Materia I1 Sistemas Operativos

Clase 3 / Miércoles 13 de marzo:[1]

Multiprogramación: mantener múltiples programas en memoria
Multitasking: cambiar rápidamente entre procesos, asignando un tiempo a cada uno

Comandos shell:

- Para ejecutar un archivo: *./file.c*

```
gcc -o cpu cpu.c… -l pthread
```

- gcc: compilador de C
- -o cpu: archivo ejecutable
- cpu.c: archivos de entrada (código fuente)
- -l pthread: busca librería de biblioteca para threads. Busca en el sistema operativo libpthread.sl

Sistemas operativos:

Es el conjunto de kernel y herramientas. Usan un **bit de modo** (kernel bit) para determinar si están en modo kernel o en modo user.

- Roles: administrador de recursos, protector servidor
- Tareas: Administración de procesos, manejo de memoria, sistemas de archivos, protección - seguridad y control de dispositivos I/O.
- Arquitecturas: monolítico, microkernel y híbrido.

Monolítico: (mucho espacio en el user space). Como todos los servicios están en el kernel space, una falla afecta a todo el kernel. Su ejecución es más rápida. Es difícil de extender pero puede usar módulos

Microkernel: La idea es mantener el kernel del menor tamaño posible. La línea de user space baja (poco en el mundo del kernel). Los drives se conectan directamente con el IPC. Si un servicio falla, este problema queda aislado. Es fácil de extender.

Kernel híbrido: Son las arquitecturas actuales. Combina lo mejor de cada uno. Es construido como monolítico pero con módulos.

[1] Todos los códigos que el profe muestra están en el servidor en la ruta /home/cruz/

Clase 4 - 5 / Miércoles 20 de marzo:

Procesos

Es la abstracción de un programa en ejecución. Se componen en la siguiente estructura:

- **Código**: (.text, información estática), se carga en modo *read only.*
- **Datos**: (.data): variables globales. Puede almacenar punteros a valores dentro del heap.
- **Stack**: cada ítem representa un llamado a la función *call frame,* contiene parámetros, variables globales y PC
- **Heap**: memoria asignada dinámicamente

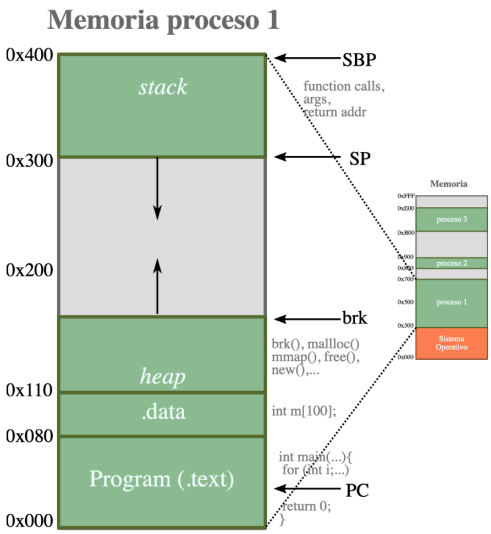
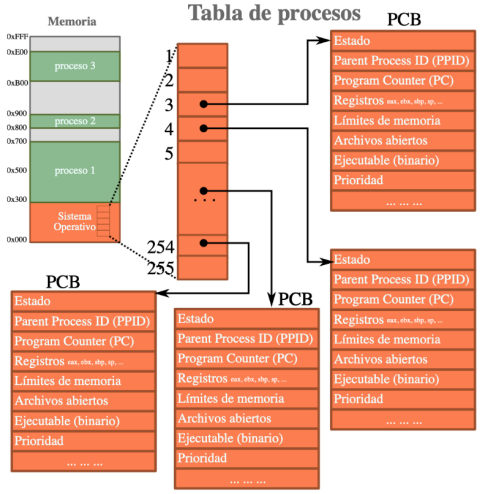


Tabla de procesos

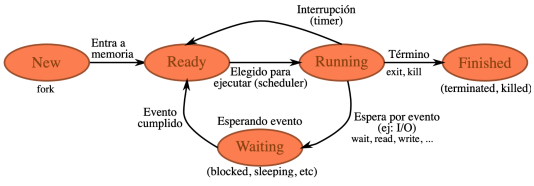
El sistema operativo tiene una tabla con la información de cada proceso. Cada uno guarda la información necesaria en su **process control block (PCB).** En esta se guarda:

- Estado
- PID
- 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, …



Los procesos tienen 5 estados:

- **New** : En creación
- **Running** : En ejecución
- **Waiting** : Esperando (I/O, signal)
- **Ready** : Listo para ejecutar. Esperando asignación de CPU
- **Terminated**: Ejecución terminada



El sistema operativo puede sacar un proceso en ejecución para empezar o continuar ejecutando otro. Esta tarea se llama context switch, y se realiza con interrupciones por parte del SO según el scheduling que esté usando.

El primer proceso se crea al iniciar el kernel. Este proceso raíz tiene PID = 1. Todos los demás procesos que se creen serán descendientes de este.

Syscalls

- **Fork:** crea otro proceso como copia del creador. Ver ejemplo de `fork3.c` en la carpeta en /home/cruz/. La línea de instrucciones se traspasa en su estado actual a los hijos del proceso activo. Retorna PID del hijo del padre, y retorna 0 al hijo. Este último comienza su ejecución en el valor de retorno del llamado a *fork.*

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.

- **Exec():** carga un binario en memoria reemplazando el código de quién lo llamó. El nuevo programa se "roba" el proceso (la memoria se sobreescribe).

- **Wait():** espera a que termine un proceso hijo. Si no se especifica el PID, va a retornar cuando termine el siguiente proceso hijo. `Wait(&valor)`, cuando el proceso hijo termine, el código de salida va a quedar almacenado en la variable `valor`.

- **Exit():** termina proceso actual y devuelve toda la memoria al SO. Como parámetro acepta un código de salida, que es 0 si salió todo bien.

- **Kill():** para indicarle a otro proceso que debe terminar. Existen varias señales:

- **Sleep():** duerme al proceso

Para manejar los procesos **huérfanos**, existe la opción de pasarle un parámetro en el *fork,* para que cuando le llegue una señal *SIGTERM*, le haga un forward a todos sus hijos. En caso que queden procesos huérfanos, estos pasan a ser hijos de init. Que está en wait().

Los **zombies** es cuando el proceso hijo termina y el padre no ha terminado o no ha hecho *await.* Una vez que el padre lo hace entonces termina el proceso. En caso que el padre nunca lo haga, una vez que este termine, el proceso zombie pasará a ser hijo de init. Este a su vez realiza una llamada *wait* cada cierto tiempo para eliminar procesos zombies. Correr el programa "waitzombies" y en otra ventana ejecutar hdoc? (para ver los procesos)

Clase 6 / Miércoles 27 de marzo:

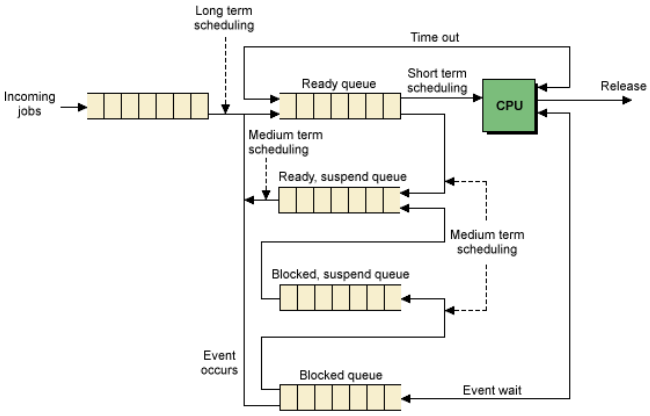
Scheduling

- **Scheduling prioritario:** es como las colas de banco, donde existen cajas especiales para clientes
- **Scheduling FIFO:** colas de supermercado puede ser multiqueue.

El scheduler es el encargado de esta área. Es como el planificador del computador.

Existen distintos niveles de *scheduling:*

- **Long-term Scheduler:** …
- **Short-term Scheduler:** …
- **Medium-term Scheduler:** swapping,

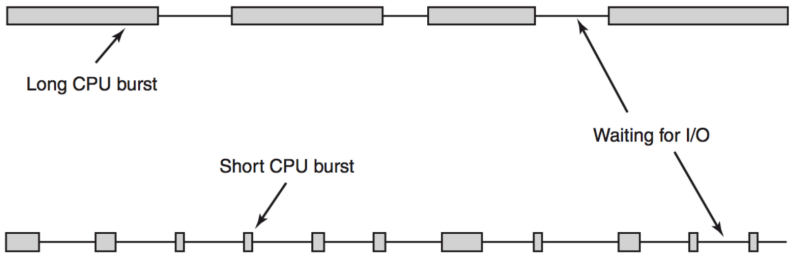


Swapping: sacar un proceso de la RAM (swap-out) y ponerlo en el disco. Sirve para liberar memoria cuando el sistema operativo necesite más. Cuando se quiera volver a usar ese proceso se trae de vuelta a la RAM (swap-in)

El caso ideal es pasar la mayor parte del tiempo en *usertime* (tiempo que estoy en la CPU), y menor tiempo posible en *systime* (tiempo que ejecuta el SO)

Los procesos tienen dos fases:

- **CPU-burst**: uso de CPU
- **I/O-burst**: espera por I/O



En la imagen vemos como un proceso usa mas CPU, mientras que el otro depende del input del usuario.

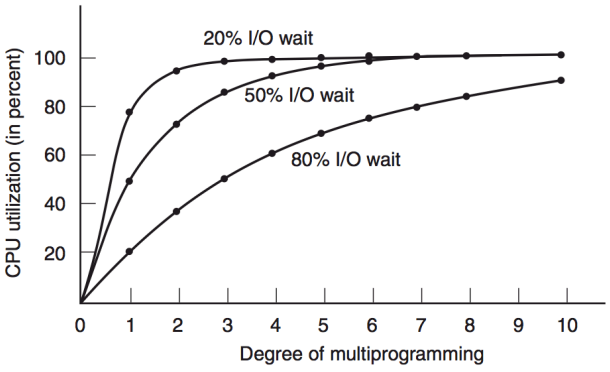
Uso de la CPU

Una CPU bajo un 100% esta sub-utilizada.

p: porcentaje de tiempo en espera por I/O

p^n: probabilidad que n procesos estén esperando por I/O.

CPU use: $CPU_n = 1 - p_n$



Tipos de scheduling

Existen distintos niveles de *scheduling* en el sistema operativo:

- **Long-term:** Admite procesos en cola *ready,* determina el grado de **multiprogramación.** Determina si es posible crear nuevos procesos. Al usar *fork* puede rechazar el llamado si queda poca memoria.
- **Medium-term:** modifica **temporalmente** el grado de **multiprogramación**. Hace el **swapping.**
- **Short-term:** a.k.a *dispatcher*. **Selecciona un proceso** de la cola *ready* para ejecutar. Ejecuta **cambio** de **contexto**.

1. **Por tipo de interrupción**
   - **Preemptive:** (expropiativo). Utiliza interrupciones (timer) para decidir cuándo sacar a un proceso de ejecución.
   - **Non-preemptive:** (no expropiativo). Permite que un proceso ejecute hasta que deja voluntariamente la CPU, ó se bloquea en I/O, ó el proceso termina.

2. **Por objetivo**
   - **Batch:** trabajo por lotes, sin interacción. Se usa para inventarios, cálculo de nóminas, etc… Generalmente son no apropiativas, o de lo contrario, tienen un *quantum* grande.
     - Minimizar *turnaround time:* tiempo desde envío hasta término
     - Maximizar throughput: número de trabajos por hora
   - **Interactive:** minimizar tiempos de respuesta, satisfacer usuarios.
   - **Real time:** diseñado para alcanzar deadlines. Su tiempo de respuesta debe ser predecible. Un ejemplo es cuando un auto autónomo debe tomar una decisión antes de llegar al semáforo.

3. **Real**
   - Alcanzar *deadlines*. Tiempo de respuesta debe ser predecible.

Algoritmos de scheduling

Batch:

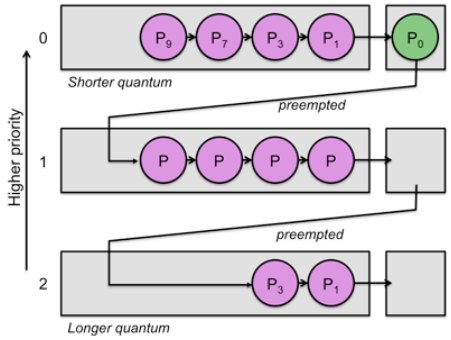
- **First Come, First Serve (FCFS)**
- **Shortest Job First:** Caso particular de priority. Tiene bajo *turnaround* time.

Interactive:

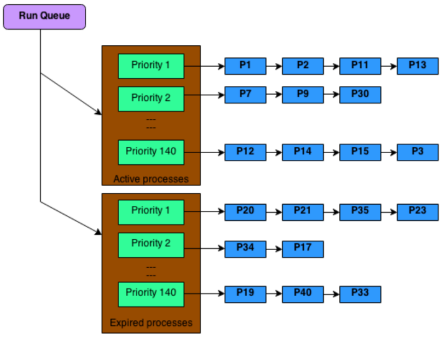
- **Round Robin (RR):** un turno (quantum) para cada uno. Buen *response time*.
- **Priority scheduling:** cada proceso tiene asociada una prioridad.
  - Prioridades iguales: FCFS o RR
  - Prioridades pueden ser estáticas o dinámicas
  - Starvation 💀
  - Aging 👴: incrementar prioridad de procesos que llevan más tiempo

- **Multilevel feedback queue (MLFQ):** optimiza *turnaround,* pero minimizando el *response time*. Tiene múltiples colas con distintas prioridades.

1. Si priority $(A) >$ priority (B), ejecutar A
2. Si priority $(A) =$ priority (B), ejecutar A y B con RR
3. Procesos entran a la cola con **mayor** prioridad
4. Si un proceso usa su *q* (acumulado en todos sus turnos), su prioridad se reduce.
5. Después de un tiempo S, todos los procesos se mueven a la cola de mayor prioridad.



- **Big O of 1 scheduler O(1):** usado en los kernel modernos de Linux. Tiene complejidad $O(1)$, mientras que las versiones anteriores eran de $O(n)$. Tiene dos colas para cada nivel de prioridad, una de procesos activos y otra de inactivos.



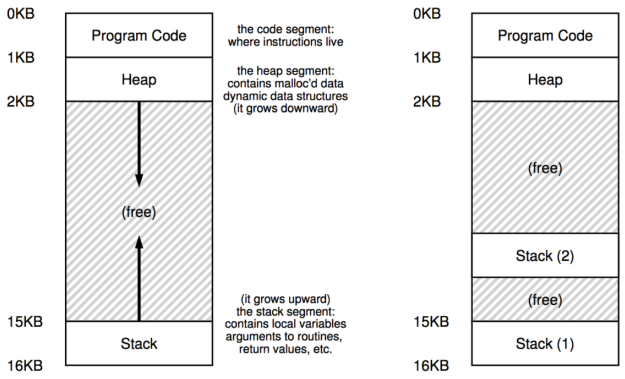
- **Completely Fair Scheduler:** intenta ser justo con todos los procesos. Tome el tiempo de cuanto ha estado cada uno en CPU, y los con menor tiempo son llevado a CPU. Usa como estructura árboles rojo-negro. Hasta antes todos usaban listas ligadas. Los procesos con menor tiempo ejecutado son puestos a la izquierda del árbol. Para elegir un proceso, se escoge el de más a la izquierda. Operaciones son $O(log n)$, pero se cachea el dato de más a la izquierda para un rápido acceso.

- **Brain Fuck Scheduler (BFS):** usa una doble lista ligada que es tratada como cola. Para insertar un proceso es $O(1)$, y para buscar el siguiente proceso a ejecutar es en el peor caso $O(n)$.

Clase 7 / Lunes 8 de abril:

Threads

Nos permite hacer cosas como: distintas partes de un código se ejecutan concurrentemente y distribuir tareas entre múltiples núcleos. Estos comparten variables globales, código, heap de memoria, archivos abiertos, procesos hijo, alarmas pendientes, etc… No tienen relación de paternidad.



En la imagen vemos como se van creando múltiples stacks.

- **Paralelo:** ocurre al mismo tiempo
- **Concurrente**: tareas están en cola *ready* al mismo tiempo.

Son como procesos pero livianos. Tienen su propio tid, PC, registros y stack. El resto lo comparten con los otros threads del mismo proceso.

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

Volatile: para que evitar optimizaciones del sistema sobre esa variable. Queremos hacerlo cuando la variable puede ser modificada “por fuera”.

Clase 8 / Miércoles 10 de abril:

Hay dos opciones para implementar threads.

1. **User space threads (N-1):** Implementados y manejado por una biblioteca en el lado del usuario. Algunas implementaciones usan variantes no-bloqueantes de *syscalls.* Cada proceso tiene una thread table.
   - 👍 Son portables
   - 👍 Context switch y scheduling más rápido
   - 👍 Mejor escalabilidad
   - 👎 Llamadas a *syscall* pueden pueden bloquear proceso (hay sólo 1 kernel thread asociado)
   - 👎 Requiere *scheduling* cooperativo

2. **Kernel space threads (1-1):** Implementados y manejado por estructuras de kernel, no usan bibliotecas. Existe una thread table común para todos los procesos. Kernel usa **thread pools,** reutilizando estructuras.
   - 👍 Sin bibliotecas, soporte nativo
   - 👍 Syscalls no bloquean el proceso.
   - 👎 Syscalls más costosas. Requiere llamadas al kernel
   - 👎 Escalabilidad limitada por el S.O
   - 👎 Semántica de fork() no es clara
   - 👎 Semántica de signals no es clara

3. **Hybrid implementations (M-N):** Kernel threads **multiplexados** entre users threads. El kernel sólo ve kernel threads. Hay varios users threads asignados a un kernel thread.
   Cada proceso tiene una tabla de threads, y cada thread de kernel apunta aun proceso (varios pueden apuntar a un mismo proceso). La thread user table se va llenando, y a medida que existan threads disponibles en el kernel, estos van tomando los procesos de la tabla de user threads y los van ejecutando.
   - 👍 Más livianos de crear
   - 👍 Syscalls no bloquean el proceso
   - 👍 Escalable como los users threads.

No me quedó clara la implementación de los hybrid threads… Hay algún video o lectura explicativa?

Los POSIX Threads no son de una implementación específica, si no que dependen del sistema operativo. Linux por ejemplo usa sólo kernel threads.

Sincronización

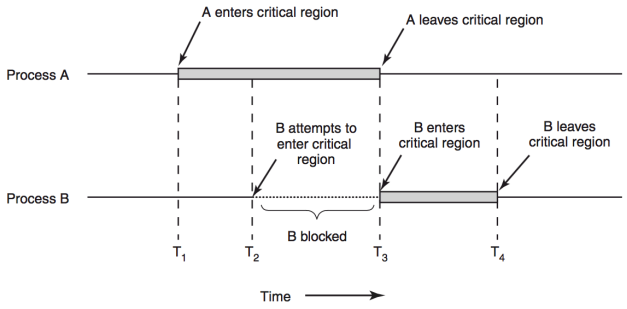
Cuando corro threads, puede que ocurran desincronizaciones. Un ejemplo es el programa pthreads2 del servidor. El counter es sumado por los dos threads de manera separada. Pero si el thread fue sacado de ejecución justo antes de guardar el nuevo valor de counter, entonces cuando vuelva a ejecución va a sobreescribir el valor que escribió el otro thread.

```
mov     0x60208c,%eax ; eax = 10
                                            mov     0x60208c,%eax ; eax = 10
                                            add     $0x1,%eax     ; eax = 11
                                            mov     %eax,0x60208c ; counter = 11, Write lost!!
add     $0x1,%eax     ; eax = 11
mov     %eax,0x60208c ; counter = 11   Dirty write :/
```

Race condition: situación en que la salida de una operación depende del orden temporal de sus operaciones internas, el cual no está bajo control del programador.

Sección crítica (sc): cuando un thread accede a recursos compartidos. Necesitamos que un sólo thread ejecute acciones sobre esta sección de recursos. Se debe cumplir:

- **Exclusión mutua:** a lo más un thread en su SC.
- **Progreso:** Al menos un thread puede entrar a su SC.
- **Espera acotada:** Si un proceso quiere ejecutar su SC, después de un tiempo finito podrá hacerlo.



Soluciones:

1. Variable *lock* compartida: Es ineficiente, cumple progreso sólo **progreso.** Los demás no por que la lectura y escritura de *lock* no son atómicas.

2. Variable *turn* compartida: Alterna entre dos threads. Cumple **exclusión mutua**. Pero los oros dos no, ya que si un thread quiere entrar, pero el thread que le corresponde no, entonces se debe esperar a que este último ejecute su SC.

3. Solución de Peterson: Cumple con las condiciones de sección crítica.

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 .../ */
}
```

Esperar usando *while(lock);* es ineficiente, ya que esta constantemente ejecutando ciclos en la CPU. Pthread ofrece una versión de ***spin lock***. Esta es más eficiente que usar un *lock* normal siempre y cuando el tiempo de espera del thread sea muy pequeño, ya que:

- No requiere verificar el estado de los demás threads
- Cuando no obtiene el lock, el thread no se va a dormir, por lo que hay menos latencia para volver a adquirirlo
- No hay problemas de cache.

El problema de **inversión de prioridad** entre dos procesos A y B que utilizan busy waiting ocurre cuando A tiene mayor prioridad que B, y B posee un recurso que A necesita, pero B no puede liberarlo porque A está en estado ready y su prioridad no permite que B ejecute. Para eliminarlo podemos usar priority inheritance que permite que un proceso con menor prioridad pueda aumentar su prioridad temporalmente para poder liberar el recurso que tiene asignado.

Semáforos: permite un número limitado de threads ejecutar una sección crítica. El clásico ejemplo es el de productor - consumidor. Incluyen un contador S y dos operaciones:

- **wait P():** pedir (intenta decrementar el valor). Es como pedir el lock.
- **release V():** votar (incrementa el valor). Es como soltar el lock.

- Ayuda a sincronizar procesos
- Se pueden usar para implementar exclusión mutua entre procesos
- Puede usarse para que un proceso notifique a otro cuando se libere un recurso.

Pueden ser implementados usando *busy waiting* o con bloqueo de threads. Son usados para *signaling*, mientras que los mutex son para proteger recursos compartidos.

Es la imagen se ve un semáforo para solucionar el problema de múltiples consumidores y múltiples productores. Notar que el semáforo y mutex funcionan de manera distinta. Este último tienen el **concepto de propiedad**, donde sólo el productor / consumidor que disminuyó su valor puede volver a subirlo.

```
mutex buffer_mutex; // similar to "semaphore
semaphore fillCount = 0;
semaphore emptyCount = BUFFER_SIZE;

procedure producer()
{
    while (true)
    {
        item = produceItem();
        down(emptyCount);
        down(buffer_mutex);
        putItemIntoBuffer(item);
        up(buffer_mutex);
        up(fillCount);
    }
}

procedure consumer()
{
    while (true)
    {
        down(fillCount);
        down(buffer_mutex);
        item = removeItemFromBuffer();
        up(buffer_mutex);
        up(emptyCount);
        consumeItem(item);
    }
}
```

Monitor: se implementan usando un **mutex** y **condition variables**. Pueden crearse a partir de semáforos, y viceversa.

Clase Lunes 22 de abril:

Direccionamiento y memoria virtual

Los procesos comparten algunas regiones de memoria con el sistema operativo, de esta manera se pueden usar y compartir bibliotecas dinámicas y evitar copiarlas para cada uno de ellos.

Direccionamiento absoluto: no usar memoria virtual, si no que acceder directamente a la dirección física de la variable. Tiene dos problemas: relocalización y protección. El primero se puede solucionar con relocalización de variables, que es acceder manualmente a la dirección física, pero sumando la dirección de inicio del proceso. De esta manera se pueden cargar múltiples procesos y acceder correctamente a las variables de cada uno.

Una mejora fue la incorporación de los registros *base y limit*. Estos registros se cargan con la información correspondiente de cada proceso, el registro *base* permite redireccionar al conocer la dirección física en la que comienza le proceso. El registro *limit* fue creado como método de protección para que un proceso no acceda a memoria que no le corresponde.

La MMU es una pieza de hardware que es la encargada de traducir la memoria virtual a memoria física.

Fragmentación externa: quedan espacios de memoria sin usar.

Compactación: fusionar los huecos. Copia memoria a otras direcciones. Es caro y por lo tanto no se hace. Para evitar espacios libres podemos usar distintas estrategias: first-fit, best-fin y worst-fit. Ninguna de estas nos aseguran que no exista fragmentación, ya que dependen como sean los procesos.

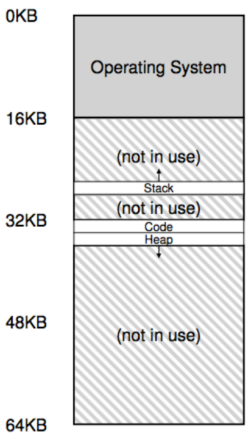
Segmentación: se divide el espacio en varios espacios más pequeños, de manera que pueda ser asignados más fácilmente. El proceso ya no está en un espacio de memoria contiguo, si no que particionado. Ayuda a disminuir la fragmentación, pero no la elimina completamente.

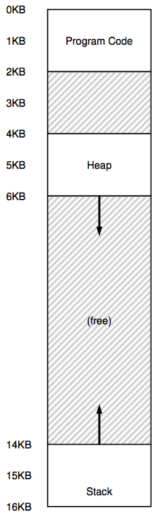
La cantidad de segmentos posibles está determinada por los primeros bits, que indican a que segmento de memoria corresponde. El resto de los bits de la dirección corresponden al *offset* dentro del segmento.

Otro campo en la tabla de segmentos

Segment	Base	Size	Upwards
`code`	32768	2048	1
`heap`	34816	2048	1
`stack`	28672	2048	0

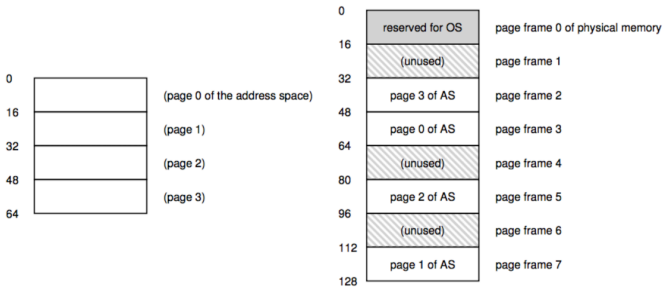
- Dirección virtual (lógica): 16300
  - Segmento `stack` (segmento 3)
  - *Offset*: complemento de 4096 (4KB, máximo offset). $4012 - 4096 = -84$
  - Base + *offset* $= 28672 - 84 = 28588$
- Dirección física: 28588 ✔





Paginación: dividir el espacio de direcciones en segmentos que sean del mismo tamaño, llamados páginas. Permite eliminar fragmentación externa, pero produce fragmentación interna.

- Memoria física: dividida en **frames** (marcos) del mismo tamaño. Deben ser también del mismo tamaño que las páginas virtuales.
- Memoria virtual: dividida en **pages** (páginas)



Se usa una **tabla de páginas** para llevar registro de páginas de cada proceso, y para mapear de memoria virtual a física. Cada proceso tiene su propia tabla de páginas.

El offset de la memoria virtual y física deben ser iguales (porte de pages y frames es igual).

- Memoria usada por tabla de páginas: #pages * (#bits para expresar frame).
- Cantidad de PTE: 2 ^ (#bits virtual - #bits offset)
- Tamaño direccionamiento proceso: 2 ^ (#bits virtual)

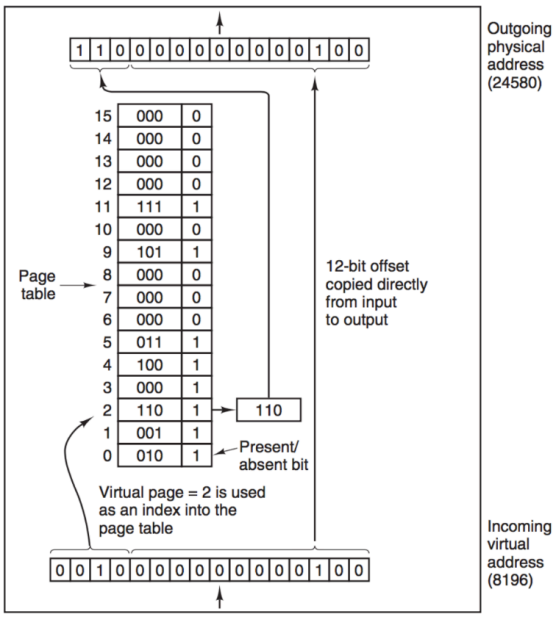
Ejemplo traducción:

1. Páginas de 16B (2^4), espacio virtual 64B (2^6), espacio físico 128B (2^7).
   Dirección virtual: 21(0b010101):
   - #Página: 1(0b01)
   - Offset: 5(0b0101)
   - Tabla: 1→7

2. Páginas de 4KB (2^{12}), dirección virtual de 16 bit (2^{16}), dirección física de 15 bit (2^{15}).

   - Tamaño de espacio virtual: $2^{16}B = 65536B = 64KB$
   - Tamaño de espacio físico: $2^{15}B = 32768B = 32KB$
   - Tamaño de página: $4KB = 2^2 \times 2^{10}B = 2^{12}B$
   - Tamaño de página: $12 (rango: 0x0000 a x FFF)$
   - Bits para #página: 4 (16 páginas)
   - Bits para #frame: 3 (8 frames)

   Dirección lógica: 8196 (0x2004)
   - #Página: 2
   - #Frame: 6
   Dirección física: 24580 (0x6004)



Ignoramos el "present/absent" bit, … por ahora

Clase Miércoles 24 de abril:

Translation look-aside buffer (TLB): memoria cache que ayuda a acelerar acceso a memoria. Guarda la traducción de memorias virtuales a memorias físicas, de esta manera no es necesario leer la page table.

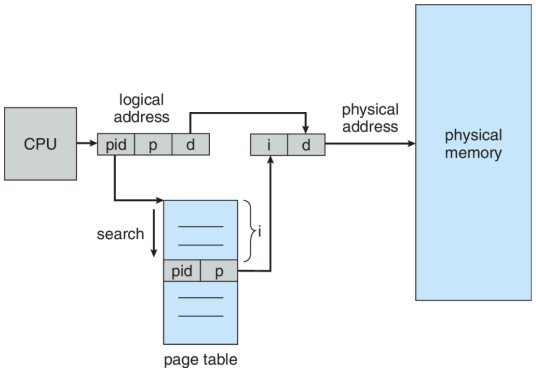
Si las páginas son muy grandes podemos tener **fragmentación interna**, que significa tener espacios sobreasignados.

Variaciones de paginación:

- **Segmentos paginados**: es tener una tabla de páginas por segmento. La tabla de páginas es más pequeña, pero vuelve la fragmentación externa.

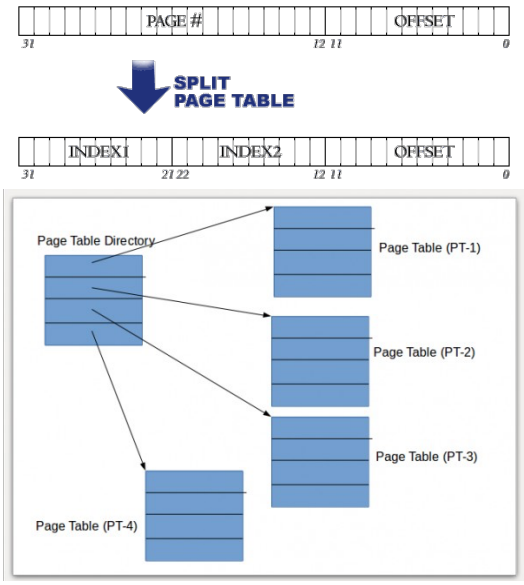
- **Tabla de páginas invertida:** Es una tabla para todo el sistema (no una por proceso). Cada página es accedida como |Pid,Page|

  Número de entradas: physical address space / page size



- **Paginación multinivel**: la idea es paginar la tabla de páginas. Pueden tener distintos tamaños.

  La idea es que el tamaño de cada tabla queda de un tamaño razonable, y que cabe en memoria. Si una page table no está usada, se puede marcar como usada en el page table directory y así no cargarla en memoria. De esta manera liberamos espacio relacionado a la paginación.



Clase Lunes 29 de abril:

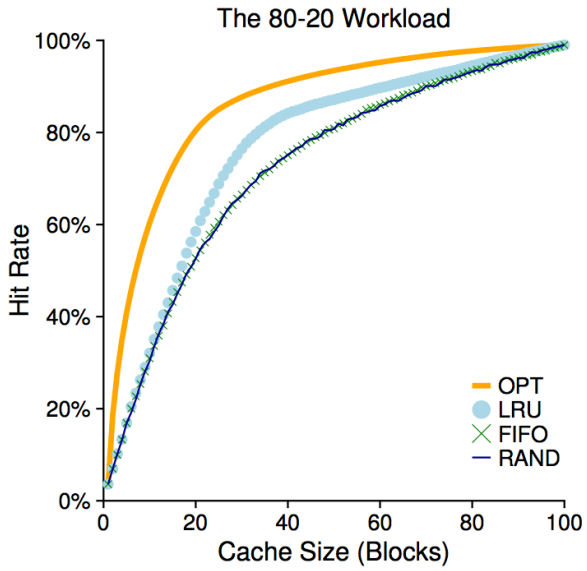
Reemplazo de páginas

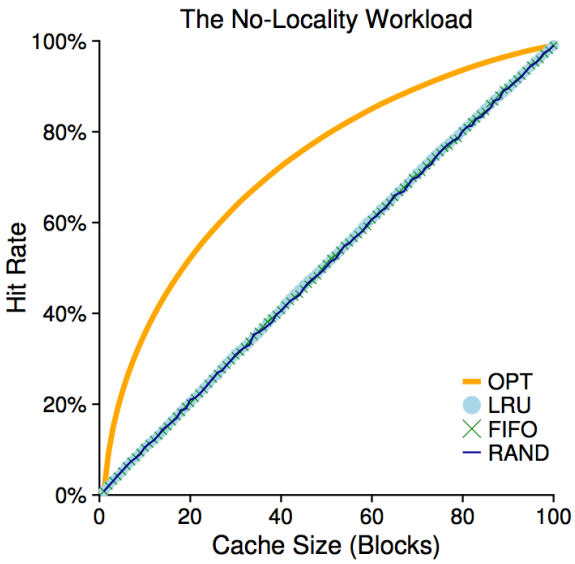
El desafío es usar efectivamente el disco (grande pero lento) para proveer la ilusión que todo el espacio virtual está en memoria. El SO se encarga de tener todas las páginas On Demand.

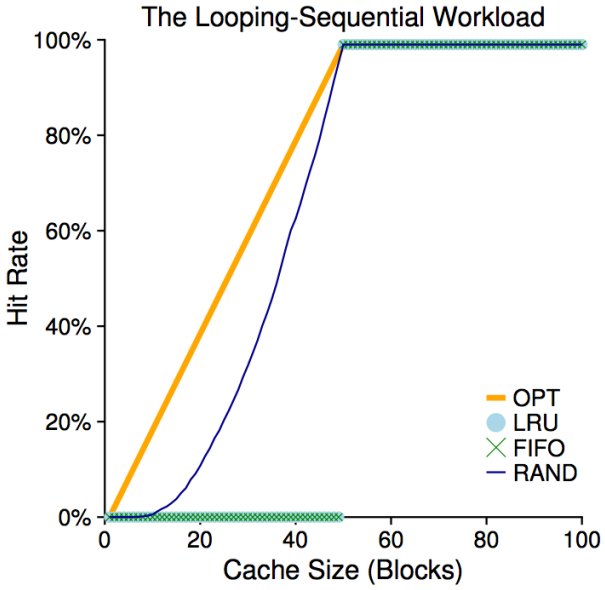
Páginas que van a ser borradas van al **swap space,** que representan parte del espacio de memoria de un proceso de ejecución. La tabla de páginas (TLB) usa un **present bit** para ver si la página buscada está cargada en memoria (en un frame). Cuando no se encuentra la página el sistema arroja un **page fault.**

Existen varias formas de decidir que página saco cuando realizo un **swap in:**

1. **Min:** reemplazo óptimo. Elijo la que será usada más tarde en un futuro. Es el mejor algoritmo pero no tenemos como conocer el futuro.
2. **FIFO**: Elegimos la página que lleva más tiempo en memoria.
3. **Random**
4. **LRU**: *least recently used*, la página que fue usada hace más tiempo.





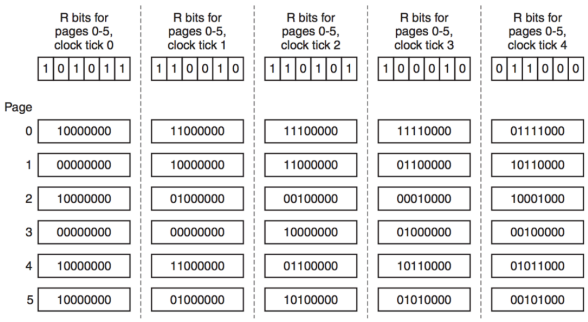


Implementando LRU: la estructura para este tipo se actualiza en cada archivo de memoria. En cambio en FIFO se actualiza sólo en cada miss. Podemos aproximarla con ayuda del hardware.

- **Algoritmo del reloj:** se apunta a una página, y al momento de elegir se mira el reference bit. Si es 1 se cambia a 0 y se mira la siguiente página. Si el bit es 0 entonces se elige esta página.

- **Algoritmo de aging:** tiene un enfoque de múltiples reference bits. La última página accedida tiene el número mayor. En cada clock los reference bits hacen un shift left.

- **Dirty bit:** 1 si la página ha sido modificada. Se pueden priorizar mejor las páginas para el algoritmo del reloj.



Page	R bits for pages 0-5, clock tick 0	R bits for pages 0-5, clock tick 1	R bits for pages 0-5, clock tick 2	R bits for pages 0-5, clock tick 3	R bits for pages 0-5, clock tick 4
	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00010000	10001000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000