

Sistemas Operativos

Swapping, VM completa y examen parcial

¿Qué pasa cuando la memoria no alcanza?

Gerson Garrido

Semana 8

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

Sesión A · Swapping y VM completa

1. Bit presente, fallo de página y swap
2. Reemplazo: OPT, FIFO, aleatoria, LRU, reloj
3. Thrashing y conjunto de trabajo
4. Experimento: presión de memoria
5. VAX/VMS, Linux y xv6

Sesión B · Examen parcial

6. Repaso: qué entra y tipos de problemas (al final de hoy)

Lectura

OSTEP cap. 21–22 *Beyond Physical Memory*, cap. 23 *Complete Virtual Memory Systems* y cap. 24.

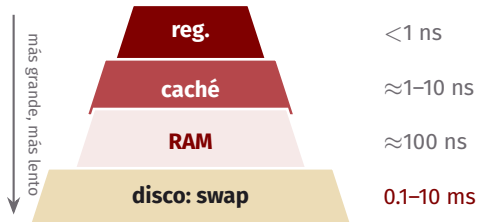
SECCIÓN 01

Más allá de la memoria física

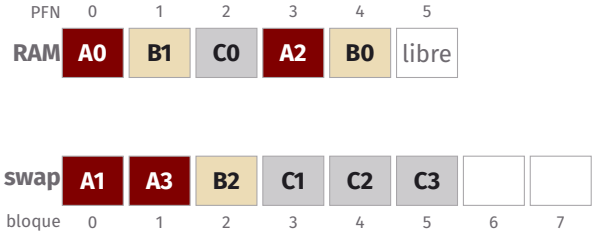
01

El problema: la suma no cabe

- Hasta ahora supusimos que todas las páginas de todos los procesos tienen un marco de página *(page frame)* en RAM.
- Pero el espacio de direcciones es enorme (Sv39: 512 GB) y hay muchos procesos.
- Idea: usar el disco como un nivel más de la jerarquía. Las páginas que no se usan esperan en el disco.
- El programa no se entera: sigue viendo una memoria grande. Es la misma ilusión de siempre, ahora más grande.



Espacio de swap *(swap space)*



El proceso A tiene 4 páginas: 2 en RAM (A0, A2) y 2 en swap (A1, A3).

El proceso C casi no está en RAM: tal vez está dormido.

- El kernel reserva una zona del disco (partición o archivo) dividida en bloques del tamaño de una página.
- Debe recordar en qué bloque está cada página que salió de la RAM.
- El código de un programa no necesita swap: si se expulsa, se vuelve a leer del ejecutable.

El bit presente

- Cada PTE dice si la página está en memoria o en disco: el bit presente *(present bit)*.
- Presente $= 1$: la MMU traduce como siempre.
- Presente $= 0$: la MMU no puede traducir y lanza una excepción: un fallo de página *(page fault)*.
- Con la página fuera, el resto de la PTE queda libre: el kernel guarda ahí el bloque de swap.

PTE de RISC-V Sv39



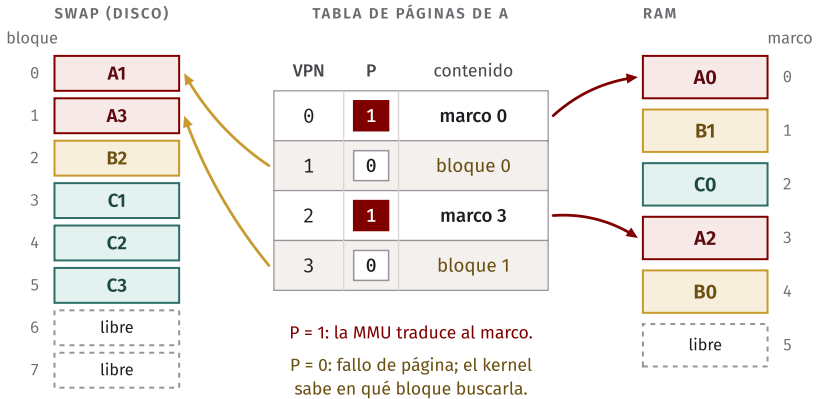
RISC-V no tiene un bit "presente" aparte: $V = 0$ cumple ese papel



la misma PTE, con $V = 0$

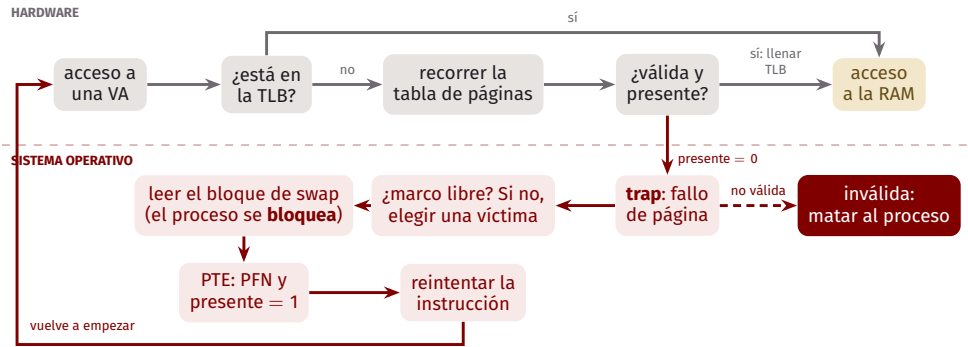
¿Por qué lo atiende el kernel y no el hardware? El disco tarda milisegundos: al lado de eso, el trap y la lógica en software no cuestan nada.

La PTE dice dónde está cada página



Con $P = 1$, la PTE da el marco; con $P = 0$, el kernel guarda ahí el bloque del swap.

Qué pasa en un fallo de página



Si la dirección ni siquiera es válida (fuera de sus regiones), el kernel termina al proceso: `Segmentation fault`.

¿Cuánto cuesta un fallo de página?

Sea p la probabilidad de que un acceso cause un fallo de página.

$$T_{\text{efectivo}} = (1 - p) \cdot T_{\text{RAM}} + p \cdot T_{\text{fallo}}$$

Con $T_{\text{RAM}} = 100\,\text{ns}$ y $T_{\text{fallo}} = 10\,\text{ms}$ (disco duro):

- $p = 1/1000$: $0.999 \cdot 100 + 0.001 \cdot 10^7 \approx 10\,100$ ns. ¡100 veces más lento!
- Para perder solo un 10 %: $p \cdot 10^7 \leq 10$, o sea $p \leq 10^{-6}$: un fallo cada millón de accesos.

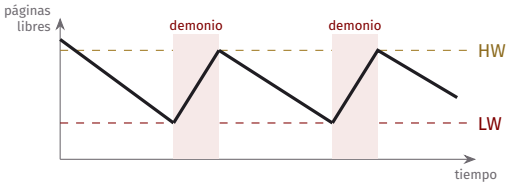
En el contenedor del curso

Swap en un SSD, medido con `memtouch` (más adelante): ≈40 µs de pasada por fallo mayor, unas 400 veces un acceso a la RAM.

Moraleja

El swap solo funciona si los fallos son *rarísimos*. La política de reemplazo importa muchísimo.

¿Cuándo se reemplaza? Marcas de agua



bash

```
$ grep -A5 'zone   Normal' /proc/zoneinfo
Node 0, zone   Normal
  pages free     1854389
        boost    0
        min      9709
        low      13232
        high     16755
```

- No conviene esperar a quedarse *sin ningún* marco libre.
- Si quedan menos de LW páginas libres, se despierta el **demonio de swap** *(swap daemon)* y expulsa páginas hasta tener HW.
- Al expulsar varias juntas, **agrupa** *(clustering)* las escrituras al disco: pocas y grandes.
- Linux: `kswapd` despierta bajo `low` (52 MB) y para en `high` (65 MB). Datos del contenedor, en páginas de 4 KB.

SECCIÓN 02

Políticas de reemplazo

02

La memoria como caché del disco

- La RAM guarda *algunas* páginas; el resto vive en el disco. Elegir qué página expulsar es una política de reemplazo.
- Meta: maximizar los aciertos *(hits)* y minimizar los fallos *(misses)*.
- Tiempo medio de acceso, con T_M el de la RAM y T_D el del disco:

$$\text{AMAT} = T_M + P_{\text{fallo}} \cdot T_D$$

La memoria del OS es totalmente asociativa: una página puede ir a cualquier marco. Por eso no hay fallos de conflicto.

Las 3 C de los fallos

Compulsivos *(cold-start)*: la primera vez que se usa la página. Inevitables.

De capacidad: la página estuvo en memoria, pero se expulsó porque no cabía todo.

De conflicto: en cachés de hardware, una página solo puede ir a ciertas posiciones.

La referencia: la política óptima (OPT)

Expulsar la página que se usará más lejos en el futuro (Belady, 1966). Secuencia de 12 referencias, 3 marcos.

ref	0	1	0	2	3	0	1	4	0	3	4	0
marco 0	0	0	0	0	0	0	0	0	0	0	0	0
marco 1	–	1	1	1	1	1	1	4	4	4	4	4
marco 2	–	–	–	2	3	3	3	3	3	3	3	3
resultado	F	F	✓	F	F	✓	✓	F	✓	✓	✓	✓
fallos	1	2	2	3	4	4	4	5	5	5	5	5

5 fallos, 7 aciertos (58 %). Los 5 fallos son fríos: con 3 marcos no se puede hacer mejor. Pero OPT necesita conocer el futuro: sirve para comparar.

entra · acierto · F fallo frío · F fallo de capacidad

FIFO: expulsar la que llegó primero

Una cola: la página que entró hace más tiempo sale primero. Fácil de implementar.

ref	0	1	0	2	3	0	1	4	0	3	4	0
marco 0	0	0	0	0	3	3	3	4	4	4	4	4
marco 1	–	1	1	1	1	0	0	0	0	3	3	3
marco 2	–	–	–	2	2	2	1	1	1	1	1	0
resultado	F	F	✓	F	F	F	F	F	✓	F	✓	F
fallos	1	2	2	3	4	5	6	7	7	8	8	9

9 fallos, 3 aciertos (25 %). 4 fallos más que OPT, todos de capacidad. FIFO expulsó dos veces a la página 0, que aparece en 5 de las 12 referencias.

entra acierto F fallo frío F fallo de capacidad

Aleatoria (RAND): elegir al azar

Expulsar una página cualquiera: no hay que guardar nada. Una corrida con la semilla 1.

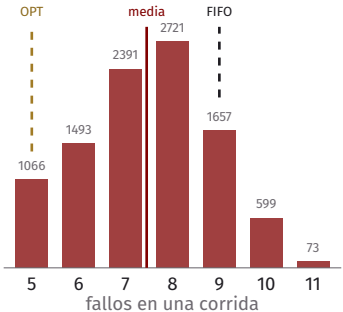
ref	0	1	0	2	3	0	1	4	0	3	4	0
marco 0	0	0	0	0	0	0	0	4	4	4	4	4
marco 1	–	1	1	1	3	3	1	1	0	0	0	0
marco 2	–	–	–	2	2	2	2	2	2	3	3	3
resultado	F	F	✓	F	F	✓	F	F	F	F	✓	✓
fallos	1	2	2	3	4	4	5	6	7	8	8	8

8 fallos, 4 aciertos (33 %) con esta semilla. Con otra semilla el resultado cambia: hay que mirar la distribución.

entra acierto F fallo frío F fallo de capacidad

RAND: 10 000 corridas

- El resultado depende de la suerte: 10 000 corridas con la misma secuencia y 3 marcos.
- `code/week-08/paging.c` simula las cinco políticas; `make examples` reproduce todos los ejemplos de hoy.
- Promedio: 7.45 fallos. A veces iguala a OPT (5), a veces es peor que FIFO.
- Nunca es tonta del todo, pero tampoco aprovecha la localidad.



`./paging -p rand -n 3 -t 10000 ...` en el contenedor del curso.

LRU: expulsar la usada hace más tiempo

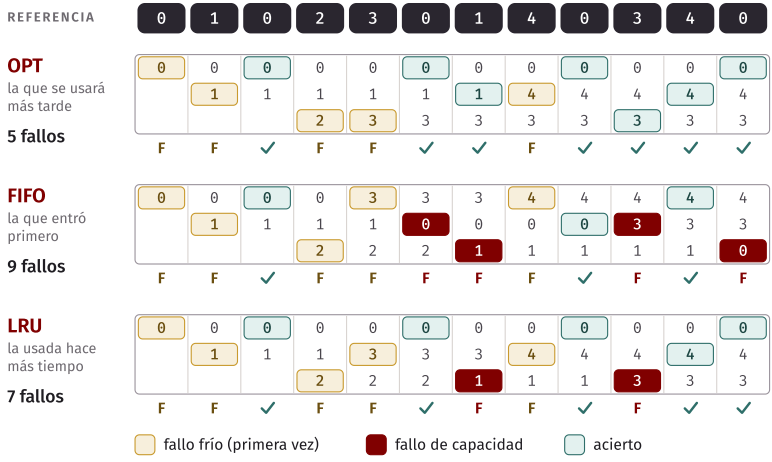
Usar el pasado para adivinar el futuro: lo que no se usó hace rato, probablemente no se use pronto.

ref	0	1	0	2	3	0	1	4	0	3	4	0
marco 0	0	0	0	0	0	0	0	0	0	0	0	0
marco 1	–	1	1	1	3	3	3	4	4	4	4	4
marco 2	–	–	–	2	2	2	1	1	1	3	3	3
resultado	F	F	✓	F	F	✓	F	F	✓	F	✓	✓
fallos	1	2	2	3	4	4	5	6	6	7	7	7

7 fallos, 5 aciertos (42 %). Mejor que FIFO (9) y que el promedio de RAND (7.45), peor que OPT (5). La página más usada nunca sale.

entra acierto F fallo frío F fallo de capacidad

OPT, FIFO y LRU, lado a lado



Depende de la carga de trabajo

- **Sin localidad** (accesos al azar): todas las políticas realistas dan lo mismo.
- **80-20** (el 20 % de las páginas recibe el 80 % de los accesos): LRU gana, porque guarda las páginas "calientes".
- **Ciclo** (recorrer $N + 1$ páginas una y otra vez con N marcos): el peor caso de LRU y FIFO.

Ciclo `0 1 2 3` repetido 3 veces, 3 marcos:

Política	Fallos	de 12
LRU	12	100 %
FIFO	12	100 %
RAND (media)	7.83	65 %

LRU expulsa siempre *justo* la página que viene después.

No hay política perfecta

LRU funciona porque los programas reales tienen localidad. Cuando no la tienen, hasta el azar le gana.

La anomalía de Belady: más memoria, más fallos

FIFO con la secuencia `0 1 2 3 0 1 4 0 1 2 3 4`.

3 marcos: 9 fallos

ref	0	1	2	3	0	1	4	0	1	2	3	4
marco 0	0	0	0	3	3	3	4	4	4	4	4	4
marco 1	–	1	1	1	0	0	0	0	0	2	2	2
marco 2	–	–	2	2	2	1	1	1	1	1	3	3
resultado	F	F	F	F	F	F	F	✓	✓	F	F	✓

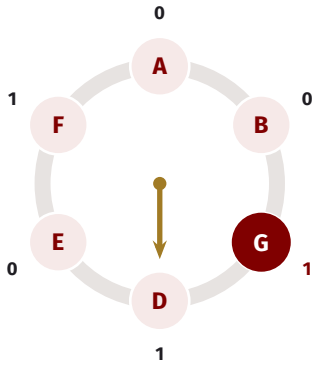
4 marcos: 10 fallos

ref	0	1	2	3	0	1	4	0	1	2	3	4
marco 0	0	0	0	0	0	0	4	4	4	4	3	3
marco 1	–	1	1	1	1	1	1	0	0	0	0	4
marco 2	–	–	2	2	2	2	2	2	1	1	1	1
marco 3	–	–	–	3	3	3	3	3	3	2	2	2
resultado	F	F	F	F	✓	✓	F	F	F	F	F	F

- Con un marco *más*, FIFO tiene un fallo *más*.
- Con 4 marcos, 0 y 1 aciertan al principio, pero luego 4 expulsa a 0 y empieza una cadena de fallos.
- LRU y OPT nunca sufren esto (**propiedad de pila**): lo que está en N marcos también está en $N + 1$.
- LRU aquí: 10 fallos con 3 marcos, 8 con 4.

LRU exacto es caro; el reloj lo aproxima

- LRU exacto: actualizar una marca de tiempo en cada acceso y, al expulsar, buscar la mínima entre millones de páginas. Demasiado caro.
- El hardware ayuda con un **bit de referencia** *(use bit)*: se pone en 1 cuando la página se usa. Solo el OS lo vuelve a 0.
- **Reloj** *(clock)*: las páginas forman un círculo y una manecilla avanza. Bit en 1: ponerlo en 0 y seguir (segunda oportunidad). Bit en 0: expulsar.



número: bit de referencia

C tiene bit 0: se expulsa. G entra con bit 1 y la manecilla avanza.

El reloj con nuestra secuencia

Subíndice: bit de referencia. Triángulo: posición de la manecilla después del acceso.

ref	0	1	0	2	3	0	1	4	0	3	4	0
marco 0	0_1	0_1	0_1	0_1	3_1	3_1	3_1	4_1	4_1	4_1	4_1	4_1
marco 1	–	1_1	1_1	1_1	1_0	0_1	0_1	0_0	0_1	0_0	0_0	0_1
marco 2	–	–	–	2_1	2_0	2_0	1_1	1_0	1_0	3_1	3_1	3_1
resultado	F	F	✓	F	F	F	F	F	✓	F	✓	✓
fallos	1	2	2	3	4	5	6	7	7	8	8	8

8 fallos, 4 aciertos (33 %). Entre FIFO (9) y LRU (7): una buena aproximación que solo necesita un bit por página.

entra acierto F fallo frío F fallo de capacidad

Páginas sucias y otras decisiones

- Página modificada: hay que escribirla al disco antes de expulsarla. Limpia: basta con olvidarla.
- El hardware marca el **bit sucio** *(dirty bit)*; en RISC-V es el bit D.
- El reloj modificado prefiere páginas *no usadas y limpias*.

usada	sucia	al expulsar
0	0	primera opción: gratis
0	1	escribir al disco
1	0	se usó hace poco
1	1	la peor opción

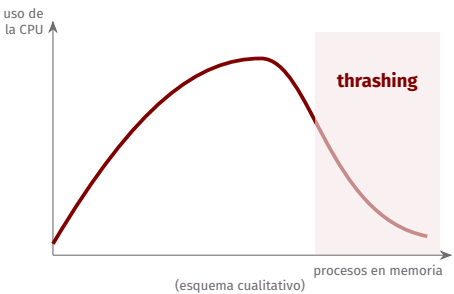
Paginación por demanda *(demand paging)*

Traer una página solo cuando se usa. O **precargar** *(prefetching)*: si se pidió P, traer también $P + 1$.

Agrupar escrituras

Muchas páginas sucias en una sola escritura grande: el disco lo agradece.

Thrashing



- Si la memoria que los procesos usan *activamente* no cabe, cada acceso falla.
- Los procesos pasan la vida esperando al disco. La CPU queda casi ociosa aunque hay mucho trabajo: es el **thrashing**.
- Remedios:
  - **control de admisión**: correr menos procesos a la vez;
  - Linux: el **OOM killer** mata a un proceso que usa mucha memoria.

Conjunto de trabajo *(working set)*

$W(t, \Delta)$: páginas distintas usadas en las últimas Δ referencias (Denning, 1968). Con $\Delta = 4$ y nuestra secuencia:

	1	2	3	4	5	6	7	8	9	10	11	12
ref	0	1	0	2	3	0	1	4	0	3	4	0
$\lvert W \rvert$	1	2	2	3	4	3	4	4	3	4	3	3

- El conjunto de trabajo llega a 4 páginas varias veces, pero solo hay 3 marcos: de ahí salen los fallos de capacidad de FIFO, LRU y el reloj.
- Idea de Denning: un proceso solo corre si su conjunto de trabajo cabe entero.

Regla práctica

Si la suma de los conjuntos de trabajo supera la RAM, aparece el thrashing. Hay que sacar procesos completos, no páginas sueltas.

SECCIÓN 03

Experimento: presión de memoria

03

`memtouch.c`: tocar N MB una y otra vez

code/week-08/memtouch.c

```c
char *buf = mmap(NULL, size,
    PROT_READ | PROT_WRITE,
    MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
madvise(buf, size,
        huge ? MADV_HUGEPAGE : MADV_NOHUGEPAGE);
for (int p = 1; p <= passes; p++) {
    getrusage(RUSAGE_SELF, &before);
    double t0 = now_s();
    for (size_t i = 0; i < size; i += page)
        buf[i] = (char)p; // queda sucia
    double dt = now_s() - t0;
    getrusage(RUSAGE_SELF, &after);
    // imprime dt, MB/s, ru_minflt, ru_majflt
}
```

- Una escritura por página: lo mínimo para que el kernel tenga que darle un marco.
- **Fallo menor**: se resuelve sin disco (página nueva en ceros, o aún en RAM).
- **Fallo mayor**: hubo que leer del disco, por ejemplo del swap.
- Contenedor con `--memory=256m`. Docker agrega por defecto otros 256 MB de swap.

Resultado: 200 MB caben, 400 MB no

bash

```
$ ./memtouch 200 2
PID 13: 200 MB, páginas de 4096 bytes, 51200 páginas
 pasada  segundos      MB/s  f.menores  f.mayores
      1     0.029      6856      51204          1
      2     0.001    209662          0          0
$ ./memtouch 400
PID 18: 400 MB, páginas de 4096 bytes, 102400 páginas
 pasada  segundos      MB/s  f.menores  f.mayores
      1     0.117      3424     102406          3
      2     1.068       375      76768      25646
      3     1.074       373      76667      25744
      4     1.086       368      76377      26038
```

- 200 MB: un fallo menor por página en la 1.ª pasada; luego, ninguno.
- 400 MB: desde la 2.ª pasada, 500 veces más lento.
- $25\,646 + 76\,768 \approx 102\,400$: fallan todas. Del swap se traen varias vecinas a la vez.

El peor caso de LRU, en vivo: siempre se necesita la página que acaba de salir. Unos 41 µs de pasada por cada fallo mayor.

Qué muestra `vmstat` y qué pasa sin swap

bash

```
$ vmstat 1|awk '{print $3,$7,$8}'
---swap--
swpd si so
1092 72 85
1092 0 0
156796 153200 304916
169100 379392 391960
163936 381444 371152
```

`si`/`so`: KB/s desde y hacia el swap con `memtouch 400`.

Ojo: `vmstat` ve toda la VM de Docker, no el límite del contenedor (ese está en `/sys/fs/cgroup`).

bash

```
$ docker run --rm -it --memory=256m \
    --memory-swap=256m ...
$ ./memtouch 400
PID 13: 400 MB, páginas de 4096 bytes, 102400 páginas
 pasada  segundos      MB/s  f.menores  f.mayores
Killed
$ echo $?
137
$ grep oom /sys/fs/cgroup/memory.events
oom 1
oom_kill 1
```

`--memory-swap` es memoria + swap: aquí, cero swap. El OOM killer mata al proceso: $137 = 128 + 9$ (`SIGKILL`). Con el swap por defecto muere `memtouch 600`: supera 512 MB.

SECCIÓN 04

Sistemas reales: VAX/VMS y Linux

04

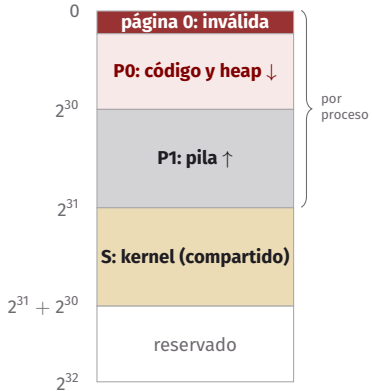
El VAX-11/780



VAX-11/780 de DEC. Joe Mabel, CC BY-SA 3.0, Wikimedia Commons.

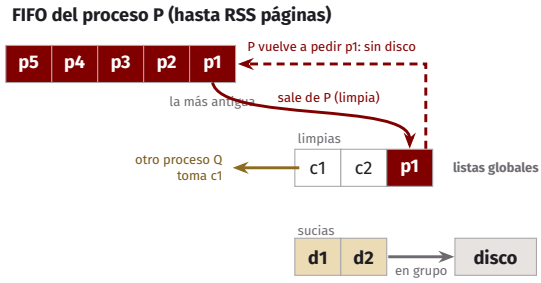
- La minicomputadora de 32 bits de DEC, del tamaño de varios armarios.
- Ahí nacieron VMS y la memoria virtual de BSD Unix.
- OSTEP la usa como caso de estudio de un sistema de VM completo.

VAX/VMS (1978): segmentación + paginación



- 32 bits, páginas de 512 bytes: VPN de 23 bits y offset de 9. Los 2 bits altos eligen el segmento.
- Una tabla lineal por segmento (P0, P1), con registros base y límite: el hueco entre heap y pila no necesita PTEs.
- Las tablas de P0 y P1 viven en la memoria *virtual* del kernel (S): se pueden mandar al swap.
- La página 0 es inválida: desreferenciar `NULL` da un fallo. Igual que hoy.

VMS: FIFO segmentado con segunda oportunidad



- La PTE del VAX no tiene bit de referencia.
- Cada proceso tiene un tope de páginas, su RSS *(resident set size)*. Al pasarse, su página más antigua va a una lista global.
- Otros procesos toman marcos de la lista de limpias.
- Si P la vuelve a pedir antes, la recupera: segunda oportunidad.
- Las sucias se escriben **agrupadas** *(clustering)*.

Dos trucos que todos copiaron

Demand zeroing

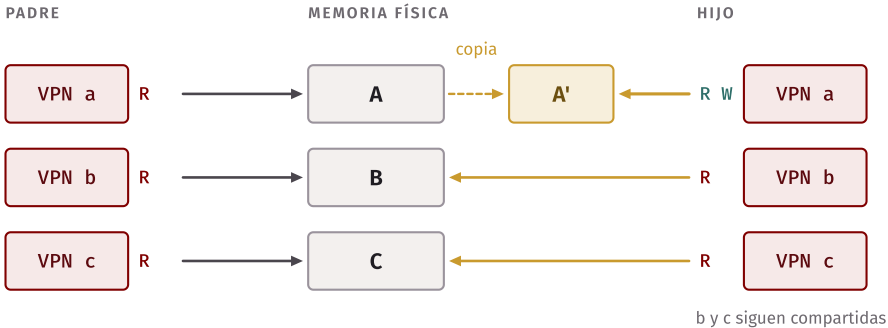
La página nueva no se llena de ceros al pedirla. Recién en el primer acceso el kernel busca un marco y lo pone en cero. Si nunca se usa, no costó nada.

Copy-on-write *(COW)*

"Copiar" es mapear la página en ambos procesos como *solo lectura*. Se copia recién cuando alguien escribe.

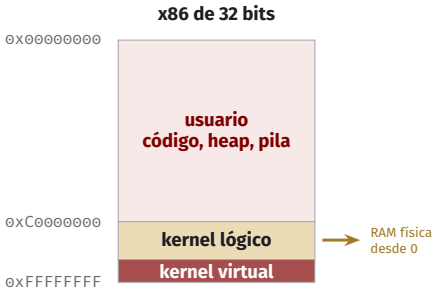
Con `fork()` y luego `exec()` casi nada se copia: el hijo descarta su espacio de inmediato.

Copy-on-write con `fork()`



4 El kernel copia A en un marco nuevo, A', y se lo da al hijo con W.

Linux: el espacio de direcciones



- El kernel está mapeado en todos los procesos: entrar al kernel no cambia de tabla de páginas (salvo con KPTI, al final).
- **Lógico** (`kmalloc`): es la RAM física con un desplazamiento fijo, contigua y nunca va al swap. Sirve para DMA.
- **Virtual** (`vmalloc`): páginas sueltas que parecen contiguas.
- En x86-64: 48 bits y 4 niveles. Usuario abajo (128 TB), kernel arriba.

Linux: páginas enormes *(huge pages)*

- Con 4 KB, una TLB de 64 entradas cubre $64 \times 4\,\text{KB} = 256\,\text{KB}$.
- Con páginas de 2 MB cubre $64 \times 2\,\text{MB} = 128\,\text{MB}$. Menos fallos de TLB y recorridos más cortos.
- **THP** *(transparent huge pages)*: el kernel las usa solo, sin cambiar el programa.
- Costo: fragmentación interna y swap más caro; algunas bases de datos piden desactivarlas.

bash

```
$ cat /sys/kernel/mm/transparent_hugepage/enabled
[always] madvise never
$ ./memtouch 200 1 | tail -1
      1     0.028      7226      51202          1
$ ./memtouch 200 1 huge | tail -1
      1     0.003     64223        102          0
```

200 MB / 2 MB = 100 páginas grandes: 102 fallos en vez de 51 202, y 9 veces más rápido.

Linux: page cache y 2Q

- La RAM libre no se desperdicia: guarda páginas de archivos leídos y escritos *(page cache)*.
- Problema de LRU: un `cat` de un archivo gigante expulsa todo lo útil.
- Linux usa una variante de **2Q**: la primera vez, la página va a la lista *inactiva*. Si se vuelve a usar, sube a la *activa*.
- Se expulsa desde la inactiva, con un algoritmo tipo reloj.



bash

```
$ grep '(file)' /proc/meminfo
Active(file):     620384 kB
Inactive(file):  4426808 kB
```

Seguridad: ASLR, Meltdown y KPTI

- **ASLR**: código, pila y heap en direcciones al azar (lo vimos en la semana 5). El kernel también (KASLR).
- **Meltdown** (2018): la ejecución especulativa de algunas CPU permitía leer memoria del kernel *desde modo usuario*, porque estaba mapeada en cada proceso.
- **KPTI** (*kernel page-table isolation*): dos tablas por proceso. En modo usuario, casi nada del kernel está mapeado. Costo: cambiar de tabla en cada trap.

xv6 ya lo hace así

Una tabla para el kernel y otra por proceso. Solo `trampoline` está en ambas: `kernel/trampoline.S` cambia `satp` en cada trap.

bash

```
$ cat /sys/devices/system/cpu/\
    vulnerabilities/meltdown
Not affected
```

La CPU ARM del contenedor no tiene ese problema.

SECCIÓN 05

Fallos de página en xv6-riscv

05



xv6 no tiene swap, pero sí fallos de página

kernel/trap.c: usertrap()

```
1   if (r_scause() == 8) {
2     // system call
3     // ...
4     syscall();
5   } else if ((which_dev = devintr()) != 0) {
6     // ok
7   } else if ((r_scause() == 15 || r_scause() == 13) &&
8              vmfault(p->pagetable, p->sz, r_stval(),
9                      (r_scause() == 13) ? 1 : 0) != 0) {
10    // page fault on lazily-allocated page
11  } else {
12    // ... imprime scause, sepc y stval
13    setkilled(p);
14  }
```

- `scause` dice la causa del trap: 8 llamada al sistema, 13 fallo al leer, 15 fallo al escribir.
- `stval` trae la dirección virtual que falló.
- Si `vmfault` no puede arreglarlo, el proceso muere. Un fallo al *ejecutar* (12) no se atiende: muere directo.

Reserva perezosa: `vmfault` en `kernel/vm.c`

kernel/vm.c: vmfault(pagetable, psz, va, read)

```
1   if (va >= psz)
2     return 0;
3   va = PGROUNDDOWN(va);
4   if (ismapped(pagetable, va)) {
5     return 0;
6   }
7   mem = (uint64)kalloc();
8   if (mem == 0)
9     return 0;
10  memset((void *)mem, 0, PGSIZE);
11  if (mappages(pagetable, va, PGSIZE, mem,
12               PTE_W | PTE_U | PTE_R) != 0) {
13    // kfree(mem) y return 0
14  }
15  return mem;
```

1. `sbrklazy(n)` solo suma `n` a `sz` (`sbrk` y `malloc` reservan de inmediato).
2. El primer acceso a esa memoria falla: `scause` 13 o 15.
3. Bajo `sz` y sin mapear: un marco nuevo en ceros (demand zeroing) y a reintentar.
4. `read` no se usa. `copyin`/`copyout` también la llaman.

Lo que viene en el bono: copy-on-write en xv6

Hoy

- `kfork()` (`kernel/proc.c`) llama a `uvmcopy()`, que hace `kalloc` y `memmove` de cada página del padre.
- Una escritura en una página mapeada pero sin `PTE_W` llega a `vmfault`, que retorna 0 porque ya está mapeada: el proceso muere.

Lab `cow` del MIT

1. `uvmcopy`: compartir los marcos, quitar `PTE_W` en ambos y marcarlos con un bit RSW.
2. En `usertrap`, `scause` 15 sobre una página COW: copiarla y mapearla escribible.
3. Contar referencias por marco en `kalloc.c`: liberar solo con la última.

Bono: semanas 11 a 16

El mismo mecanismo que hoy usa xv6 para la reserva perezosa, aplicado a `fork()`. Requiere entender bien `usertrap`, la PTE y `kalloc`.

SECCIÓN 06

Repaso para el parcial

06

Qué entra: semanas 1 a 8

Virtualización de la CPU

S1–2 Procesos, estados, `fork`/`exec`/`wait`, ejecución directa limitada, traps y cambio de contexto.

S3 FIFO, SJF, STCF, RR; tiempos de retorno y de respuesta. MLFQ.

S4 Lotería, stride, CFS, multiprocesadores.

Virtualización de la memoria

S5 Espacio de direcciones, `malloc`/`free`, base y límite, segmentación.

S6 Espacio libre, paginación, VPN y offset.

S7 TLB, tablas multinivel, Sv39.

S8 Swap, fallos de página, políticas de reemplazo, VMS y Linux.

Formato: problemas de cálculo como los de los simuladores, lectura de código y preguntas cortas de diseño. Detalles en `exams/README.md`.

Tipos de problemas (1): planificación y base y límite

Tiempo de retorno promedio

A, B y C llegan en $t = 0$ y duran 6, 2 y 4. ¿FIFO (A, B, C) o SJF?

FIFO: terminan en 6, 8, 12
$\Rightarrow (6 + 8 + 12)/3 \approx 8.67$.
SJF: B, C, A terminan en 2, 6, 12 $\Rightarrow 20/3 \approx 6.67$.

Traducir con base y límite

Base $= 32$ KB, límite $= 4$ KB. ¿A dónde van las direcciones virtuales 1000 y 5000?

$1000 < 4096$: física $= 32\,768 + 1000 = 33\,768$.
$5000 \geq 4096$: excepción, el kernel termina al proceso.

Cómo se responde: con el procedimiento a la vista (Gantt, bits separados, comparación con el límite). Un número solo no alcanza.

Tipos de problemas (2): paginación, TLB y reemplazo

Traducir con una tabla de páginas

Espacio de 16 KB, páginas de 1 KB, y la VPN 2 está en el marco 7. Traduzcan `0x0A4C`.

14 bits = 4 de VPN + 10 de offset: `0010 1001001100`.
VPN = 2, offset = `0x24C`. Física = $7 \cdot 1024 + \texttt{0x24C} = \texttt{0x1E4C}$.

Tiempo efectivo con TLB

TLB con 98 % de aciertos, tabla de 2 niveles, 100 ns por acceso a memoria.

Acierto: 1 acceso. Fallo: 2 de la tabla + 1 del dato.
$0.98 \cdot 100 + 0.02 \cdot 300 = 104$ ns.

Reemplazo, para practicar: OPT, FIFO y LRU a mano con `0 1 2 0 1 3 0 3 1 2 1` y 3 marcos (pregunta de repaso). Verifiquen con `./paging`.

Cómo estudiar

1. Las **preguntas de repaso** de las guías de las semanas 1 a 8.
2. Los simuladores de `ostep-homework` con semillas nuevas (`-s 7`, `-s 42`…). Resuelvan a mano y *después* verifiquen con `-c`.
3. Rehagan los ejemplos resueltos de las diapositivas sin mirar la respuesta.
4. Lean el código de xv6 que vimos: `struct proc`, `usertrap`, `scheduler`, `walk`, `vmfault`.

El error más común

Confundir dirección virtual con física, o bits con bytes. Escriban siempre las unidades y cuántos bits tiene cada campo.

Resumen

- El **swap** usa el disco para dar más memoria de la que hay. El **bit presente** (en RISC-V, *V*) dice si la página está en RAM; si no, hay un **fallo de página**.
- Un fallo cuesta miles de accesos a RAM: los fallos deben ser rarísimos. El demonio de swap libera páginas entre las marcas **LW** y **HW**.
- Con nuestra secuencia: OPT 5 fallos, LRU 7, reloj 8, FIFO 9, RAND 7.45 en promedio. FIFO sufre la **anomalía de Belady**; LRU y OPT no.
- El **reloj** aproxima LRU con un bit por página; prefiere páginas limpias. Si el conjunto de trabajo no cabe: **thrashing**.
- VMS: FIFO segmentado, demand zeroing y copy-on-write. Linux: page cache con 2Q, huge pages y KPTI.
- xv6 no tiene swap, pero atiende fallos (`scause` 13 y 15) para la reserva perezosa.

Para la próxima clase

- **Sesión B: examen parcial**, semanas 1 a 8.
- Laboratorio: `paging-policy.py` (preguntas 1–4 del cap. 22); `memtouch` o `mem.c` con `vmstat 1` en un contenedor con `--memory`; ejercicios de traducción en grupo.
- Semana 9: leer el apéndice *Virtual Machine Monitors* y los cap. 26–27 (hilos).

Entregas

P2 (lotería) se entrega esta semana. Se publica P3 (lab `pgtbl`), con entrega en la semana 10. El bono COW empieza en la semana 11.

Pregunta para pensar

Con copy-on-write, ¿cuántas páginas copia `fork()` si el hijo hace `exec()` de inmediato? ¿Y si el padre escribe en todo su heap antes?