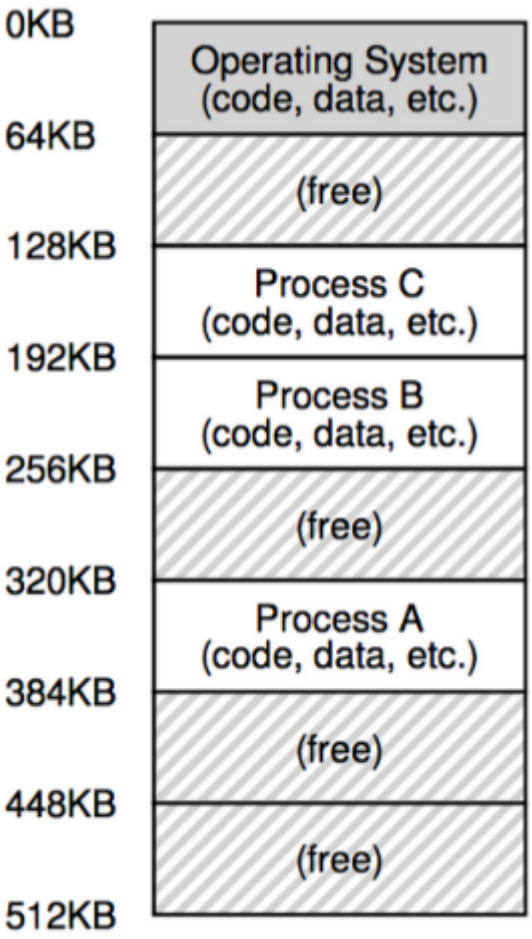
Administración de memoria

La memoria del computador está organizada como un arreglo muy grande de *bytes*. Cada byte tiene una dirección única. La introducción de la multiprogramación trajo consigo dos problemas:

- Las variables no siempre están en la misma dirección. Requisito: relocalización de variables.
- Un proceso podría leer y modificar memoria de otro. Requisito: protección de memoria.

Es evidente entonces, que con multiprogramación, no nos sirven direcciones absolutas. Así, una solución es que el compilador no genera direcciones absolutas, si no que direcciones reubicables (e.g. JMP 32+X). Esto permite utilizar un espacio de direcciones.

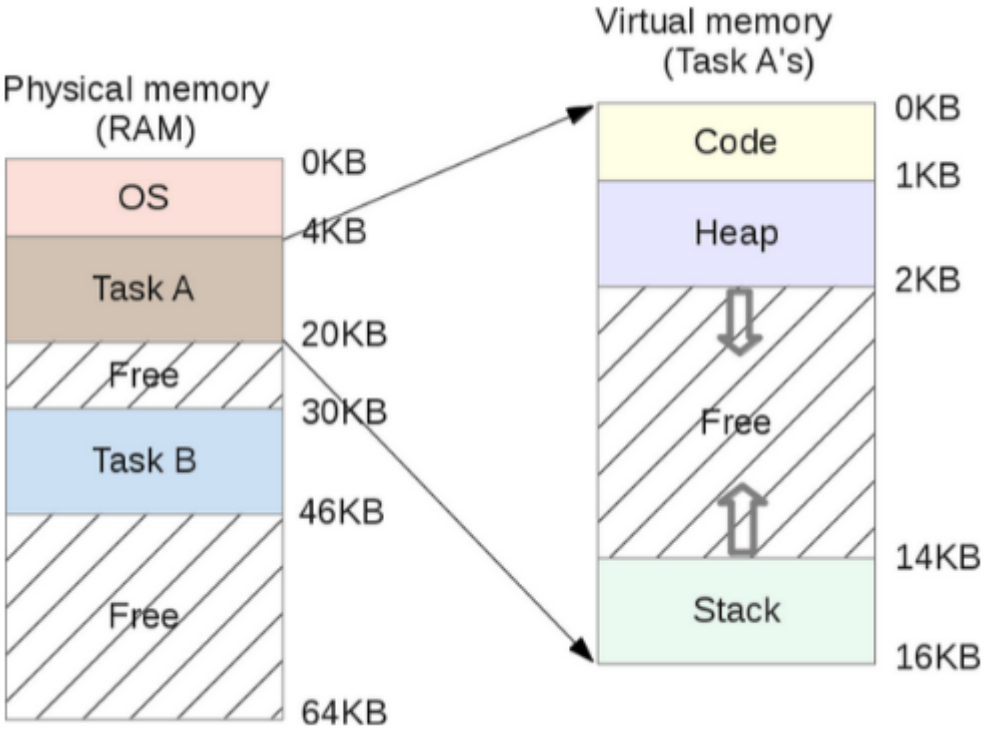


PROF

Espacio de direcciones

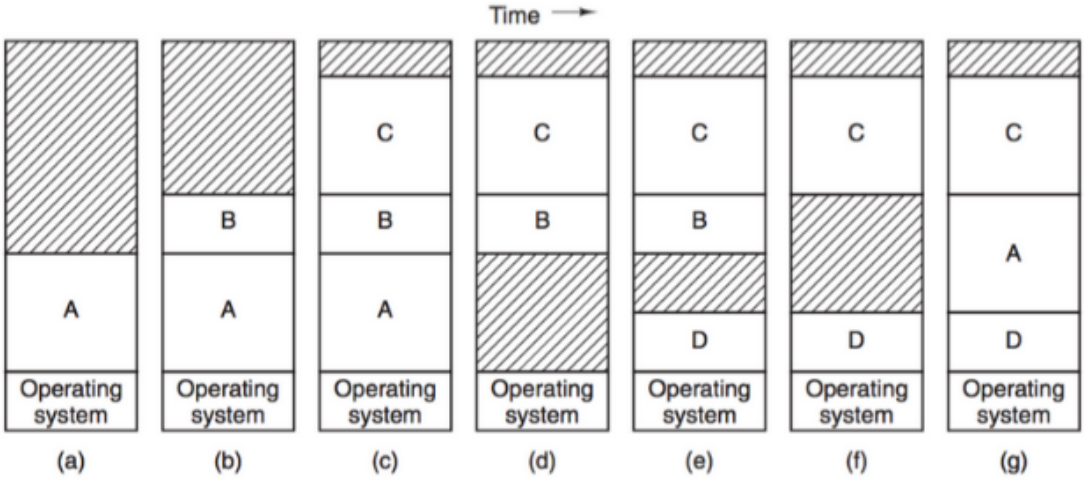
Cuando uno obtiene las direcciones de memoria de un programa, lo que recibe son direcciones virtuales! Esta abstracción se llama espacio de direcciones.

El proceso mantiene un espacio **único** y **secuencial** (lineal) de direcciones. Un proceso de 16KiB utiliza direcciones de 0 a 16383. El sistema operativo mapea estas direcciones a direcciones físicas. Esto lo hace por *hardware* en la *MMU*.



Sobrecarga de memoria: ¿qué pasa si la memoria se llena?

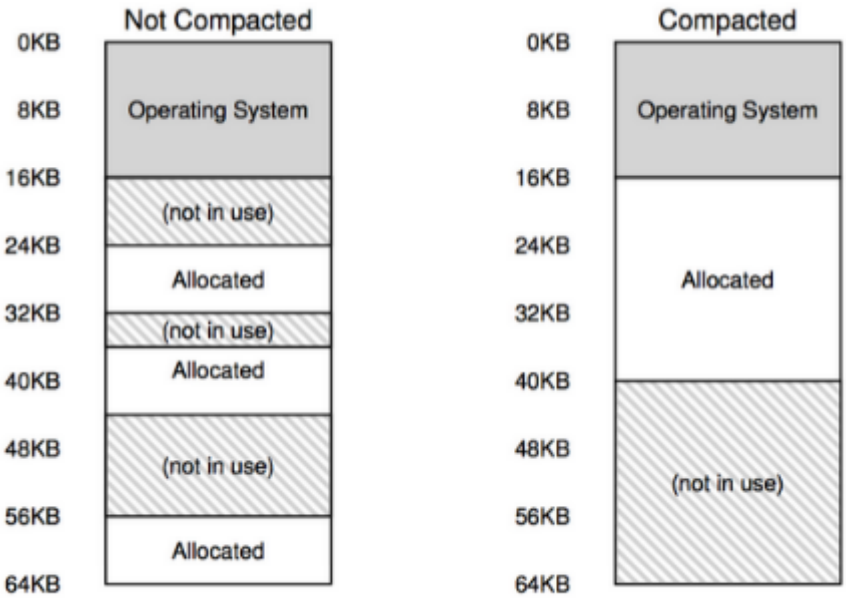
El *mid term scheduler* determina cuando hay que hacer operaciones de *swapping*. Esta operación utiliza una porción del disco, denominado espacio de *swap*. Acá se guardan imágenes de memoria de procesos. Así, los procesos pueden ser cargados y descargados de la memoria, lo que puede dejar algunos huecos...



PROF

Compactación

Una opción es realizar una operación de **compactación**, fusionando los huecos. Esto es muy costoso, ya que requiere mover todos los procesos en memoria. Algo mejor es asignar de manera inteligente los espacios de memoria para que nunca deba compactar. Esto da paso a las estrategias de alocación de memoria.



Estrategias

- *First-fit*: asigna el primer hueco que encuentre.
- *Best-fit*: asigna el hueco más pequeño que encuentre.
- *Worst-fit*: asigna el hueco más grande que encuentre.

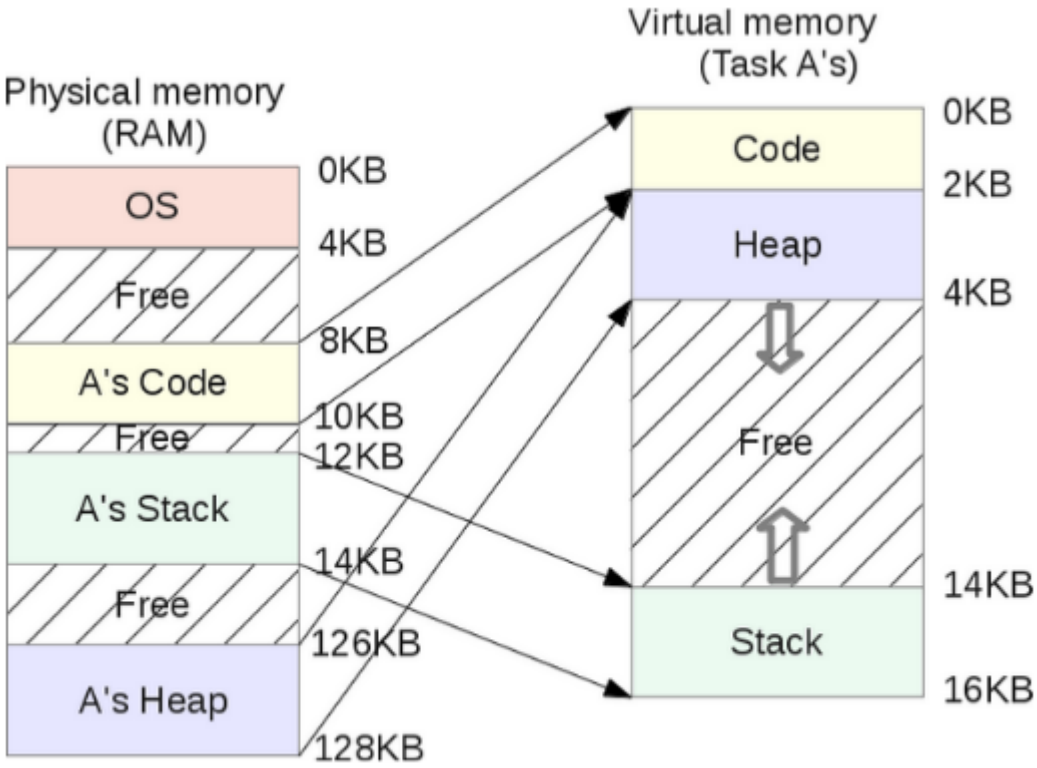
Cuando los espacios libres quedan separados se dice que hay **fragmentación**.

Segmentación

Muchas veces los procesos requieren harta memoria, en general, es difícil tener un espacio de memoria físico en donde quepa por completo. La segmentación permite dividir un proceso en segmentos más pequeños, cada uno con su propio espacio de direcciones. Esto permite que cada segmento pueda ser cargado en memoria independientemente. Los segmentos son espacios de direcciones contiguos (segmentos: code, heap, stack, ...
).

PROF

La *MMU* mapea cada segmento a un espacio de memoria físico. Queda especificado en la tabla de segmentos. Esta tabla además especifica el tamaño de memoria que se la ha asignado, por lo que, todos reciben una asignación de memoria contigua, eliminando la fragmentación externa. Sin embargo, la fragmentación interna sigue existiendo. Esta consiste en los espacios sin utilizar dentro de un segmento.



Necesito conocer el segmento y el *offset*. Con lógica de *bits* es muy fácil.

- Dirección virtual:
  $4200 = 0b01000001101000 = 0x1068$
  - Bits de segmento: $0b01 = 1 = 0x1$
  - Bits de offset:
    $0b000001101000 = 104 = 0x068$

PROF

```
SEG_MASK = 0x3000;
OFFSET_MASK = 0xFFF;
SEG_SHIFT = 12;
virtualAddress = 4200;
segment = (virtualAddress & SEG_MASK) >> SEG_SHIFT;
offset = virtualAddress & OFFSET_MASK;
if(offset >= size[segment])
   raise(SEG_FAULT);
else
   physicalAddress = base[segment] + offset;
```

Ahora, hay que tener cuidado, porque crece en sentido contrario. Así, se agrega un bit que indica el sentido en el que crece el segmento.

La segmentación elimina la fragmentación externa (espacios libres (sin asignar, no sin ocupar) no contiguos), sin embargo, sigue existiendo fragmentación (fragmentación interna, cuando se asigna más de lo pedido). Además, es difícil anticipar el tamaño de los segmentos.

Visualización de segmentos

el comando pmap permite visualizar los segmentos de un proceso.

Paginación

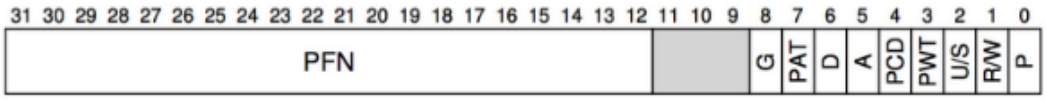
Podemos añadir una segunda idea para reducir la fragmentación: que los segmentos sean del mismo tamaño, así podemos utilizar páginas de memoria y una tabla de páginas.

- Espacio virtual: páginas.
- Espacio físico: marcos de página (*frames* o *page frames*).
- Páginas y *frames* del mismo tamaño.

Tabla de páginas

- *offset*: offset o desplazamiento dentro de la página.
- VPN: número de página virtual.
- PFN: número de marco de página físico.
- VPN es traducido a PFN por la tabla de páginas.

En una arquitectura con espacio de direcciones virtuales de 32 bits, si cada página tiene 4KiB, entonces la tabla de páginas tiene 2^20 entradas (PTE, *page table entry*). Esto genera un problema enorme: la tabla de páginas tiene un espacio de 4MiB por proceso, espacio del sistema operativo.



PTE en x86

PROF

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

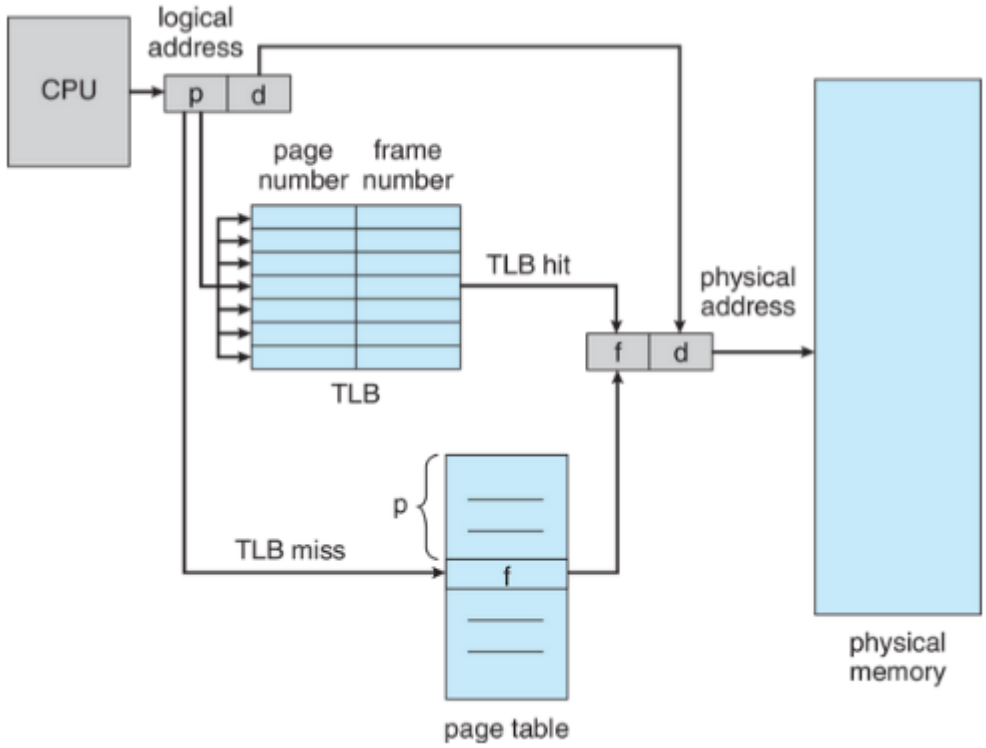
La dirección de la tabla de páginas se guarda en el PCB (*process control block*), como PTBR (*page table base register*). Así, cada acceso a memoria se convierte en dos accesos: primero se accede a la tabla de páginas, y luego a la memoria.

Paginación con TLB

Recordando que cada vez que accedo a memoria, tenemos que primero acceder a la tabla de páginas, y luego a la memoria física. Esto es muy lento, por lo que se añade un caché de páginas,

llamado TLB (*translation lookaside buffer*). Esta caché es una memoria *fully-asociative*, que guarda las últimas páginas accedidas. Así, se reduce el tiempo de acceso a memoria.

- Si la dirección está en la caché se produce un *TLB hit* y se responde inmediatamente.
- Si la dirección no está en la caché se produce un *TLB miss* y se lee desde la memoria y se actualiza el TLB.



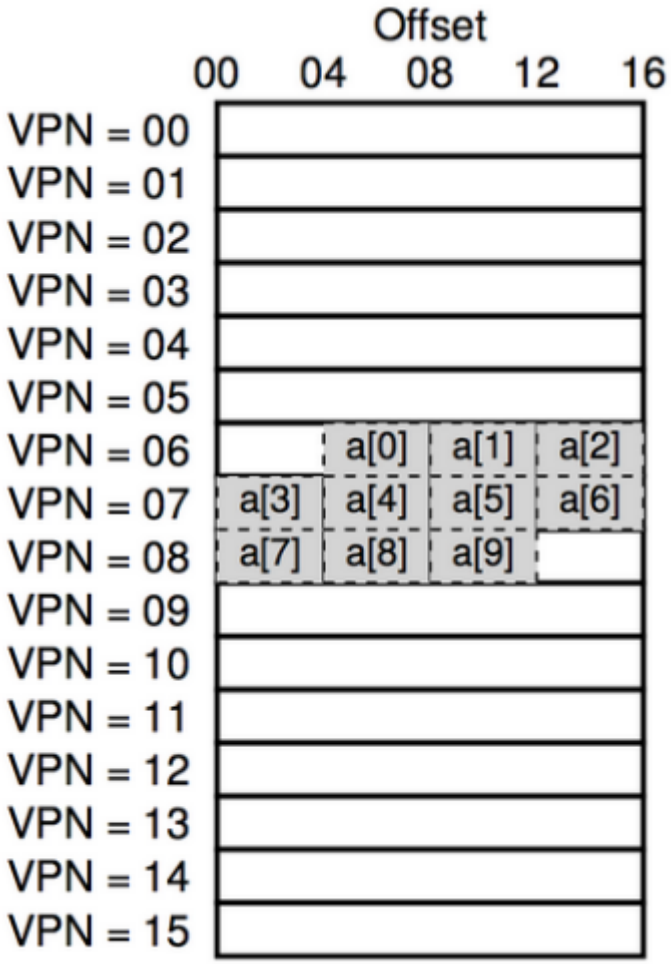
Así, recordando que el TLB se encuentra en la MMU, el acceso al TLB es considerablemente más rápdio que el acceso a la memoria principal para la tabla de páginas.

PROF

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

Efectividad del TLB

Consideremos que queremos hacer acceso a a[0] hasta a[9]. Sin TLB se tienen 20 accesos a memoria, mientras que con TLB se accede a la tabla de páginas una vez por página, para traerla al TLB; además de los 10 accesos a memoria "obligatorios".



PROF

No siempre es ideal, hay que hacer cálculos para ver si vale la pena. Si el acceso al TLB es de 1ns y a memoria es de 100ns, significa que sin TLB el tiempo total fue de 2000ns y con TLB de 1310ns. Entonces con TLB es 1.52 veces de rápido. En realidad, el *hit rate* suele ser más cercano a 99%. Esto ocurre por localidad espacial y localidad temporal.

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

Las *page table entrys* (PTE) de un proceso no sirven para otro proceso, así, la mejor opción es guardar en las *entrys* del TLB el *address space ID* (ASID) del proceso. Así, cuando se cambia de contexto, sabemos que las entrys con el mismo ASID que el proceso en contexto son válidas, y las demás no.

Variantes de paginación

Recordando el problema de que las tablas de páginas pueden llegar a ser muy grandes, y que estas viven en el OS. Una solución es tener páginas más grandes, lo que permite necesitar menos páginas y una tabla de páginas más pequeña.

Fragmentación interna

Esta corresponde a tener **espacio sobreasignado**, es decir, que a un proceso se le asigna más memoria de la que realmente necesita. Aumentar el tamaño de las páginas aumenta el problema de fragmentación interna!

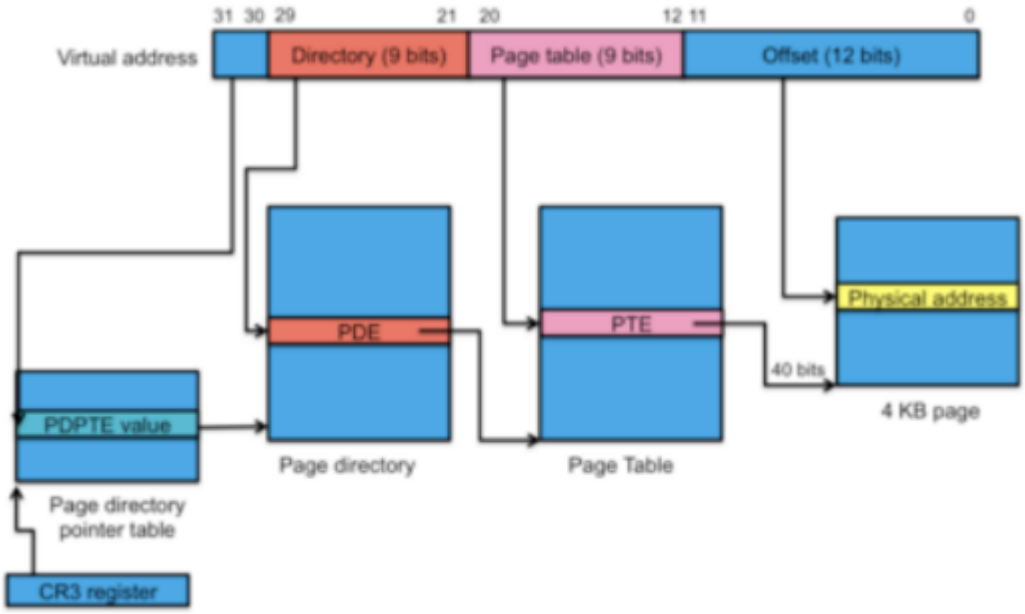
Segmentación de la tabla de páginas

Por otro lado, los proceso no suelen ocupar todas sus páginas, por lo que con **segmentos paginados**, se tiene una tabla de páginas por segmento, por lo que se tiene tablas de páginas mas pequeñas. Pero vuelve la fragmentación externa!

Paginación multinivel

La idea es la siguiente, en vez de segmentar la tabla de páginas, paginarla de nuevo.

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



PROF

La paginación multinivel no cambia la cantidad de memoria direccionable total, sino que, cambia la cantidad de entradas de tablas de páginas necesarias en memoria.

Reemplazo de páginas

Procesos pueden tener parte de su memoria no cargada aun (en disco) o en *swap* (en disco).
El desafío es como utilizar efectivamente el disco (grande pero lento) para proveer la ilusión de que todo el espacio virtual está en memoria.

Swap space

- Tabla de páginas utiliza un bit de presencia para saber si una página está en un *page frame* o no. En caso negativo, se genera un *page fault* lo que activa el mecanismo de recuperación de una página desde disco a memoria física:
- Sistema operativo atiende el *page fault*
- Cuando el *page fault* ha sido resuelto, el proceso puede continuar.

Como dijimos, ante un *page fault*, el sistema operativo va a buscar la página. Algoritmo para buscar la página:

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

PROF

Algoritmo para panejar un *page fault*:

```
PFN = FindFreePhysicalPage();   // busca frame libre
if(PFN == -1)                  '// no había frame libre :'(
PFN = ReplacePage();         // rutina de reemplazo de página

// Copia desde el disco a la memoria. Proceso queda "waiting on I/O"
DiskRead(PTE.DiskAddress, PFN);

// Actualiza tabla de páginas
PTE.present = TRUE;
PTE.PFN = PFN;

// Vuelve a ejecutar la instrucción que generó el PAGE_FAULT
Retry();
```

Aloritmos de reemplazo: ¿qué página reemplazar?

Cualquier algoritmo de decisión debe intentar minimizar la ocurrencia de *page faults*.

OPT / MIN

Elegir la página que será utilizada lo más lejos posible en el futuro. Es el algoritmo ideal, pero no se puede implementar porque no se conoce el futuro 😢

FIFO

- Muy simple de implementar.
- Elegir la página que lleva más tiempo en memoria.
- Uno podría pensar que a mayor cantidad de *frames* se comporta mejor, pero esto no siempore es cierto.

RANDOM

- Puede tener rendimiento promedio mejor a FIFO.
- Al igual que FIFO, no aprovecha la localidad de referencia.

LRU (*least recently used*)

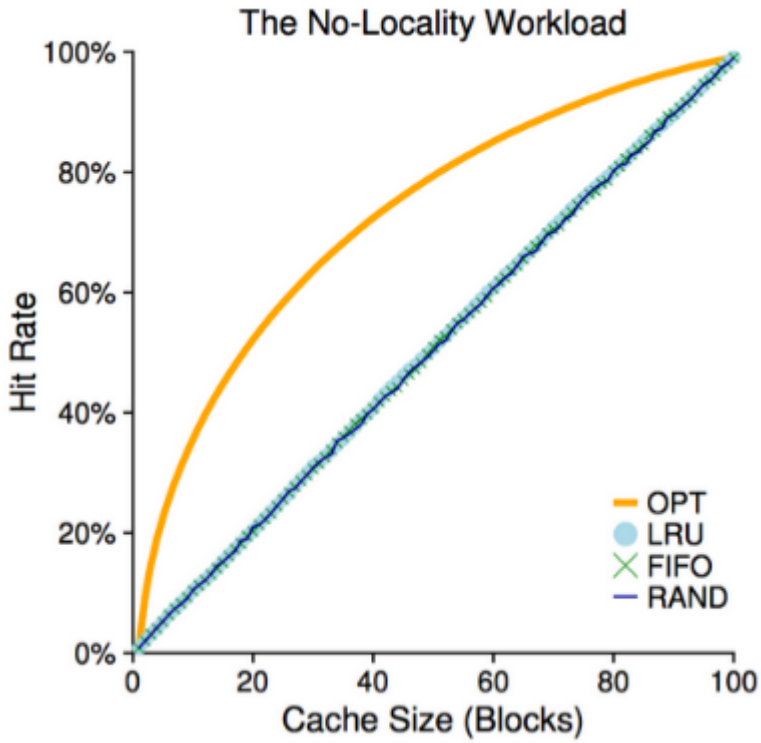
- Elegir la página que no ha sido utilizada por más tiempo.
- Rendimiento similar a MIN
- Es implementable
- Nos queda la duda si es realmente eficiente por tener que estar manejando una cola.

Workload

***Workload* sin localidad**

PROF

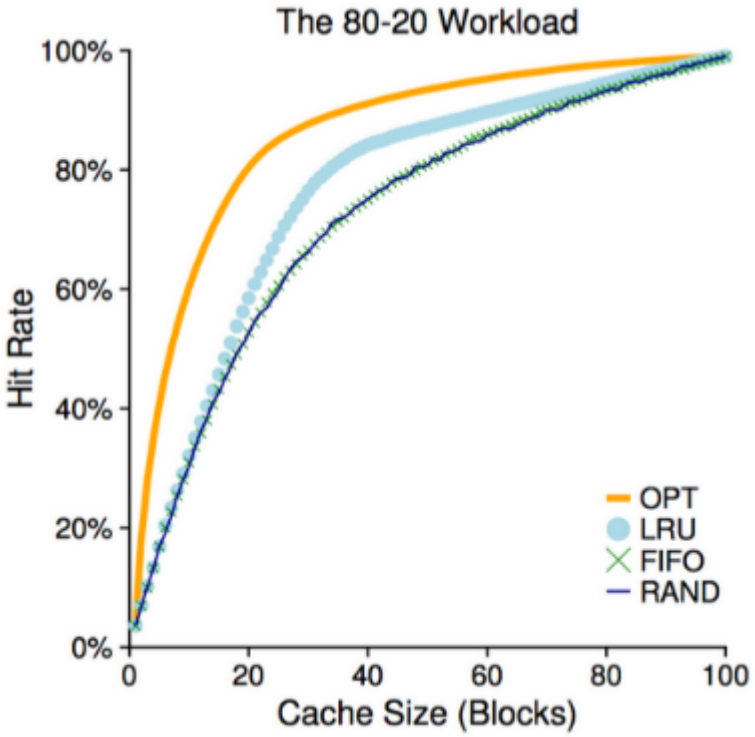
- Todas las políticas que podemos implementar se comportan casi igual.
- Con suficientes *frames* disponibles se llega a un *hit rate* de 1.



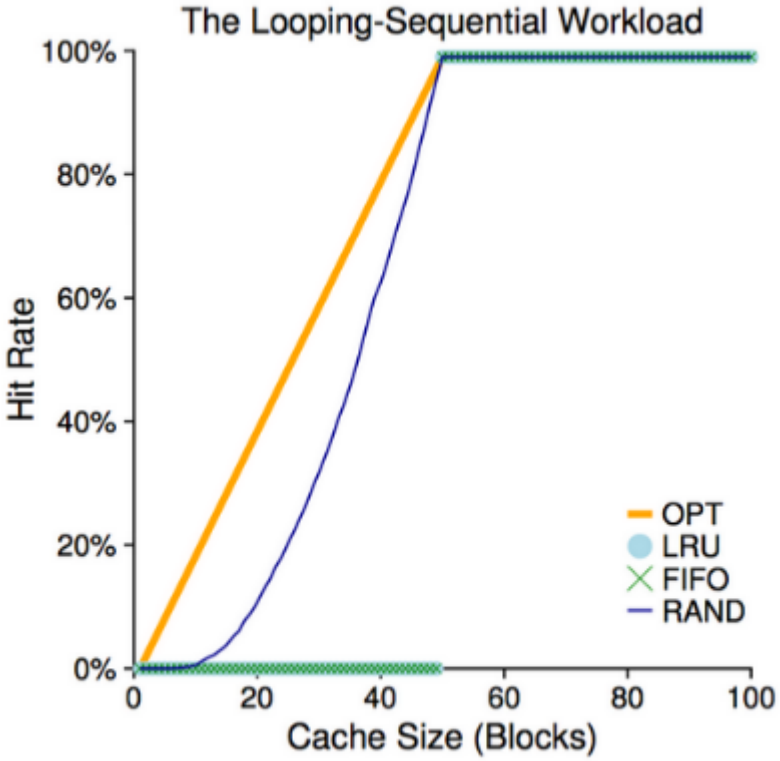
***Workload* con localidad**

- LRU mejor que FIFO y RANDOM
- Patrón más común

PROF



Workload con _loop_ secuencial * FIFO y LRU eliminan las páginas más antiguas * RANDOM se comporta mejor * Es el peor caso para FIFO y LRU * Es un patrón común en muchas aplicaciones



Implementando LRU

- Hay que actualizar una cola **para cada acceso**. Mantener esta cola ordenada es caro.
- Podemos agregar un *timestamp* por *hardware* a cada página. Utilizar esto requiere comparar todos los *timestamps* para encontrar el mínimo, lo que es caro.

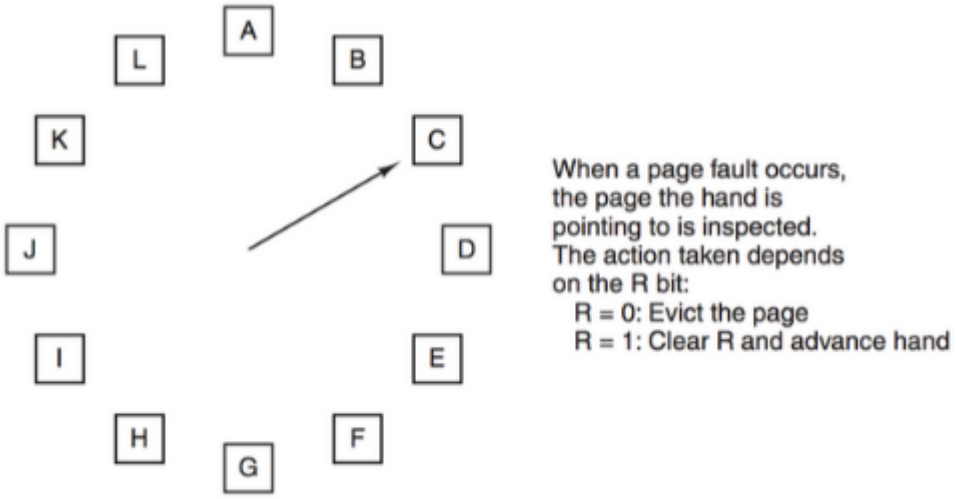
Aproximando LRU

Algoritmo del reloj (*clock*)

Aproximación basada en el *reference bit*

PROF

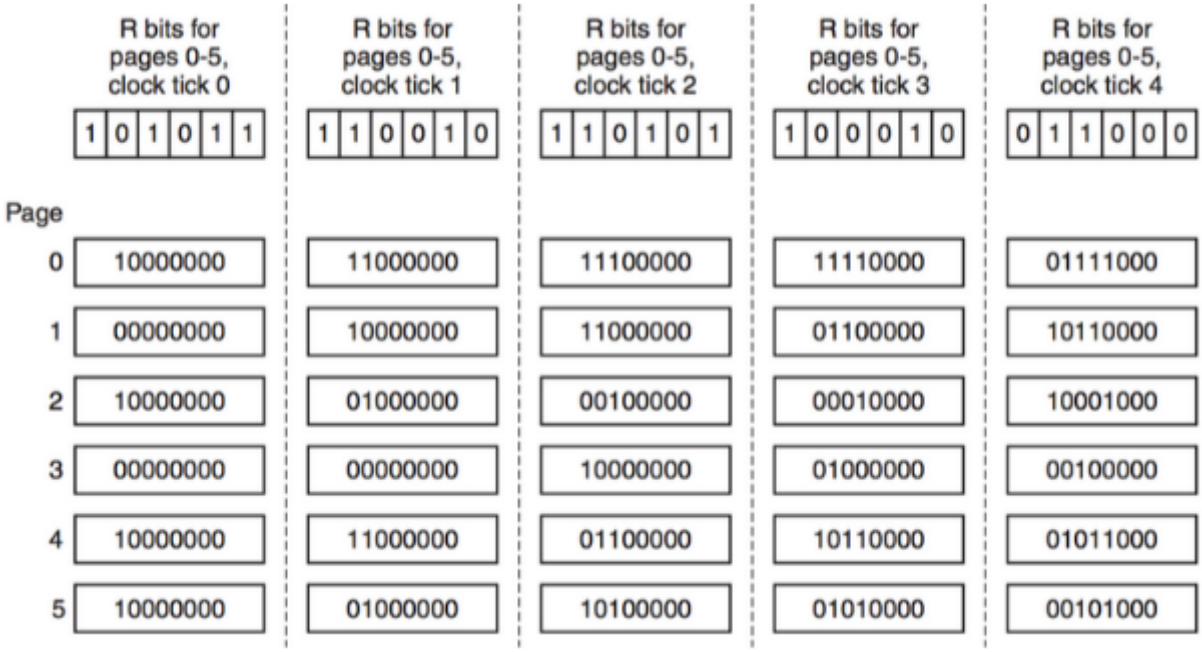
- Algoritmo apunta a una página cualquiera
- Al momento de elegir se mira el *reference bit*:
  - Si es 1, se cambia a 0 y se pasa a la siguiente página
  - Si es 0 se elige
- Aproxima LRU bastante bien y es eficiente



Algoritmo de *aging*

Si nos enfocamos en múltiples *reference bits* se obtiene una mejor aproximación de LRU.

- Contadores se actualizan por *hardware* a intervalos regulares
- En cada *tick*, los *reference bit* se copian a cada contador (*shift left*) y se borran.
- La secuencia de bits sirve como historial.
- La página con el menor contador es la menos utilizada.



PROF

Usando el *dirty bit*

Permite priorizar mejor las páginas para el algoritmo de reloj.

Reference	Dirty	Descripción
0	0	No usada recientemente ni modificada. Buena candidata.

Reference	Dirty	Descripción
0	1	No usada recientemente pero modificada.
1	0	Usada recientemente pero no modificada.
1	1	Usada recientemente y modificada.

Cuando una página no ha sido modificada no es necesario escribirla en disco! 😄

Working set y *Thrashing*

El modelo *working set* $w_\Delta (t)$ intenta determinar si un proceso tiene una cantidad apropiada de *frames* asignados. Se define como el conjunto de páginas utilizadas por un proceso en los últimos Δ accesos a memoria.

- Se le puede considerar una medida efectia de la demanada por *frames* de un proceso.
- Si no se le puede asignar $|w_\Delta (t)|$ *frames*, entonces no conviene tenerlo en ejecución.
- Si la suma de los *wotrking set* de todos los procesos es mayor que la cantidad de *frames* disponibles, entonces se produce *thrashing*.
- El *working set* es usado por el *medium term scheduler* y puede ser implementado utilizando los *reference bits*.