria es significativa → Hay que tratar de reducir el tamaño de la tabla

Ojo: Esto afecta al tiempo de acceso (al doble practicamente)

Dado que siempre para buscar una variable, primero hay que cargar la tabla de paginas para traducir, y luego ir a buscar

Por lo tanto: **CADA ACCESO A MEMORIA SE DUPLICA (**a 2 accesos**)**



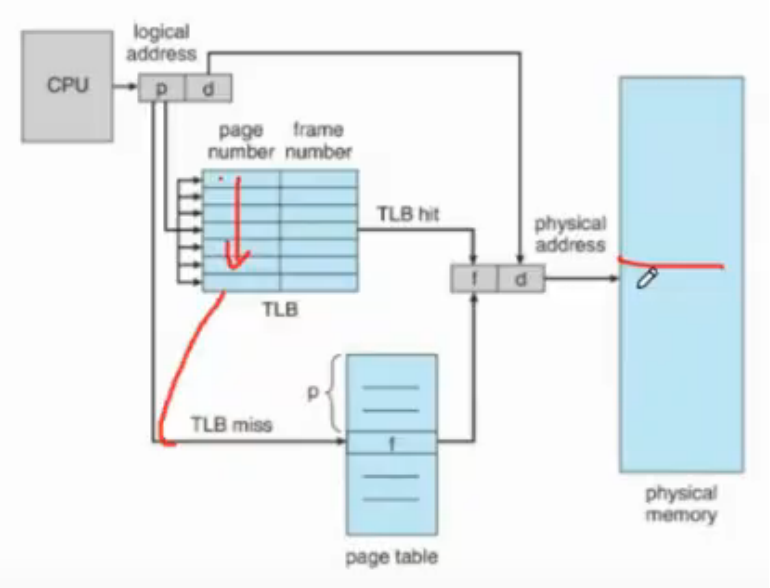
Solucion: Para acelerar acceso a memoria: fully associative **caché de direcciones en MMU**

Para esto utilizaremos un **CACHE** llamado **TLB**

TLB Translation look-aside buffer (TLB)

- Traducciones de VPN → PFN
- Se almacenara dentro de la memoria de la cpu, y no dentro de la memoria principal
- TLB HIT: Encontramos traduccion
- TLB Miss: Traducción no esta en caché

Con esto, los accesos a memorias se verían asi:



Con esto, si obtenemos un TLB Miss → Nos demorariamos mas, porque tenemos que buscar las mismas operaciones que antes (buscar en page table y physical memory) sumado a buscar en la tlb y no encontrar, si tenemos un tbl hit, nos ahorramos buscar en la page table, tlb es mucho mas rapido de buscar (no es significiativo realmente si hay tlb miss)

EJEMPLO DE TLB

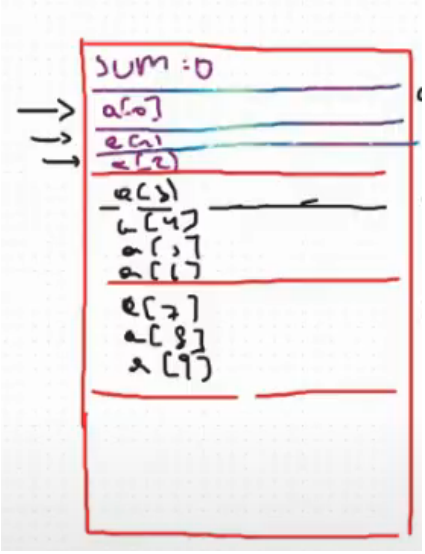
```
int sum = 0
for (i = 0; i<10; i++){
    sum += a[i];
}
```

Si solo hacemos accesos a memoria sin tlb

Tenemos que hacer las 2 operaciones por cada acceso a memoria a[i]

Por lo tanto son 10*2 = 20 Accesos a memoria

Con TLB, primero veamos como esta almacenado en la memoria del programa



Esto, dado que el espacio virtual de 8 bit, con paginas de 16B, almacenando un arreglo de 10 int (4B)

Sum = 0 → 4B

a[0] → 4B

a[1] → 4B

a[2] → 4B

Ahi ya tenemos 16B, por lo que saltamos a otra pagina

a[3], a[4], a[5], a[6] caen en la siguiente pagina, y asi...

Notemos que, cada vez que acceda a un valor por pagina, suponiendo que no estaba en la cache, nos da un miss, pero los siguientes, ya estan cargados en la tlb, por lo tanto, son hits:



Asi tenemos Miss*3 , Hit*7 → 70% **TLB Hit Rate** (7 de 10 accesos ahorrados)

Con paginas mas grandes → El Hit Rate Aumenta!!!

TIEMPO ACCESOO PROMEDIO: T(hit) * Hit_Rate(%) + T(miss)* (1 - hit_rate)

ej: tiempo del ejemplo de acceso: 101ns x 0,7 + 201ns x 0,3 = 131 ns

Clase 8

09/09/2024

Explicaciones de porqué **TLB Hit Rate REALES! ~= 99%**

Localidad Espacial

Estamos escribiendo en un arreglo → Se guarda de forma contigua en memoria

Como el programa probablemente vaya a acceder a memoria cercana a la buscada (memoria espacialmente cercena)

Luego de un **miss** → Vendrán muchos **HITS**

Localidad Temporal

Acceder a variables que ya habiamos accedido con anterioridad

Como el programa accedio a una variable, probablemente será accedida en un futuro

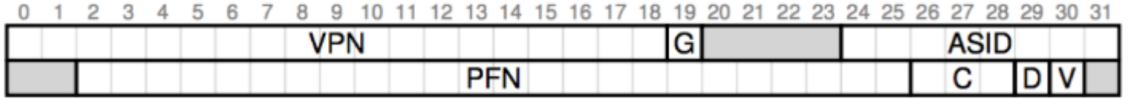
Ahora bien, como identificamos las traducciones? Sobre todo en los cambios de contexto

Recordatorio: Cambio contexto: Cambiar espacio de memoria que estoy utilizando, desde un proceso A a un proceso B

Porque la **TLB NO! se almacena en memoria del S.O.** (como la Tabla de Páginas), **ni en el PCB**

PTE's de un proceso, no le sirven al siguiente!

Si por ejemplo tuviera 0 (pag)→ 255 (frame), ¿A que proceso le pertenece esta traduccion? → Tenemos un ASID (funciona como un PID)



ASID: Adress Space Identifier:

- Cada proceso tiene su unico ASID (en su PCB)
- Campo ASID permite que coexistan entradas de distintos procesos en la TLB

V = Bit de validez: Si la entrada es valida o no

Con esto

Si un proceso que ejecuta, sale de la cpu → En lugar d eperder traducciones, al volver a entrar, las puede reutilizar sin tener Tantos Miss's

PROBLEMA: Tablas de paginas muy grandes

Una Arquitectura con

1. Espacio de direcciones virtuales de 32 bit

- 32 bit → 2**32 Direcciones posibles → 2**32 Bytes
- 2**32 Bytes = 2**2 * 2**30 Bytes = 4 * 1GB = **4GB Maximo de memoria Virtual!**

2. Paginas de 4KB. 20 bit para pagina, 12 para offset

- Cuantas paginas caben en la memoria virtual:
  - 4GB/4KB = 2**2 * 2**30/2**2 * 2**10 = **2**20 Paginas**
- Si tengo 2**20 paginas → ¿Cuantos bits? = 20 bits para VPN
- Se cumple que son 20!!!

3. La tabla de paginas almacena 1 entrada: que almacena la traduccion de 1 pag a cierto frame

- Por lo tanto: 2**20 pag → 2**20 PTE's (page table entry) (entrada) → Guarda PFN + metadatos (32 bits aprox : 4B)
- Tamaño Tabla paginas: 2*20 × 2**2B = 4MB (no es tanto)

- Consideran que hay 1 tabla de paginas por cada proceso
- Y suponiendo que tenemos corriendo 100 procesos → 400MB del S.O Solo en Tablas de PAG

Conclusion: ES DEMASIADO!!!!

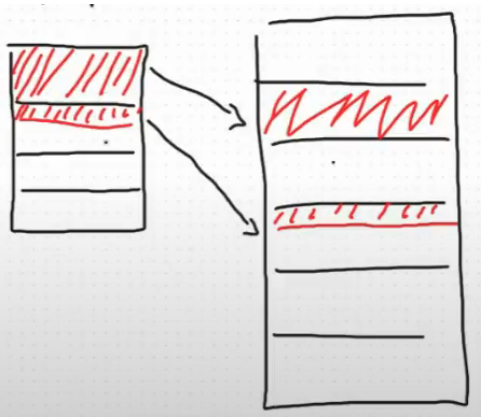
SOLUCION (NO FIABLE)

Usar Paginas **MAS GRANDES!!!**

Si usamos paginas mas grandes, en lugar de usar Pag de 4KB, usamos paginas de 16KB

- 4GB/ 16KB: 2**2 * 2**30 / 2**4 ** 2**10 = 2**16
- 2**16 es mucho menos que 2**20 !! Reducimos la cantidad de entradas de la pagina (PTE)
- 2**16 PTES necesarias

PROBLEMA CON ESTO: Como las paginas son muy grandes, y por ejemplo un proceso uso un poquito de una, tendremos mucho Espacio SobreAsignado



Por lo tanto, **NO ES UNA SOLUCION FIABLE !!!!!**

A este problema se le llama **FRAGMENTACION INTERNA!**

```
Importante saberse:
1. Fragmentacion Externa: Espacios libres NO CONTIGUOS (sol:
2. Fragmentacion Interna: Espacios Sobreasignados!
```

Si utilizamos **Paginas muy grandes** → Tendremos Fragmentacion Interna

Si utlizamos **Paginas muy pequeñas →** Cada vez que tenga que cargar un proceso, tengo que cargar muchas traducciones en la TLB, y cargar mas paginas : Mas costoso! NEcesito un tamaño ideal que combine ambos mundos

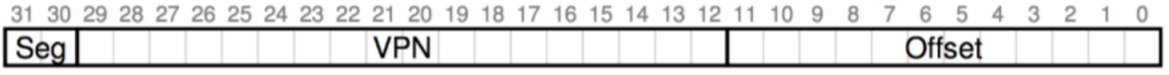
Que solucione la fragmentacion interna, y que a la vez no tenga que cargar tantas paginas en memoria cuando quiero cargar un proceso

Necesito encontontrar un Trade Off que tenga un equilibrio

En LINUX: Se usan paginas de 8KB (en alpha AXP) y 4KB (en sistema intel x86)

ESTRATEGIAS PARA SOLUCIONAR EL TAMAÑO DE TABLA DE PAG

Segmentos Paginados:



Seg:

- Limite y base de segmento
- LA direccion virtual (primeros dos bits: Indican 4 segmentos posibles)

Dentro de cada segmento → Paginas

- (VPN)

La idea es: Cada **SEGMENTO** tiene su propia **TABLA DE PAGINAS!**

Los segmentos en teoria, tienen un tamaño fijo a diferencia del heap, por lo tanto no va a usar todas las paginas que tiene asociada a el!!

ESPACIO VIRUTAL: 16 PAGINAS

- Un Proceso utiliza (EFECTIVAMENTE): 4 Paginas
- 12 entradas vacias (75%) en tabla de paginas



Con esto en la tabla de pagina, bastaría con poner menos (solo las utilizadas)

Con eso → Reduzco la gran cantidad de Entradas en paginas

OTRA ESTRATEGIA: **PAGINACION MULTI-NIVEL**

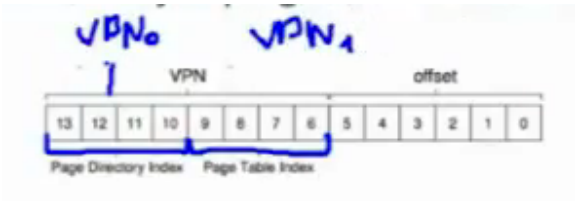
Es lo mismo que la estrategia anterior → Pero mas generica!

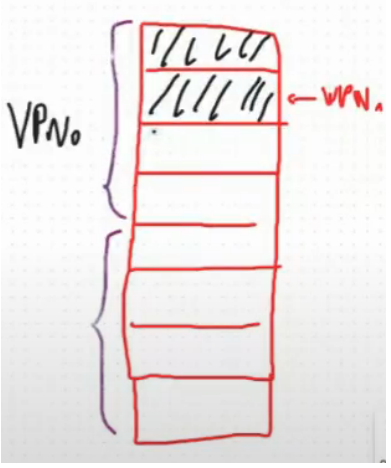
Esta es la que utilizan los sitemas modernos

En vez de segmentar, PAGINAR la tabla de paginas

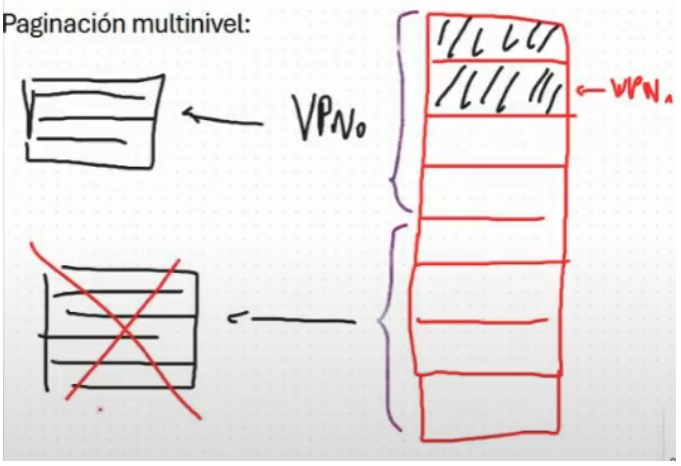
A grandes Rasgos:

1. Toma el VPN y lo divide en la cantidad de niveles que necesitemos
2. Si queremos 2 niveles, dividimos en 2 el vpn (no necesaraimente tiene que ser igual cantidad de bits las mitades)





Por cada seccion grande, tendre una tabla de paginas, luego si hay alguna seccion que no este utilizando, no tengo porque tener guardada dentro de la memoria su tabla → (ahorro espacio)

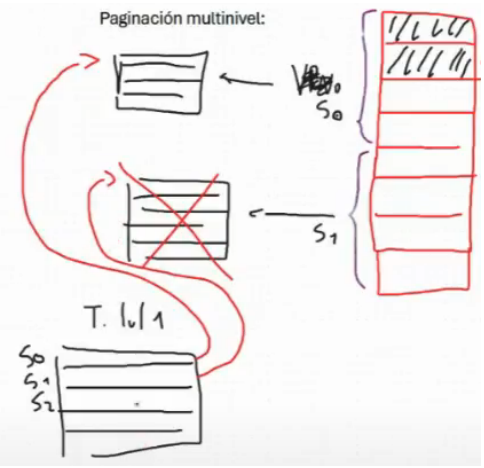


EJEMPLO

Dirección Virtual de 14 bit, paginas de 64 B

1. Esquemas de 1 Nivel: 2**14 / 2**6 = 2**8 paginas posibles → 256 PTE´s
2. Esquemas de 2 Nivel: Como vimos en el ejemplo de arriba (de los 14 bit: 8 son de vpn0, vpn1)
    - 4 bits para vpn0: 16 PTE en tabla Principal
    - 4 bits para vpn1: 16 PTE en la tablas de 2o Nivel

Graficamente se ve como que una tabla (principal), lleva a una seccion (1er nivel), y dentro de ese nivel, la 2da tabla (del vpn1), nos indica la direccion dentro de la 2da

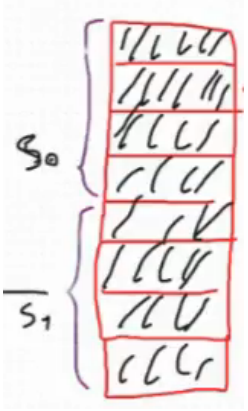


Ahora bien, el **AHORRO, solo** es significativo, cuando **no estoy usando toda la memoria**,

- si por ejemplo tuviera que usar toda

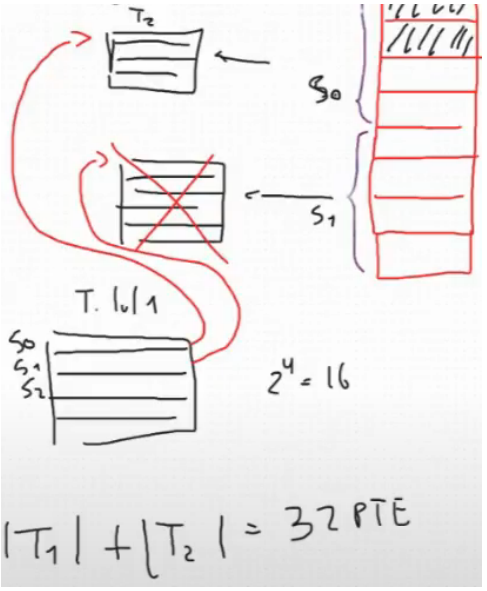
  → No tendria ahorro alguno

- Es **dificil** que un proceso utilice **TODA su memoria virtual**
- A la larga, usaría mas por tener mas tablas

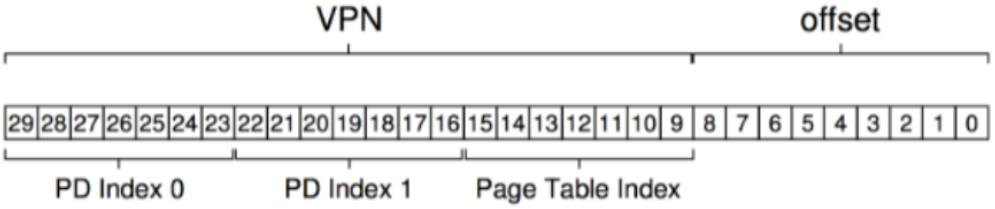


Sin embargo, si hay tablas que no estoy contando, estoy ahorrando espacio!!

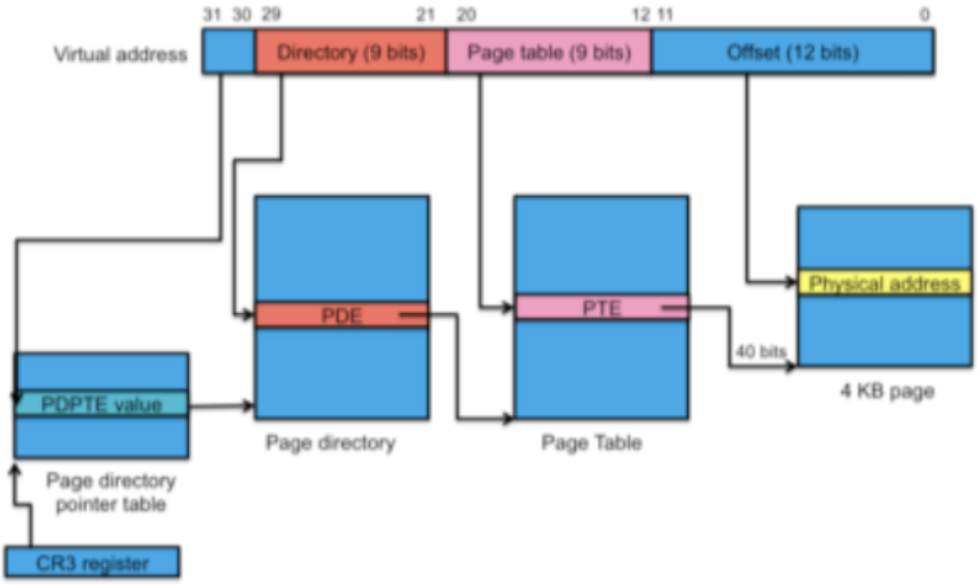
Redujimos los 256 PTES de usar todo el nivel cargado (tabla) → **A 32 PTEs**



A 3 NIVELES



Se vería algo asi:



En memoria, se necesita la tabla de paginas del **1er NIVEL**

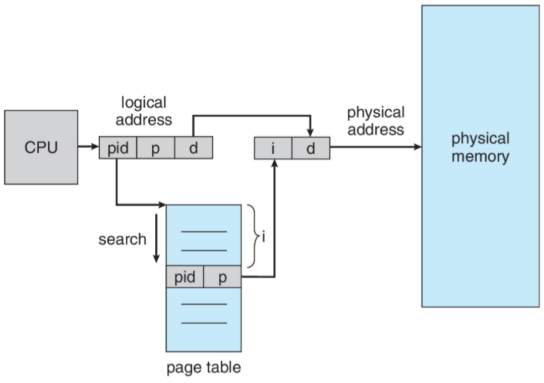
Y una tabla de cada Nivel para poder Resolver una Direccion Fisica

PAGE TABLE: La tabla de ultimo nivel guarda las traducciones a (VPN → PFN)

LA otras tablas: Guardan Punteros a Tablas

PAGINACION INVERTIDA

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





Una unica tabla de pagiunas indexada por frames

→ Busqueda es mas costosa!

- Lo mas importante de aprender de memoria talvez, sea **PAGINACION, no todo!**

Resumen

Vimos las estrategias para usar la memoria de manera eficiente

1. Segmentacion
2. Paginacion
3. Segmento Paginado
4. Paginacion Multi-nivel
5. Paginacion Inversa

Ahora veremos **REEMPLAZO DE PAGINAS**

Suponemos que la suma de todos los espacios virtuales caben en la memoria fisica → La realidad no es así

Habiamos visto que cierto espacio de disco, la ibamos a usar como “memoria principal” fake (SWAP SPACE)

Usamos el Disco (grande y lento) entonces, para proveer la ilusion que todo espacio virtual está en memoria

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

lo que nos queda por hacer, es llevar y traer paginas.

SWAP SPACE

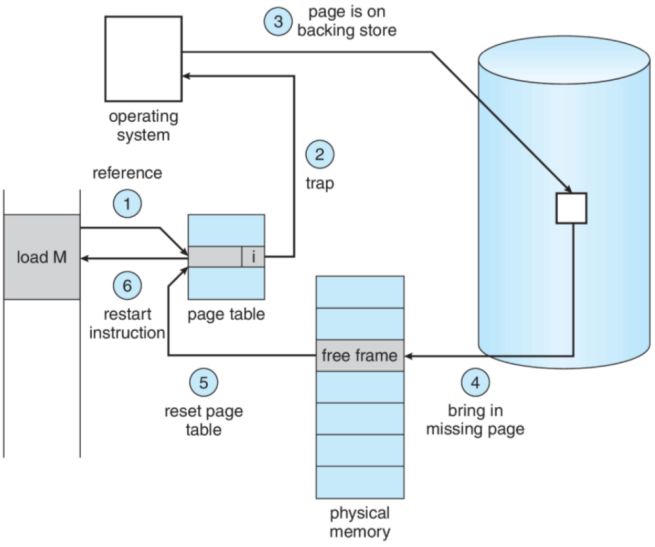
Paginas borradas de memoria principal, van a espacio de SWAP

- Representan parte del espacio de memoria de un proceso en ejecución
- Tenemos un metadato de bit de presente (**PRESENT BIT**) → Para saber si la pagina esta en **FRAME o no (mem fisica)**

- 0 si solo tengo una copia en el espacio SWAP, y no esta cargada en la memoria principal: esto generá la **INTERRUPCION**:
    - **Page Fault**: Activa el mecanismo de recuperación de una pagina desde disco a memoria fisica

Ante el Page Fault → El SO va a buscar la pagina

Mecanismo de rutina page-fault handler



1. Estoy intentando cargar una direccion de memoria
2. Me voy a la tabla de pagina → Pero me dice que el bit de presencia esta en 0
3. Genera una interrupcion (Page Fault)
4. Le avisa al sistema operativo
5. El OS tendra que ejecutar una rutina, para tener que hacer un intercambio de pagina
6. El sistema operativo va a buscar la pagina
7. La cargamos en el espacio libre (free frame)
8. Reseteamos la info en la tabla paginas
9. Re ejecutar la instruccion que genero la interrupcion

¿Que pagina Reemplazar?

Ahora bien, necesitamos encontrar algun ALGORITMO que permita cada vez que **no tengamos espacio libre**, sacar de la memoria principal aquella pagina que sea la mejor opción (aquella que no vaya a utilizar en un futuro cercano)

recordar : La memoria principal tiene las paginas a utilizar, las que no se utilicen tanto, mejor dejarlas en swap space, asi simulamos tener mas espacio del que realmente te

Clase 9

11/09/24

Dentro de un espacio del Disco Duro (Swap Space) se van a guardar paginas, para simular que tenemos mas espacio del que realmente tenemos en la memoria principal

Esto es lento, y queremos que la mayoria de pag se encuentren en la mem principal

Y aquellos que no son tan utilizados (o estan en espera), se ecnuentren dentro del **espacio swap**

Para ello necesitamos ciertos algoritmos que reemplazen aquellas paginas al momento de necesitar espacio libre en la memoria principal, saquen aquellas que no vayamos a utilizar!!

En esta clase se verán los algoritmos que permiten hacer **REEMPLAZO DE PAGINA**

Clase 10

30/09/24

ARCHIVOS

S.O nos abstrae del hardware

Implementa capas de abstraccion

La primera → Device Driver

La segunda → Generic Block Layer

La tercera → File System

la Cuarta → Application

El sistema operativo nos brinda un **SISTEMA DE ARCHIVOS**

- Este nos permite ejecutar funciones
  - open
  - read
  - write
  - close, etc

Como nos comunicamos con dispositivos I/O

4 Tipos de forma de transferir datos entre dispositivos (I/O)

1. Lectura Directa → Comunicacion Unbuffered → Bloquea la cpu hasta que termine de leer (**LENTO**)
2. Buffer en el espacio de USUARIO → Esto permite **CEDER en la CPU (pagina puede ser swapped out)**, → Al terminar tendremos en el buffer la memoria
3. Buffer en el espacio **KERNELL →** Las paginas **Nunca seran Swapeadas**, siempre estaran disponibles para leer
4. Double-Buffer → Version para escribir en varios buffers, ej: pantallas → Generan imagenes

Lo que nos interesa realmente de esta materia → MEMORIA **SECUNDARIA**

Memoira Secundaria

1. Almacenamiento de **Alta capacidad**
2. Memoria **No volatil**
3. Mas lento (y barato) que la memoria principal

1. DISCO DURO

DISCO DURO: Esta armado a traves de brazos,

- tiene que ejecutar movimientos para trabajar
  - ROTAR
  - Esto implica tiempos de demora distintos
- Latencias:
  - Latencia de Rotacion
  - Seek Time (mover el brazo lector (para leer los discos))
  - Hay ciertas formas
  - Defragmentar → Posicionar memoria util, para evitar tener que hacer tantos movimientos al leer la información

2. DISCO ESTADO SOLIDO (SSD)

Mejor que el Disco Duro → No tiene brazos

- Dado que eliminamos la **latencia de rotacion y Seek Time**
- Son mas **rapidos**
- Mas **costosos** por MB
- **Menos consumo** electrico
- CELDAS LIMITADAS de reescrituras

- Usan celdas para descargar informacion

FORMATEO

Dejar la memoria secundaria en un formato listo para hacer algo

Podemos hacer formateos a distintos niveles

1. FORMATEO BAJO NIVEL

- Formateo de fabrica : ESTRUCTURA : header + data + trailer

2. PARTICIONAMIENTO

Formateo que permite tener varias particiones

Un espacio para administrar las particiones, y luego las particiones, donde se pueden hacer unidades independientes por ejemplo, instalar un S.O alternativo, etc

Sistemas antiguos utilizan **MBR (master boot record)**

Sistemas modernos usan **GPT (Guide partition table)**

→ Permite tener una cantidad mayor de particiones (sigue siendo limitada)

FORMATEO LOGICO

El que se encarga de hacer el **sistema de archivos**

Divido memoria en bloques, y a cada uno le doy una interpretacion

Por ejemplo, un bloque es un directorio, otro es un archivo, etc

Schedulling de Accesos

Para minimizar los movimientos a la hora de buscar informacion en el disco

El S.O solicita datos del disco especificando:

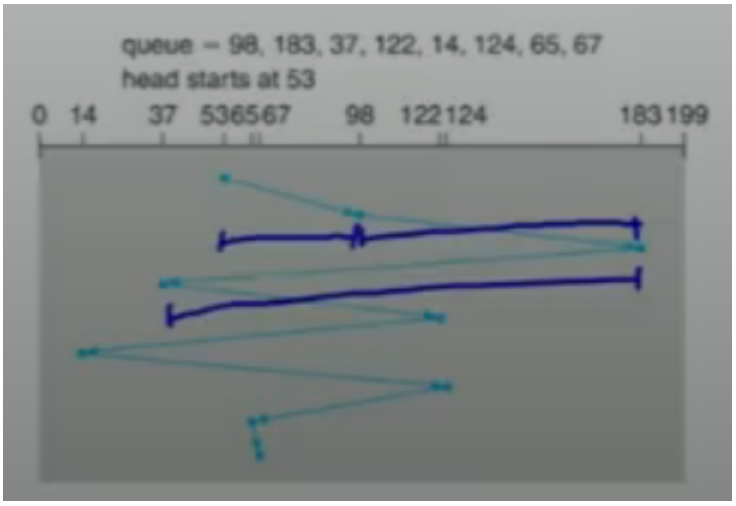
- Lectura? o Escritura?
- Direccion del Disco → **CHS o LBA**
- Direccion de memoria fisica (**Buffer**)
- Numeros de secotres a transferir
    - El orden de atencion → Determina el **seek time** de la transferencia

Objetivo: Minimizar la latencia de escritura (movimientos en un disco duro)

1. **SCHEDULING FCFS (FIFO)**

Primero en llegar, primero en ser atendido

Justo, pero no necesariamente el de mejor servicio



Este es el movimiento del brazo

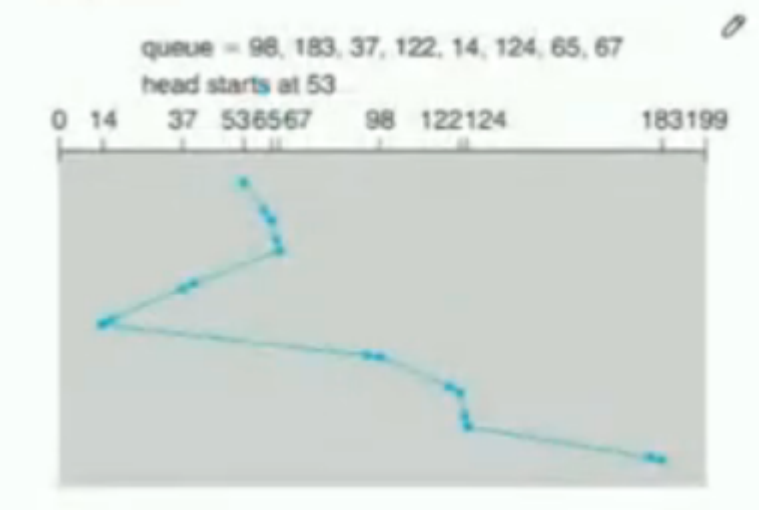
Si contamos en total las distancias que recorrio:

- Desplazamiento de 640

FCFS dependemos mucho del orden de acceso → No podemos asumir nada

2. **SCHEDULING SSTF : (**Shortest Seek Time First)

Busca moverse lo menos posible → Busca en minimos locales los mas cercanos



- Desplazamiento de 208 (mucho menos)

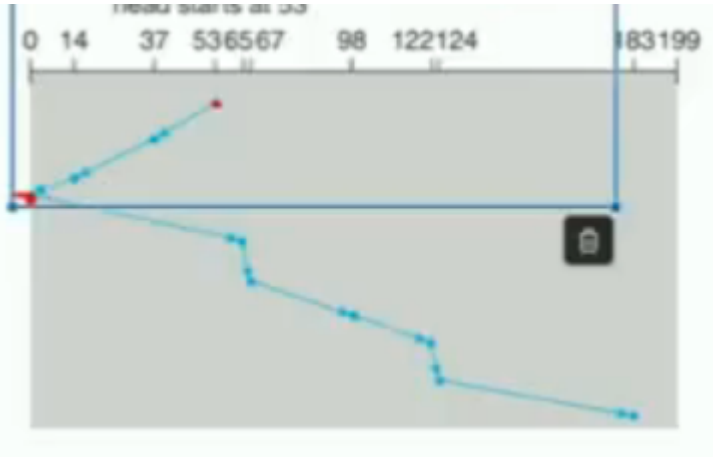
No es optimo → **Posible inanicion**

😭 Mientras estamos leyendo, pueden llegar peticiones cercanas, y provoca inanicion en los lejanos!

3. **SCHEDULING SCAN**

Elevador → Se va a ir moviendo completamente hacia un lado, y luego hacia el otro

Lee todo lo que encuentre en el camino



Tiempo de respuesta menos variado que SSTF

El brazo lector no se desgasta tanto → No tiene que cambiar de direccion

Pero es ineficiente igualmente, probablemente mas que los anteriores

4. C-SCAN: Lo mismo pero va hasta la orilla y lee en una direccion (no lee entremedio, se dirige hasta la esquina y de ahi empieza a leer hacia adelante)

5. LOOK y C-LOOK → Es SCAN pero sin llegar a los extremos

ARCHIVOS

Los archivos son la unidad minima de almacenamiento para el usuario

Simplemente es una agrupacion de bytes (tiene que estar continua o linkeadas (los frames))

Con el formato, sabemos como interpretar un archivo → El s.o sabra con que abrir el archivo

Tienen datos:

- Nombre → Low level (para el S.O)
- Nombre → Simbolico, leible para el humano “hellowoeld.py”
- tipo
- ubicacion

- tamaño
- proteccion
- etc

DIRECTORIOS

Contenedores de archivos o otro directorios (de manera de alto nivel (para usuarios))

- Puede contener:
  - Lista de pares (nombre leible, low-level nombre)
  - Otros directiors : Jerarqui o arboles de directorios
    - Jerarquia empieza con : **(la raiz): "/"**
    - Usa separador /
    - Unidades son Subdirectorios

Rutas absolutas : Partencon desde la raiz (con /) → **UNICAS**

v/s Rutas relativas: Parten desde el directorio donde nos encontremos

Clase 11

2/10/24

Directorios en bajo nivel:

Tablas que tengan punteros a otros directorios y/o otros archivos

API de sistema de archivos

- Creacion: Open
- Lectura/Escritura: read, write

- Lectura/Escritura (no secuencial): lseek
- escritura inmediata : fsync
- renaming mv
- eliminar rm

DIRECTORIOS

Creacion: mkdir

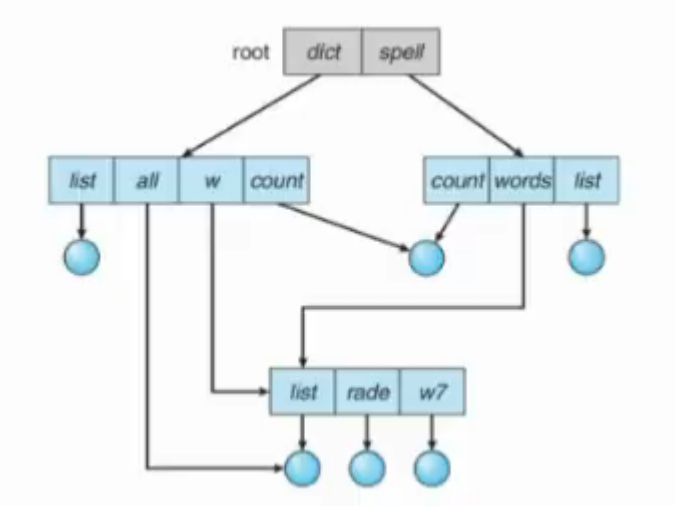
Lectura: opendir, readdir, closedir

eliminacion rmdir

LINKS

Referencias a los procesos/archivos

Existen 2



Hardlinks:

Apuntan directo al (Inode) dato en el disco, apunta al archivo real!

Softlinks:

Guarda una **referencia con el path**

En el dibujo: **HARD**link es por ejemplo el puntero de count al nodo (**DIRECTO**)

SOFTlink seria por ejemplo el puntero que w apunta a otra ruta (**PATH**), pero no directamnete al nodo

- Si borro el hardlink, **no necesariamente borro el archivo**,

  → Este se borra cuando se borran todos sus **hardlinks**

  Por lo tanto el nodo, deberia tener un contador de cuantos lo estan apuntando

- Cuando **borro** un archivo original o lo **muevo** → **EL SOFTLINK SE INVALIDA!**

MONTURA SISTEMA DE ARCHIVOS

Incorporar un dispositivo nuevo a nuestro sistema archivos, EJ: Montar memoria USB

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

En Windows → NTFS

En un USB podriamos usar FAT32

En Linux → ext4

VFS (virtual file system) → Lo veremos mas adelante!

Cuando yo monto un volumen nuevo, se hace en la carpeta **MNT**

El S.O Debe tener una tabla, para saber que sistema de archivos se esta usando en cada directorio



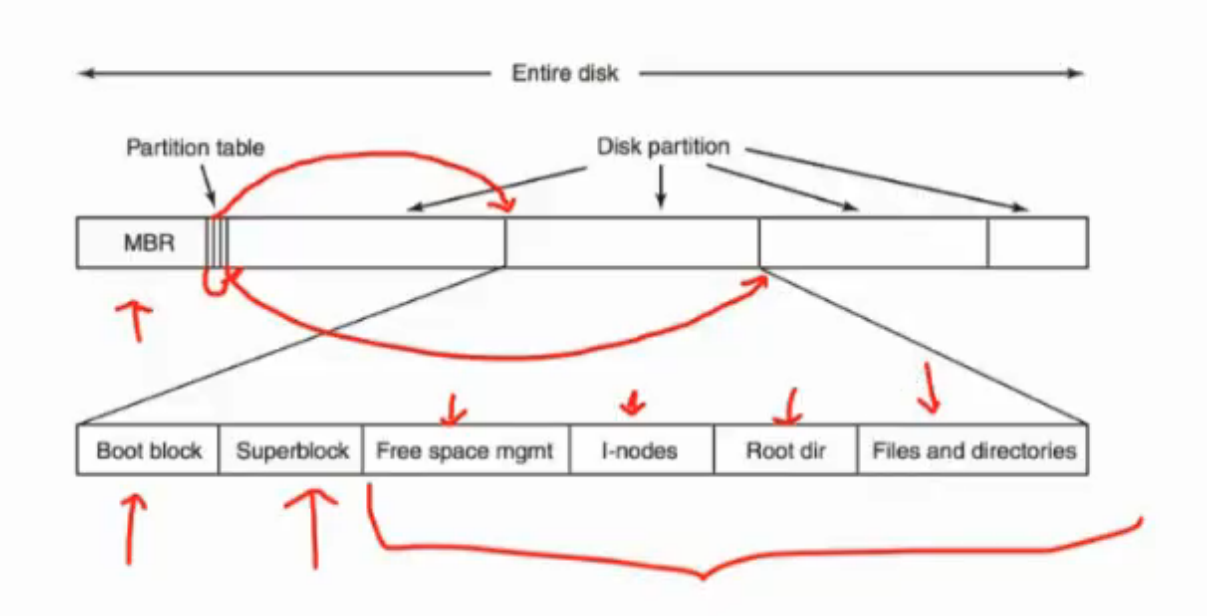
- El rojo podria ser NTFS y azul FAT32

TABLA DE PARTICIONES

Al inicio tenemos MBR (master boot record), indica partition Table

Boot control block: Permite cargar el sistema operativo

Volume Control Block: Datos de formato del sistema de archivos



Virtual File System, es una interfaz comun para distintos sistemas de archivos

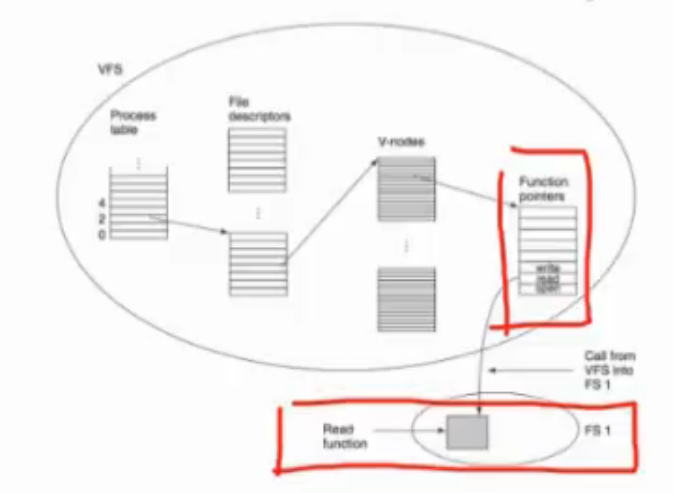
Llama a sistema de archivos especifico, que utiliza cierto directorio / archivo

Esto hace archivos tengan compatibilidad entre sistemas operativos

VFS:

Tiene una **Tabla de archivos de procesos**,

- esta apunta a una **Tabla Global de archivos abiertos**
  - Tiene una entrada unica por cada archivo abierto
  - Cada entrada apunta a un virtual node **V-Node**
    - Cada V-Node apunta a una tabla de funciones (funciones que utiliza el archivo en especifico)



BLOQUES

Bloque de directorio

- File Control Block (bloque que contiene info para poder leer datos de un archivo)
  - Read (index)
  - Voy a consultar tabla de archivos por procesos
  - Luego voy a la tabla de archivos del sistema
  - Luego voy a consultar a v-nodes para saber que funcion read (segun sistema de archivos) necesito utilizar
    - Leer Bloques de datos
    - Bytes que son en si el archivo

- Bloques indice
- Bloques Indireccion (Simple, Doble)

Los archivos no son solo blouqes de datos

Hay un tamaño real, y un tamaño que ocupa en disco

Necesito almacenar metadatos, datos extras

PAra leer un archivo, tengo que cargar y cargar bloques

La idea es no tener que andar dando tantas vueltas → Cada S.O Tendra una estrategia Distinta

ESTRATEGIAS

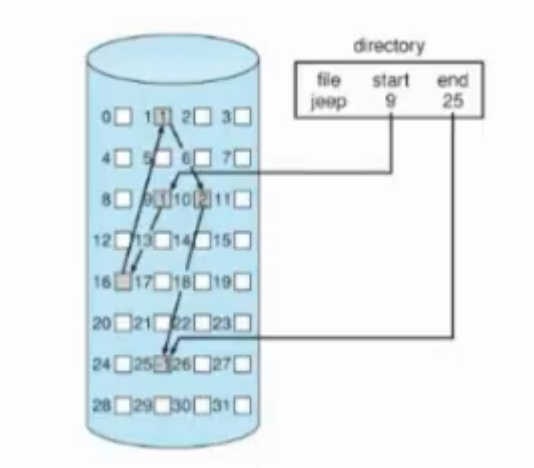
1. **ASGINACION CONTIGUA**

- que los archivos guarden sus bloque de datos de manera contigua
- Es bien sencillo de almacenar y leer
- - Problema → No se pueden guardar archivos pequeños → Compactar el archivo
    - La compactacion es costoso → queremos evitarlo

2. **ASIGNACION ENLAZADA (LINKED)**

Almacena **punteros a bloque siguiente**

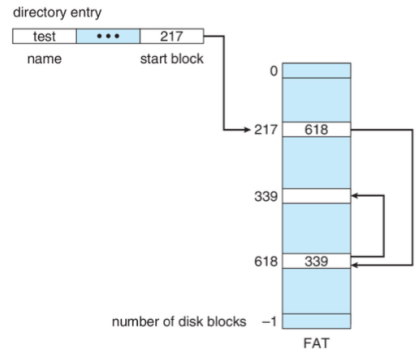
- Una parte de bloque es util para el dato (del archivo)
- Y una pequeña parte apunta al siguiente bloque



- - Problema O(n): Si quiero leer algo al final del archivo, al ser lista ligada, tengo que recorrer todos los bloques anteriores!!!
- Solucion: Lista Doblemente enlazada, pero igual para busquedas entremedios el mismo problema

3. ASIGNACION LINKED FAT

- **FAT**: File allocation table
- Cada entrada de la tabla me apunta hacia otro bloque
- No tengo la referencia dentro del bloque de datos → Lo tengo dentro de una tabla
- Ahora solo me dirijo a la tabla para buscar cierto puntero, asociar un bloque con otro

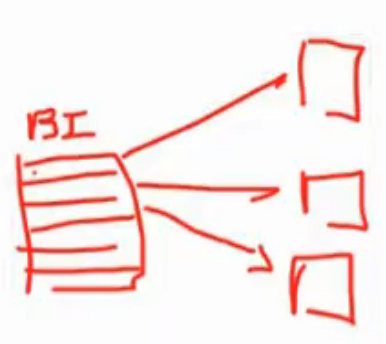


- - Problema : **Limitado por tamaño de tabla** → Para FAT32 : 4GB por archivo

direcciones 4 bytes → 32 bits : 2**32 = 4GB

4. ASIGNACION INDEXADA

- Punteros (bloques de indices) hacia **Bloque de datos**
- Cada archivo presenta su propio bloque de indices (se puede interpretar como una tabla)
- Cada bloque de indice → Apunta a un **BLOQUE DE DATOS**



Se pierde espacio por tamaño de index block

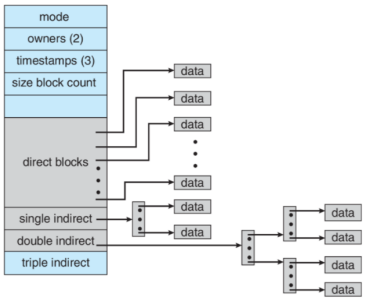
Puede haber sobreasignacion dado que los bloques tienen tamaños constantes → y si un archivo por ejemplo pesa menos de 4kb cae dentro de un bloque de indice, sobran n-1

- ESQUEMA ENLZADO: Ultimo bloque de indice, apunta a la siguiente tabla de bloques de indices → No muy util → Mejor tener :

INDICE MUTINIVEL!!!!

5. ASGINACION COMBINADA (BLOQUES DE INDIRECCION)

Bloques indirectos → Apuntas a bloques de indices → Apuntan a bloques de datos





Ej: Direccion simple → Como se ve en la figura : Bloque indireccion simple → Bloque indice → Bloque de datos

Tamaño posible: 1024*1024*32KB = 4 TB !!!

Memoria libre

Como encontramos bloque libres → BITMAP de bloques libres

- Sencillo, eficiente, implementable en hardware
- - Problema: No escala tan bien

Otra Estrategia, es que cada bloque libre, tiene un puntero al siguiente bloque libre

- Poco eficiente
- Mejora: Agrupar espacio contiguos