Sistemas Operativos: Teoricas

Introduccion a los sistemas operativos

Que es un SO?

Una forma de dividir a los sistemas informaticos en hardware y software => requieren un intermediario

Maneja la contencion y concurrencia para lograr buen rendimiento y hacerlo correctamente

Elementos Basicos

- **Drivers:** Programas parte del SO y manejan los detalles de bajo nivel relacionados con la operacion de los distintos dispositivos.
- **Nucleo o kernel:** El SO propiamente dicho. Se encarga de las tareas fundamentales y contiene los subsistemas.
- **Interprete de comandos o shell:** Permite al usuario interactuar con el SO. Grafico o command line.
- **Proceso:** Programa en ejecucion + su espacio de memoria asociado
- **Archivo:** Secuencia de bits con un nombre y artributos que indican permisos
- **Directorio:** Coleccion de archivos y directorios que contiene un nombre y se organiza jerarquicamente
- **Dispositivo virtual:** Una abstraccion de un dispositivo fisico bajo la forma de un archivo
- **Sistema de archivos:** Forma de organizar datos en el disco
- **Directorios del sistema:** Directorios donde el propio SO guarda archivos que necesita para su funcionamiento
- **Binario del sistema:** Archivos que viven en los directorios del sistema. Proveen utilidades basicas
- **Archivo de configuracion:**
- **Usuario:** Para aislar informacion entre si y establecer limitaciones
- **Grupo:** Coleccion de usuarios

Procesos y API del SO

Procesos

Un programa compilado en codigo objeto y ejecutandose es un proceso. Cada uno tiene un PID (Process ID)

Desde la memoria el proceso esta compuesto por:

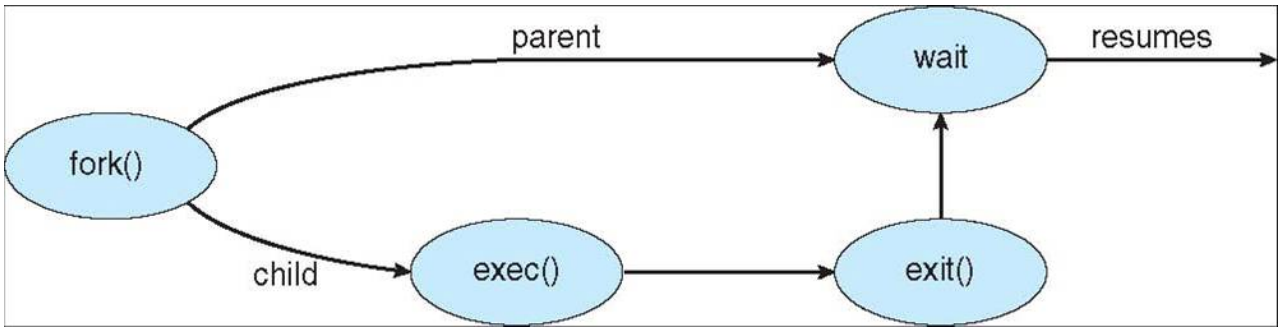
- *Texto:* Codigo
- *Datos:* Donde se almacena el heap
- *Stack*

Y puede:

- *Terminar:* exit(). Indica al SO que puede liberar sus recursos y indica status de terminacion.
- *Lanzar un hijo:* Los procesos se organizan como arbol. El primero se llama root o init
  - *fork():* crea un proceso igual al actual, devuelve el pid del hijo. El padre puede decidir suspenderse hasta que termine el hijo con *wait()*. Cuando termina el padre obtiene el codigo de status del hijo
  - system() = fork() + wait()
  - *exec():* El hijo hace algo distinto que el padre.

- *Ejecutar en CPU*
- *Hacer Syscall*
- *Hacer E/S*

Cuando lanzamos un programa desde el shell:



Scheduler

Decide a que proceso le corresponde ejecutar en cada momento.

Para cambiar de tarea hay que hacer un **cambio de contexto** y los datos necesarios para esto se guardan en el **PCB (Process Control Block)**

Politicas de scheduling

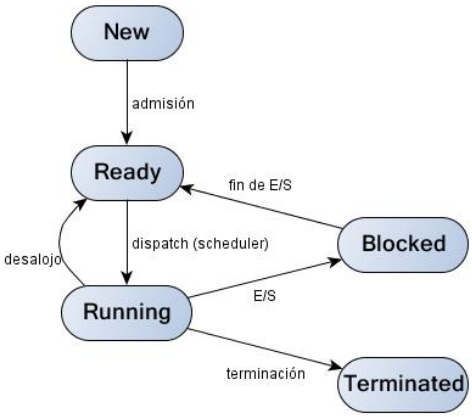
Que optimizar?

- Fairness
- Eficiencia
- Carga del sistema
- Tiempo de respuesta
- Latencia
- Tiempo de ejecucion
- Rendimiento
- Liberacion de recursos

Cuando actua el scheduler?

El scheduler decide si el proceso debe seguir o no si...

1. Se bloquea.
2. Cuando un proceso nuevo se carga o algun proceso pasa cierto tiempo en ejecución (quantum)
3. Cuando un proceso se desbloquea.
4. Cuando un proceso termina.

• *Sin desalojo:* Cuando las decisiones solo ocurren en 1. y 4. • *Con desalojo:* con el reloj. Requiere clock con interrupciones y no da garantias de contunuidad a los procesos. • Cooperativo: cuando el kernel toma control (syscalls).	 *Con desalojo*

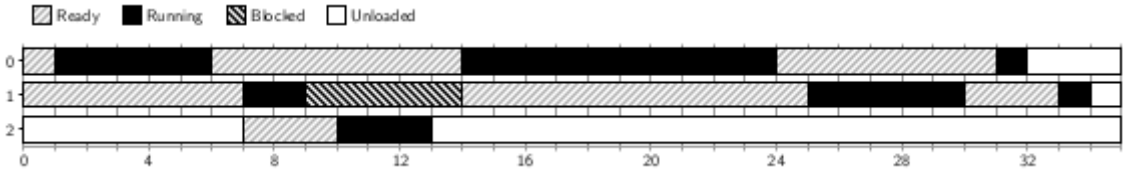
Como elijo que tarea hacer?

- *FCFS:* First Come First Serve. En orden sin interrumpir hasta que termine de ejecutarse.
- *SRTF:* Shortest Remaining Time First. (prioridades variables)
- *Sala de espera:* FIFO supone que todos los procesos son iguales => un proceso puede taponar
  => agregamos prioridades => puede generar *starvation*
  => aumento la prioridad a medida que los procesos envejecen (prioridades variables)
- *Round Robin:* Darle un **quantum** (tiempo fijo de ejecucion) a cada proceso e ir alternando entre ellos
- *Multiples colas:* La cola con mas prioridad es la que tiene menos quanta. Cuando a un proveso no le alcanza su cuota de CPU se pasa a la cola siguiente, disminuye su prioridad y y se le asigna mas tiempo de CPU en su proximo turno. Los procesos de maxima prioridad van a la cola de maxima prioridad.
- *Trabajo mas corto primero (SJF, Shortest Job First):* Intenta maximizar rendimiento. Sin desalojo
- *Scheduling para RT:* Se usan cuando hay deadlines estrictas. => EDF (Earliest Deadline First)
- *Scheduling en SMP:* Con el concepto de *afinidad al procesador* intentamos usar el mismo procesador para una tarea aunque tarde mas en liberarse para aprovechar la cache.
  Depende que tan estricto sea es *afinidad dura o afinidad blanda*

LEER DEL LIBRO

Diagrama de GANTT

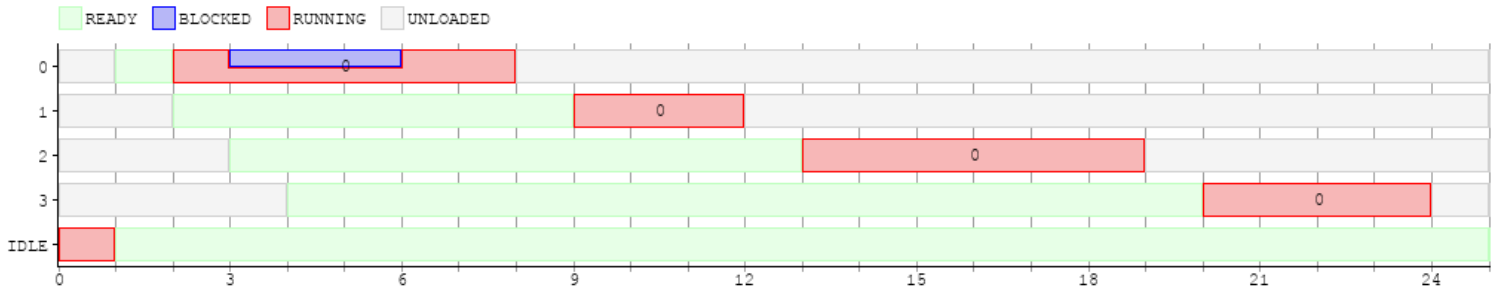
Muestra el estado de cada uno de los procesos existentes en un sistema durante un período de tiempo determinado.



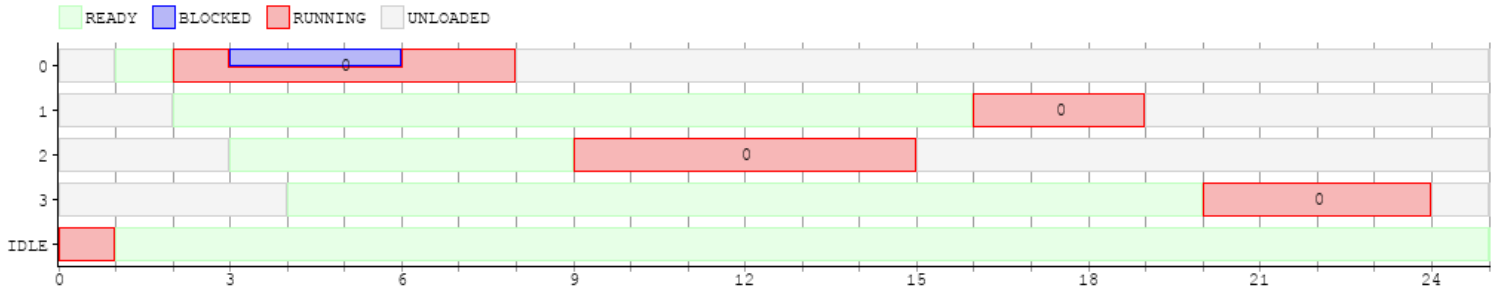
Procesos vs. Diagramas

Proceso	Tiempo de llegada	Tiempo de ejecución	Tiempo de bloqueo (incl.)	Prioridad
P0	1	6	2-4	3
P1	2	3	-	2
P2	3	6	-	1
P3	4	4	-	4

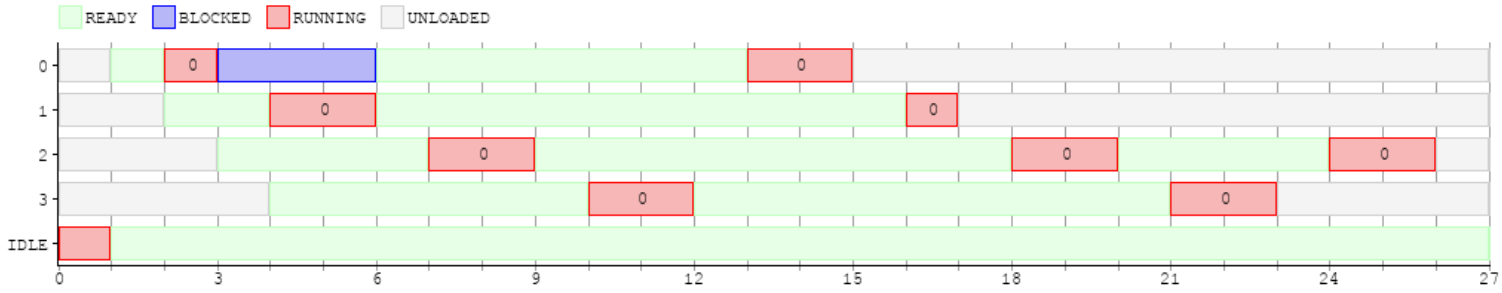
Cambio de contexto = 1
FCFS



Prioridades sin desalojo



Round Robin (q=2)



Evaluaciones de rendimiento

- *Fairness:* "Justicia" en la asignación del CPU.
- *Tiempo de* respuesta: Tiempo que el proceso tarda en empezar a ejecutarse.
- *Throughput:* Cantidad de procesos que terminan por unidad de tiempo.
- *Turnaround:* Tiempo total que le toma a un proceso ejecutar completamente.
- *Waiting time:* Tiempo que un proceso pasa en estado ready.

Sincronizacion

Tenemos dos problemas que generan la programacion paralela y estructuras compartidas: contencion y concurrencia.

Condicion de carrera: El resultado obtenido baria dependiendo en que momento u orden se ejecuten las cosas. Toda ejecución debería dar un resultado equivalente a alguna ejecución secuencial de los mismos procesos

Veamos posibles soluciones

- Secciones criticas (CRIT)
  1. Sólo hay un proceso a la vez en CRIT.
  2. Todo proceso que esté esperando entrar a CRIT va a entrar.
  3. Ningún proceso fuera de CRIT puede bloquear a otro.

  Implementaciones *Con dos llamados:* uno para salir, otro para entrar.
- *Suspender todas las interrupciones en la seccion critica*
- *Locks:* Variable booleanas que cuando entro en la seccion critica pongo en 1 y cuando termino en 0. Si esta en 1 espero que este en 0. Para que funcione bien me conviene usar TAS.
- *Test and set (TAS):* Se pone una variable en 1 y devuelve el valor anterior de manera atomica (indivisible). Para esperar que termine me conviene ponerla en un ***while (TestAndSet (& lock))*** para esperar e intentar obtener un lock y cuando salgo de ahi y termino el codigo critico ***lock = FALSE.*** Esto se llama **busy waiting** y consume mucha CPU.

- *Sleep:* Si es mucho, perdemos tiempo, si es poco desperdiciamos CPU. Tambien desde otro lado podemos llamar a *wakeup* para terminar con el sleep cuando ya no es necesario.
- *Semaforos:* Variable entera con las siguientes caracteristicas:
  - *sem(int value):* Se inicializa en cualquier valor
  - Tiene dos operaciones que se ejecutan sin interrupciones:
    - *wait(): while (s<=0) dormir(); s--;*
    - *signal(): s++; if (alguien espera por s) despertar a alguno;*
  - La version binaria se llama *mutex()*
  - Si me olvido un signal o invierto el orden, genero deadlock
- *Atomic bools:* tienen *getAndSet(bool b)* (guarda b y devuelve lo que habia en el registro) y *testAndSet()* (= a GAS pero con b = true)
- *Atomic ints:* tienen *getAndInc() / getAndDec(), getAndAdd(int) y compareAndSwap(int,int)* => no generan busy waiting (wait free)
- *Atomic queue:* tienen *enqueue(elem) y dequeue(*elem)*
- *Atomic<T> objeto:* tienen *get(), set(), compareAndSwap(T test, T value)*
- *TAS lock (Spin lock):* Tiene *create(), reg.lock()* y *reg.unlock()*. No se puede usar recursivamente (deadlock) y hace busy waiting.

  atomic < bool > reg ;

  void create () { reg . set (false); }

  void lock () { while (reg . TestAndSet ()) {} }

  void unlock () { reg . set (false); }
- *TTAS lock (local spinning):* Testear antes de TAS. c*reate(), lock() y unlock()*

void create () { mtx . set (false); } *void lock () {* *while (true) {* *while (mtx . get ()) {}* *if (! mtx . testAndSet ()) return ;* *}* *}* *void unlock () { mutex . set (false); }*	Es mas eficiente porque: • while hace *get()* en vez de *testAndSet()* • cache hit mientras true • Cuando *unlock()* hay cache miss

Tenemos el problema de garantizar exclusion mutua:

- Si la seccion critica es una funcion => menor concurrencia
- Si la seccion critica es un bloque => mayor concurrencia

Condiciones para la existencia de deadlock

- **Exclusión mutua:** Un recurso no puede estar asignado a más de un proceso.
- **Hold and wait:** Los procesos que ya tienen algún recurso pueden solicitar otro.
- **No preemption:** No hay mecanismo compulsivo para quitarle los recursos a un proceso.
- **Espera circular:** Tiene que haber un ciclo de $N \geq 2$ procesos, tal que P_i espera un recurso que tiene P_{i+1}.

Resumen de problemas de sincronizacion

- Problemas
  - Race condition
  - Deadlock
  - Starvation
- Prevención
  - Patrones de diseño
  - Reglas de programación
  - Prioridades
  - Protocolo (e.g., Priority Inheritance)
- Detección
  - Análisis de programas
    - Análisis estático
    - Análisis dinámico
  - En tiempo de ejecución
    - Preventivo (antes que ocurra)
    - Recuperación (deadlock recovery)

Razonamiento en paralelo

Correctitud de programas distribuidos

- Plantear propiedades de **safety:** cosas malas no suceden (por ejemplo, deadlock).
- Plantear propiedades de **progreso** (o **liveness**): en algún momento algo bueno sucede.
- Demostrar (o argumentar) que la combinación de esas propiedades implica el comportamiento deseado.

Tipos de Propiedades:

- *Contraejemplo:* Sucesión de pasos que muestra una ejecución del sistema que viola cierta propiedad.
- *Safety:* "nada malo sucede".
  - Ejemplos: exclusión mutua, ausencia de deadlock, no pérdida de mensajes, "los relojes nunca están más de δ unidades desincronizados".
- *Liveness:* "en algún momento algo bueno sí va a suceder"
  - Ejemplos: "si se presiona el botón de stop, el tren frena", "cada vez que se recibe un estímulo, el sistema responde", "siempre en el futuro el sistema avanza", no inanición.
- *Fairness:* "Los procesos reciben su turno con infinita frecuencia".
  - Incondicional: El proceso es ejecutado "regularmente" si está habilitado siempre.
  - Fuerte: El proceso es ejecutado "regularmente" si está habilitado con infinita frecuencia.
  - Débil: El proceso es ejecutado "regularmente" si está continuamente habilitado a partir de determinado momento.

"No se van a dar escenarios poco realistas donde alguien es postergado para siempre"

En general, fairness se suele suponer como válida para probar otras propiedades (liveness en general).

Propiedades

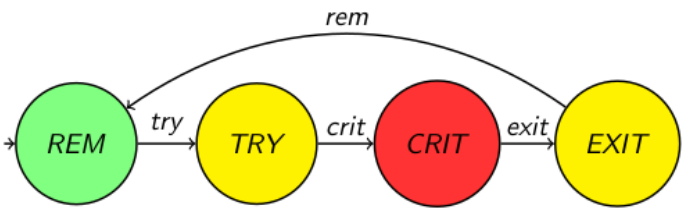
Propiedad barrera o rendezvous

Cada $P_i, i \in [0..N-1]$, tiene que ejecutar a(i); b(i).

Propiedad BARRERA a garantizar: b(j) se ejecuta después de todos los a(i)

Pero, no hay que restringir de más: no hay que imponer ningún orden entre los a(i) ni los b(i)

Modelo de proceso



- Estado:
  $\sigma : [0..N-1] \rightarrow \{REM, TRY, CRIT, EXIT\}$
- Transicion:
  $\sigma \rightarrow^{l} \sigma', l \in \{rem, try, crit, exit\}$
- Ejecucion:
  $\tau = \tau_0 \rightarrow^{l} \tau_1 \ldots$

Wait-Freedom

"Todo proceso que intenta acceder a la sección crítica, en algún momento lo logra, cada vez que lo intenta".

Esta propiedad se llama WAIT-FREEDOM.

$$\forall \tau, \forall k, \forall i . \tau_k(i) = TRY \Rightarrow \exists k' > k . \tau_{k'}(i) = CRIT$$

Intuición: "libre de procesos que esperan (para siempre)". Es una garantía muy fuerte.

Fairness

Para toda ejecución τ y todo proceso i,

si i puede hacer una transición l_i en una cantidad infinita de estados de τ

entonces existe un k tal que $\tau_k \rightarrow^{l_i} \tau_{k+1}$

Exclusion mutua

Para toda ejecución τ y estado τ_k, no puede haber más de **un** proceso i tal que $\tau_k(i) = CRIT$

$$EXCL \equiv cant(CRIT) \leq 1$$

Progreso del sistema

Para toda ejecución τ y estado τ_k,

si en τ_k hay un proceso i en TRY y ningún i' en CRIT

entonces $\exists j > k$, t. q. en el estado τ_j algún proceso i' está en CRIT .

$$LOCK-FREEDOM \equiv (cant(TRY) \leq 1 \wedge cant(CRIT) = 0 \Rightarrow cant(CRIT) > 0)$$

Predicados auxiliares:

- Lograr entrar: $IN(i) \equiv i \in TRY \Rightarrow i \in CRIT$
- Salir: $OUT(i) \equiv i \in CRIT \Rightarrow i \in REM$

Progreso global dependiente (deadlock-, lockout-, o starvation-freedom)

Para toda ejecución τ ,

si para todo estado τ_k y proceso i tal que $\tau_k(i) = CRIT$,

$\exists j > k$, tal que $\tau_j(i) = REM$

entonces para todo estado $\tau_{k'}$ y todo proceso i',

si $\tau_{k'}(i') = TRY$

entonces $\exists j' > k'$, tal que $\tau_{j'}(i') = CRIT$

$$STARVATION - FREEDOM \equiv \forall i . OUT(i) \Rightarrow \forall i . IN(i)$$

Progreso global absoluto

Para toda ejecución τ , estado τ_k y todo proceso i,

si $\tau_k(i) = TRY$

entonces $\exists j > k$, tal que $\tau_j(i) = CRIT$.

$$WAIT - FREEDOM \equiv \forall i . IN(i)$$

Livelock

Un conjunto de procesos está en livelock si estos continuamente cambian su estado en respuesta a los cambios de estado de los otros.

Seccion critica de M <= N

Propiedad a garantizar

$\forall \tau , \forall k$

1. $cant(i | \tau_k(i) = CRIT) \leq M$
2. $\forall i . \tau_k(i) = TRY \wedge cant(j | \tau_k(j) = CRIT) < M \Rightarrow \exists k' > k . \tau_{k'}(i) = CRIT$

SWMR (Single-Writer/Multiple-Readers)

Una variable compartida que los escritores necesitan acceso exclusivo pero los lectores pueden leer simultaneamente

Registros RW

Caracteristicas:

- Si read() y write() NO se solapan => read() devuelve el último valor escrito.
- Si read() y write() se solapan
  - “Safe”: read() devuelve cualquier valor
  - Regular: read() devuelve algun valor escrito.
  - Atomic : read() devuelve un valor consistente con una serialización.

Ejemplos de algoritmos EXCL con registros RW

	Dijkstra	*Panaderia de lamport*
Registros	• flag[i]: atomic single-writer / multi-reader • turn: atomic multi-writer / multi-reader	• choosing[i], number[i]: atomic single-writer / multi-reader

Proceso i	/* TRY */
	L : flag [i] = 1;
	while (turn 6 = i)
	if (flag [turn] == 0) turn = i ;
	flag [i] = 2;
	foreach j 6 = i
	if (flag [j] == 2) goto L ;
	/* CRIT */
	...
	/* EXIT */
	flag [i] = 0;
		/* TRY */
		choosing [i] = 1;
		number [i] = 1 + max j6 = i number [j];
		choosing [i] = 0;
		foreach j 6 = i {
		waitfor choosing [j]==0;
		waitfor number [j]==0 ‖
		(number [i] , i) < (number [j] , j);
		}
		/* CRIT */
		...
		/* EXIT */
		number [i] = 0;

Caract.	• Garantiza EXCL. • Suponiendo FAIRNESS, garantiza LOCK-FREEDOM, pero no WAIT-FREEDOM.	• Garantiza EXCL, LOCK-FREEDOM y WAIT-FREEDOM.
Complej.	Los algoritmos vistos requieren O(n) registros RW.	

Teorema: No se puede garantizar EXCL y LOCK-FREEDOM con menos de n registros RW

	Fischer	**Problema del consenso**
Registros	• turn: multi-writer / multi-reader	Dados: • *Valores:* V = {0, 1} • *Inicio:* Todo proceso *i* empieza c/ *init(i)* ∈ *V* . • Decisión: Todo proceso i decide un valor decide(i) ∈ V . El problema de consenso requiere: • *Acuerdo:* Para todo i ≠ j, decide(i) = decide(j). • *Validez:* Existe i, tal que init(i) = decide(i). • *Terminación:* Todo i decide en un número finito de transiciones (WAIT-FREEDOM). *Teorema:* No se puede garantizar consenso para un n arbitrario con registros RW atómicos.
Proceso i	/* TRY */ L : waitfor turn = 0; turn = i ; *tarda a lo sumo δ* pause Δ; if turn 6 = i goto L ; /* CRIT */ ... /* EXIT */ turn = 0;	
Caract.	• Garantiza EXCL. • Suponiendo FAIRNESS, garantiza LOCK-FREEDOM si Δ > δ.	

Consensus number

Es la cantidad de procesos para los que resuelve consenso

- Registros RW atómicos = 1
- Colas, pilas = 2
- (TAS) getAndSet() = 2
- compareAndSwap() tiene consensus number infinito.

Administracion de memoria

El manejador de memoria se encarga de:

1. Asegurar la disponibilidad de memoria.

   Memoria virtual: Hacerle creer al proceso que dispone de más memoria de la que realmente tiene.

   - Cantidad de bytes de memoria física = MEM SIZE.
   - Cantidad de unidades de direccionamiento de memoria física = MEM SIZE / DIR UNIT.
   - Cantidad de unidades de direccionamiento de memoria virtual = 2^DIR_BITS .
   - Cantidad de bytes de memoria virtual = 2^DIR_BITS * DIR UNIT.

2. Asignar y liberar memoria.

- Asignar = reservar una porción de memoria para un proceso.
  - La porción de memoria pasa a estar ocupada por el proceso que la solicitó.
  - Tenemos que saber quién es el dueño de esa porción de memoria.
- Liberar = un porción de la memoria vuelve a estar disponible para cualquier proceso.

3. Organizar la memoria disponible.

El MM debe elegir que porcion de memoria asignar

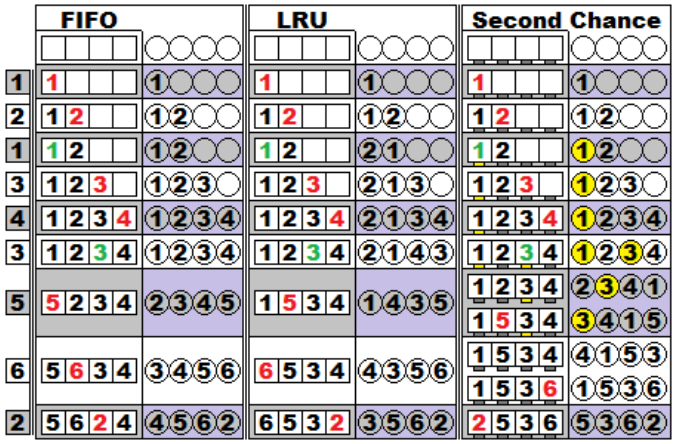
- ¿Cómo agrupamos la memoria?
  - *En unidades de direccionamiento.*
  - *En Bloques de tamaño fijo.*
  - *En Bloques de tamaño variable.*
- ¿Cómo organizamos la memoria libre?
  - *Con un mapa de bits.*
  - *Con una lista enlazada.*
- Mecanismos más sofisticados:
  - *Segmentación:* Separar la memoria en segmentos.
    - Cada segmento tiene una base y un límite
    - Se acceden mediante **direcciones lógicas.**
  - *Paginación:* Dividir la memoria en paginas.
    - Nos permite mapear mucha más memoria de la que realmente tiene el sistema, Si una página no está cargada en ningún marco de página, el MM se encarga de ir a buscarla al disco y cargarla en memoria.
    - Algoritmos de remoción:
      - FIFO, LRU
      - Segunda oportunidad: si fue referenciada le doy otra chance
      - Not Recently Used: Primero desalojo las ni referenciadas ni modificadas. Después las referenciadas y por último las modificadas.

Tengo un sistema con 6 páginas y sólo 4 marcos de página. La memoria comienza vacía.

Llegan los siguientes pedidos de memoria (número de página) en ese orden:

1, 2, 1, 3, 4, 3, 5, 6, 2

Hit-Rate= Páginas qué pedí y ya estaban cargadas en memoria/páginas totales pedidas.



- *Segmentación + paginación.*

Direcciones logicas vs. Fisicas

- *Memoria física:* Una celda dentro de la memoria del sistema. El tamaño está determinado por el hardware.
  - Un **marco de página** es una porción de memoria física.
- *Memoria virtual:* Una representación de la información almacenada. El tamaño depende de la unidad de direccionamiento y la cantidad de bits de direccionamiento.
  - Una **página** es una porción de memoria virtual.

4. Asegurar la protección de la memoria.

Un proceso no debería poder usar memoria que no reservó. Soluciones:

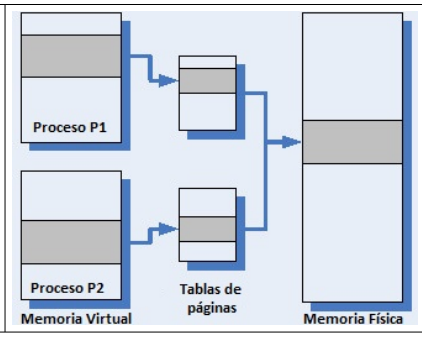
- *Paginación:* Cada proceso tiene su propia tabla de páginas.
- *Segmentación:* Cada proceso tiene su propia tabla de segmentos.

5. Permitir acceso a memoria compartida.

Queremos que dos procesos lean y escriban sobre una misma variable.

- *Paginación:* Podemos mapear dos páginas al mismo marco de página.

***malloc* no es syscall** sino un "memory manager" provisto por la lib-C. Ergo, *malloc* vive y se ejecuta en el espacio de usuario, al igual que *calloc, realloc y free*



En linux malloc pide memoria al kernel con:

- *brk() y sbrk():* cambian el límite del heap del proceso
- *mmap():* mapea una página en el espacio de direcciones del proceso.

Como pide mas memoria?

1. busca entre las porciones libres de memoria del Heap uno o más buckets contiguos que puedan satisfacer el tamaño pedido.
2. Si eso falla, se intentará extender el Heap del proceso a través brk()/sbrk().
3. La llamadas brk()/sbrk() al Kernel ajustarán, dentro del mm struct del proceso, el tope del Heap para que éste sea más extenso. Esto no aumenta la memoria directamente, ya que no se está mapeando memoria física al proceso, sino que lo que se hace es ampliar el espacio de direcciones reservado para el Heap.
4. Finalmente, cuando alguna de esas posiciones de memorias no mapeadas es accedida (generalmente por una lectura/escritura de la implementación de malloc) se produce una excepción de Page Fault. Dicha excepción será atrapada por el Kernel y producirá una invocación al administrador de páginas para obtener una nueva página de memoria física para el frame que generó la excepción.

6. Controlar el swapping.

Si solo hay un proceso, no hace falta compartir. Si hay mas de uno hay que hacer **swapping,** es decir, pasar a disco el espacio de memoria de los procesos que no se estan ejecutando. Esto nos puede causar problemas como:

- *Reubicacion:* Puede ser que cuando vuelva a la memoria, el espacio que les corresponda no sea el mismo, lo que se puede solucionar teniendo un registro que haga de base y que las direcciones del programa sean relativas.
- *Proteccion:* Como nos aseguramos que un proceso no lea los datos de otro (memoria privada)?
- *Manejo de espacio libre:* Como sabemos que pedazos de memoria tenemos libres y donde conviene ubicar un programa? (Evitar fragmentacion)

Fragmentacion

Ocurre cuando tenemos suficiente memoria para atender una solicitud pero no es continua.

Puede ser:

- *Externa:* Bloques pequeños de memoria no contiguos.
- *Interna:* Memoria desperdiciada dentro de una partición (un bloque o página).

Veamos posibles soluciones:

- *Compactar:* requiere reubicacion, es muy costoso en tiempo
- *Organizar la memoria:*
  - *Bitmap:* Dividir en bloques de igual tam la memoria. Cada posicion del bitmap representa un bloque (0 = libre, 1 = ocupado). Encontrar bloques consecutivos requiere barrida lineal.
  - *Lista enlazada:* Cada nodo representa un proceso o bloque libre, y dicen el tam y sus limites. Liberar es simple. Asignar puede ser (ninguna se usa en la vida real):
    - First fit: La primera sección de memoria contigua del tamaño necesario.
      Rapido, tiende a fragmentar la memoria
    - Best fit: De todas las secciones de tamaño mayor o igual al tamaño necesario, tomo la más chica.
      Lento, no es mejor (llena la memoria de bloquecitos inservibles)
    - Worst fit: Toma la mas grande
    - Quick fit: Mantengo una lista de bloques libres de los tamanios mas frecuentemente solicitados.
    - Buddy sistem: usa splitting de bloques
  - *Memoria virtual:* Requiere un MMU. Espacio de direcciones: tamaños de la mem física + swap.
    Los programas usan direcciones virtuales. La memoria virtual se divide en **paginas** de tam fijo y la fisica en **page frames.**
    La MMU traduce paginas a frames y se swappean las paginas. Si la pagina no esta en memoria, la MMU hace page fault y la atrapa el SO.
    La MMU es una tabla de paginas, y para hacerla mas eficiente conviene hacerla multinivel: Los primeros bits nos llevan hacia la tabla que tenemos que consultar. Hace que no haga falta tener toda la tabla en memoria y se pueden swappear sus partes.
  - *Memoria asociativa:* Para mejorar la performance, podemos agregar un cache a la memoria virutal, lo que nos permite mapear directamente páginas a frames sin consultar las tablas de páginas (que pueden tener varios niveles).

Page Fault

¿Qué pasa cuando una página no está cargada en la memoria?

1. Un proceso accede a una dirección (virtual) de memoria.
2. La MMU traduce la dirección virtual a dirección física (accede a la entrada en la última tabla de páginas).

3. Lee el atributo correspondiente a presencia en la memoria.
4. Si es negativo, se produce la interrupción Page Fault.
5. Se ejecuta la RAI correspondiente.
6. Si la memoria está llena, se ejecuta el algoritmo de remoción.
7. Si la página que se va a desalojar fue modificada, hay que bajarla al disco.
8. Se carga en el lugar liberado la página solicitada.
9. Se vuelve a ejecutar la instrucción del proceso que accede a la dirección solicitada.

Thrashing

Situación en la que el SO pasa más tiempo cargando páginas que ejecutando procesos.

Supongan que tenemos 2 procesos y un solo marco de página disponible.

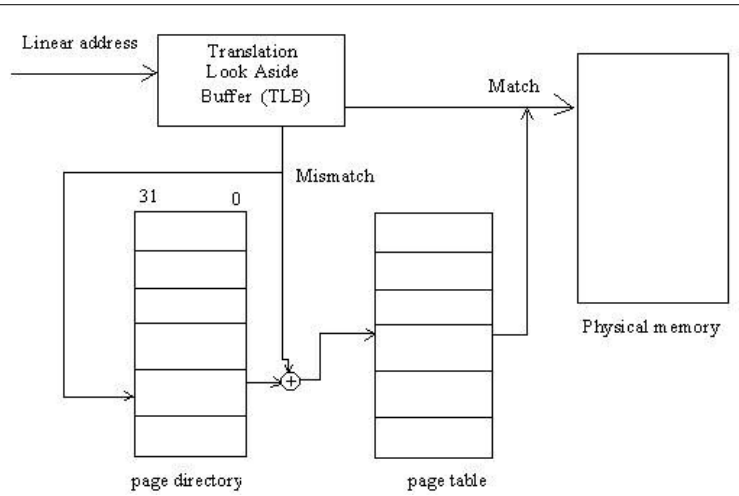
Cada proceso usa una sóla página, pero cada vez que ejecute, su página no va a estar cargada.

Siempre va a estar cargada la página del otro.

TLB (Translation Lookaside Buffer)

Buffer de Traducción Adelantada.

- Es una caché que guarda ’traducciones’.
- Paginación de 4 niveles: Cuatro accesos a memoria (1 por cada tabla) más uno para leer la página.



Copy-on-write

Al crear un nuevo proceso se duplica toda su memoria.

Sabemos que, en general después de un fork() viene un exec().

exec() inutiliza todas las páginas de memoria, entonces ¿para qué nos gastamos en duplicar todo?

Copy-on-Write: Sólo duplico (copy) cuando alguno de los procesos escribe (write).