

Sistemas Operativos

TLB y tablas de páginas avanzadas

Traducir rápido y con tablas pequeñas

Gerson Garrido

Semana 7

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. La TLB: una caché de traducciones dentro de la MMU
2. Aciertos, fallos, quién los atiende y qué pasa en un cambio de contexto
3. Medir la TLB de esta máquina con `tlb.c`
4. Tablas más pequeñas: páginas grandes, híbrido, multinivel, invertidas
5. xv6-riscv: Sv39, `walk()`, `satp` y `sfence.vma`

> **Lectura**
>
> OSTEP cap. 19 *Paging: Faster Translations (TLBs)* (sesión A) y cap. 20 *Paging: Smaller Tables* (sesión B).

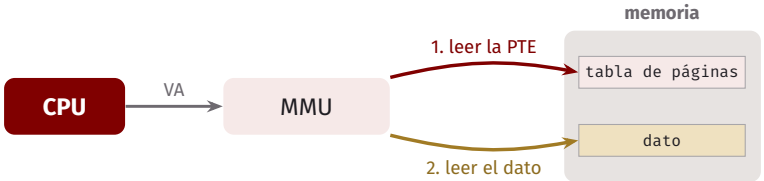
SECCIÓN 01

La TLB

01



El problema: la paginación es lenta

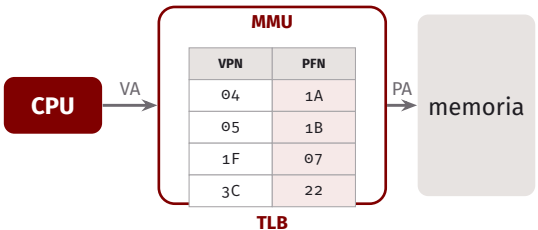


- Con una tabla lineal, cada acceso del programa se convierte en dos: primero la PTE, después el dato. Incluso buscar *instrucciones*.
- El programa correría a la mitad de velocidad. Y la tabla ocupa megabytes.

Las dos preguntas de hoy

¿Cómo traducir sin ir a memoria cada vez? ¿Cómo lograr que la tabla sea pequeña?

La idea: una caché de traducciones

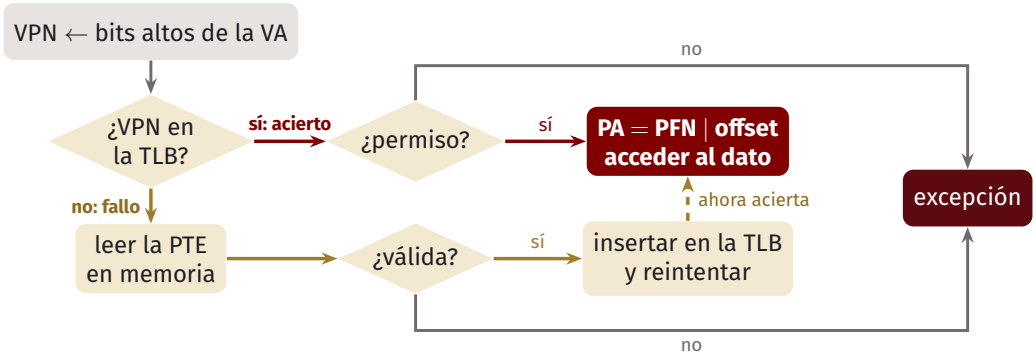


TLB ***(translation lookaside buffer)***

Una caché pequeña y muy rápida, dentro de la MMU, que guarda traducciones recientes VPN $\rightarrow$ PFN.

- Decenas a pocos miles de entradas.
- Se busca en paralelo en todas las entradas *(fully associative)*: en un ciclo.
- Si la traducción está, no se lee la tabla de páginas.

Algoritmo de control: acierto y fallo



El fallo *(TLB miss)* paga un acceso extra: la PTE está en PTBR + VPN × tamaño(PTE).

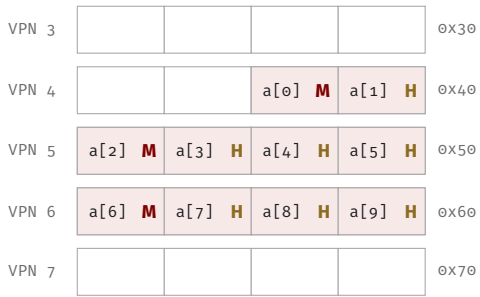
El mismo algoritmo, en pseudocódigo

hardware (pseudocódigo)

```
1  vpn = (va & VPN_MASK) >> SHIFT;
2  (hit, e) = tlb_lookup(vpn);
3  if (hit) {
4      check(e.prot);    // o PROTECTION_FAULT
5      pa = (e.pfn << SHIFT) | (va & OFFSET_MASK);
6      access(pa);
7  } else {
8      pte = read_memory(PTBR + vpn * PTE_SIZE);
9      if (!pte.valid)
10         raise(SEGFAULT);
11     check(pte.prot);
12     tlb_insert(vpn, pte.pfn, pte.prot);
13     retry_instruction();
14 }
```

- **Líneas 3–6**: el camino rápido. Ni un acceso de más.
- **Línea 8**: el fallo lee la PTE desde la tabla en memoria.
- **Línea 13**: la instrucción se repite y ahora acierta.
- Como toda caché, funciona si los fallos son raros.

Ejemplo: recorrer un arreglo



VA de 8 bits, páginas de 16 bytes, `int a[10]` desde `0x48`. M: fallo, H: acierto *(hit)*.

TLB

VPN	PFN
4	0E
5	09
6	1C

3 fallos, 7 aciertos: tasa de aciertos del 70 %.

Solo se falla con el *primer* elemento de cada página.

Localidad: por qué la TLB funciona

Localidad espacial

Si se usa una dirección, pronto se usarán las *cercanas*. En el arreglo: un fallo por página y el resto aciertos.

Localidad temporal

Si se usa una dirección, pronto se volverá a usar. Una segunda pasada por el arreglo: 10 aciertos de 10 (si la TLB conserva las 3 entradas).

Páginas de	Fallos	Aciertos
16 B	3	70 %
32 B	2	80 %
4 KB	1	90 %

Con páginas más grandes, el mismo arreglo cruza menos páginas.

Lo mismo pasa con el código: las instrucciones de un bucle están en pocas páginas.

Con 32 B y `a` en `0x48`: VPN 2 y 3. Con 4 KB: una sola.

¿Cuánto se gana? Tiempo efectivo de acceso

Supongamos: buscar en la TLB cuesta ≈ 0, un acceso a memoria cuesta **100 ns**.

- Acierto: 1 acceso $=$ 100 ns.
- Fallo (tabla lineal): PTE $+$ dato $=$ 200 ns.
- Con tasa de aciertos h:

$$T = h \cdot 100 + (1 - h) \cdot 200$$

h	T
0 % (sin TLB)	200 ns
90 %	110 ns
99 %	101 ns

Conclusión

Con un 99 % de aciertos, la paginación cuesta casi lo mismo que no tenerla. Los programas reales suelen superar el 99 %; los que no, sufren (ley de Culler: *la RAM no siempre es RAM*).

¿Quién atiende el fallo de TLB?

El hardware *(hardware-managed)*

- El hardware conoce el formato de la tabla y dónde está (`CR3` en x86, `satp` en RISC-V).
- En un fallo, la MMU recorre la tabla *(page walk)*, inserta la entrada y reintenta.
- El kernel no se entera, salvo que la PTE sea inválida. Así son x86, ARM y RISC-V.

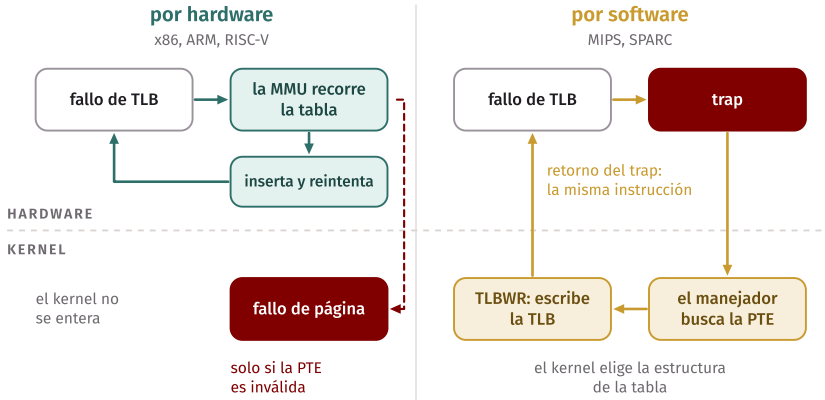
El software *(software-managed)*

- El hardware solo levanta una excepción.
- El kernel busca la PTE en *su* estructura, escribe la TLB con instrucciones privilegiadas y retorna del trap.
- MIPS, SPARC. Ventaja: flexibilidad; el kernel elige la estructura de la tabla.

Para el resto del curso

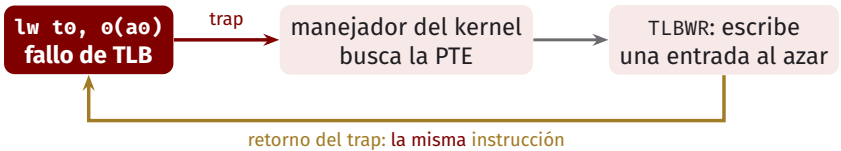
xv6-riscv usa el modelo de hardware: la MMU de RISC-V recorre sola la tabla Sv39.

Dónde se resuelve el fallo



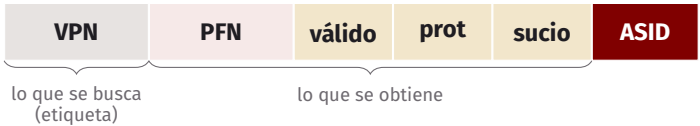
Por hardware, el kernel solo ve los fallos de página. Por software, cada fallo de TLB es un trap.

TLB por software: dos detalles delicados



1. **Retorno distinto**: tras una llamada al sistema se sigue con la instrucción *siguiente*; tras un fallo de TLB se repite *la misma*. El hardware debe guardar el PC correcto.
2. **Fallos en cadena**: si el código del manejador también falla en la TLB, nunca termina. Solución: el manejador vive en memoria no traducida, o en entradas fijas *(wired)* que nunca se reemplazan.

¿Qué hay en una entrada de la TLB?



- **válido** de la TLB $\neq$ válido de la PTE: aquí significa “esta entrada tiene una traducción”, no “la página está asignada”.
- **prot** y **sucio** se copian de la PTE: el acierto también verifica permisos.
- **ASID**: a qué espacio de direcciones pertenece. Lo vemos ahora.

Cambio de contexto: dos procesos, la misma VPN

TLB

VPN	PFN	válido	
10	100	1	**P1**
10	170	1	**P2**

La TLB no sabe de quién es cada entrada. Si P2 corre, ¿la VPN 10 va al marco 100 o al 170?

Opción 1: vaciar la TLB *(flush)*

En cada cambio de contexto, poner todos los bits válidos en 0. Simple, pero el proceso nuevo empieza con puros fallos.

Opción 2: ASID

Un identificador de espacio de direcciones en cada entrada. La TLB solo acierta si coincide con el ASID actual.

ASID: varios procesos comparten la TLB

VPN	PFN	válido	ASID
10	100	1	1
10	170	1	2
50	101	1	1
50	101	1	2

código compartido

- Al cambiar de proceso, el kernel escribe el ASID actual en un registro privilegiado.
- Con ASID 1, la VPN 10 da el marco 100; con ASID 2, el 170.
- Dos procesos pueden mapear el mismo marco (101): código compartido de una biblioteca, por ejemplo.
- Los ASID son pocos (8 bits en MIPS, 16 en RISC-V): con más procesos, el kernel recicla ASIDs y vacía las entradas viejas.

¿Qué entrada se reemplaza?

LRU ***(least recently used)***

Sacar la entrada que lleva más tiempo sin usarse. Aprovecha la localidad.

Aleatoria ***(random)***

Sacar cualquiera. Simple en hardware y sin casos patológicos.

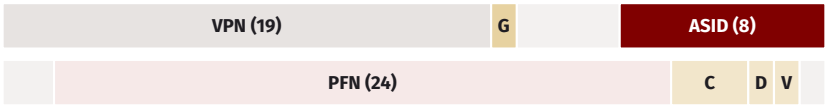
El caso feo de LRU: TLB de 3 entradas, bucle sobre 4 páginas.

0	1	2	3	0	1	2	3
M	M	M	M	M	M	M	M

LRU siempre saca justo la página que se va a necesitar: 0 % de aciertos. Una política aleatoria acierta a veces.

Volveremos a estas políticas, en grande, cuando estudiemos el reemplazo de páginas en disco (semana 8).

Una entrada real: MIPS R4000



Entrada simplificada de 64 bits; en gris, bits sin usar.

- VA de 32 bits, páginas de 4 KB: el VPN tendría 20 bits, pero la mitad alta es del kernel: bastan 19.
- PFN de 24 bits: hasta $2^{24} \times 4$ KB $= 64$ GB de memoria física.
- **G** (global): la entrada vale para todos los procesos; se ignora el ASID.
- **C**: cómo se cachea la página. **D**: sucia. **V**: válida.
- TLB por software: instrucciones `TLBP`, `TLBR`, `TLBWI`, `TLBWR` (privilegiadas).
- Registro *wired*: cuántas entradas reserva el kernel para sí.

SECCIÓN 02

Medir la TLB

02

`tlb.c`: tocar *N* páginas una y otra vez

code/week-07/tlb.c (resumido)

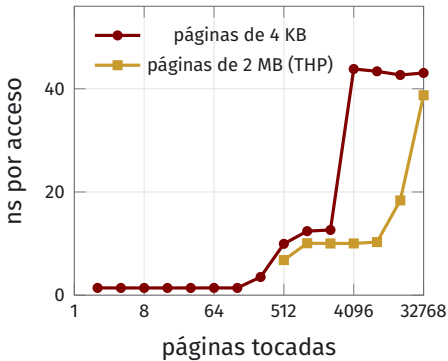
```
1  char *buf = mmap(...);     // N páginas
2  madvise(buf, len, MADV_NOHUGEPAGE);
3  memset(buf, 1, len);       // tocar todas
4  // cada página guarda dónde está la siguiente
5  for (long i = 0; i < pages; i++)
6    *(long *)(buf + slot(i, pgsz)) =
7        slot((i + 1) % pages, pgsz);
8
9  double t0 = now_ns();      // clock_gettime
10 for (long t = 0; t < trials; t++)
11   for (long i = 0; i < pages; i++)
12     cur = *(volatile long *)(buf + cur);
13 ns = (now_ns() - t0) / (trials * pages);
```

- Fijado a la CPU 0 con `sched_setaffinity`: cada núcleo tiene su TLB.
- `MADV_NOHUGEPAGE`: páginas de 4 KB de verdad.
- Se tocan las páginas antes de medir: sin fallos de página en el tiempo.
- Cada lectura depende de la anterior: la CPU no puede solapar los fallos.
- `volatile`: el compilador no puede borrar el bucle.

Resultados en esta máquina

make bench

```
$ ./run-tlb.sh
páginas     memoria    ns 4KB    ns THP  KB en THP
      2        8 KB      1.41      1.40          0
    128      512 KB      1.40      1.40          0
    256     1024 KB      3.52      3.52          0
    512     2048 KB      9.95      6.80       2048
   2048     8192 KB     12.64     10.02       8192
   4096    16384 KB     43.84     10.03      16384
   8192    32768 KB     43.39     10.29      32768
  16384    65536 KB     42.68     18.37      65536
  32768   131072 KB     43.07     38.75     131072
```



ns por acceso (mínimo de 5 repeticiones). Filas seleccionadas.

¿Qué nos dice la curva?

- Hasta 128 páginas: **1.4 ns**. Todo acierta en la TLB más rápida.
- De 512 a 2048 páginas: ≈**10–12 ns**, y casi igual con THP (solo 1 a 4 páginas grandes). Este escalón no lo explica el tamaño de página.
- Desde 4096 páginas (16 MB), con 4 KB: ≈**43 ns**, 30 veces un acierto. Con THP, los mismos datos siguen en 10 ns: la diferencia es la traducción.
- Con THP el salto llega a 64–128 MB: 4 a 8 veces más tarde. En teoría cada entrada cubre 512 veces más memoria; aquí no sabemos qué lo limita.

Con honestidad: cómo se midió

Linux aarch64 en la máquina virtual de Docker, sobre una Apple M4 Max: *dos* traducciones y núcleos de dos tipos. Los números cambian entre corridas.

SECCIÓN 03

Tablas más pequeñas

03

El otro problema: el tamaño de la tabla

VA de 32 bits, páginas de 4 KB, PTE de 4 B

- offset: 12 bits; VPN: 20 bits.
- 2^{20} PTE $\times$ 4 B $=$ 4 MB por proceso.
- 100 procesos: 400 MB solo en tablas.

Sv39: VA de 39 bits, PTE de 8 B

- VPN: 27 bits.
- $2^{27} \times 8$ B $=$ 1 GB por proceso.

¿Qué hay en esos megabytes?

Casi todo son PTE inválidas: el hueco enorme entre el heap y la pila. Un programa típico usa unas pocas decenas de páginas al principio y unas pocas al final del espacio.

Primera idea: páginas más grandes

VA de 32 bits, PTE de 4 B:

Página	VPN	Tabla
4 KB	20 bits	4 MB
16 KB	18 bits	1 MB
4 MB	10 bits	4 KB

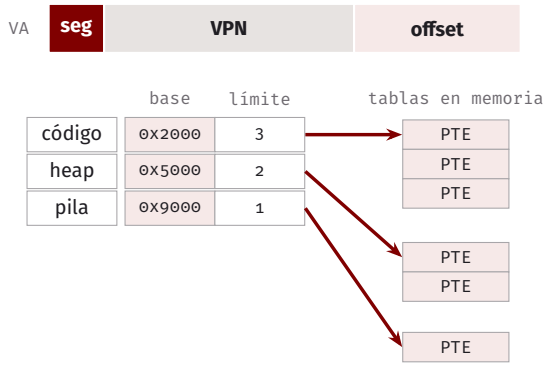
El costo: fragmentación interna

Una pila de 5 KB ocupa una página entera. Con páginas de 4 MB, cada región pequeña desperdicia megabytes.

En la práctica: varios tamaños

Páginas de 4 KB por defecto y páginas grandes (*superpages*: 2 MB o 1 GB en x86-64 y RISC-V) para regiones grandes, como la de la medición anterior o una base de datos. Ayudan a la TLB más que a la tabla.

Segunda idea: híbrido de segmentación y paginación



- Una tabla de páginas por segmento.
- La base apunta a la tabla; el límite dice cuántas PTE tiene.
- El hueco entre heap y pila no tiene PTE: no ocupa nada.
- Problemas: un heap grande y disperso desperdicia igual, y las tablas vuelven a ser de tamaño variable: fragmentación externa.

Tercera idea: paginar la tabla de páginas

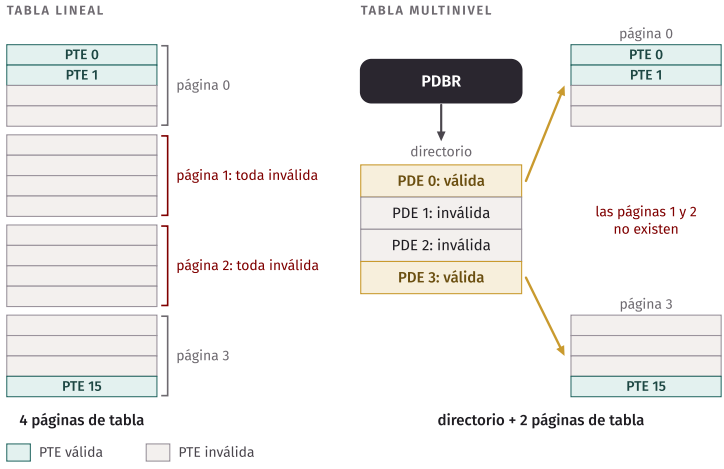


Tabla multinivel: cómo se parte la dirección

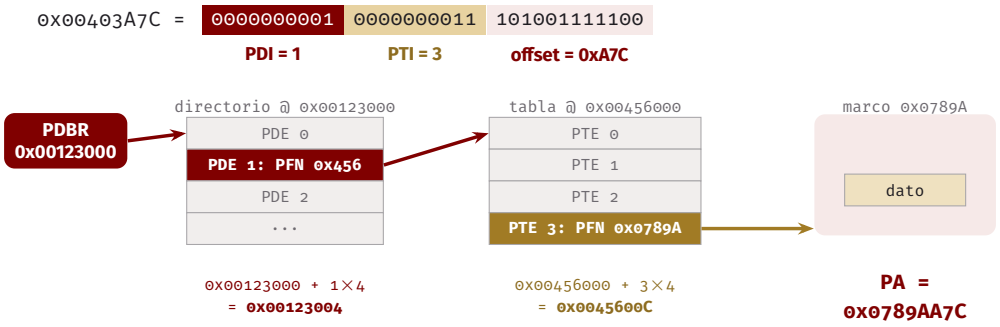


- Una página de 4 KB guarda $4096/4 = 2^{10}$ PTE: el índice de la tabla tiene 10 bits.
- Los otros 10 bits eligen una de las 1024 **PDE** *(page directory entry)*. El directorio también cabe en una página.
- Cada PDE tiene un bit válido y el PFN de una página de PTE. PDE inválida: esa página de PTE no existe.
- El registro de la tabla ahora apunta al directorio (PDBR).

Así era x86 de 32 bits

Esquema 10/10/12, con el registro `CR3` apuntando al directorio.

Recorrido de dos niveles: VA = 0x00403A7C



Paso 4: PA = PFN $\ll$ 12 | offset. Leer el dato (acceso 3). Si la TLB hubiera tenido la VPN 0x00403, solo el acceso 3.

¿Cuánto se ahorra?

Un proceso usa 3 páginas de código en `0x00400000`, el heap en `0x08000000` y la pila en `0xBFFFF000`.

- Directorio: **1** página (siempre).
- Código: PDI $=$ 1. Heap: PDI $=$ 0x20. Pila: PDI $=$ 0x2FF. Tres PDE válidas $\Rightarrow$ **3** páginas de PTE.
- Total: 4 páginas $=$ 16 KB, contra 4 MB de la lineal: 256 veces menos.

code/week-07

```
$ ./ptcalc 32 4096 4 0x00400000 \
    0x00401000 0x00402000 \
    0x08000000 0xbffff000
...
niveles: 2   bits por índice: 10 10
tabla lineal: 2^20 PTE x 4 B = 4 MB
...
nivel 1: 1 página(s)
nivel 0: 3 página(s)
multinivel: 4 páginas x 4096 B = 16 KB
```

Peor caso: si se usaran *todas* las páginas, la multinivel ocupa 4 MB $+$ 4 KB, un poco *más* que la lineal. Gana con espacios dispersos *(sparse)*.

Ventajas y precio

Ventajas

- El tamaño es proporcional a la memoria usada, no al espacio de direcciones.
- Cada pedazo de la tabla es una página: el kernel la saca de su lista de marcos libres. Sin fragmentación externa.

Precio

- Un fallo de TLB cuesta dos lecturas de tabla en vez de una.
- Más complejidad en el hardware o en el kernel.

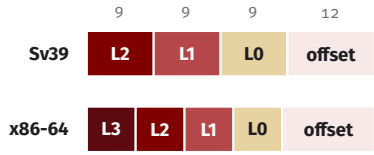
Intercambio tiempo/espacio *(time-space trade-off)*

Menos memoria a cambio de fallos más caros. Si la TLB acierta casi siempre, el precio casi no se nota.

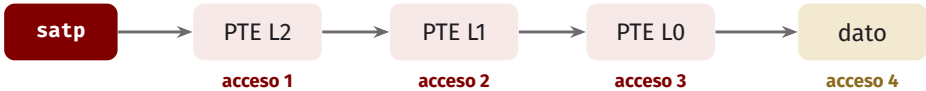
Más de dos niveles

Regla: cada nodo de la tabla debe caber en una página.

- Sv39: páginas de 4 KB, PTE de 8 B $\Rightarrow$ 512 PTE por página $\Rightarrow$ 9 bits por nivel.
- VPN de 27 bits $= 9 + 9 + 9$: **tres** niveles.
- Con dos niveles, el directorio tendría 2^{18} entradas: 2 MB. No cabe.
- x86-64 usa 48 bits: $9 + 9 + 9 + 9 + 12$, **cuatro** niveles (y cinco en los procesadores más nuevos).



El costo de un fallo de TLB con tres niveles



Fallo: **3 lecturas de tabla** $+$ **1 dato** $=$ **4 accesos**. Acierto: 1.

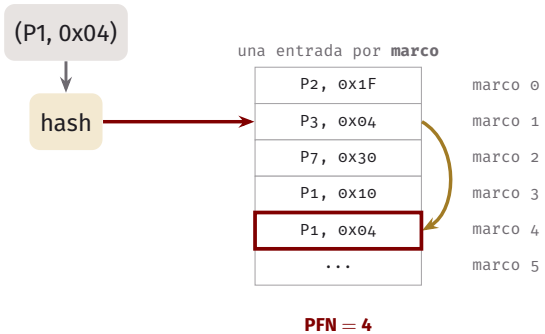
Con 100 ns por acceso:

$$T = h \cdot 100 + (1 - h) \cdot 400$$

h	lineal	3 niveles
90 %	110 ns	130 ns
99 %	101 ns	103 ns

En la práctica, las cachés de datos guardan las PTE más usadas y las CPU tienen cachés especiales para los niveles altos del recorrido: el fallo cuesta menos que 4 accesos a la RAM.

Tablas invertidas



- Una sola tabla para todo el sistema: una entrada por marco físico, con el proceso y la VPN que lo usan.
- Recorrerla entera sería lento: se usa una tabla hash. Las colisiones forman una cadena.
- El índice de la entrada encontrada *es* el PFN.
- Tamaño proporcional a la memoria física. Ejemplo: PowerPC.

Comparación

Esquema	Tamaño de la tabla	Costo
Lineal	$2^{\text{bits del VPN}}$ PTE, siempre	1 lectura de tabla por fallo
Páginas grandes	menos PTE	fragmentación interna
Híbrido	según el tamaño de cada segmento	tablas de tamaño variable
Multinivel	según la memoria *usada*	1 lectura por nivel en cada fallo
Invertida	según la memoria física	hash, colisiones; compartir es difícil

Lo que usa casi todo el mundo

Tablas multinivel con TLB por hardware y algunas superpages: x86-64, ARM64 y RISC-V. Es lo que tiene xv6.

SECCIÓN 04

En xv6-riscv: Sv39

04



La dirección virtual en Sv39



kernel/riscv.h

```
1  // extract the three 9-bit page table indices from a virtual address.
2  #define PXMASK          0x1FF // 9 bits
3  #define PXSHIFT(level)  (PGSHIFT + (9 * (level)))
4  #define PX(level, va)   ((((uint64)(va)) >> PXSHIFT(level)) & PXMASK)
```

- `PX(2, va)`: bits 30–38. `PX(0, va)`: bits 12–20.
- Cada página de tabla: 512 PTE de 8 bytes = 4 KB.
- `va = 0x80604567`: L2 = 2, L1 = 3, L0 = 4, offset = `0x567`.
- xv6 usa solo 38 bits (`MAXVA`) para no extender el signo.

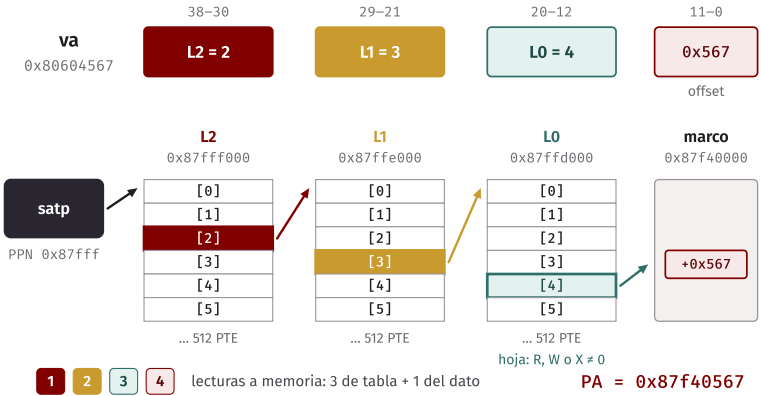
`walk()`: el recorrido, en software

kernel/vm.c

```
1  pte_t *walk(pagetable_t pagetable, uint64 va, int alloc)
2  {
3    if (va >= MAXVA)
4      panic("walk");
5    for (int level = 2; level > 0; level--) {
6      pte_t *pte = &pagetable[PX(level, va)];
7      if (*pte & PTE_V) {
8        pagetable = (pagetable_t)PTE2PA(*pte);
9      } else {
10       if (!alloc || (pagetable = (pde_t *)kalloc()) == 0)
11         return 0;
12       memset(pagetable, 0, PGSIZE);
13       *pte = PA2PTE(pagetable) | PTE_V;
14     }
15   }
16   return &pagetable[PX(0, va)];
17 }
```

- Hace lo mismo que la MMU, pero lo usa el kernel para leer y modificar tablas.
- Niveles 2 y 1: si la PTE es válida, bajar a la página que apunta.
- Si no, con `alloc` crea una página de tabla *vacía*: la tabla crece solo donde hace falta.
- Retorna la *dirección* de la PTE de nivel 0; `mappages()` la llena.

Recorrido Sv39 de 0x80604567



La PTE [4] es una hoja; el dato es el acceso 4. PA ilustrativas: `kalloc()` reparte desde el final de la RAM.

`satp` y `sfence.vma`

kernel/riscv.h

```
#define SATP_SV39 (8L << 60)
#define MAKE_SATP(pagetable) (SATP_SV39 | (((uint64)pagetable) >> 12))
  // the zero, zero means flush all TLB entries.
  asm volatile("sfence.vma zero, zero" ::: "memory");
```

kernel/vm.c: kvminithart()

```
// wait for any previous writes to
// the page table memory to finish.
sfence_vma();
w_satp(MAKE_SATP(kernel_pagetable));
// flush stale entries from the TLB.
sfence_vma();
```



- Modo Sv39 y PPN de la raíz: el PTBR de RISC-V.
- ASID en 0: xv6 no usa ASIDs.
- `trampoline.S` vacía la TLB en *cada* entrada y salida del kernel.

Superpages en Sv39: adelanto de P3

- Una PTE es hoja si tiene R, W o X. Si no, apunta a la siguiente tabla.
- Hoja en el nivel 1: mapea **2 MB** (los 9 bits de L0 pasan al offset). En el nivel 2: **1 GB**.
- En teoría, una entrada de TLB cubre 512 páginas. En la medición con THP, el salto llegó 4 a 8 veces más tarde.
- `walk()` siempre baja hasta el nivel 0. En **P3** (lab `pgtbl` del MIT) agregarán superpages a xv6.

kernel/vm.c: freewalk()

```
1  for (int i = 0; i < 512; i++) {
2    pte_t pte = pagetable[i];
3    if ((pte & PTE_V) && (pte & (PTE_R | PTE_W | PTE_X)) == 0) {
4      // this PTE points to a lower-level page table.
5      uint64 child = PTE2PA(pte);
6      freewalk((pagetable_t)child);
```

Laboratorio: paginación y medición de la TLB

1. `paging-linear-translate.py` (cap. 18): preguntas 1–5. Calculen el tamaño de la tabla lineal y traduzcan a mano.
2. `code/week-07`: `make bench`. ¿Dónde están los saltos en *su* computadora? ¿Qué cambia con páginas grandes (columna THP)?
3. `paging-multilevel-translate.py` (cap. 20): preguntas 1–3. Verifiquen la separación de cada dirección con `./ptcalc 15 32 1 <va>`.

Entregable

La tabla y el gráfico de `tlb`, con una estimación del alcance de la TLB y una explicación de los saltos. Un escalón que aparece igual en las dos columnas no se le puede atribuir a la TLB.

Resumen

- La **TLB** guarda traducciones recientes. Por la localidad, acierta casi siempre y la paginación cuesta casi nada.
- El fallo de TLB lo atiende el **hardware** (x86, RISC-V) o el **software** (MIPS). En un cambio de contexto se **vacía** la TLB o se usan **ASIDs**.
- Medir importa: en esta máquina, un acceso pasa de 1.4 ns a $\approx$43 ns cuando las páginas tocadas superan lo que cubre la TLB.
- Las **tablas multinivel** solo crean las páginas de tabla que se usan: 16 KB en vez de 4 MB. El precio es un acceso más por nivel en cada fallo.
- Otras ideas: páginas grandes, híbrido segmentación + paginación, tablas invertidas.
- xv6-riscv usa **Sv39**: tres niveles de 9 bits, `walk()`, `satp` y `sfence.vma` sin ASIDs.

Para la próxima clase

- Leer OSTEP cap. 21–24 (swapping, políticas de reemplazo, sistemas de memoria virtual completos).
- Laboratorio: los tres ejercicios de esta semana.
- Repasar las preguntas de traducción de las semanas 5–7: entran al parcial.

Semana 8

P2 (lotería) se entrega. Se publica P3 (lab `pgtbl`), con entrega en la semana 10. Examen parcial: sesión B.

Pregunta para pensar

Un arreglo de 10 enteros cruza 3 páginas. ¿Cuántos aciertos y fallos de TLB hay en la primera pasada? ¿Y en la segunda, si entre ambas hubo un cambio de contexto en xv6?