Chapter 8: Memory Management

Operating Systems

Memory hierarchy and caches | Base and limit registers | Address binding and the MMU | Dynamic loading and linking | Swapping | Contiguous allocation and fit algorithms | Fragmentation and bitmaps | Paging | TLB and access time | Hierarchical, hashed, inverted tables | x86-64 and huge pages | Segmentation, x86 and ARMv8

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/os/slides/Chapter_08_Memory_Management.html

Learning Outcomes

Abstraction

Explain the memory hierarchy, cache lines and average access time, the address space, and the problems of protection and relocation.

Addresses

Explain binding times, logical vs. physical addresses, and the MMU's base–limit check.

Linking

Discuss dynamic loading, static and dynamic linking, shared libraries via PLT/GOT, and swapping.

Contiguous allocation

Apply the fit algorithms; analyze internal/external fragmentation, bitmaps, and linked lists.

Paging

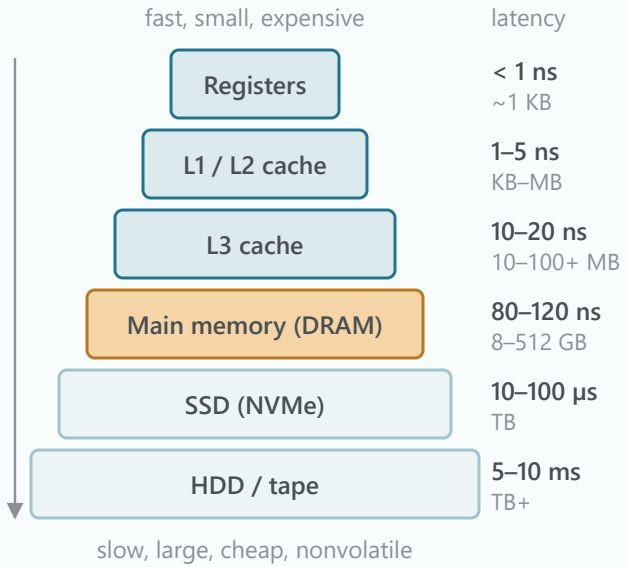
Split an address into (p, d), translate it with a page table and TLB, and compute the effective access time.

Real systems

Compare hierarchical, hashed, and inverted tables; x86-64 four/five-level paging, huge pages, segmentation, and ARMv8.

The Memory Hierarchy and the Memory Manager

- The only storage the CPU can access **directly** is **registers** and **main memory**; disk only through I/O.
- The memory unit sees only a stream: *address + read* or *address + data + write*. It doesn't know where the address came from.
- A register is read in one clock cycle; a DRAM access takes **hundreds of cycles** and the CPU stalls. A **cache** bridges the gap.
- Programmers want memory that is *unlimited, fast, nonvolatile, and protected*; the real answer is a hierarchy.
- The **memory manager** tracks free and used space, allocates it to processes, reclaims it, and protects processes from each other.



Typical orders of magnitude as of 2026; they vary by core and model. Caches are managed by hardware, main memory by the OS.

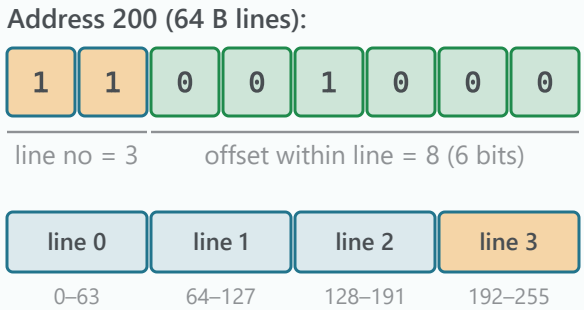
Kinds of Main Memory

Type	Properties	Used for
RAM (random access)	Read/write, fast, **volatile**. DRAM needs refreshing; SRAM is faster and costlier.	Main memory (DRAM: DDR5, LPDDR5X), caches (SRAM)
ROM (read-only)	Written at manufacture, unchangeable, nonvolatile, cheap.	Simple devices, fixed code in microcontrollers
EEPROM / Flash	Electrically erasable and rewritable, nonvolatile; writes are slow and write endurance is limited.	**UEFI/BIOS firmware** (SPI flash), SSDs, USB sticks, phone and music-player storage (NAND flash)
CMOS / NVRAM	A small area kept alive by a battery or flash.	Clock, boot settings

In today's computers the BIOS/UEFI code lives in **flash memory**, not ROM, which is why it can be updated. Portable music players also use NAND flash, not disks.

Cache Lines

- Memory is divided into **cache lines** (typically **64 bytes**): line 0 = 0–63, line 1 = 64–127, …
- For address *a*, the line number is ⌊a / 64⌋ and the offset within the line is a mod 64.
- When the CPU reads a word, hardware checks the cache first: a **hit** → ~4–5 cycles in L1; a **miss** → a lower level or DRAM, hundreds of cycles.
- Caches are expensive, so they are small; that's why they are built in levels (L1, L2, L3).
- Why does it work? **Temporal locality** (what was just used will be used again) and **spatial locality** (nearby addresses are used together).



How Does a Read Travel Through the Hierarchy?

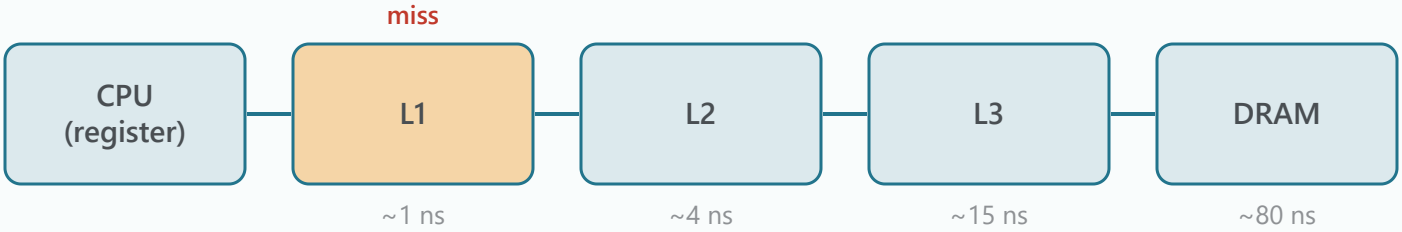


total latency: 0 ns

The CPU wants to read variable `x` (`LOAD R1, [x]`). The closest, fastest level is checked first.

Step 1 / 5

How Does a Read Travel Through the Hierarchy?



total latency: ~1 ns

Not in L1: a **miss**. ~1 ns lost; the search continues one level down.

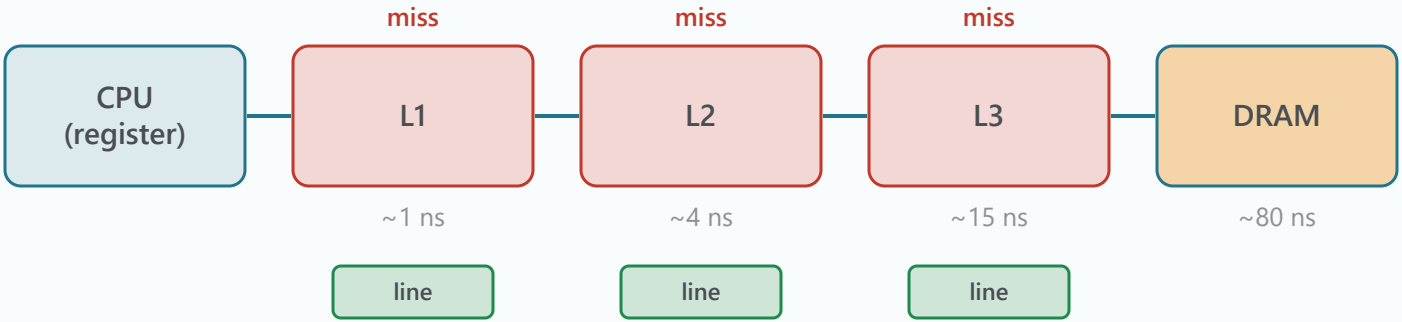
How Does a Read Travel Through the Hierarchy?



total latency: ~20 ns

Misses in L2 and L3 too. Each level is a little larger but slower; ~20 ns in total.

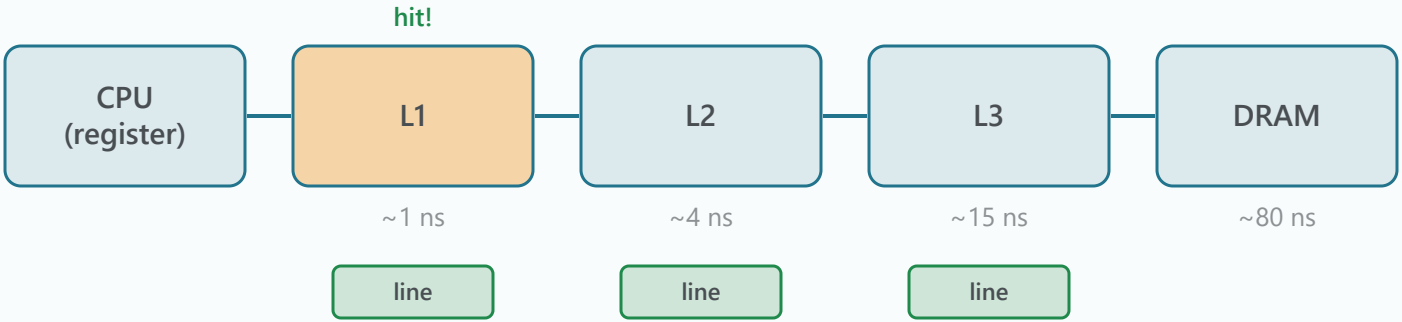
How Does a Read Travel Through the Hierarchy?



total latency: ~100 ns

The data comes from DRAM (~80 ns). Not just one word: **the whole 64-byte line** is placed in L3, L2, and L1.

How Does a Read Travel Through the Hierarchy?



total latency: ~1 ns

A second access to data in the same line is now an **L1 hit**: ~1 ns. Caches work thanks to **locality**.

Code Trace: A Direct-Mapped Cache

Access sequence

0	8	64	256	0	72

Cache (4 sets)

set	block (tag)
0	–
1	–
2	–
3	–

The cache starts empty (−1). 64 B lines, **4 sets**, direct mapping: each block can go to only one set.

Step 1 / 14

```
1 int addr[6] = {0, 8, 64, 256, 0, 72};
2 int tag[4] = {-1, -1, -1, -1};
▶ int hits = 0, misses = 0;
4 for (int i = 0; i < 6; i++) {
5   int block = addr[i] / 64;
6   int set = block % 4;
7   if (tag[set] == block) hits++;
8   else { tag[set] = block; misses++; }
9 }
```

VARIABLES

`tag = [-1, -1, -1, -1]` `hits = 0` `misses = 0`

Code Trace: A Direct-Mapped Cache

Access sequence

0	8	64	256	0	72

Cache (4 sets)

set	block (tag)
0	–
1	–
2	–
3	–

addr[0] = 0 → block = 0 / 64 = **0**, set = 0 mod 4 = **0**.

Step 2 / 14

```
1 int addr[6] = {0, 8, 64, 256, 0, 72};
2 int tag[4] = {-1, -1, -1, -1};
3 int hits = 0, misses = 0;
4 for (int i = 0; i < 6; i++) {
▶   int block = addr[i] / 64;
▶   int set = block % 4;
7   if (tag[set] == block) hits++;
8   else { tag[set] = block; misses++; }
9 }
```

VARIABLES

`i = 0` `block = 0` `set = 0` `tag = [-1, -1, -1, -1]` `hits = 0` `misses = 0`

Code Trace: A Direct-Mapped Cache

Access sequence

0	8	64	256	0	72
M					

Cache (4 sets)

set	block (tag)
0	block 0
1	–
2	–
3	–

Set 0 is empty → **miss**; block 0 is loaded.

Step 3 / 14

```
1 int addr[6] = {0, 8, 64, 256, 0, 72};
2 int tag[4] = {-1, -1, -1, -1};
3 int hits = 0, misses = 0;
4 for (int i = 0; i < 6; i++) {
5   int block = addr[i] / 64;
6   int set = block % 4;
7   if (tag[set] == block) hits++;
▶   else { tag[set] = block; misses++; }
9 }
```

VARIABLES

`i = 0` `block = 0` `set = 0` `tag = [0, -1, -1, -1]` `hits = 0` `misses = 1`

Code Trace: A Direct-Mapped Cache

Access sequence

0	8	64	256	0	72
M					

Cache (4 sets)

set	block (tag)
0	block 0
1	–
2	–
3	–

addr[1] = 8 → block = 8 / 64 = **0**, set = 0 mod 4 = **0**.

Step 4 / 14

```
1 int addr[6] = {0, 8, 64, 256, 0, 72};
2 int tag[4] = {-1, -1, -1, -1};
3 int hits = 0, misses = 0;
4 for (int i = 0; i < 6; i++) {
▶   int block = addr[i] / 64;
▶   int set = block % 4;
7   if (tag[set] == block) hits++;
8   else { tag[set] = block; misses++; }
9 }
```

VARIABLES

`i = 1` `block = 0` `set = 0` `tag = [0, -1, -1, -1]` `hits = 0` `misses = 1`

Code Trace: A Direct-Mapped Cache

Access sequence

0	8	64	256	0	72
M	H				

Cache (4 sets)

set	block (tag)
0	block 0
1	–
2	–
3	–

tag[0] = 0 = block → **hit**. 8 is in the same 64-byte line as 0: **spatial locality**.

Step 5 / 14

```
1 int addr[6] = {0, 8, 64, 256, 0, 72};
2 int tag[4] = {-1, -1, -1, -1};
3 int hits = 0, misses = 0;
4 for (int i = 0; i < 6; i++) {
5   int block = addr[i] / 64;
6   int set = block % 4;
▶   if (tag[set] == block) hits++;
8   else { tag[set] = block; misses++; }
9 }
```

VARIABLES

`i = 1` `block = 0` `set = 0` `tag = [0, -1, -1, -1]` `hits = 1` `misses = 1`

Code Trace: A Direct-Mapped Cache

Access sequence

0	8	64	256	0	72
M	H				

Cache (4 sets)

set	block (tag)
0	block 0
1	–
2	–
3	–

addr[2] = 64 → block = 64 / 64 = **1**, set = 1 mod 4 = **1**.

Step 6 / 14

```
1 int addr[6] = {0, 8, 64, 256, 0, 72};
2 int tag[4] = {-1, -1, -1, -1};
3 int hits = 0, misses = 0;
4 for (int i = 0; i < 6; i++) {
▶   int block = addr[i] / 64;
▶   int set = block % 4;
7   if (tag[set] == block) hits++;
8   else { tag[set] = block; misses++; }
9 }
```

VARIABLES

`i = 2` `block = 1` `set = 1` `tag = [0, -1, -1, -1]` `hits = 1` `misses = 1`

Code Trace: A Direct-Mapped Cache

Access sequence

0	8	64	256	0	72
M	H	M			

Cache (4 sets)

set	block (tag)
0	block 0
1	block 1
2	–
3	–

Set 1 is empty → **miss**; block 1 is loaded.

Step 7 / 14

```
1 int addr[6] = {0, 8, 64, 256, 0, 72};
2 int tag[4] = {-1, -1, -1, -1};
3 int hits = 0, misses = 0;
4 for (int i = 0; i < 6; i++) {
5   int block = addr[i] / 64;
6   int set = block % 4;
7   if (tag[set] == block) hits++;
▶   else { tag[set] = block; misses++; }
9 }
```

VARIABLES

`i = 2` `block = 1` `set = 1` `tag = [0, 1, -1, -1]` `hits = 1` `misses = 2`

Code Trace: A Direct-Mapped Cache

Access sequence

0	8	64	256	0	72
M	H	M			

Cache (4 sets)

set	block (tag)
0	block 0
1	block 1
2	–
3	–

addr[3] = 256 → block = 256 / 64 = **4**, set = 4 mod 4 = **0**.

Step 8 / 14

```
1 int addr[6] = {0, 8, 64, 256, 0, 72};
2 int tag[4] = {-1, -1, -1, -1};
3 int hits = 0, misses = 0;
4 for (int i = 0; i < 6; i++) {
▶   int block = addr[i] / 64;
▶   int set = block % 4;
7   if (tag[set] == block) hits++;
8   else { tag[set] = block; misses++; }
9 }
```

VARIABLES

`i = 3` `block = 4` `set = 0` `tag = [0, 1, -1, -1]` `hits = 1` `misses = 2`

Code Trace: A Direct-Mapped Cache

Access sequence

0	8	64	256	0	72
M	H	M	M		

Cache (4 sets)

set	block (tag)
0	block 4
1	block 1
2	–
3	–

Set 0 holds block 0 → **miss**; block 4 **evicts** it (a conflict miss).

Step 9 / 14

```
1 int addr[6] = {0, 8, 64, 256, 0, 72};
2 int tag[4] = {-1, -1, -1, -1};
3 int hits = 0, misses = 0;
4 for (int i = 0; i < 6; i++) {
5   int block = addr[i] / 64;
6   int set = block % 4;
7   if (tag[set] == block) hits++;
▶   else { tag[set] = block; misses++; }
9 }
```

VARIABLES

`i = 3` `block = 4` `set = 0` `tag = [4, 1, -1, -1]` `hits = 1` `misses = 3`

Code Trace: A Direct-Mapped Cache

Access sequence

0	8	64	256	0	72
M	H	M	M		

Cache (4 sets)

set	block (tag)
0	block 4
1	block 1
2	–
3	–

addr[4] = 0 → block = 0 / 64 = **0**, set = 0 mod 4 = **0**.

Step 10 / 14

```
1 int addr[6] = {0, 8, 64, 256, 0, 72};
2 int tag[4] = {-1, -1, -1, -1};
3 int hits = 0, misses = 0;
4 for (int i = 0; i < 6; i++) {
▶   int block = addr[i] / 64;
▶   int set = block % 4;
7   if (tag[set] == block) hits++;
8   else { tag[set] = block; misses++; }
9 }
```

VARIABLES

`i = 4` `block = 0` `set = 0` `tag = [4, 1, -1, -1]` `hits = 1` `misses = 3`

Code Trace: A Direct-Mapped Cache

Access sequence

0	8	64	256	0	72
M	H	M	M	M	

Cache (4 sets)

set	block (tag)
0	block 0
1	block 1
2	–
3	–

Set 0 holds block 4 → **miss**; block 0 **evicts** it (a conflict miss).

Step 11 / 14

```
1 int addr[6] = {0, 8, 64, 256, 0, 72};
2 int tag[4] = {-1, -1, -1, -1};
3 int hits = 0, misses = 0;
4 for (int i = 0; i < 6; i++) {
5   int block = addr[i] / 64;
6   int set = block % 4;
7   if (tag[set] == block) hits++;
▶   else { tag[set] = block; misses++; }
9 }
```

VARIABLES

`i = 4` `block = 0` `set = 0` `tag = [0, 1, -1, -1]` `hits = 1` `misses = 4`

Code Trace: A Direct-Mapped Cache

Access sequence

0	8	64	256	0	72
M	H	M	M	M	

Cache (4 sets)

set	block (tag)
0	block 0
1	block 1
2	–
3	–

addr[5] = 72 → block = 72 / 64 = **1**, set = 1 mod 4 = **1**.

Step 12 / 14

```
1 int addr[6] = {0, 8, 64, 256, 0, 72};
2 int tag[4] = {-1, -1, -1, -1};
3 int hits = 0, misses = 0;
4 for (int i = 0; i < 6; i++) {
▶   int block = addr[i] / 64;
▶   int set = block % 4;
7   if (tag[set] == block) hits++;
8   else { tag[set] = block; misses++; }
9 }
```

VARIABLES

`i = 5` `block = 1` `set = 1` `tag = [0, 1, -1, -1]` `hits = 1` `misses = 4`

Code Trace: A Direct-Mapped Cache

Access sequence

0	8	64	256	0	72
M	H	M	M	M	H

Cache (4 sets)

set	block (tag)
0	block 0
1	block 1
2	–
3	–

tag[1] = 1 = block → **hit**. 72 is in the same line as 64.

Step 13 / 14

```
1 int addr[6] = {0, 8, 64, 256, 0, 72};
2 int tag[4] = {-1, -1, -1, -1};
3 int hits = 0, misses = 0;
4 for (int i = 0; i < 6; i++) {
5   int block = addr[i] / 64;
6   int set = block % 4;
▶   if (tag[set] == block) hits++;
8   else { tag[set] = block; misses++; }
9 }
```

VARIABLES

`i = 5` `block = 1` `set = 1` `tag = [0, 1, -1, -1]` `hits = 2` `misses = 4`

Code Trace: A Direct-Mapped Cache

Access sequence

0	8	64	256	0	72
M	H	M	M	M	H

Cache (4 sets)

set	block (tag)
0	block 0
1	block 1
2	–
3	–

The loop ends: **2 hits, 4 misses** (33% hit ratio). 0 and 256 map to the same set and evict each other; a 2-way set-associative cache would avoid this. (Here the whole block number is stored instead of a tag.)

Step 14 / 14

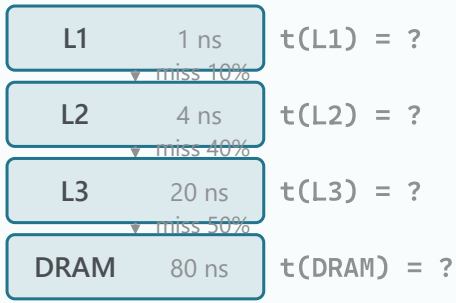
```
1 int addr[6] = {0, 8, 64, 256, 0, 72};
2 int tag[4] = {-1, -1, -1, -1};
3 int hits = 0, misses = 0;
4 for (int i = 0; i < 6; i++) {
5   int block = addr[i] / 64;
6   int set = block % 4;
7   if (tag[set] == block) hits++;
8   else { tag[set] = block; misses++; }
▶ }
```

VARIABLES

`tag = [0, 1, -1, -1]` `hits = 2` `misses = 4`

Code Trace: Average Memory Access Time

```
▶ /* L1, L2, L3, DRAM access time (ns) */
▶ double lat[4] = {1, 4, 20, 80};
▶ double miss[3] = {0.10, 0.40, 0.50};
4 double t = lat[3];          /* DRAM */
5 for (int k = 2; k >= 0; k--)
6   t = lat[k] + miss[k] * t;
7 printf("average = %.1f ns\n", t);
```



Four levels: access times and each cache level's **miss rate**. On a miss at one level, the request goes down to the next.

Step 1 / 10

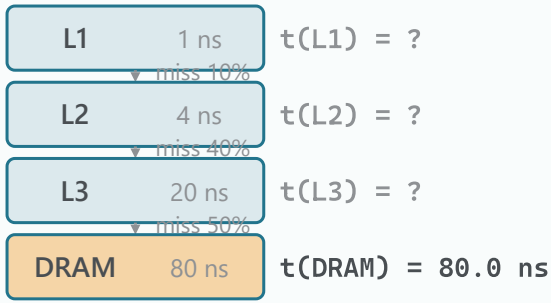
VARIABLES

`t = ?` `k = ?`

OUTPUT

Code Trace: Average Memory Access Time

```
1 /* L1, L2, L3, DRAM access time (ns) */
2 double lat[4] = {1, 4, 20, 80};
3 double miss[3] = {0.10, 0.40, 0.50};
▶ double t = lat[3];          /* DRAM */
5 for (int k = 2; k >= 0; k--)
6   t = lat[k] + miss[k] * t;
7 printf("average = %.1f ns\n", t);
```



`t = 80`

The bottom level, DRAM, satisfies every request: starting there, **t = 80 ns**.

Step 2 / 10

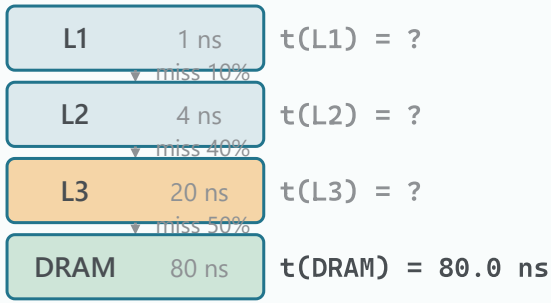
VARIABLES

`t = 80.0` `k = ?`

OUTPUT

Code Trace: Average Memory Access Time

```
1 /* L1, L2, L3, DRAM access time (ns) */
2 double lat[4] = {1, 4, 20, 80};
3 double miss[3] = {0.10, 0.40, 0.50};
4 double t = lat[3];          /* DRAM */
▶ for (int k = 2; k >= 0; k--)
6   t = lat[k] + miss[k] * t;
7 printf("average = %.1f ns\n", t);
```



k = 2 ≥ 0: move up one level (L3).

Step 3 / 10

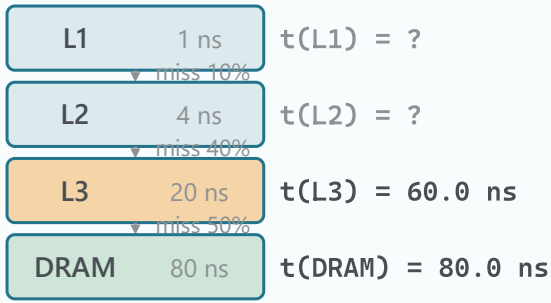
VARIABLES

`t = 80.0` `k = 2`

OUTPUT

Code Trace: Average Memory Access Time

```
1 /* L1, L2, L3, DRAM access time (ns) */
2 double lat[4] = {1, 4, 20, 80};
3 double miss[3] = {0.10, 0.40, 0.50};
4 double t = lat[3];          /* DRAM */
5 for (int k = 2; k >= 0; k--)
▶   t = lat[k] + miss[k] * t;
7 printf("average = %.1f ns\n", t);
```



`t = 20 + 0.50 × 80.0 = 60.0`

L3 spends 20 ns on every request; for 50% of requests the 80 ns of the level below is added: **t = 60 ns**.

Step 4 / 10

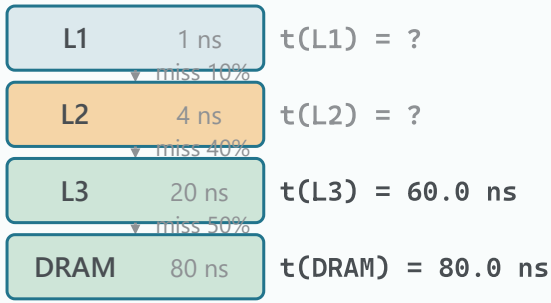
VARIABLES

`t = 60.0` `k = 2`

OUTPUT

Code Trace: Average Memory Access Time

```
1 /* L1, L2, L3, DRAM access time (ns) */
2 double lat[4] = {1, 4, 20, 80};
3 double miss[3] = {0.10, 0.40, 0.50};
4 double t = lat[3];          /* DRAM */
▶ for (int k = 2; k >= 0; k--)
6   t = lat[k] + miss[k] * t;
7 printf("average = %.1f ns\n", t);
```



k = 1 ≥ 0: move up one level (L2).

Step 5 / 10

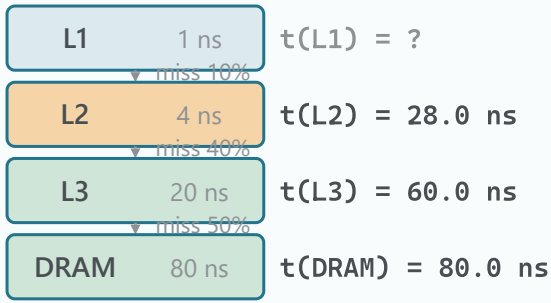
VARIABLES

`t = 60.0` `k = 1`

OUTPUT

Code Trace: Average Memory Access Time

```
1 /* L1, L2, L3, DRAM access time (ns) */
2 double lat[4] = {1, 4, 20, 80};
3 double miss[3] = {0.10, 0.40, 0.50};
4 double t = lat[3];          /* DRAM */
5 for (int k = 2; k >= 0; k--)
▶   t = lat[k] + miss[k] * t;
7 printf("average = %.1f ns\n", t);
```



`t = 4 + 0.40 × 60.0 = 28.0`

L2 spends 4 ns on every request; for 40% of requests the 60 ns of the level below is added: **t = 28 ns**.

Step 6 / 10

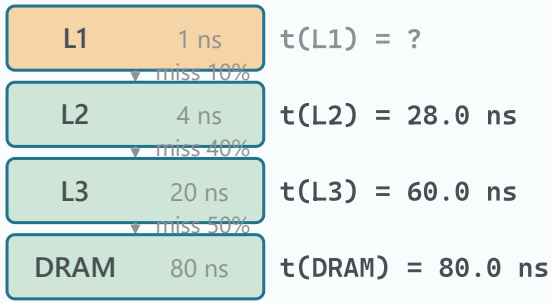
VARIABLES

`t = 28.0` `k = 1`

OUTPUT

Code Trace: Average Memory Access Time

```
1 /* L1, L2, L3, DRAM access time (ns) */
2 double lat[4] = {1, 4, 20, 80};
3 double miss[3] = {0.10, 0.40, 0.50};
4 double t = lat[3];          /* DRAM */
▶ for (int k = 2; k >= 0; k--)
6   t = lat[k] + miss[k] * t;
7 printf("average = %.1f ns\n", t);
```



k = 0 ≥ 0: move up one level (L1).

Step 7 / 10

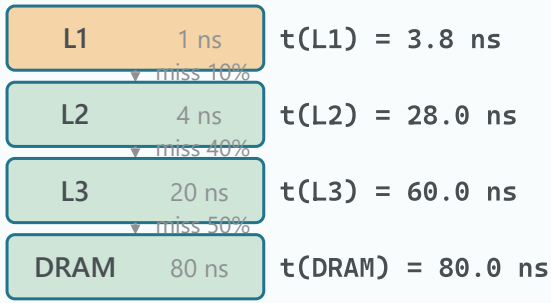
VARIABLES

`t = 28.0` `k = 0`

OUTPUT

Code Trace: Average Memory Access Time

```
1 /* L1, L2, L3, DRAM access time (ns) */
2 double lat[4] = {1, 4, 20, 80};
3 double miss[3] = {0.10, 0.40, 0.50};
4 double t = lat[3];          /* DRAM */
5 for (int k = 2; k >= 0; k--)
▶   t = lat[k] + miss[k] * t;
7 printf("average = %.1f ns\n", t);
```



`t = 1 + 0.10 × 28.0 = 3.8`

L1 spends 1 ns on every request; for 10% of requests the 28 ns of the level below is added: **t = 3.8 ns**.

Step 8 / 10

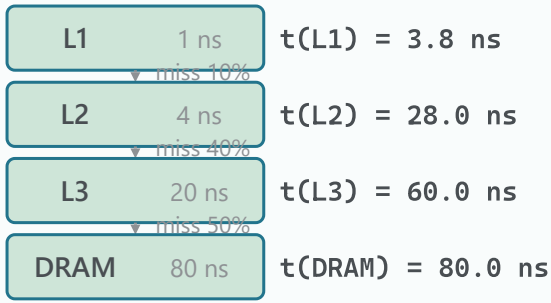
VARIABLES

`t = 3.8` `k = 0`

OUTPUT

Code Trace: Average Memory Access Time

```
1 /* L1, L2, L3, DRAM access time (ns) */
2 double lat[4] = {1, 4, 20, 80};
3 double miss[3] = {0.10, 0.40, 0.50};
4 double t = lat[3];          /* DRAM */
▶ for (int k = 2; k >= 0; k--)
6   t = lat[k] + miss[k] * t;
7 printf("average = %.1f ns\n", t);
```



k = −1: the loop ends.

Step 9 / 10

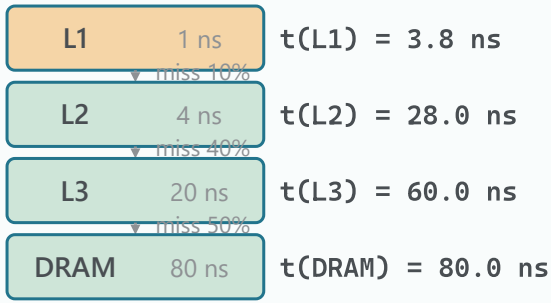
VARIABLES

`t = 3.8` `k = -1`

OUTPUT

Code Trace: Average Memory Access Time

```
1 /* L1, L2, L3, DRAM access time (ns) */
2 double lat[4] = {1, 4, 20, 80};
3 double miss[3] = {0.10, 0.40, 0.50};
4 double t = lat[3];          /* DRAM */
5 for (int k = 2; k >= 0; k--)
6   t = lat[k] + miss[k] * t;
▶ printf("average = %.1f ns\n", t);
```



`average = 3.8 ns`

Even though 10% of accesses miss L1, the average is **3.8 ns**: about 21 times faster than DRAM's 80 ns. A high hit ratio is what makes the hierarchy work.

Step 10 / 10

VARIABLES

`t = 3.8` `k = -1`

OUTPUT

```
average = 3.8 ns
```

Cache Design Questions

1. When to place?

Usually on a miss; hardware also prefetches the following lines.

2. Which line?

Mapping: direct-mapped, *k*-way set-associative, or fully associative.

3. Which one leaves when space is needed?

Replacement policy: LRU approximations, random, FIFO.

4. Where does the evicted line go?

If modified ("dirty") it is written to the level below (write-back); or every write goes down at once (write-through).

Size

A larger cache gets more hits but is slower and costlier.

Replacement

A balance between performance and hardware complexity.

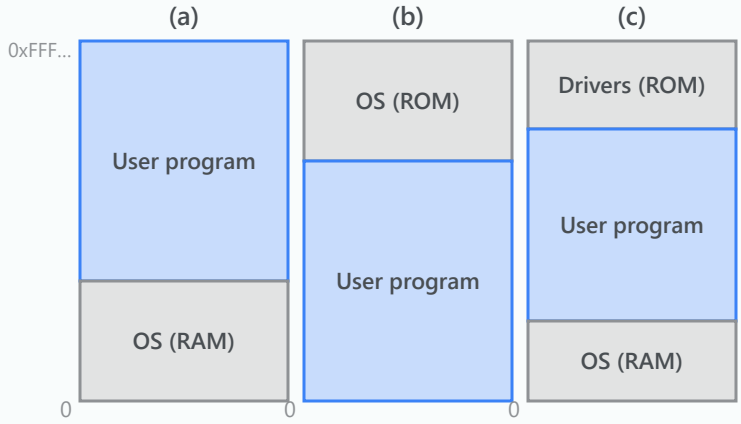
Coherence

On multicore, copies of the same line must stay up to date (protocols such as MESI).

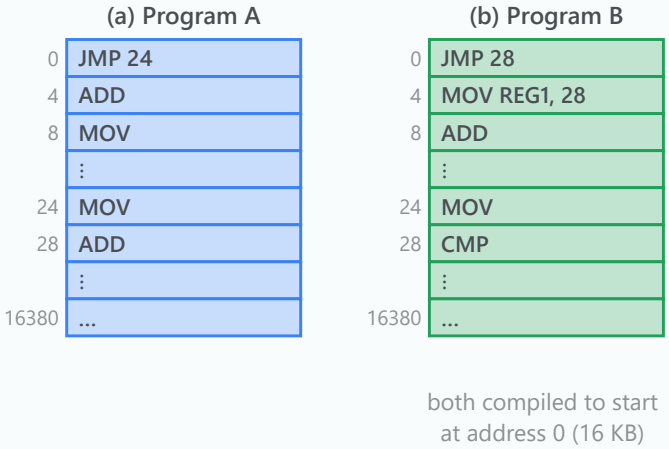
The same four questions come up in the "caches" the OS manages: the page cache, virtual memory, the TLB.

Memory Without Abstraction

- On early computers (mainframes before the 1960s, PCs before the 1980s) every program saw **physical memory** directly: `MOV REGISTER1, 1000` reads physical address 1000.
- **Two programs cannot** be in memory at once: both write to the same physical addresses and one clobbers the other.
- In (a) and (c), a bug in a user program can **crash** the OS.
- (b) was used in some embedded systems, (c) in early PCs with MS-DOS (BIOS drivers in ROM).
- MMU-less microcontrollers still work this way today; most protect regions with at least an **MPU** (memory protection unit).

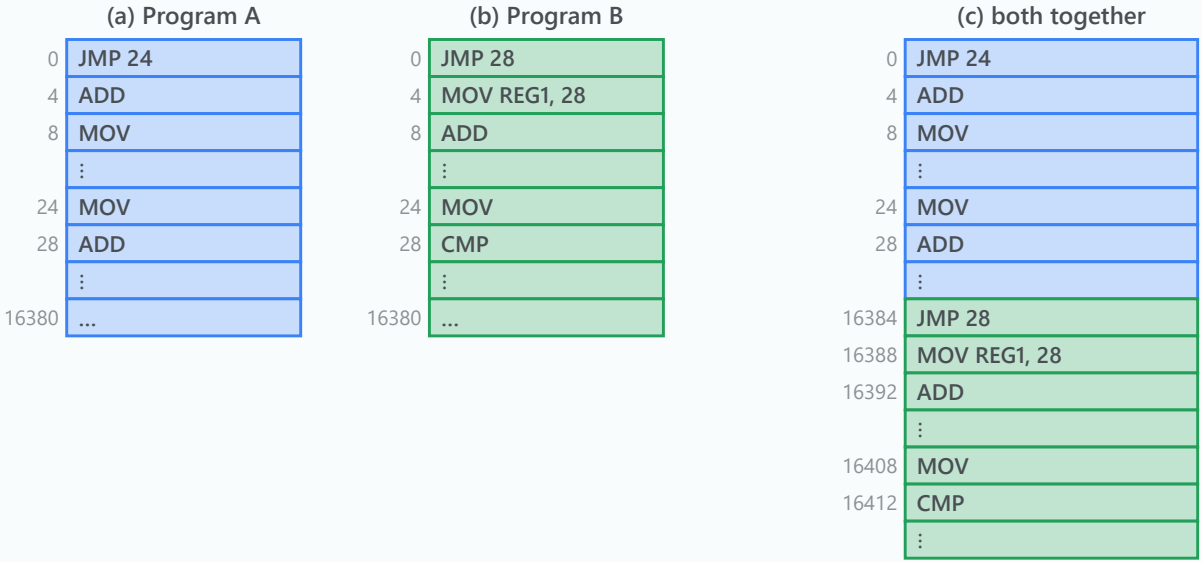


The Relocation Problem



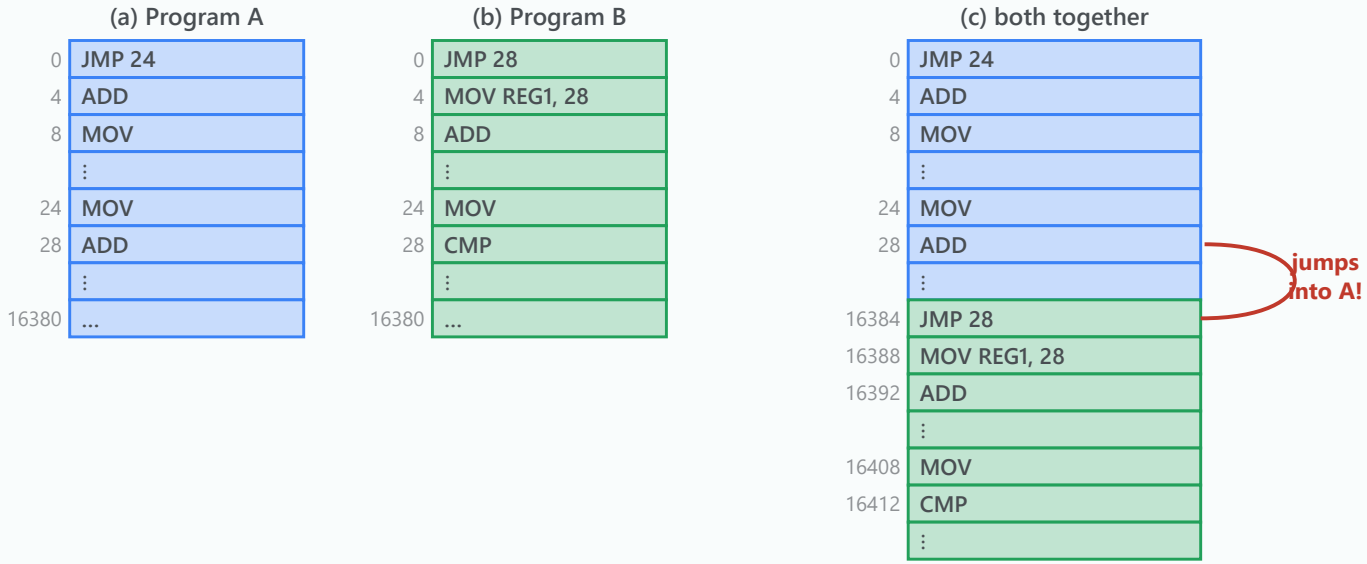
A and B are two 16 KB programs, both compiled assuming they start at **address 0**.

The Relocation Problem



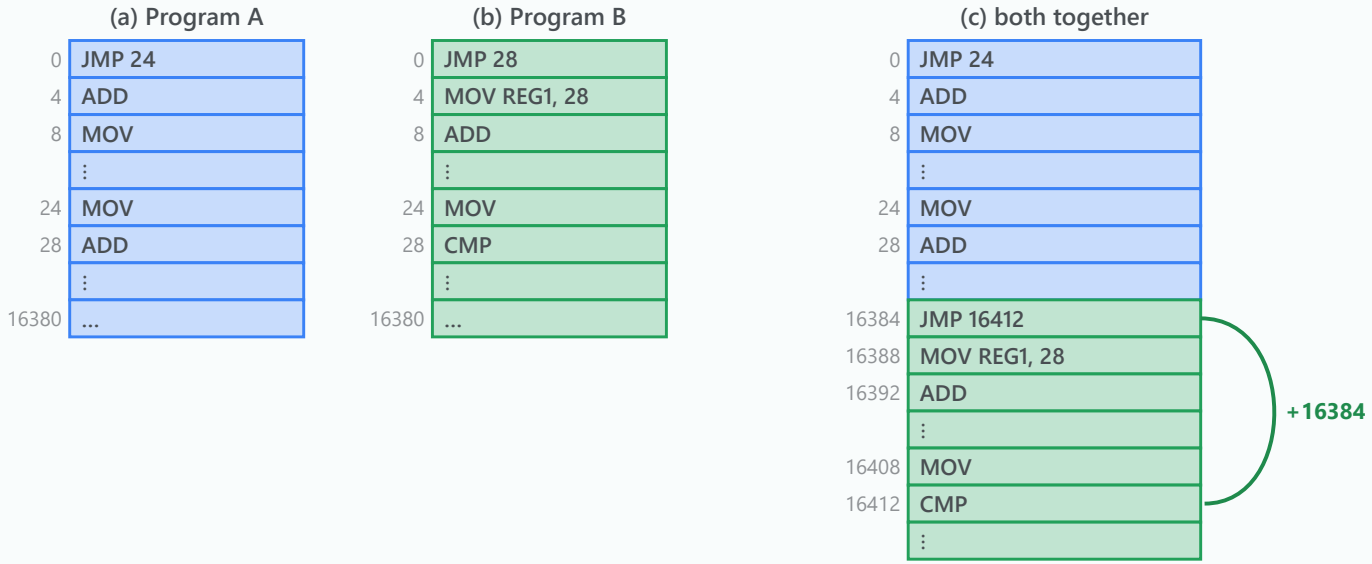
The OS loads A at 0, then B right above it, at **16384**.

The Relocation Problem



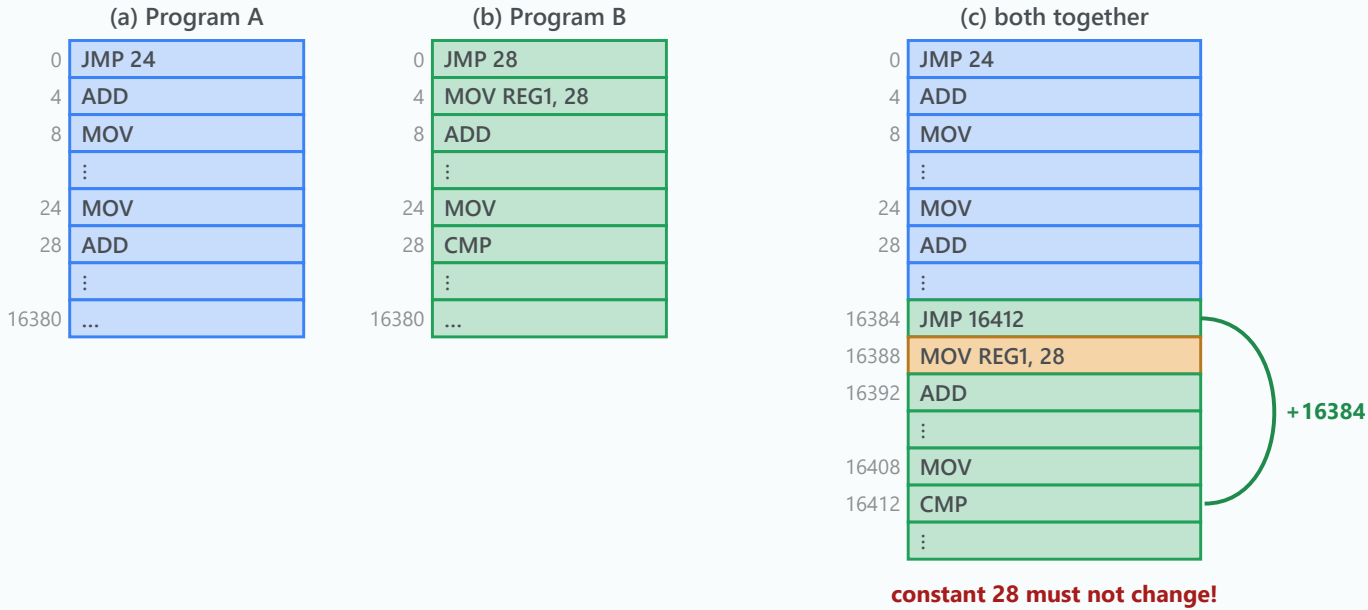
B's first instruction, `JMP 28`, uses an absolute address: it jumps to physical 28, i.e., into **A's** `ADD` instruction, and the program crashes.

The Relocation Problem



Static relocation (IBM 360): the loader adds the start address to every address word: `JMP 16412`. Loading slows down.

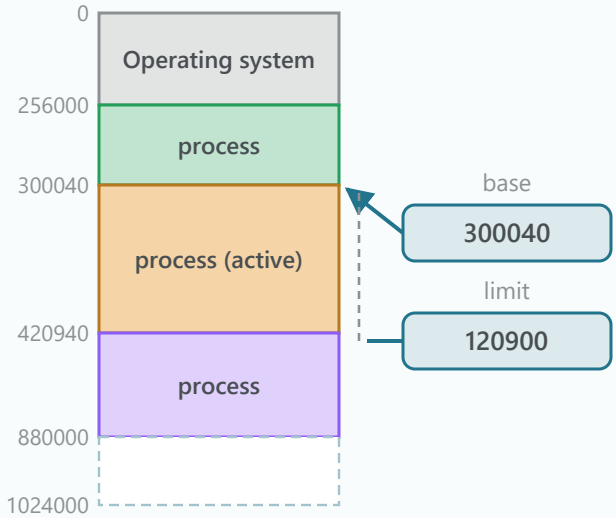
The Relocation Problem



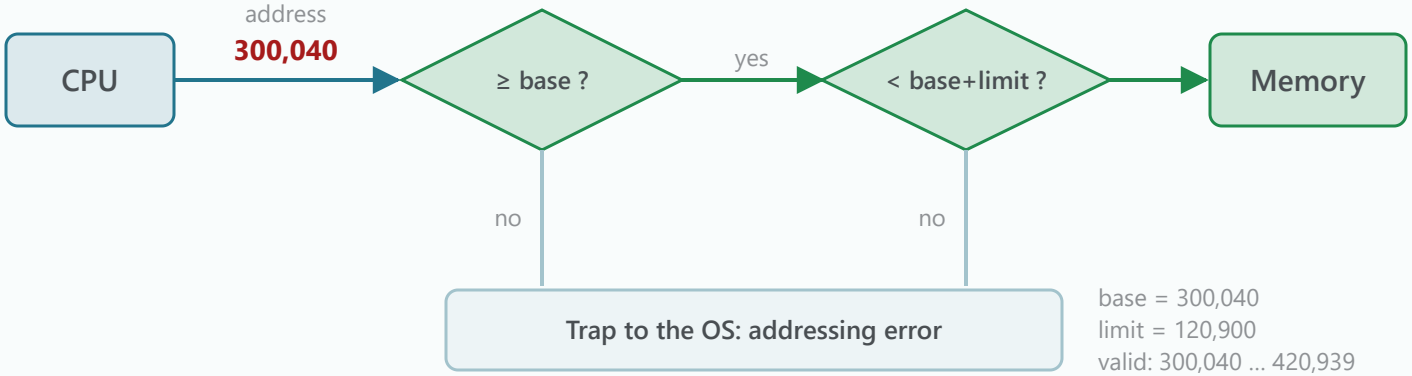
But the 28 in `MOV REG1, 28` is a **constant** and must not change. To know which words are addresses, the loader needs extra **relocation information**.

Address Spaces: Base and Limit Registers

- **Address space**: the set of addresses a process can use. Every process has its own address space, independent of the others.
- Two problems to solve: **protection** (no access to another's space) and **relocation** (loadable anywhere).
- The simplest hardware: **base** (starting physical address) and **limit** (length) registers. The program is loaded as is into a contiguous area.
- The registers are loaded only in **kernel mode** by privileged instructions; on a context switch the new process's values are installed.
- The CDC 6600 worked this way. The Intel 8088's segment registers were **base only**; with no limit there was no protection.



Hardware Address Protection

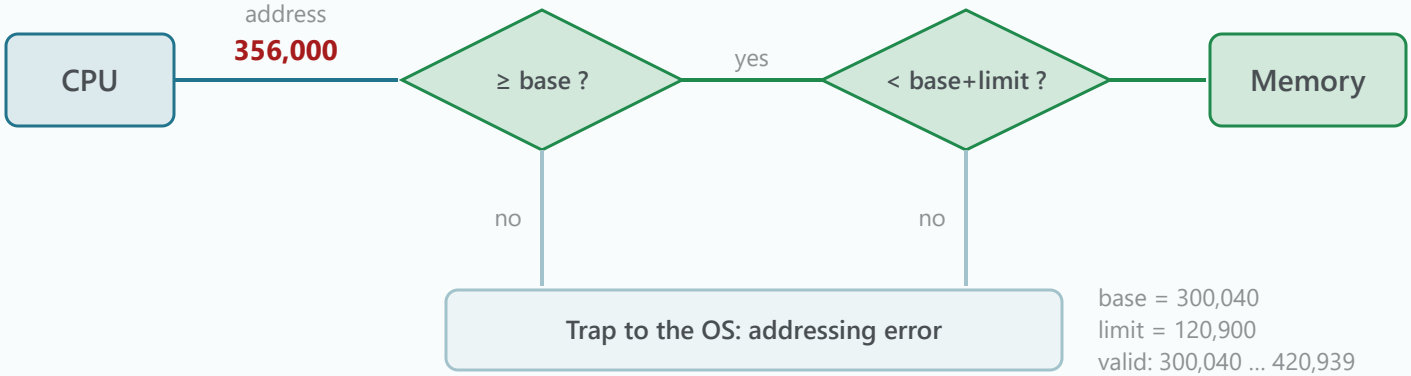


300,040 = base: the first valid address; both comparisons pass and the access proceeds.

Step 1 / 5

Every user-mode access is checked. In kernel mode there is no check; the OS must be able to access all of memory (Silberschatz Figure 9.2).

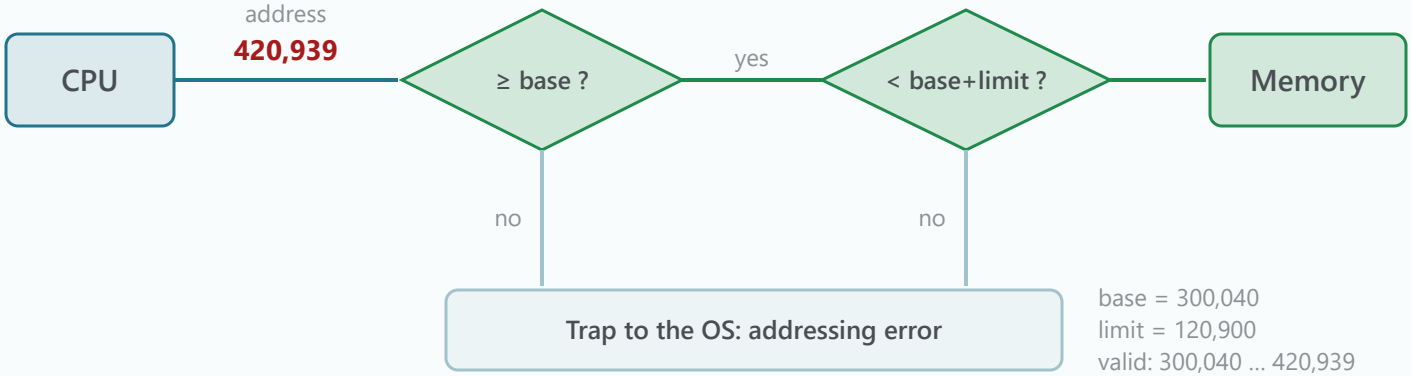
Hardware Address Protection



356,000 is inside the range; the access proceeds. The check happens on every access, in parallel hardware.

Every user-mode access is checked. In kernel mode there is no check; the OS must be able to access all of memory (Silberschatz Figure 9.2).

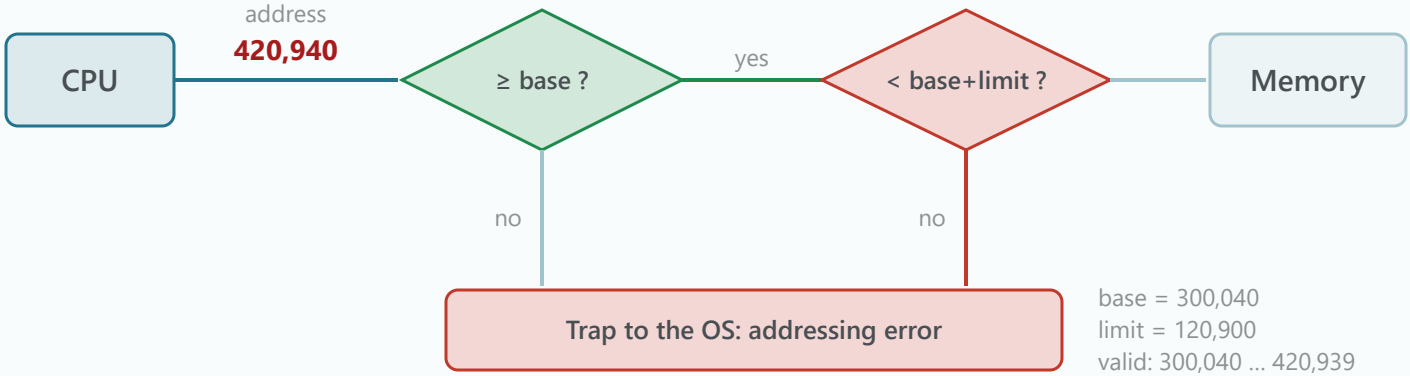
Hardware Address Protection



420,939 = base + limit − 1: the last valid address.

Every user-mode access is checked. In kernel mode there is no check; the OS must be able to access all of memory (Silberschatz Figure 9.2).

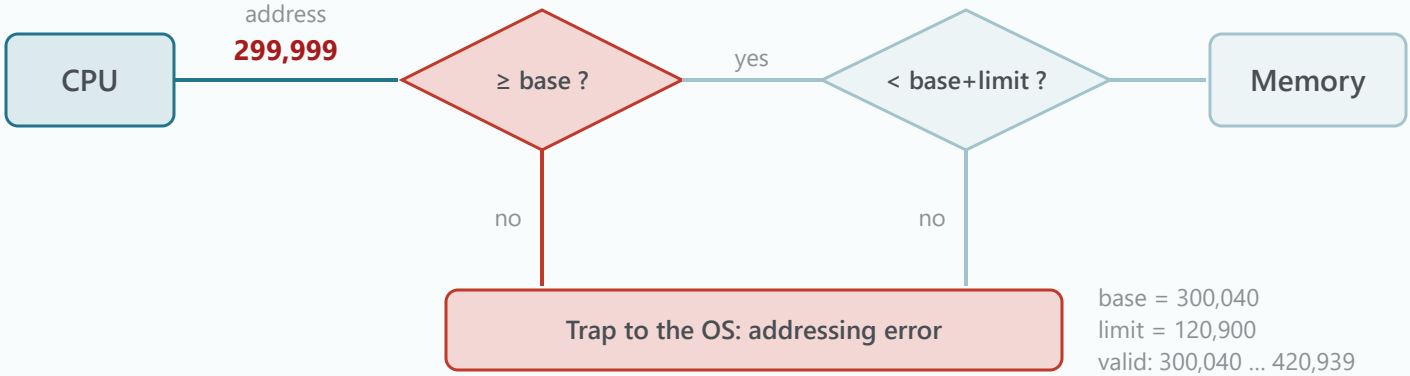
Hardware Address Protection



420,940 = base + limit: the second comparison fails → **trap**. The kernel usually terminates the process (`SIGSEGV` on Unix).

Every user-mode access is checked. In kernel mode there is no check; the OS must be able to access all of memory (Silberschatz Figure 9.2).

Hardware Address Protection

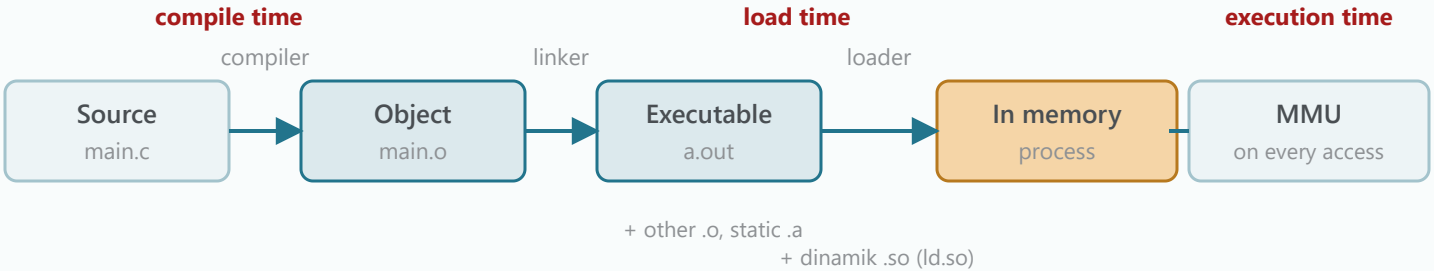


299,999 is below the base: the first comparison fails → the other process's space is protected.

Step 5 / 5

Every user-mode access is checked. In kernel mode there is no check; the OS must be able to access all of memory (Silberschatz Figure 9.2).

Address Binding Times

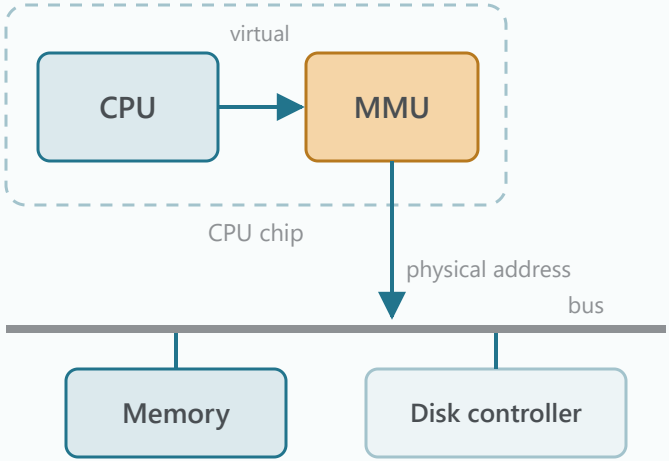


Binding time	Generated code	If location changes	Hardware
Compile	Absolute code (e.g., MS-DOS `.COM`)	recompile	none
Load	Relocatable code; the loader fixes up addresses	reload	none
Execution	Logical addresses; the process can move while running	nothing	MMU (base–limit, paging)

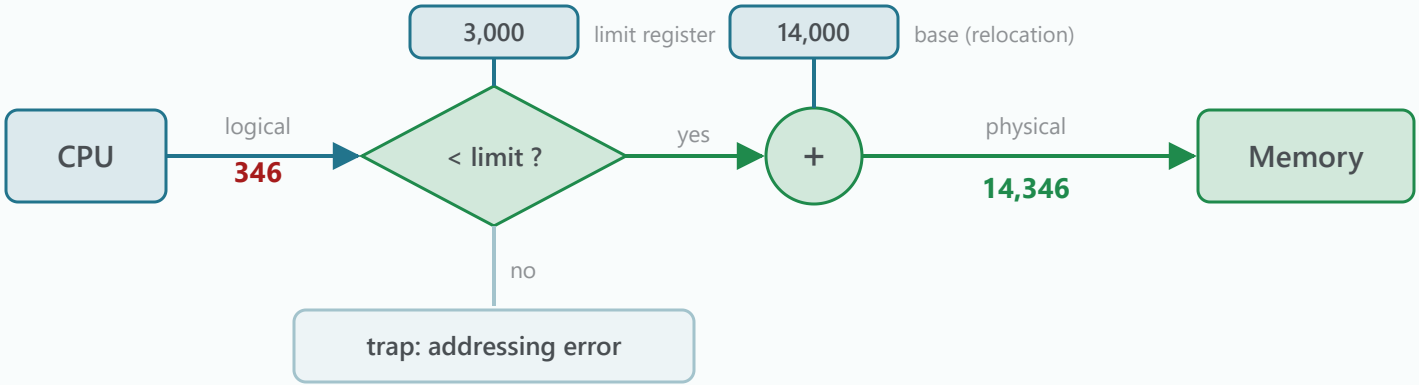
In source code addresses are *symbolic* (`count`); the compiler binds them to *relocatable* addresses ("14 bytes from the start of this module"), the linker/loader to *absolute* addresses (74014). All modern general-purpose systems use execution-time binding.

Logical and Physical Addresses, the MMU

- **Logical (virtual) address**: the address the CPU generates. **Physical address**: the one the memory unit sees, loaded into the memory-address register (MAR).
- With compile- and load-time binding the two are **the same**; with execution-time binding they **differ**.
- Logical address space: the set of addresses a program generates; physical address space: the corresponding physical addresses.
- The run-time mapping is done by hardware, the **MMU** (memory-management unit). The program **never sees** physical addresses.
- In today's processors the MMU, with its TLBs and page-table walker, sits inside every core.



Dynamic Relocation with an MMU

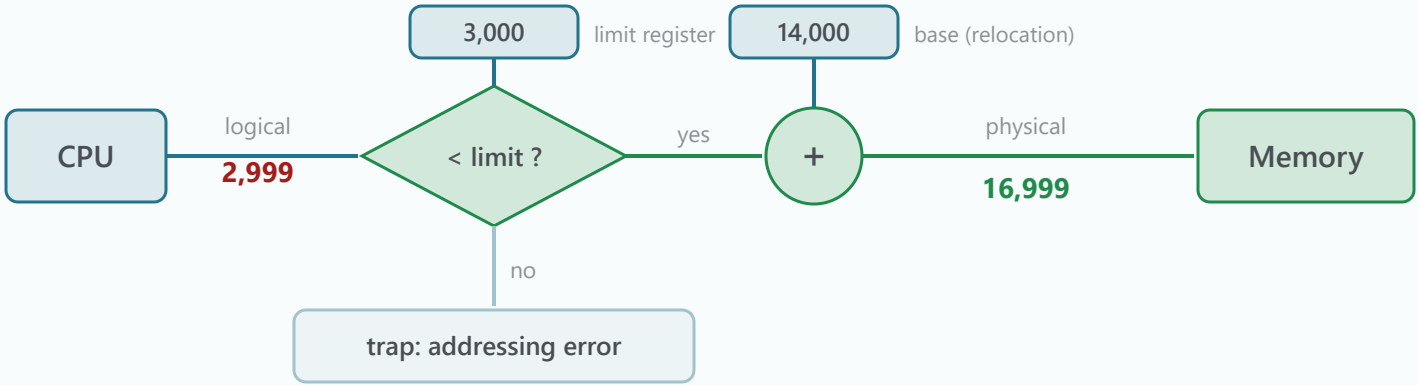


Logical **346** < 3,000 → physical 346 + 14,000 = **14,346**. The program sees 0 through 2,999.

Step 1 / 4

Silberschatz's *relocation register* is Tanenbaum's *base register*. Every access needs an addition and a comparison; without special circuitry, the addition is slow because of carry propagation.

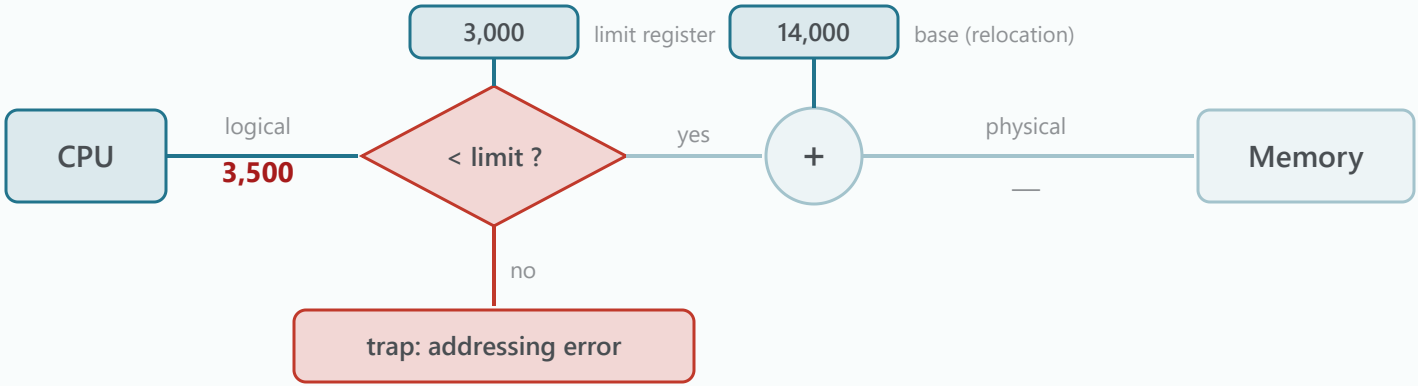
Dynamic Relocation with an MMU



2,999, the last valid logical address → **16,999**.

Silberschatz's *relocation register* is Tanenbaum's *base register*. Every access needs an addition and a comparison; without special circuitry, the addition is slow because of carry propagation.

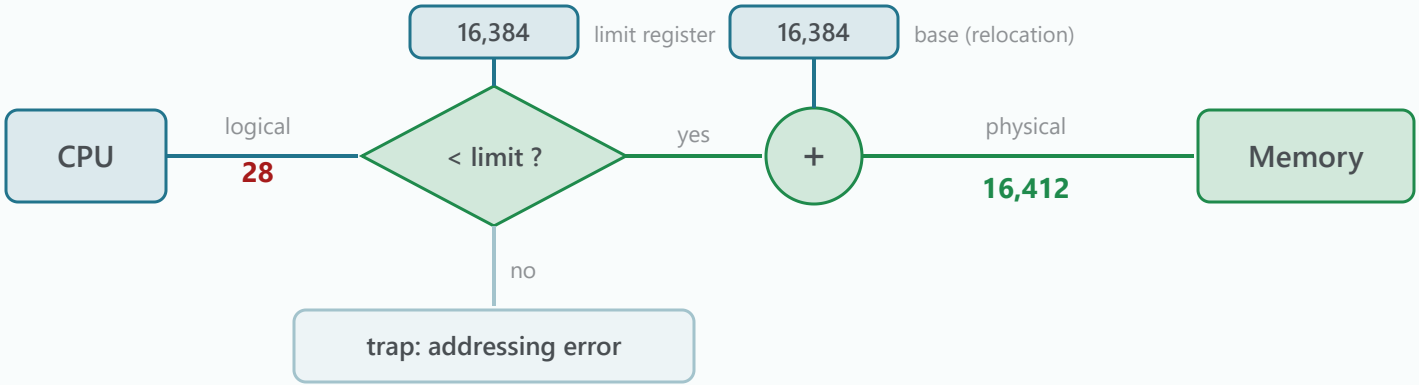
Dynamic Relocation with an MMU



3,500 ≥ limit → **trap** without adding. The comparison is on the logical address, so it is the same whatever the base.

Silberschatz's *relocation register* is Tanenbaum's *base register*. Every access needs an addition and a comparison; without special circuitry, the addition is slow because of carry propagation.

Dynamic Relocation with an MMU



Program B from the earlier example: base = limit = 16,384. `JMP 28` stays unchanged; the MMU produces 28 + 16,384 = **16,412**. No static relocation is needed.

Silberschatz's *relocation register* is Tanenbaum's *base register*. Every access needs an addition and a comparison; without special circuitry, the addition is slow because of carry propagation.

Dynamic Loading

```
1  #include <dlfcn.h>
2  #include <stdio.h>
3
▶ int main(void) {
5    void *h = dlopen("libm.so.6", RTLD_LAZY);
6    if (!h) { puts(dlerror()); return 1; }
7    double (*root)(double) =
8        (double (*)(double)) dlsym(h, "sqrt");
9    printf("%.3f\n", root(2.0));
10   dlclose(h);
11   return 0;
12 }
```



The program has started: the address space holds only `main`, `libc`, the heap, and the stack; **libm is not loaded**.

Step 1 / 5

VARIABLES

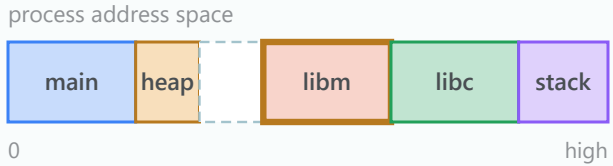
`h = –` `root = –`

OUTPUT

A routine isn't loaded until it is called; unused code is never loaded. No hardware support is needed; the program is designed this way and the OS provides a library (`dlopen`; `LoadLibrary` / `GetProcAddress` on Windows). Since glibc 2.34, `-ldl` is unnecessary.

Dynamic Loading

```
1  #include <dlfcn.h>
2  #include <stdio.h>
3
4  int main(void) {
▶    void *h = dlopen("libm.so.6", RTLD_LAZY);
6    if (!h) { puts(dlerror()); return 1; }
7    double (*root)(double) =
8        (double (*)(double)) dlsym(h, "sqrt");
9    printf("%.3f\n", root(2.0));
10   dlclose(h);
11   return 0;
12 }
```



`dlopen` : the dynamic linker (`ld.so`) finds the file and maps it into the address space with `mmap` . Loading happens **now**, at the program's request.

Step 2 / 5

VARIABLES

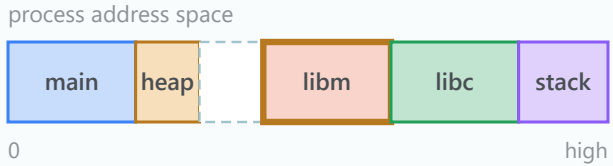
`h = handle` `root = –`

OUTPUT

A routine isn't loaded until it is called; unused code is never loaded. No hardware support is needed; the program is designed this way and the OS provides a library (`dlopen` ; `LoadLibrary` / `GetProcAddress` on Windows). Since glibc 2.34, `-ldl` is unnecessary.

Dynamic Loading

```
1  #include <dlfcn.h>
2  #include <stdio.h>
3
4  int main(void) {
5    void *h = dlopen("libm.so.6", RTLD_LAZY);
6    if (!h) { puts(dlerror()); return 1; }
▶    double (*root)(double) =
▶        (double (*)(double)) dlsym(h, "sqrt");
9    printf("%.3f\n", root(2.0));
10   dlclose(h);
11   return 0;
12 }
```



`dlsym` looks up `sqrt` in the symbol table and returns its address in this process's address space.

Step 3 / 5

VARIABLES

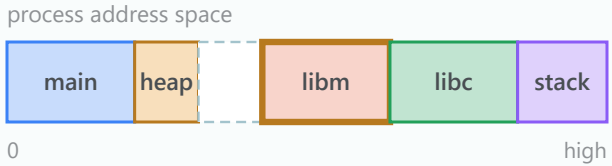
`h = handle` `root = &sqrt`

OUTPUT

A routine isn't loaded until it is called; unused code is never loaded. No hardware support is needed; the program is designed this way and the OS provides a library (`dlopen`; `LoadLibrary` / `GetProcAddress` on Windows). Since glibc 2.34, `-ldl` is unnecessary.

Dynamic Loading

```
1  #include <dlfcn.h>
2  #include <stdio.h>
3
4  int main(void) {
5    void *h = dlopen("libm.so.6", RTLD_LAZY);
6    if (!h) { puts(dlerror()); return 1; }
7    double (*root)(double) =
8        (double (*)(double)) dlsym(h, "sqrt");
▶    printf("%.3f\n", root(2.0));
10   dlclose(h);
11   return 0;
12 }
```



The call goes through the function pointer; the output is `1.414`.

Step 4 / 5

VARIABLES

`h = handle` `root = &sqrt`

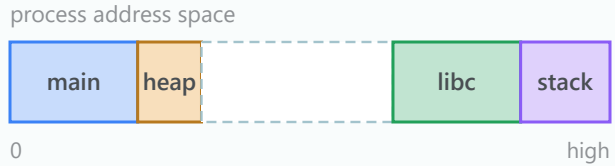
OUTPUT

```
1.414
```

A routine isn't loaded until it is called; unused code is never loaded. No hardware support is needed; the program is designed this way and the OS provides a library (`dlopen`; `LoadLibrary` / `GetProcAddress` on Windows). Since glibc 2.34, `-ldl` is unnecessary.

Dynamic Loading

```
1  #include <dlfcn.h>
2  #include <stdio.h>
3
4  int main(void) {
5    void *h = dlopen("libm.so.6", RTLD_LAZY);
6    if (!h) { puts(dlerror()); return 1; }
7    double (*root)(double) =
8        (double (*)(double)) dlsym(h, "sqrt");
9    printf("%.3f\n", root(2.0));
▶    dlclose(h);
11   return 0;
12 }
```



`dlclose` : when the reference count reaches 0, the library is removed from the address space.

Step 5 / 5

VARIABLES

`h = closed` `root = –`

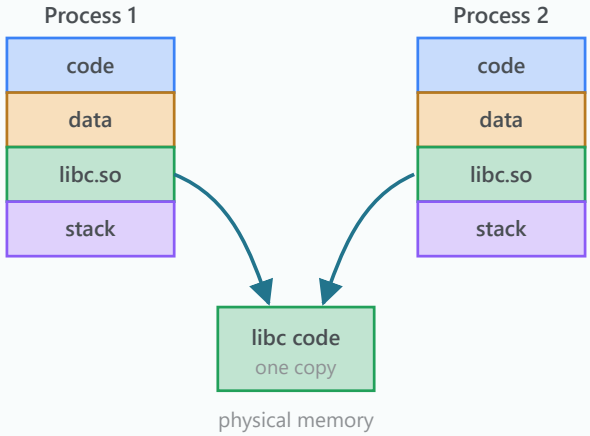
OUTPUT

```
1.414
```

A routine isn't loaded until it is called; unused code is never loaded. No hardware support is needed; the program is designed this way and the OS provides a library (`dlopen` ; `LoadLibrary` / `GetProcAddress` on Windows). Since glibc 2.34, `-ldl` is unnecessary.

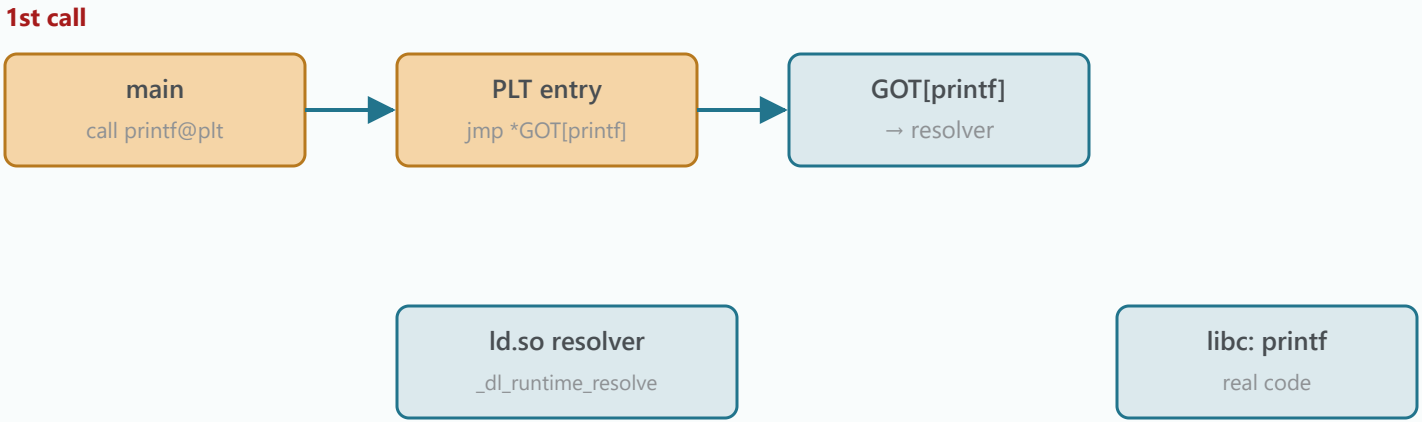
Static and Dynamic Linking

	Static linking	Dynamic linking
Who, and when?	The linker (`ld`), after compilation	The dynamic linker (`ld.so`), at load or run time
File	`.a`, `.lib`	`.so`, `.dll`, `.dylib`
Memory	A separate copy in every program	Code pages are in memory once, shared
Updates	Must relink	Replacing the library suffices (versioning: *soname*)
Risk	Large files; security patches are delayed	Version mismatches ("DLL hell")



- Static: library routines are copied into the binary (`ld *.o -lc -lm`).
- Dynamic: only a **stub** remains in the file; the address is resolved at execution time.

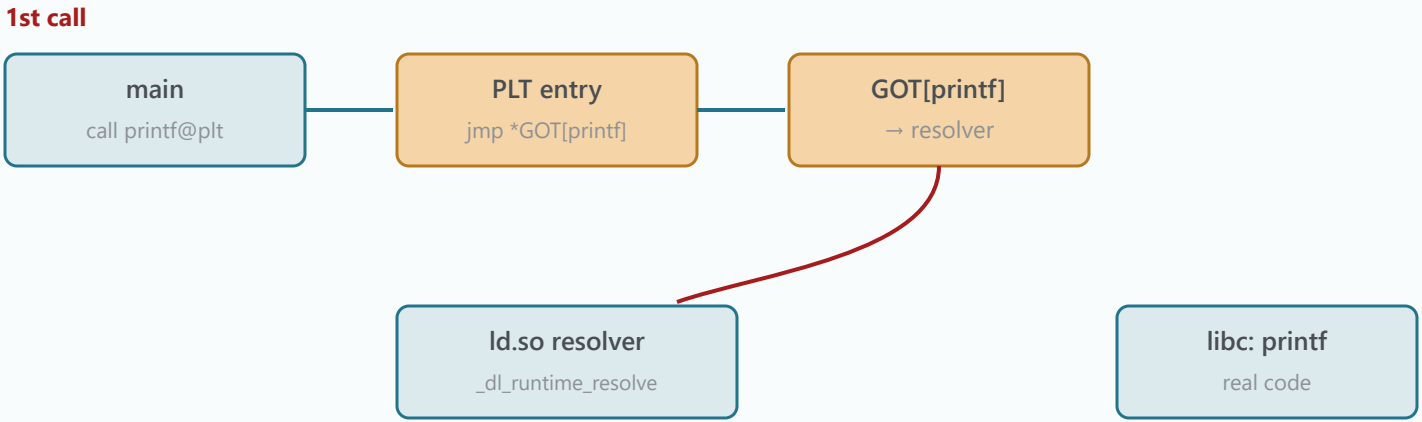
Lazy Binding: PLT and GOT



The compiler doesn't know `printf`'s address; the call goes to a small **PLT** (procedure linkage table) stub. This is the textbooks' *stub*.

The textbook account, "the stub replaces itself with the routine's address", is conceptual; on ELF systems what changes is the GOT entry, not code. And it's done by the dynamic linker in user space, not by the OS.

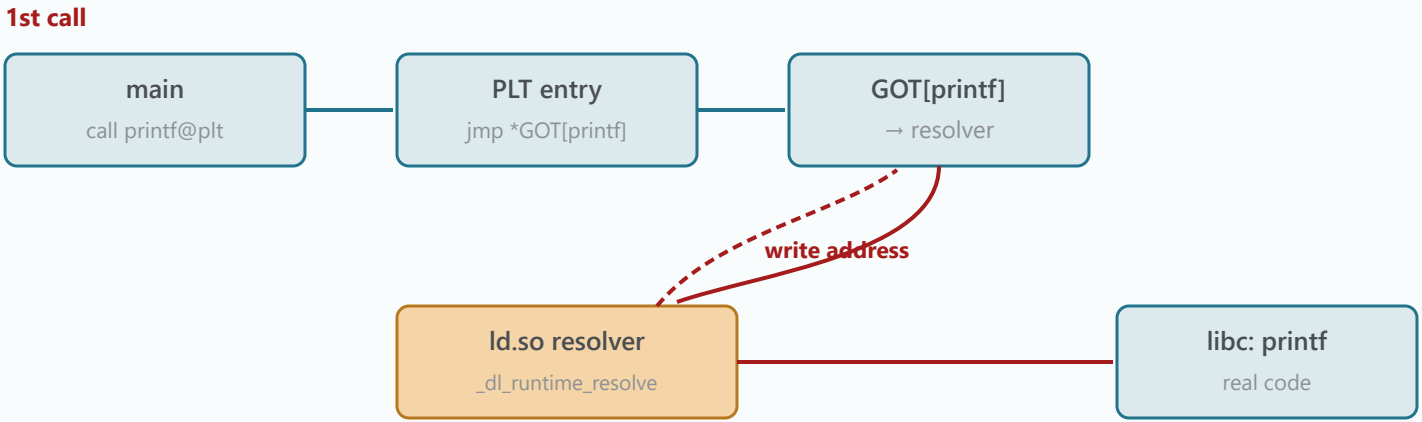
Lazy Binding: PLT and GOT



The PLT jumps indirectly through the address in the **GOT** (global offset table). Initially the GOT entry points to the resolver.

The textbook account, "the stub replaces itself with the routine's address", is conceptual; on ELF systems what changes is the GOT entry, not code. And it's done by the dynamic linker in user space, not by the OS.

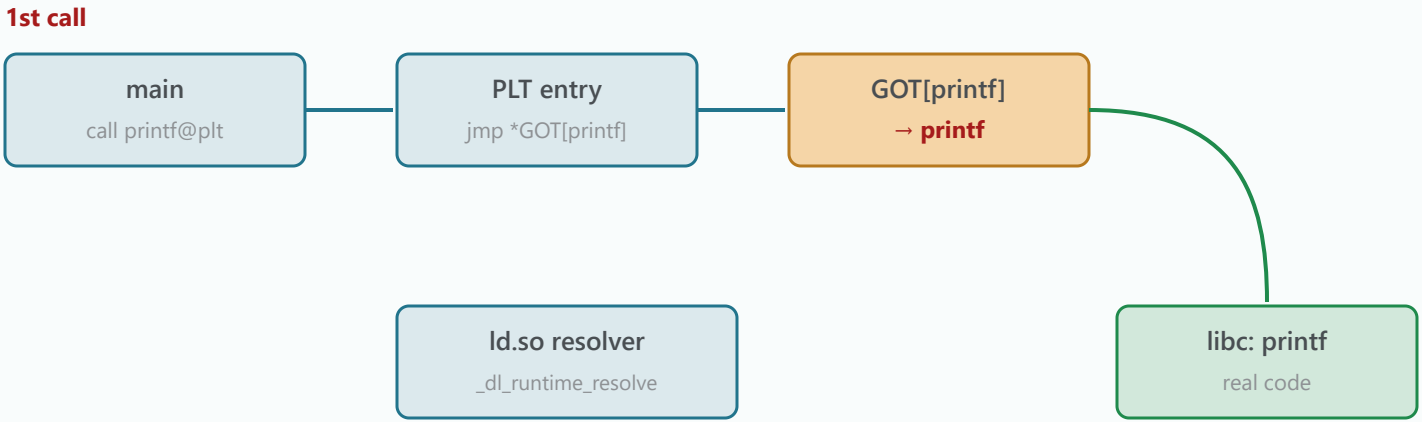
Lazy Binding: PLT and GOT



The `ld.so` resolver looks the symbol up in the loaded libraries and **writes** the address it finds **into the GOT**.

The textbook account, "the stub replaces itself with the routine's address", is conceptual; on ELF systems what changes is the GOT entry, not code. And it's done by the dynamic linker in user space, not by the OS.

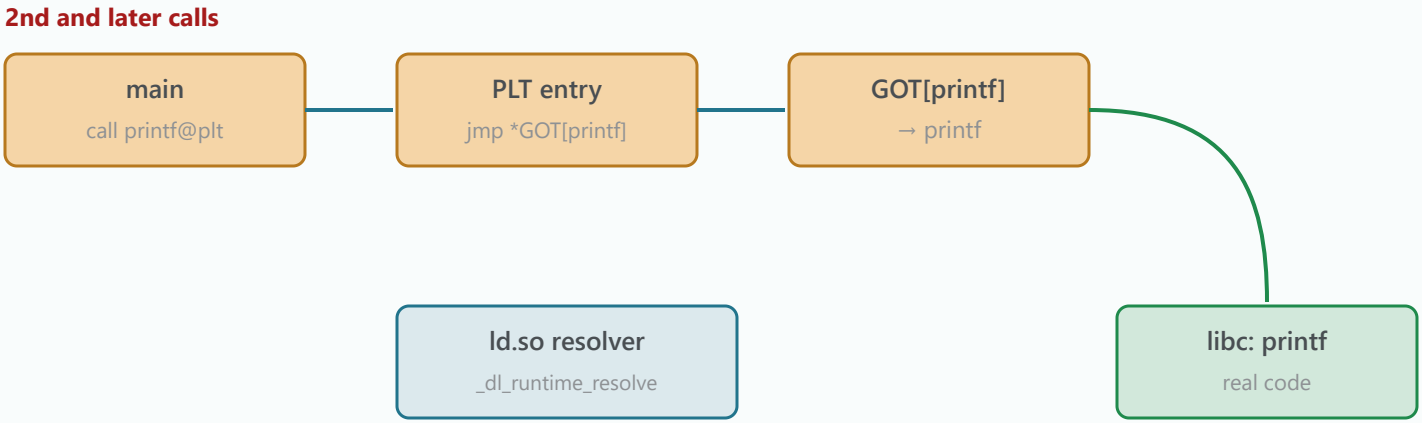
Lazy Binding: PLT and GOT



The resolver jumps to `printf`. The code pages didn't change; only the per-process GOT page did, so the code stays shareable.

The textbook account, "the stub replaces itself with the routine's address", is conceptual; on ELF systems what changes is the GOT entry, not code. And it's done by the dynamic linker in user space, not by the OS.

Lazy Binding: PLT and GOT



Later calls go PLT → GOT → `printf`: just one indirect jump. For security, many distributions use `-z now` (full RELRO) to resolve all symbols at load time and make the GOT read-only.

The textbook account, "the stub replaces itself with the routine's address", is conceptual; on ELF systems what changes is the GOT entry, not code. And it's done by the dynamic linker in user space, not by the OS.

Building Libraries: Static and Shared

Static library

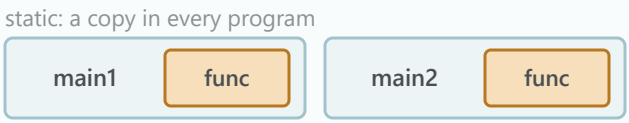
```
$ gcc -c func.c -o func.o
$ ar rcs libfunc.a func.o
$ gcc main.c -o main -static -L. -lfunc
$ ./main
```

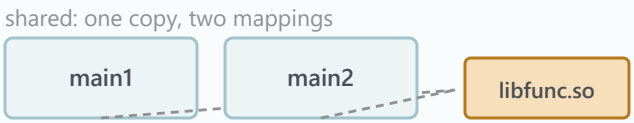
- `ar` archives object files; the linker copies only the objects it needs.
- `-static` also requires the static version of libc (`libc.a`).

Shared library

```
$ gcc -fPIC -c func.c -o func.o
$ gcc -shared -o libfunc.so func.o
$ gcc main.c -o main -L. -lfunc
$ export LD_LIBRARY_PATH=$(pwd)
$ ./main
$ ldd ./main   (libfunc.so => ./libfunc.so)
```

- `-fPIC` : **position-independent code**: PC-relative addressing, data via the GOT.
- So one physical page can map to a **different virtual address** in each process (ASLR).
- Permanent alternative to `LD_LIBRARY_PATH` : `-Wl,-rpath,'$ORIGIN'` or `ldconfig` .

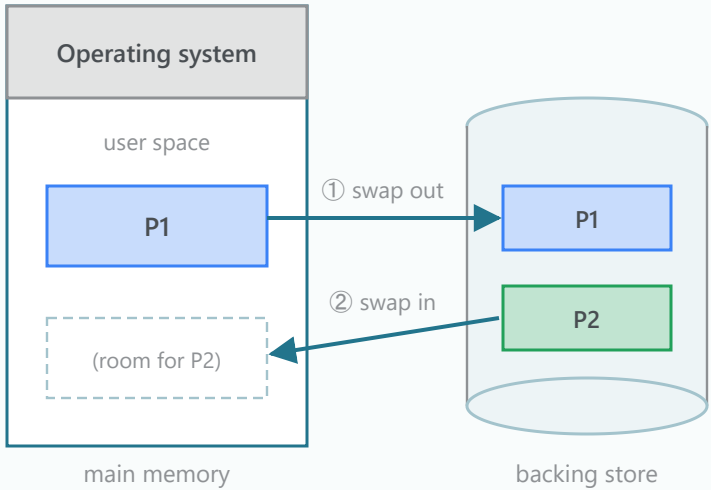




Distributions now build programs as PIE too, so ASLR loads the program itself at a random address.

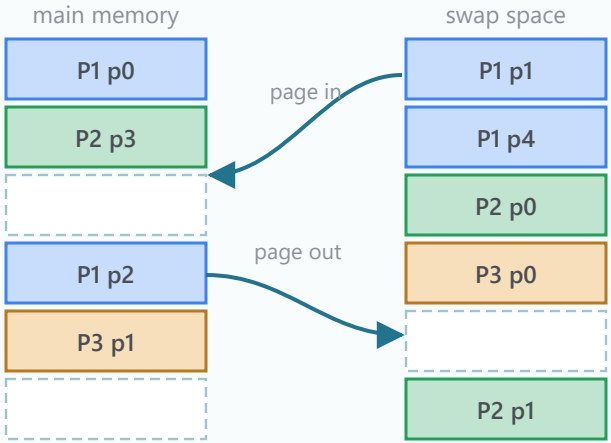
Swapping

- **Standard swapping**: an *entire* process is temporarily written to a backing store (disk), then brought back.
- Total memory demand can exceed physical memory; the degree of multiprogramming rises.
- A returning process may land at another address → **execution-time binding** is required.
- The cost: a 100 MB process, a 50 MB/s disk → 2 s out + 2 s in = **4 s** (Silberschatz).
- A process with pending I/O is not swapped (DMA would write to the old address), or its I/O goes through kernel buffers.



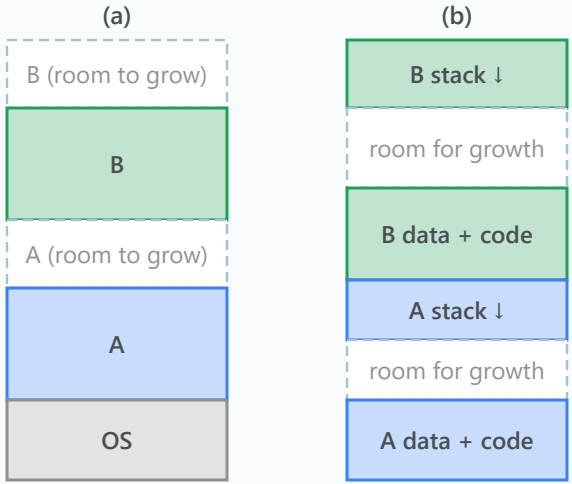
Page Swapping and Modern Systems

- Moving a whole process is unacceptable today: a 1 GiB process, 3 GB/s NVMe → round trip ≈ **0.72 s**.
- Linux, Windows, and macOS swap **pages**: only long-unused (cold) pages go out, and a 4 KiB page comes back in tens of µs.
- **Compression** before going to disk: Linux zswap and zram, Windows memory compression, the macOS compressor.
- Mobile: traditional swapping is absent or limited because of flash wear and energy; Android uses zram + `lmkd`, iOS compression + app termination (jetsam).



Contiguous Memory Allocation

- Memory is split into two regions: the **operating system** and **user processes**. Linux and Windows put the kernel in high memory.
- Each process sits in a single **contiguous** region, protected by base–limit.
- **Fixed partitions** (IBM OS/360 MFT): number of partitions = degree of multiprogramming; a small process in a large partition → *internal fragmentation*.
- **Variable partitions** (MVT): each partition is as large as its process; free blocks are called **holes**.
- Processes grow: **room is left** for the data segment (`malloc`) and the stack. A process returning from swap is brought in at the size it uses, not with its spare room.



Variable Partitions



holes: 896 K @ 128

Only the OS is in memory.

Step 1 / 8

- The OS keeps allocated partitions and holes in a table (or a list).
- When a process arrives, a large enough hole is found and the rest stays a hole; when it ends, its space returns as a hole and merges with its neighbors.

Variable Partitions

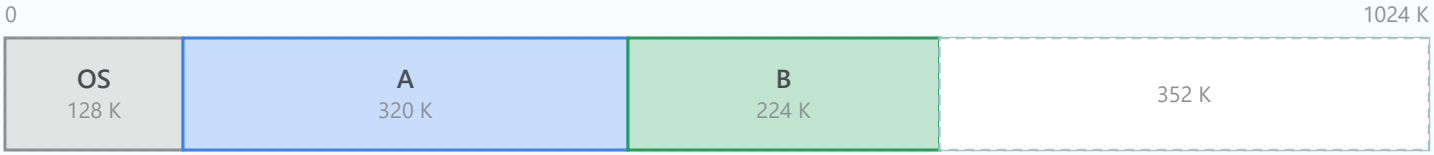


holes: 576 K @ 448

A (320 K) arrives and goes into the first hole that fits.

- The OS keeps allocated partitions and holes in a table (or a list).
- When a process arrives, a large enough hole is found and the rest stays a hole; when it ends, its space returns as a hole and merges with its neighbors.

Variable Partitions

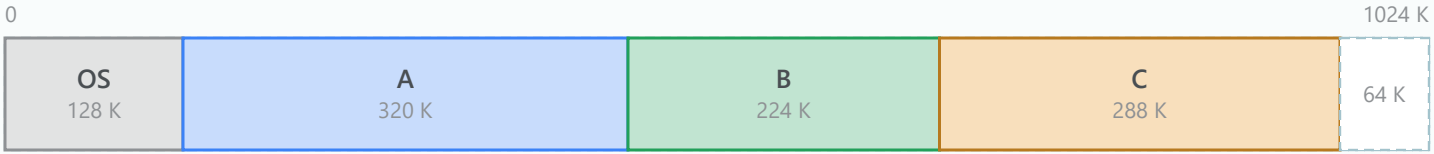


holes: 352 K @ 672

B (224 K) arrives and goes into the first hole that fits.

- The OS keeps allocated partitions and holes in a table (or a list).
- When a process arrives, a large enough hole is found and the rest stays a hole; when it ends, its space returns as a hole and merges with its neighbors.

Variable Partitions

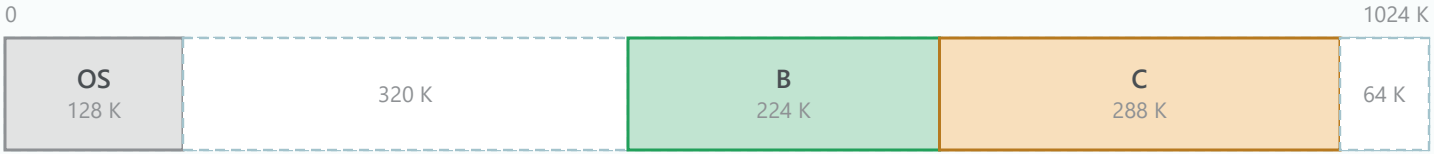


holes: 64 K @ 960

C (288 K) arrives and goes into the first hole that fits.

- The OS keeps allocated partitions and holes in a table (or a list).
- When a process arrives, a large enough hole is found and the rest stays a hole; when it ends, its space returns as a hole and merges with its neighbors.

Variable Partitions

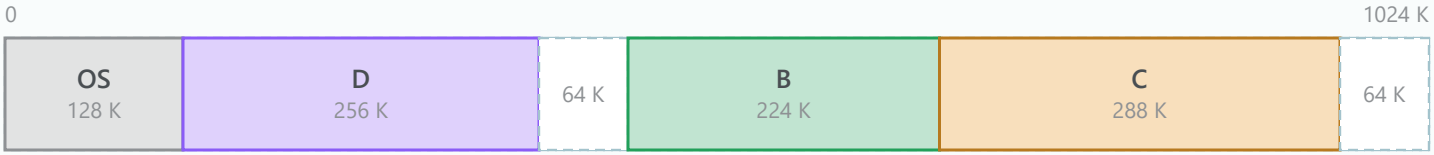


holes: 320 K @ 128, 64 K @ 960

A terminates: 128–448 becomes a **320 K** hole.

- The OS keeps allocated partitions and holes in a table (or a list).
- When a process arrives, a large enough hole is found and the rest stays a hole; when it ends, its space returns as a hole and merges with its neighbors.

Variable Partitions

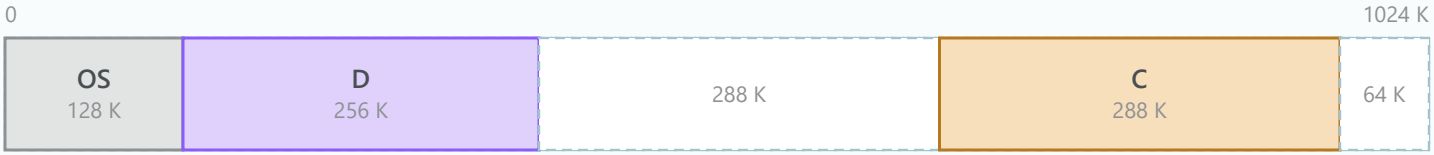


holes: 64 K @ 384, 64 K @ 960

D (256 K) goes into A's hole, leaving a small **64 K** hole.

- The OS keeps allocated partitions and holes in a table (or a list).
- When a process arrives, a large enough hole is found and the rest stays a hole; when it ends, its space returns as a hole and merges with its neighbors.

Variable Partitions



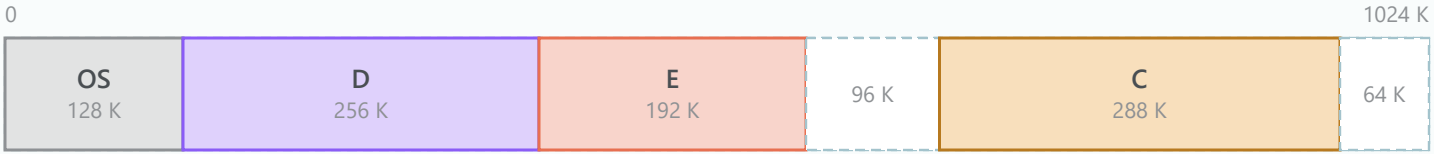
holes: 288 K @ 384, 64 K @ 960

B terminates: its space **merges** with the 64 K hole to its left → 288 K. Adjacent holes are coalesced at once.

Step 7 / 8

- The OS keeps allocated partitions and holes in a table (or a list).
- When a process arrives, a large enough hole is found and the rest stays a hole; when it ends, its space returns as a hole and merges with its neighbors.

Variable Partitions



holes: 96 K @ 576, 64 K @ 960

E (192 K) goes into the 288 K hole. 160 K is free in total but split **into two pieces**: external fragmentation is growing.

- The OS keeps allocated partitions and holes in a table (or a list).
- When a process arrives, a large enough hole is found and the rest stays a hole; when it ends, its space returns as a hole and merges with its neighbors.

Placement Algorithms

First fit

Scan from the start of the list and stop at the first hole that is big enough. Fast; small holes pile up near the front.

Next fit

The search resumes where it left off; slightly *worse* than first fit (Bays, 1977).

Best fit

Scan the whole list and pick the hole closest to the request. Slow; leaves **tiny**, useless holes.

Worst fit

Pick the largest hole so the leftover is still useful. Does poorly in simulations.

Quick fit

Separate lists for common sizes (4 K, 8 K, 12 K …). Finding is fast, merging is expensive.

A common improvement

Separate lists for processes and holes. Today's `malloc` implementations (jemalloc, tcmalloc) use size classes.

Simulations (Silberschatz, Knuth) show that first fit and best fit beat worst fit in both time and storage utilization; first fit is generally faster.

Best Fit: Code Trace

```
1  int best_fit(int hole[], int n, int req) {
▶    int best = -1;
3    for (int i = 0; i < n; i++)
4      if (hole[i] >= req &&
5          (best == -1 || hole[i] < hole[best]))
6        best = i;
7    if (best != -1) hole[best] -= req;
8    return best;
9  }
10 /* hole = 120, 450, 230, 300, 180 (K) */
11 /* best_fit(hole, 5, 210); best_fit(hole, 5, 160); */
```



The hole list is 120, 450, 230, 300, 180 K. The first request is **210 K**; `best = −1` (no candidate).

Step 1 / 12

VARIABLES

`req = 210` `i = –` `best = -1`

`hole = [120 450 230 300 180]`

OUTPUT

Best Fit: Code Trace

```
1  int best_fit(int hole[], int n, int req) {
2    int best = -1;
3    for (int i = 0; i < n; i++)
▶      if (hole[i] >= req &&
5          (best == -1 || hole[i] < hole[best]))
6        best = i;
7    if (best != -1) hole[best] -= req;
8    return best;
9  }
10 /* hole = 120, 450, 230, 300, 180 (K) */
11 /* best_fit(hole, 5, 210); best_fit(hole, 5, 160); */
```



[0] = 120 < 210: doesn't fit, skip.

Step 2 / 12

VARIABLES

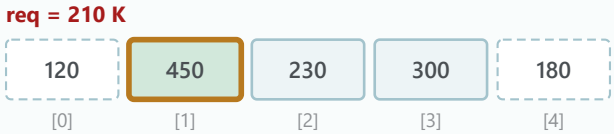
`req = 210` `i = 0` `best = -1`

`hole = [120 450 230 300 180]`

OUTPUT

Best Fit: Code Trace

```
1  int best_fit(int hole[], int n, int req) {
2    int best = -1;
3    for (int i = 0; i < n; i++)
▶      if (hole[i] >= req &&
5          (best == -1 || hole[i] < hole[best]))
▶        best = i;
7    if (best != -1) hole[best] -= req;
8    return best;
9  }
10 /* hole = 120, 450, 230, 300, 180 (K) */
11 /* best_fit(hole, 5, 210); best_fit(hole, 5, 160); */
```



[1] = 450 ≥ 210 and the first candidate → `best = 1`.

Step 3 / 12

VARIABLES

`req = 210` `i = 1` `best = 1`

`hole = [120 450 230 300 180]`

OUTPUT

Best Fit: Code Trace

```
1  int best_fit(int hole[], int n, int req) {
2    int best = -1;
3    for (int i = 0; i < n; i++)
▶      if (hole[i] >= req &&
5          (best == -1 || hole[i] < hole[best]))
▶        best = i;
7    if (best != -1) hole[best] -= req;
8    return best;
9  }
10 /* hole = 120, 450, 230, 300, 180 (K) */
11 /* best_fit(hole, 5, 210); best_fit(hole, 5, 160); */
```



[2] = 230 ≥ 210 and smaller than 450 → new best `best = 2`.

Step 4 / 12

VARIABLES

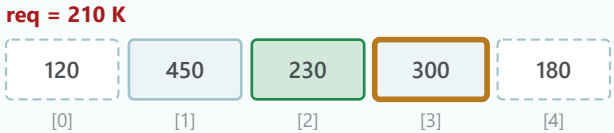
`req = 210` `i = 2` `best = 2`

`hole = [120 450 230 300 180]`

OUTPUT

Best Fit: Code Trace

```
1  int best_fit(int hole[], int n, int req) {
2    int best = -1;
3    for (int i = 0; i < n; i++)
▶      if (hole[i] >= req &&
▶          (best == -1 || hole[i] < hole[best]))
6        best = i;
7    if (best != -1) hole[best] -= req;
8    return best;
9  }
10 /* hole = 120, 450, 230, 300, 180 (K) */
11 /* best_fit(hole, 5, 210); best_fit(hole, 5, 160); */
```



[3] = 300 fits, but the current best, 230, is smaller: no change.

Step 5 / 12

VARIABLES

`req = 210` `i = 3` `best = 2`

`hole = [120 450 230 300 180]`

OUTPUT

Best Fit: Code Trace

```
1  int best_fit(int hole[], int n, int req) {
2    int best = -1;
3    for (int i = 0; i < n; i++)
▶      if (hole[i] >= req &&
5          (best == -1 || hole[i] < hole[best]))
6        best = i;
7    if (best != -1) hole[best] -= req;
8    return best;
9  }
10 /* hole = 120, 450, 230, 300, 180 (K) */
11 /* best_fit(hole, 5, 210); best_fit(hole, 5, 160); */
```



[4] = 180 < 210: doesn't fit, skip.

Step 6 / 12

VARIABLES

`req = 210` `i = 4` `best = 2`

`hole = [120 450 230 300 180]`

OUTPUT

Best Fit: Code Trace

```
1  int best_fit(int hole[], int n, int req) {
2    int best = -1;
3    for (int i = 0; i < n; i++)
4      if (hole[i] >= req &&
5          (best == -1 || hole[i] < hole[best]))
6        best = i;
▶    if (best != -1) hole[best] -= req;
▶    return best;
9  }
10 /* hole = 120, 450, 230, 300, 180 (K) */
11 /* best_fit(hole, 5, 210); best_fit(hole, 5, 160); */
```



210 K is carved from hole [2], leaving a small **20 K** hole. Returns 2. First fit would have split the 450 for the same request.

Step 7 / 12

VARIABLES

`req = 210` `best = 2`

`hole = [120 450 20 300 180]`

OUTPUT

```
best_fit(210) = 2
```

Best Fit: Code Trace

```
1  int best_fit(int hole[], int n, int req) {
2    int best = -1;
3    for (int i = 0; i < n; i++)
4      if (hole[i] >= req &&
5          (best == -1 || hole[i] < hole[best]))
6        best = i;
7    if (best != -1) hole[best] -= req;
8    return best;
9  }
10 /* hole = 120, 450, 230, 300, 180 (K) */
11 /* best_fit(hole, 5, 210); best_fit(hole, 5, 160); */
```



The second request is **160 K**; hole [2] is now 20 K. The scan starts over and walks the whole list.

Step 8 / 12

VARIABLES

`req = 160` `i = –` `best = -1`

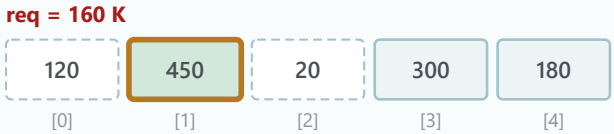
`hole = [120 450 20 300 180]`

OUTPUT

```
best_fit(210) = 2
```

Best Fit: Code Trace

```
1  int best_fit(int hole[], int n, int req) {
2    int best = -1;
3    for (int i = 0; i < n; i++)
▶      if (hole[i] >= req &&
5          (best == -1 || hole[i] < hole[best]))
▶        best = i;
7    if (best != -1) hole[best] -= req;
8    return best;
9  }
10 /* hole = 120, 450, 230, 300, 180 (K) */
11 /* best_fit(hole, 5, 210); best_fit(hole, 5, 160); */
```



[0] = 120 doesn't fit; [1] = 450 ≥ 160 and the first candidate → `best = 1`.

Step 9 / 12

VARIABLES

`req = 160` `i = 1` `best = 1`

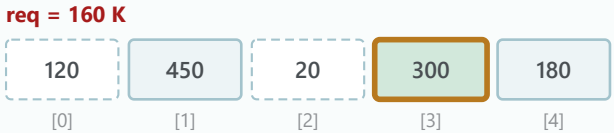
`hole = [120 450 20 300 180]`

OUTPUT

```
best_fit(210) = 2
```

Best Fit: Code Trace

```
1  int best_fit(int hole[], int n, int req) {
2    int best = -1;
3    for (int i = 0; i < n; i++)
▶      if (hole[i] >= req &&
5          (best == -1 || hole[i] < hole[best]))
▶        best = i;
7    if (best != -1) hole[best] -= req;
8    return best;
9  }
10 /* hole = 120, 450, 230, 300, 180 (K) */
11 /* best_fit(hole, 5, 210); best_fit(hole, 5, 160); */
```



[2] = 20 doesn't fit; [3] = 300 ≥ 160 and smaller than 450 → new best `best = 3`.

Step 10 / 12

VARIABLES

`req = 160` `i = 3` `best = 3`

`hole = [120 450 20 300 180]`

OUTPUT

```
best_fit(210) = 2
```

Best Fit: Code Trace

```
1  int best_fit(int hole[], int n, int req) {
2    int best = -1;
3    for (int i = 0; i < n; i++)
▶      if (hole[i] >= req &&
5          (best == -1 || hole[i] < hole[best]))
▶        best = i;
7    if (best != -1) hole[best] -= req;
8    return best;
9  }
10 /* hole = 120, 450, 230, 300, 180 (K) */
11 /* best_fit(hole, 5, 210); best_fit(hole, 5, 160); */
```



[4] = 180 ≥ 160 and smaller than 300 → new best `best = 4`.

Step 11 / 12

VARIABLES

`req = 160` `i = 4` `best = 4`

`hole = [120 450 20 300 180]`

OUTPUT

```
best_fit(210) = 2
```

Best Fit: Code Trace

```
1  int best_fit(int hole[], int n, int req) {
2    int best = -1;
3    for (int i = 0; i < n; i++)
4      if (hole[i] >= req &&
5          (best == -1 || hole[i] < hole[best]))
6        best = i;
▶    if (best != -1) hole[best] -= req;
▶    return best;
9  }
10 /* hole = 120, 450, 230, 300, 180 (K) */
11 /* best_fit(hole, 5, 210); best_fit(hole, 5, 160); */
```



160 K is carved from hole [4], leaving a small **20 K** hole. Returns 4. Best fit has now left two useless 20 K holes.

Step 12 / 12

VARIABLES

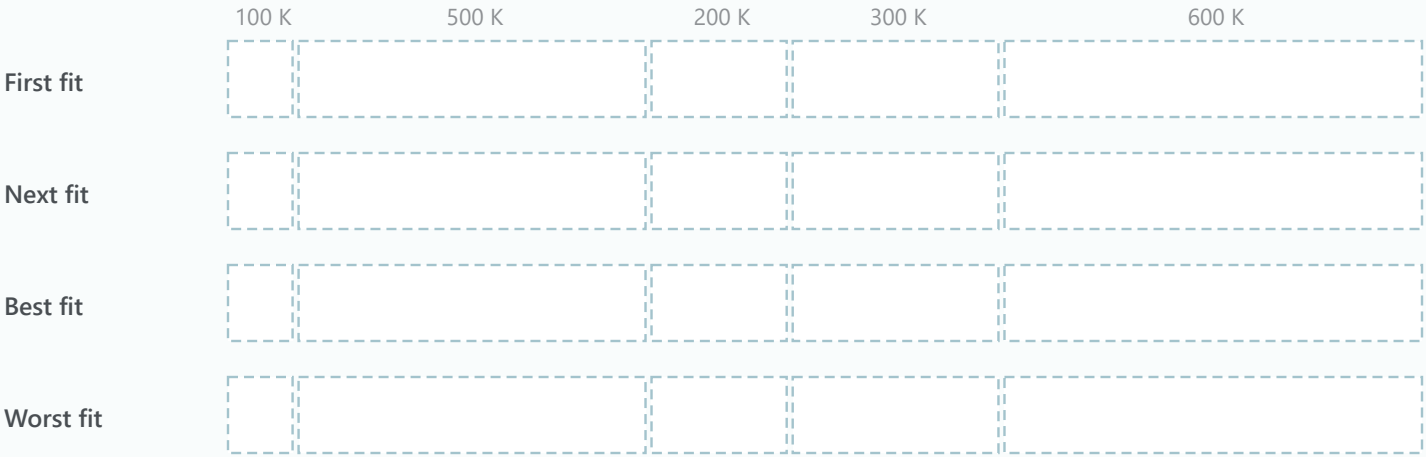
`req = 160` `best = 4`

`hole = [120 450 20 300 20]`

OUTPUT

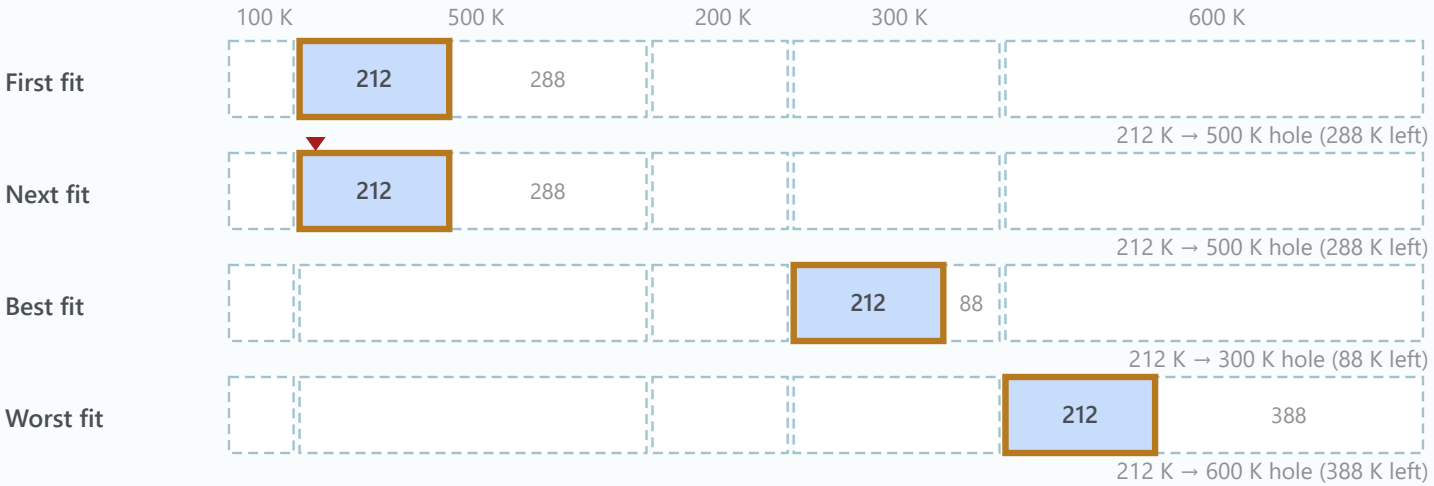
```
best_fit(210) = 2
best_fit(160) = 4
```

Placement: An Example



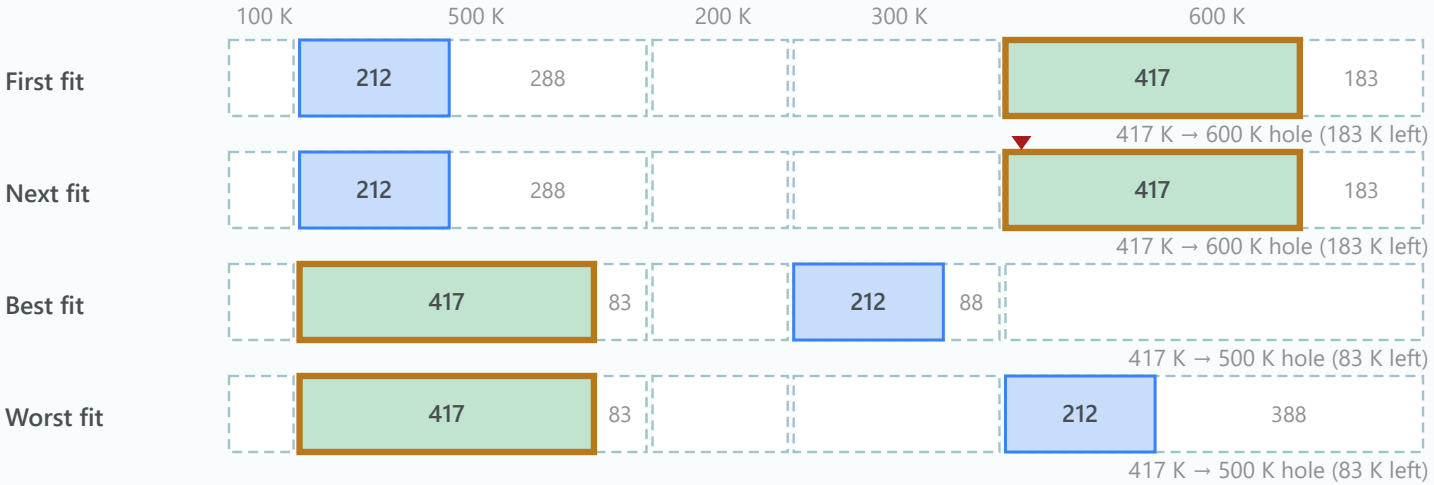
The holes, in order, are **100, 500, 200, 300, 600 K**. The requests, in order: **212, 417, 112, 426 K**.

Placement: An Example



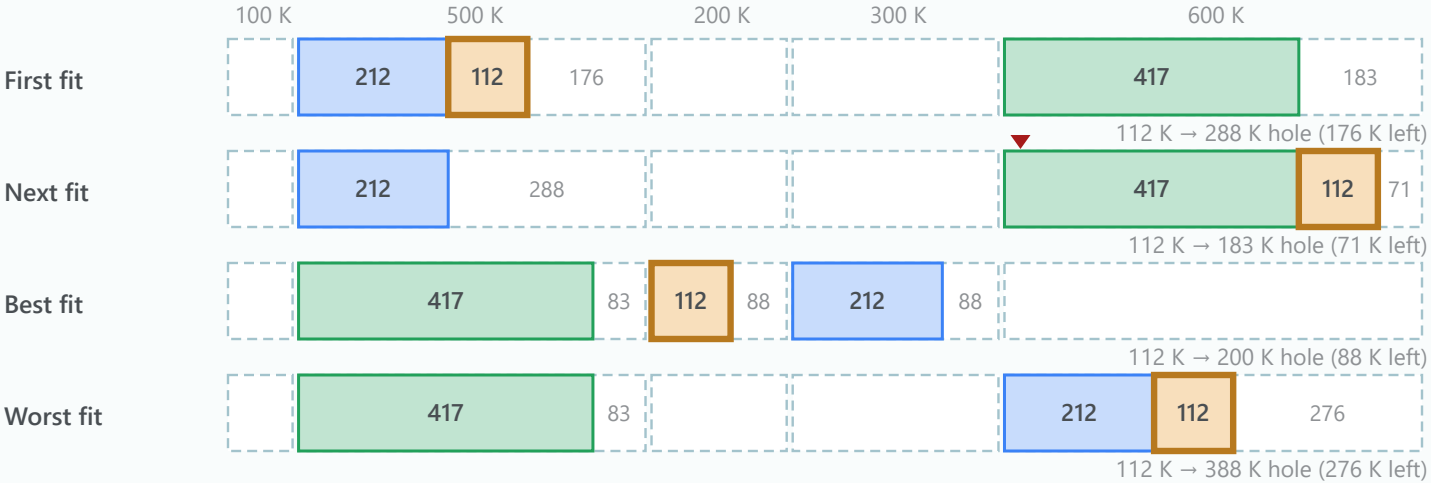
212 K: first/next fit put it in 500, best fit in 300 (88 left), worst fit in 600.

Placement: An Example



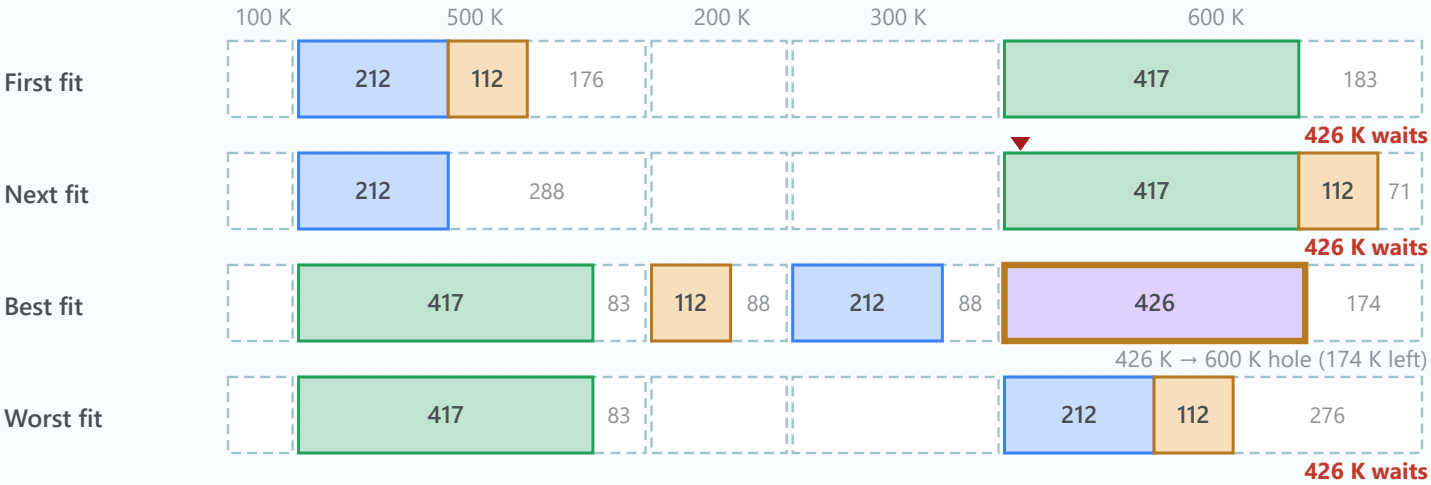
417 K: first and next fit use 600; best and worst fit use 500 (83 left).

Placement: An Example



112 K: first fit uses the 288 leftover; next fit the 183 where it last stopped (71 left); best fit 200; worst fit 388.

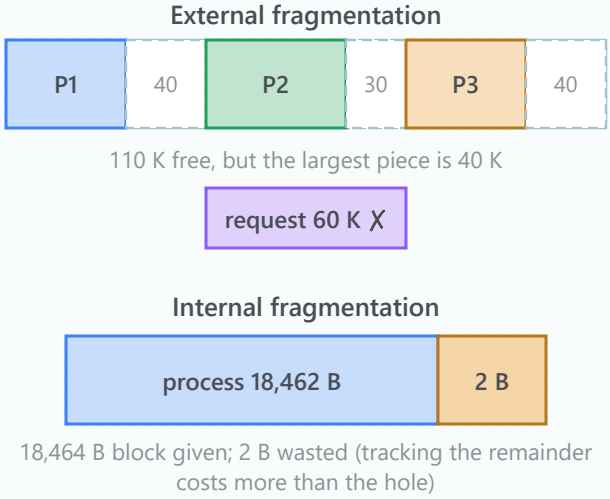
Placement: An Example



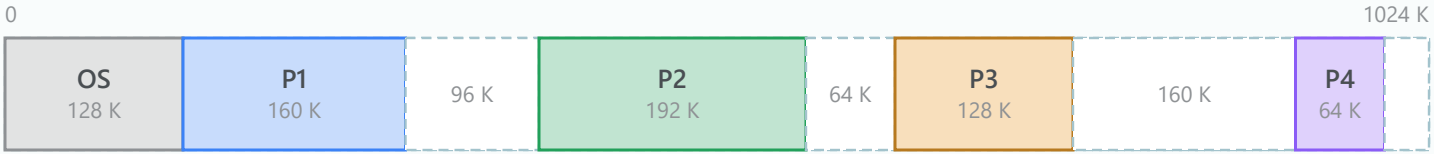
426 K: only **best fit** can place it (600 K is still whole). In the others there is enough free space in total, but not in one piece: **external fragmentation**.

Internal and External Fragmentation

- **External fragmentation**: there is enough total free memory for a request, but it is **not contiguous**. A problem of variable partitions.
- **The 50-percent rule** (Knuth): with first fit and N allocated blocks, about 0.5 N blocks are lost to fragmentation → **one third** of memory is unusable.
- **Internal fragmentation**: the block given is larger than requested; the difference is wasted *inside* the block. Seen with fixed partitions, paging, and the buddy system.
- Solutions: **compaction** (external), or letting a process's address space be **noncontiguous** (paging, segmentation).



Compaction

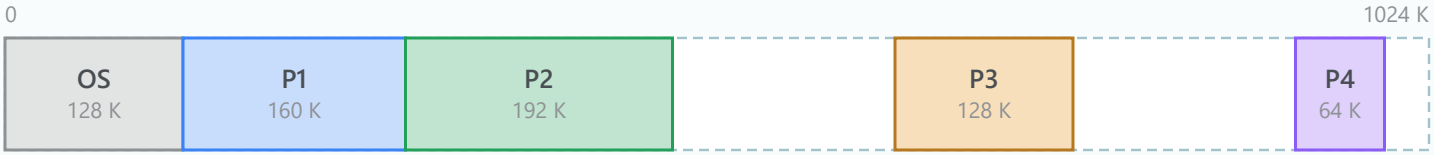


Initially there is **352 K** free among four processes (96 + 64 + 160 + 32), but the largest hole is 160 K.

Step 1 / 4

- **Compaction**: shuffle processes to one end and gather the holes into one block. Possible only with **execution-time binding** (a base register).
- Expensive: per Tanenbaum, 16 GB of memory at 8 bytes per 8 ns takes ≈ **16 s**; even at today's ≈20 GB/s copy rate it's ≈ 0.9 s, and processes stop meanwhile.
- Not to be confused with data *compression*: the goal isn't to shrink data but to make blocks *contiguous*.
- Today: Linux performs **page migration** to create contiguous physical frames for huge pages (`kcompactd`).

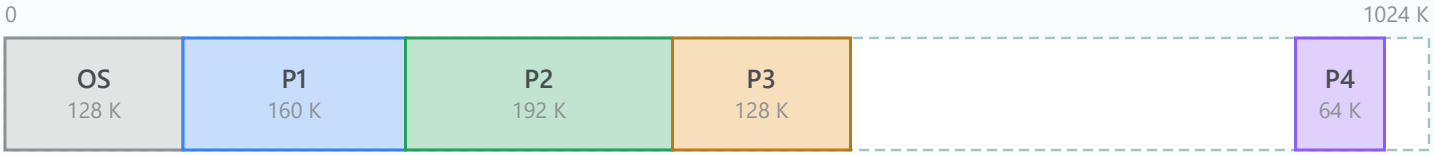
Compaction



P2 slides down: 384 → 288. The moved process's base register is updated.

- **Compaction**: shuffle processes to one end and gather the holes into one block. Possible only with **execution-time binding** (a base register).
- Expensive: per Tanenbaum, 16 GB of memory at 8 bytes per 8 ns takes ≈ **16 s**; even at today's ≈20 GB/s copy rate it's ≈ 0.9 s, and processes stop meanwhile.
- Not to be confused with data *compression*: the goal isn't to shrink data but to make blocks *contiguous*.
- Today: Linux performs **page migration** to create contiguous physical frames for huge pages (`kcompactd`).

Compaction

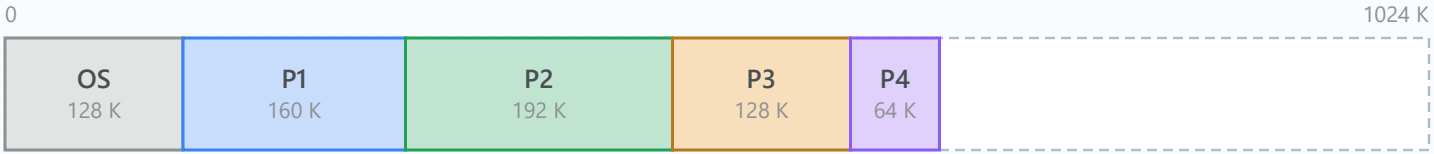


P3: 640 → 480.

Step 3 / 4

- **Compaction**: shuffle processes to one end and gather the holes into one block. Possible only with **execution-time binding** (a base register).
- Expensive: per Tanenbaum, 16 GB of memory at 8 bytes per 8 ns takes ≈ **16 s**; even at today's ≈20 GB/s copy rate it's ≈ 0.9 s, and processes stop meanwhile.
- Not to be confused with data *compression*: the goal isn't to shrink data but to make blocks *contiguous*.
- Today: Linux performs **page migration** to create contiguous physical frames for huge pages (`kcompactd`).

Compaction

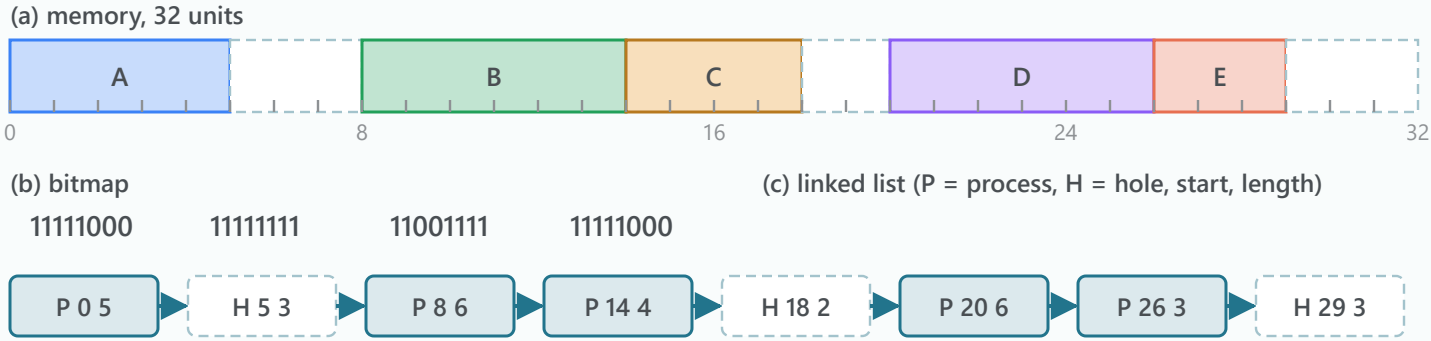


P4: 928 → 608. Now there is a single **352 K** hole. 384 K were copied in total.

Step 4 / 4

- **Compaction**: shuffle processes to one end and gather the holes into one block. Possible only with **execution-time binding** (a base register).
- Expensive: per Tanenbaum, 16 GB of memory at 8 bytes per 8 ns takes ≈ **16 s**; even at today's ≈20 GB/s copy rate it's ≈ 0.9 s, and processes stop meanwhile.
- Not to be confused with data *compression*: the goal isn't to shrink data but to make blocks *contiguous*.
- Today: Linux performs **page migration** to create contiguous physical frames for huge pages (`kcompactd`).

Tracking Free Memory: Bitmaps and Linked Lists



	Bitmap	**Linked list**
Structure	1 bit per allocation unit (0 free, 1 used)	A node per process/hole (P/H, start, length)
Size	Fixed: with 4 B units, 1/33 of memory; 16 GiB, 4 KiB units → 512 KiB	Proportional to the number of processes + holes
Search	Finding k consecutive 0s is **slow**	Walk the hole list; kept sorted by address, merging is easy

Searching a Bitmap for Free Space: Code Trace

```
1  /* find k consecutive free (0) units */
2  int find(const char bit[], int n, int k) {
▶    int start = 0, len = 0;
4    for (int i = 0; i < n; i++) {
5      if (bit[i] == 0) {
6        if (len == 0) start = i; /* new run */
7        if (++len == k) return start;
8      } else
9        len = 0;                 /* run broken */
10   }
11   return -1;
12 }
```



orange: current free run · green: area found

Memory has 10 units; used units are 1. We look for three consecutive 0s for a process of **k = 3** units.

Step 1 / 10

VARIABLES

`k = 3` `i = –` `start = 0` `len = 0`

OUTPUT

Searching a Bitmap for Free Space: Code Trace

```
1  /* find k consecutive free (0) units */
2  int find(const char bit[], int n, int k) {
3    int start = 0, len = 0;
4    for (int i = 0; i < n; i++) {
▶      if (bit[i] == 0) {
6        if (len == 0) start = i; /* new run */
7        if (++len == k) return start;
8      } else
▶        len = 0;                 /* run broken */
10   }
11   return -1;
12 }
```



orange: current free run · green: area found

bit[0] = 1: a used unit, len stays 0.

Step 2 / 10

VARIABLES

`k = 3` `i = 0` `start = 0` `len = 0`

OUTPUT

Searching a Bitmap for Free Space: Code Trace

```
1  /* find k consecutive free (0) units */
2  int find(const char bit[], int n, int k) {
3    int start = 0, len = 0;
4    for (int i = 0; i < n; i++) {
▶      if (bit[i] == 0) {
6        if (len == 0) start = i; /* new run */
7        if (++len == k) return start;
8      } else
▶        len = 0;                 /* run broken */
10   }
11   return -1;
12 }
```



orange: current free run · green: area found

bit[1] = 1: a used unit, len stays 0.

Step 3 / 10

VARIABLES

`k = 3` `i = 1` `start = 0` `len = 0`

OUTPUT

Searching a Bitmap for Free Space: Code Trace

```
1  /* find k consecutive free (0) units */
2  int find(const char bit[], int n, int k) {
3    int start = 0, len = 0;
4    for (int i = 0; i < n; i++) {
▶      if (bit[i] == 0) {
▶        if (len == 0) start = i; /* new run */
▶        if (++len == k) return start;
8      } else
9        len = 0;                 /* run broken */
10   }
11   return -1;
12 }
```



orange: current free run · green: area found

bit[2] = 0 with no run in progress → a new run, `start = 2`, len = 1.

Step 4 / 10

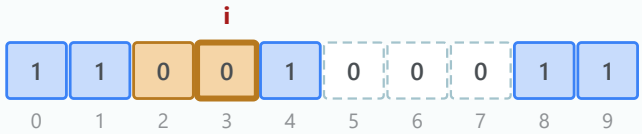
VARIABLES

`k = 3` `i = 2` `start = 2` `len = 1`

OUTPUT

Searching a Bitmap for Free Space: Code Trace

```
1  /* find k consecutive free (0) units */
2  int find(const char bit[], int n, int k) {
3    int start = 0, len = 0;
4    for (int i = 0; i < n; i++) {
▶      if (bit[i] == 0) {
6        if (len == 0) start = i; /* new run */
▶        if (++len == k) return start;
8      } else
9        len = 0;                 /* run broken */
10   }
11   return -1;
12 }
```



orange: current free run · green: area found

bit[3] = 0 → the run grows: len = 2; not yet k.

Step 5 / 10

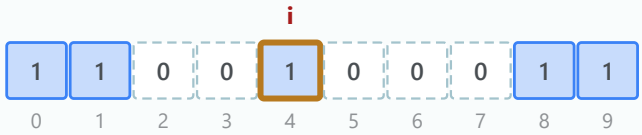
VARIABLES

`k = 3` `i = 3` `start = 2` `len = 2`

OUTPUT

Searching a Bitmap for Free Space: Code Trace

```
1  /* find k consecutive free (0) units */
2  int find(const char bit[], int n, int k) {
3    int start = 0, len = 0;
4    for (int i = 0; i < n; i++) {
▶      if (bit[i] == 0) {
6        if (len == 0) start = i; /* new run */
7        if (++len == k) return start;
8      } else
▶        len = 0;                 /* run broken */
10   }
11   return -1;
12 }
```



orange: current free run · green: area found

bit[4] = 1: the 2-unit run broke too soon → **len = 0**. The hole at 2–3 is too small for this process.

Step 6 / 10

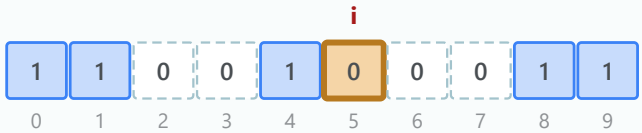
VARIABLES

`k = 3` `i = 4` `start = 2` `len = 0`

OUTPUT

Searching a Bitmap for Free Space: Code Trace

```
1  /* find k consecutive free (0) units */
2  int find(const char bit[], int n, int k) {
3    int start = 0, len = 0;
4    for (int i = 0; i < n; i++) {
▶      if (bit[i] == 0) {
▶        if (len == 0) start = i; /* new run */
▶        if (++len == k) return start;
8      } else
9        len = 0;                 /* run broken */
10   }
11   return -1;
12 }
```



orange: current free run · green: area found

bit[5] = 0 with no run in progress → a new run, `start = 5`, len = 1.

Step 7 / 10

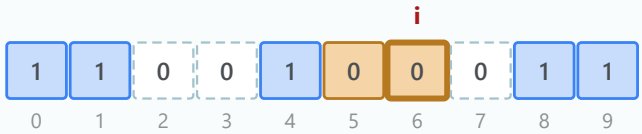
VARIABLES

`k = 3` `i = 5` `start = 5` `len = 1`

OUTPUT

Searching a Bitmap for Free Space: Code Trace

```
1  /* find k consecutive free (0) units */
2  int find(const char bit[], int n, int k) {
3    int start = 0, len = 0;
4    for (int i = 0; i < n; i++) {
▶      if (bit[i] == 0) {
6        if (len == 0) start = i; /* new run */
▶        if (++len == k) return start;
8      } else
9        len = 0;                 /* run broken */
10   }
11   return -1;
12 }
```



orange: current free run · green: area found

bit[6] = 0 → the run grows: len = 2; not yet k.

Step 8 / 10

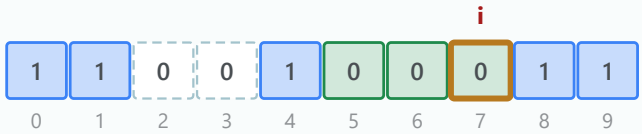
VARIABLES

`k = 3` `i = 6` `start = 5` `len = 2`

OUTPUT

Searching a Bitmap for Free Space: Code Trace

```
1  /* find k consecutive free (0) units */
2  int find(const char bit[], int n, int k) {
3    int start = 0, len = 0;
4    for (int i = 0; i < n; i++) {
▶      if (bit[i] == 0) {
6        if (len == 0) start = i; /* new run */
▶        if (++len == k) return start;
8      } else
9        len = 0;                 /* run broken */
10   }
11   return -1;
12 }
```



orange: current free run · green: area found

bit[7] = 0 → len = **3 = k**: the run is long enough; the start, **5**, is returned.

Step 9 / 10

VARIABLES

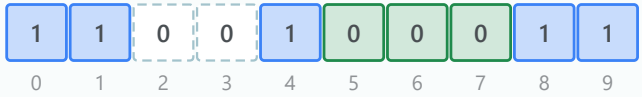
`k = 3` `i = 7` `start = 5` `len = 3`

OUTPUT

```
find = 5
```

Searching a Bitmap for Free Space: Code Trace

```
1  /* find k consecutive free (0) units */
2  int find(const char bit[], int n, int k) {
3    int start = 0, len = 0;
4    for (int i = 0; i < n; i++) {
5      if (bit[i] == 0) {
6        if (len == 0) start = i; /* new run */
▶        if (++len == k) return start;
8      } else
9        len = 0;                /* run broken */
10   }
11   return -1;
12 }
```



orange: current free run · green: area found

The allocator sets bits 5, 6, and 7 to 1. The search moves linearly from the start of the array, which is why it is **slow** for a large bitmap.

Step 10 / 10

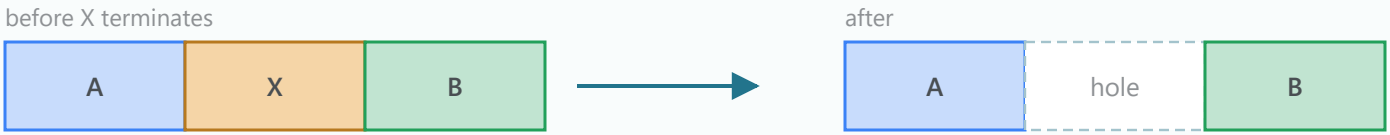
VARIABLES

`k = 3` `start = 5` `len = 3`

OUTPUT

```
find = 5
```

Freeing in a Linked List



Both neighbors are processes: X's node becomes an **H**; the list length stays the same.

Step 1 / 4

- If the list is **sorted by address**, merging takes constant time by looking at the neighbors.
- Keeping holes in a separate list **sorted by size** speeds up best fit (first fit becomes best fit) but slows down freeing.
- The hole list can be stored *inside* the holes themselves: the first words of each hole hold its size and a pointer to the next hole.

Freeing in a Linked List



Hole on the right: X and the hole on its right **merge**, and one node is deleted.

Step 2 / 4

- If the list is **sorted by address**, merging takes constant time by looking at the neighbors.
- Keeping holes in a separate list **sorted by size** speeds up best fit (first fit becomes best fit) but slows down freeing.
- The hole list can be stored *inside* the holes themselves: the first words of each hole hold its size and a pointer to the next hole.

Freeing in a Linked List



Hole on the left: the left hole grows by X's size, and X's node is deleted.

- If the list is **sorted by address**, merging takes constant time by looking at the neighbors.
- Keeping holes in a separate list **sorted by size** speeds up best fit (first fit becomes best fit) but slows down freeing.
- The hole list can be stored *inside* the holes themselves: the first words of each hole hold its size and a pointer to the next hole.

Freeing in a Linked List

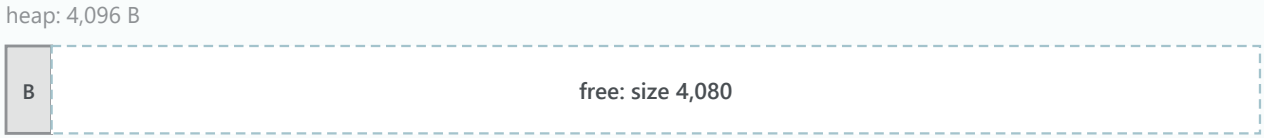


Holes on both sides: three nodes collapse into **one hole**, and two nodes are deleted. A **doubly linked list** makes finding the previous node easy.

Step 4 / 4

- If the list is **sorted by address**, merging takes constant time by looking at the neighbors.
- Keeping holes in a separate list **sorted by size** speeds up best fit (first fit becomes best fit) but slows down freeing.
- The hole list can be stored *inside* the holes themselves: the first words of each hole hold its size and a pointer to the next hole.

Inside malloc: Headers and an Embedded Free List



Initially: one free chunk. The free-list node lives **inside** the chunk; the allocator can't call `malloc` for a separate data structure.

Step 1 / 5

Magic number

If the fixed pattern in the header is corrupted (overflow, double `free`, a bad pointer), the library can notice and abort the program.

The real request

`malloc(n)` consumes at least n + header bytes, rounded up for alignment (glibc: 16 B).

A system call?

No, a library function. When space runs out it grows the heap with `brk` / `sbrk`; large requests (≥ 128 KiB in glibc) come straight from `mmap`.

Inside malloc: Headers and an Embedded Free List



`p1 = malloc(200)` : the first-fit chunk is **split**. The size and magic number go in the header.

Magic number

If the fixed pattern in the header is corrupted (overflow, double `free`, a bad pointer), the library can notice and abort the program.

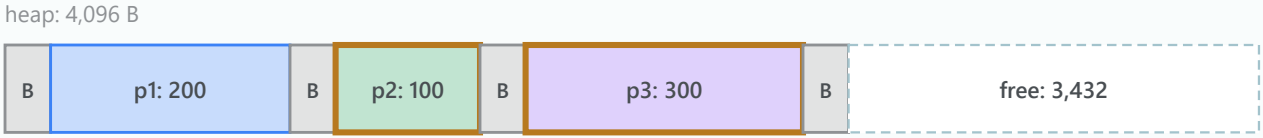
The real request

`malloc(n)` consumes at least n + header bytes, rounded up for alignment (glibc: 16 B).

A system call?

No, a library function. When space runs out it grows the heap with `brk` / `sbrk`; large requests (≥ 128 KiB in glibc) come straight from `mmap`.

Inside malloc: Headers and an Embedded Free List



H = 16 B header: in use {size, magic number}, free {size, next}

free list: **head → [3,432] → NULL**

3,864 − (100 + 16) − (300 + 16) = 3,432

`p2 = malloc(100)`, `p3 = malloc(300)`: the same splitting repeats.

Magic number

If the fixed pattern in the header is corrupted (overflow, double `free`, a bad pointer), the library can notice and abort the program.

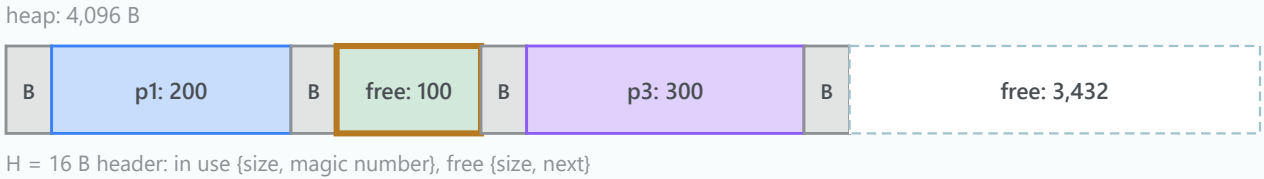
The real request

`malloc(n)` consumes at least n + header bytes, rounded up for alignment (glibc: 16 B).

A system call?

No, a library function. When space runs out it grows the heap with `brk` / `sbrk`; large requests (≥ 128 KiB in glibc) come straight from `mmap`.

Inside malloc: Headers and an Embedded Free List



free list: **head → [100] → [3,432] → NULL**

free(p2): header = (char *)p2 − 16 → reads size 100, puts the chunk at the head of the list

`free(p2)` takes no size: it is read from the header. Now a 150 B request won't fit the 100 chunk: **external fragmentation**.

Magic number

If the fixed pattern in the header is corrupted (overflow, double `free`, a bad pointer), the library can notice and abort the program.

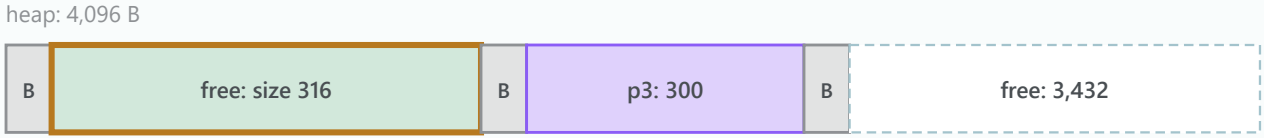
The real request

`malloc(n)` consumes at least n + header bytes, rounded up for alignment (glibc: 16 B).

A system call?

No, a library function. When space runs out it grows the heap with `brk` / `sbrk`; large requests (≥ 128 KiB in glibc) come straight from `mmap`.

Inside malloc: Headers and an Embedded Free List



H = 16 B header: in use {size, magic number}, free {size, next}

free list: **head → [316] → [3,432] → NULL**

free(p1): coalesces with the free neighbor: 216 + 116 = 332 B → size 332 − 16 = 316

`free(p1)` : two free chunks adjacent by address are **coalesced**; the list gets shorter.

Step 5 / 5

Magic number

If the fixed pattern in the header is corrupted (overflow, double `free`, a bad pointer), the library can notice and abort the program.

The real request

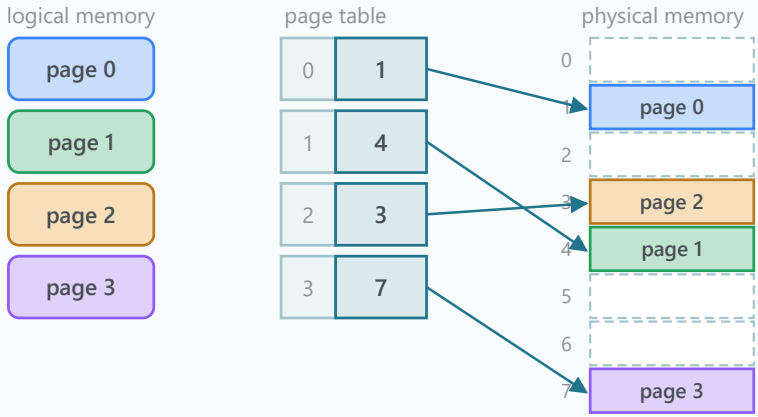
`malloc(n)` consumes at least n + header bytes, rounded up for alignment (glibc: 16 B).

A system call?

No, a library function. When space runs out it grows the heap with `brk` / `sbrk`; large requests (≥ 128 KiB in glibc) come straight from `mmap`.

Paging: The Basic Idea

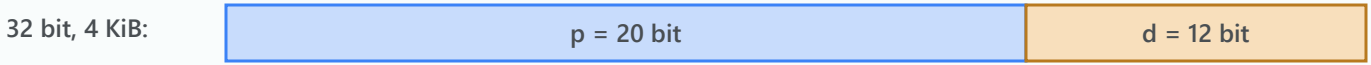
- Physical memory is divided into fixed-size **frames**, logical memory into **pages** of the same size.
- A process's physical space **need not be contiguous** → no external fragmentation; only internal fragmentation in the last page.
- A process of N pages needs N free frames; the OS keeps track of free frames.
- The **page table** maps page numbers to frame numbers; each process has its own table.
- The backing store is divided into blocks of the same size.



Splitting the Address: (p, d)



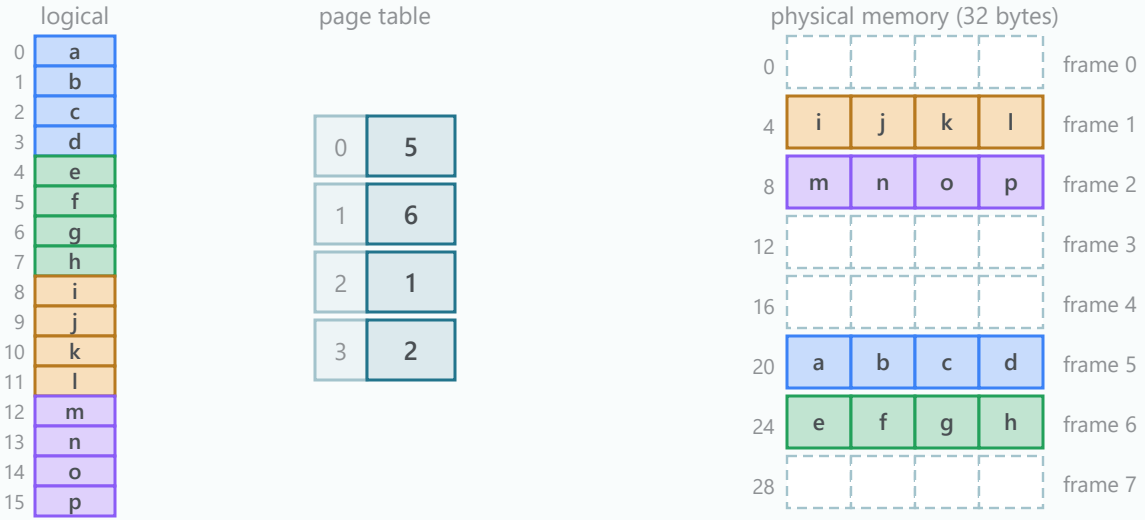
logical address space 2^m, page size 2^n



Example	Computation	Result
32-bit address, 4 KiB pages	4096 = 2^{12} → d is 12 bits, p = 32 − 12 = 20 bits	2^{20} = 1,048,576 pages
`0x12345678`	p = `0x12345678 >> 12`, d = `0x12345678 & 0xFFF`	p = `0x12345` (74,565), d = `0x678` (1656)
16-bit address, 4 KiB (Tanenbaum)	8196 = 2·4096 + 4	p = 2, d = 4
48-bit (x86-64), 4 KiB	p = 48 − 12	36-bit page number

Page sizes are powers of 2, so division and remainder are just a shift and a mask; hardware does them for free. Physical address = frame × page size + d.

Translation with a Page Table

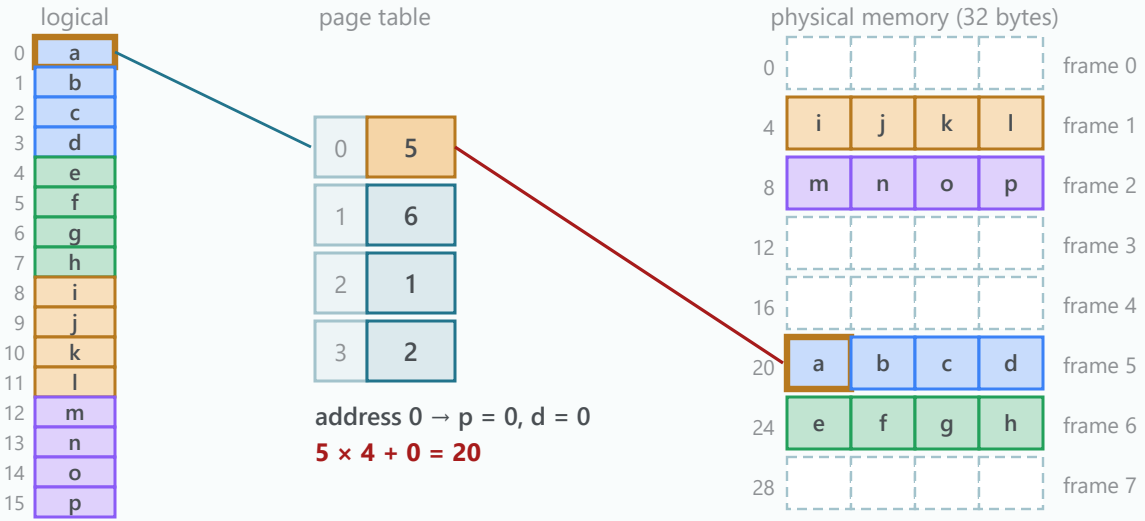


Page size **4 bytes**, physical memory **32 bytes** (8 frames). 16 bytes of logical memory, a…p; page table 0→5, 1→6, 2→1, 3→2.

Step 1 / 5

The example from Silberschatz Figure 9.10. The page-table entry's index is the page number; the page number itself isn't stored in the table.

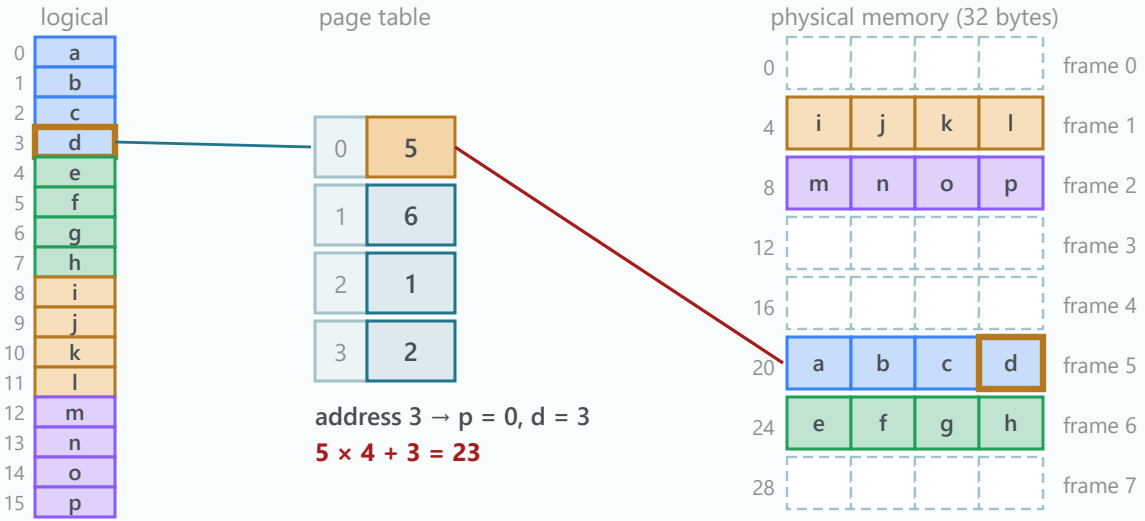
Translation with a Page Table



Logical **0** (`a`): page 0, offset 0 → frame 5 → physical **20**.

The example from Silberschatz Figure 9.10. The page-table entry's index is the page number; the page number itself isn't stored in the table.

Translation with a Page Table

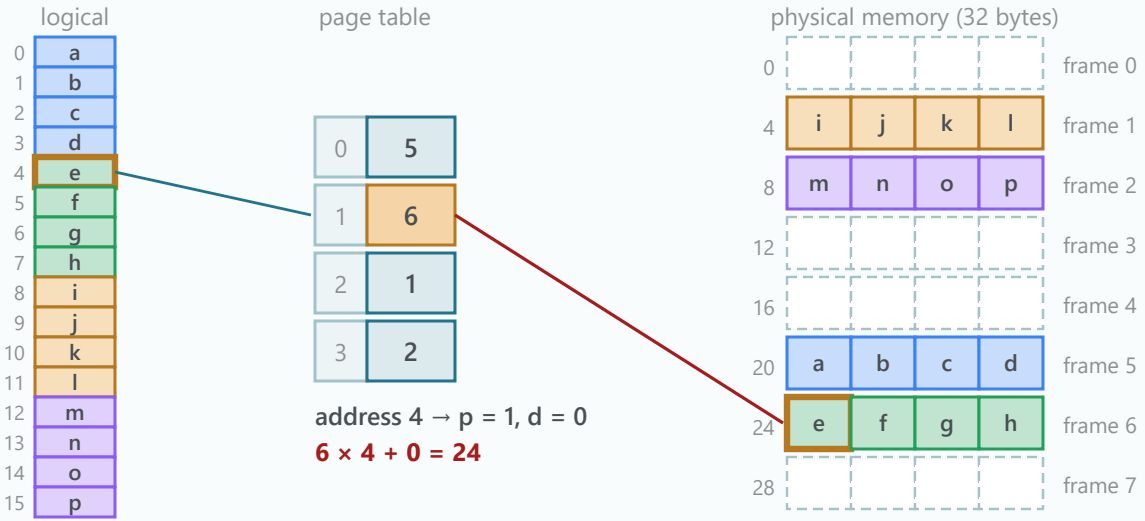


Logical **3** (d): page 0, offset 3 → frame 5 → physical **23**.

Step 3 / 5

The example from Silberschatz Figure 9.10. The page-table entry's index is the page number; the page number itself isn't stored in the table.

Translation with a Page Table

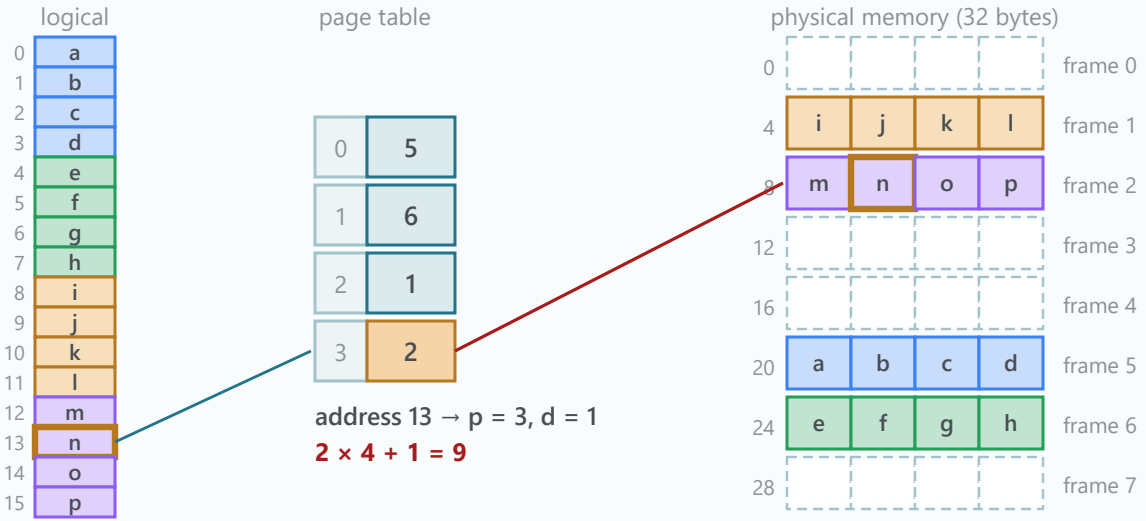


Logical **4** (e): page 1, offset 0 → frame 6 → physical **24**.

Step 4 / 5

The example from Silberschatz Figure 9.10. The page-table entry's index is the page number; the page number itself isn't stored in the table.

Translation with a Page Table



Logical **13** (`n`): page 3, offset 1 → frame 2 → physical **9**. The process's order isn't preserved in physical memory, yet the program sees it as contiguous.

The example from Silberschatz Figure 9.10. The page-table entry's index is the page number; the page number itself isn't stored in the table.

Translation Inside the MMU

```
1  /* 16-bit virtual address, 4 KiB
   pages */
▶  MOV REG, 0
3  MOV REG, 8192
4  MOV REG, 20500
5  MOV REG, 8196
6  MOV REG, 32780
7
8  /* page table (page → frame) */
9  /* 0→2 1→1 2→6 3→0 4→4 5→3 */
10 /* 9→5 11→7, others: absent */
```



Page 0 → frame 2: **8192**. The upper 4 bits are the page number; the lower 12 bits are copied unchanged.

Step 1 / 5

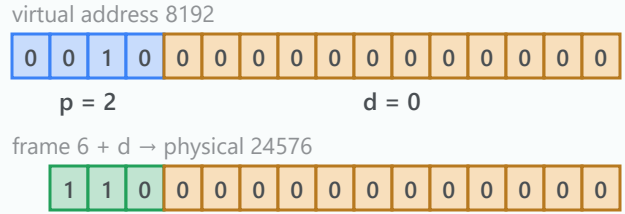
VARIABLES

`virtual = 0` `p = 0` `d = 0` `frame = 2` `physical = 8192`

64 KiB virtual address space (16 pages), 32 KiB physical memory (8 frames → 3-bit frame number).

Translation Inside the MMU

```
1  /* 16-bit virtual address, 4 KiB
   pages */
2  MOV REG, 0
▶  MOV REG, 8192
4  MOV REG, 20500
5  MOV REG, 8196
6  MOV REG, 32780
7
8  /* page table (page → frame) */
9  /* 0→2 1→1 2→6 3→0 4→4 5→3 */
10 /* 9→5 11→7, others: absent */
```



8192 = page 2, offset 0 → frame 6: **24576**.

Step 2 / 5

VARIABLES

`virtual = 8192` `p = 2` `d = 0` `frame = 6` `physical = 24576`

64 KiB virtual address space (16 pages), 32 KiB physical memory (8 frames → 3-bit frame number).

Translation Inside the MMU

```
1  /* 16-bit virtual address, 4 KiB
   pages */
2  MOV REG, 0
3  MOV REG, 8192
▶  MOV REG, 20500
5  MOV REG, 8196
6  MOV REG, 32780
7
8  /* page table (page → frame) */
9  /* 0→2 1→1 2→6 3→0 4→4 5→3 */
10 /* 9→5 11→7, others: absent */
```

virtual address 20500



frame 3 + d → physical 12308



20500 = 5·4096 + 20 → frame 3: 12288 + 20 = **12308**.

Step 3 / 5

VARIABLES

`virtual = 20500` `p = 5` `d = 20` `frame = 3` `physical = 12308`

64 KiB virtual address space (16 pages), 32 KiB physical memory (8 frames → 3-bit frame number).

Translation Inside the MMU

```
1  /* 16-bit virtual address, 4 KiB
   pages */
2  MOV REG, 0
3  MOV REG, 8192
4  MOV REG, 20500
▶  MOV REG, 8196
6  MOV REG, 32780
7
8  /* page table (page → frame) */
9  /* 0→2 1→1 2→6 3→0 4→4 5→3 */
10 /* 9→5 11→7, others: absent */
```

virtual address 8196



frame 6 + d → physical 24580



8196 = 2·4096 + 4 → 6·4096 + 4 = **24580** (Tanenbaum Figure 3-10).

Step 4 / 5

VARIABLES

`virtual = 8196` `p = 2` `d = 4` `frame = 6` `physical = 24580`

64 KiB virtual address space (16 pages), 32 KiB physical memory (8 frames → 3-bit frame number).

Translation Inside the MMU

```
1  /* 16-bit virtual address, 4 KiB
   pages */
2  MOV REG, 0
3  MOV REG, 8192
4  MOV REG, 20500
5  MOV REG, 8196
▶  MOV REG, 32780
7
8  /* page table (page → frame) */
9  /* 0→2 1→1 2→6 3→0 4→4 5→3 */
10 /* 9→5 11→7, others: absent */
```



32780 = 8·4096 + 12; page 8 is unmapped (present/absent bit 0) → **page fault**; the OS takes over.

Step 5 / 5

VARIABLES

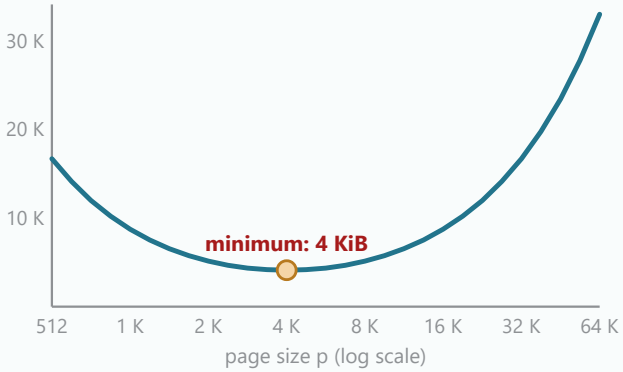
`virtual = 32780` `p = 8` `d = 12` `frame = –`

`physical = –`

64 KiB virtual address space (16 pages), 32 KiB physical memory (8 frames → 3-bit frame number).

Page Size and Internal Fragmentation

- Example: page 2,048 B, process 72,766 B → 35 pages + 1,086 B → **36 frames**, 962 B of internal fragmentation in the last page.
- Worst-case internal fragmentation is 1 page − 1 byte; on average, half a page.
- Small pages: little waste, but large page tables and many TLB misses. Large pages: efficient disk I/O, fewer table entries.
- Overhead = $s \cdot e/p + p/2$ (s process size, e entry size). Setting the derivative to zero: p = √(2se); s = 1 MiB, e = 8 B → **p = 4096**.
- Today: x86-64 4 KiB (+ 2 MiB, 1 GiB huge pages), ARM64 4/16/64 KiB, Apple Silicon 16 KiB.



Free Frames



- The OS keeps a **frame table**: for each physical frame, whether it is free and, if not, which page of which process it holds.
- When a user process passes an address (e.g., a `read` buffer), the kernel translates it to a physical address through the process's page table.
- On a context switch the dispatcher activates the new process's page table (`CR3` on x86): **context-switch time increases**.

The Page-Table Entry

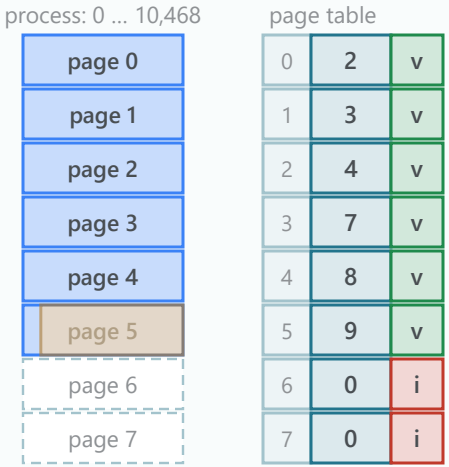
x86-64 page-table entry (64 bits, 4 KiB page)



Bit	Meaning (Tanenbaum's name)	Who writes it?
P	Present/absent: if 0, an access causes a page fault	OS
R/W, U/S, NX	Protection: writable? accessible in user mode? executable?	OS
A	Accessed (referenced): set on a read or write; used by page replacement	hardware (the OS clears it)
D	Dirty (modified): set on a write; a dirty page is written to disk when evicted	hardware
PCD, PWT, PAT	Cache disable / write policy: for memory-mapped I/O	OS
G	Global: not flushed from the TLB on a context switch (kernel pages)	OS

The disk address isn't kept in the PTE: when the page isn't in memory, the entry's remaining bits belong to the OS (Linux encodes the swap entry there).

The Valid–Invalid Bit



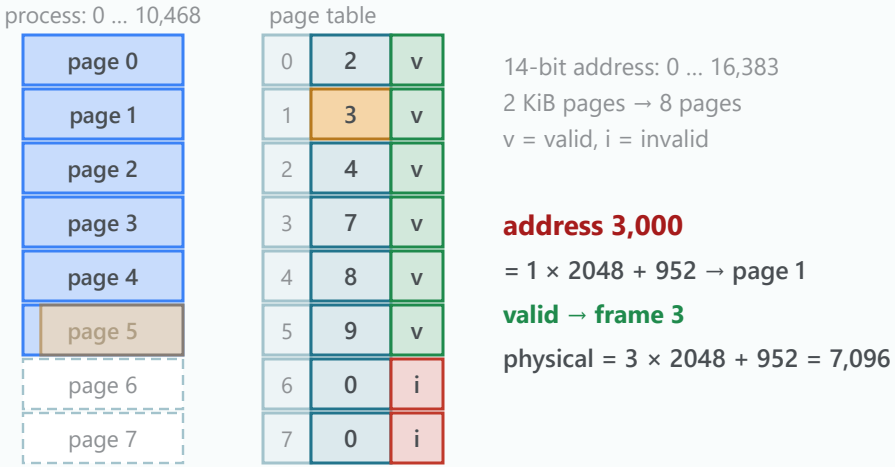
14-bit address: 0 … 16,383
2 KiB pages → 8 pages
v = valid, i = invalid

The process uses **0–10,468**: pages 0–5 are valid (frames 2, 3, 4, 7, 8, 9); pages 6 and 7 are **invalid**.

Step 1 / 5

Instead of a full table, a **page-table length register** (PTLR) can be used: a page number beyond the length causes a trap (Silberschatz Figure 9.13).

The Valid–Invalid Bit



14-bit address: 0 … 16,383
2 KiB pages → 8 pages
v = valid, i = invalid

address 3,000

= 1 × 2048 + 952 → page 1

valid → frame 3

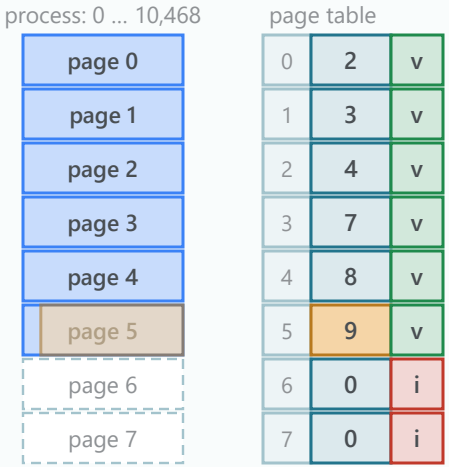
physical = 3 × 2048 + 952 = 7,096

3,000: page 1, offset 952 → frame 3 → **7,096**.

Step 2 / 5

Instead of a full table, a **page-table length register** (PTLR) can be used: a page number beyond the length causes a trap (Silberschatz Figure 9.13).

The Valid–Invalid Bit



14-bit address: 0 … 16,383
2 KiB pages → 8 pages
v = valid, i = invalid

address 10,468

= 5 × 2048 + 228 → page 5

valid → frame 9

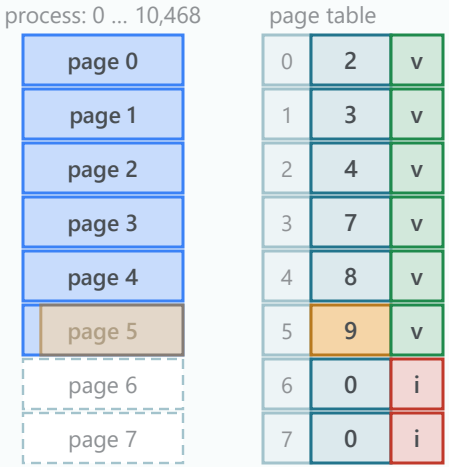
physical = 9 × 2048 + 228 = 18,660

10,468: inside page 5 → **18,660**. The process's last address.

Step 3 / 5

Instead of a full table, a **page-table length register** (PTLR) can be used: a page number beyond the length causes a trap (Silberschatz Figure 9.13).

The Valid–Invalid Bit



14-bit address: 0 … 16,383
2 KiB pages → 8 pages
v = valid, i = invalid

address 12,000

= 5 × 2048 + 1760 → page 5

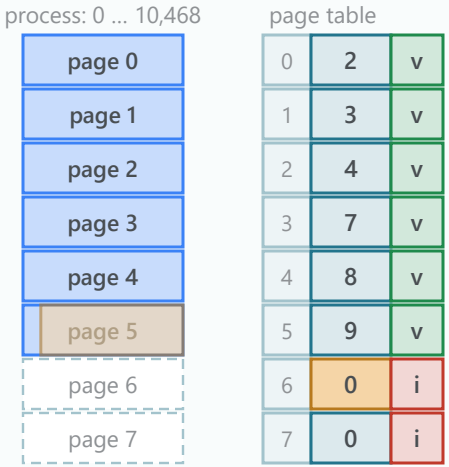
valid → frame 9

physical = 9 × 2048 + 1760 = 20,192

12,000 lies outside the program but inside page 5 (10,240–12,287) → the hardware **doesn't catch it**: 20,192. The price of page granularity, i.e., internal fragmentation.

Instead of a full table, a **page-table length register** (PTLR) can be used: a page number beyond the length causes a trap (Silberschatz Figure 9.13).

The Valid–Invalid Bit



14-bit address: 0 … 16,383
2 KiB pages → 8 pages
v = valid, i = invalid

address 12,500

= 6 × 2048 + 212 → page 6

invalid bit → trap (protection fault)

12,500: page 6, invalid → **trap**. The OS distinguishes an illegal access (SIGSEGV) from a page not yet loaded.

Instead of a full table, a **page-table length register** (PTLR) can be used: a page number beyond the length causes a trap (Silberschatz Figure 9.13).

Protection Bits

Read / write (R/W)

Code pages are mapped read-only; an attempt to write to code causes a protection fault.

User / supervisor (U/S)

Kernel pages are inaccessible in user mode. The same table can hold both kernel and process mappings.

No-execute (NX / XD)

Code cannot run from data, stack, or heap pages. OSes enforce **W^X**: a page is either writable or executable.

SMEP / SMAP

In kernel mode, prevents executing code in user pages (SMEP) and unauthorized access to user data (SMAP).

Protection keys (PKU)

A 4-bit key in the PTE plus a user register (PKRU) change a group of pages' permissions without a system call.

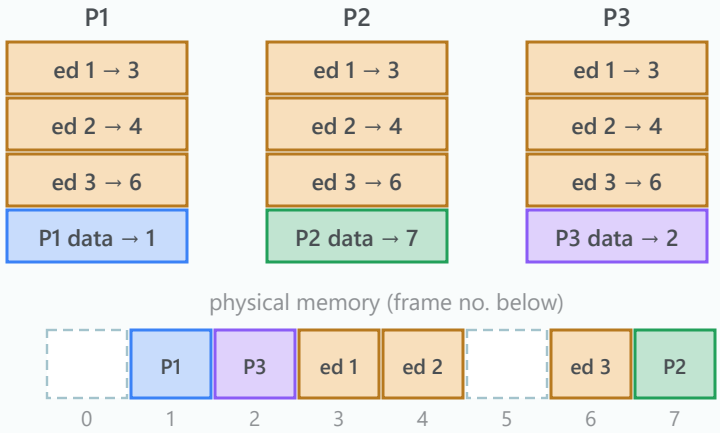
Valid / invalid

Access control at the coarsest level: if a page isn't in the address space at all, every access traps.

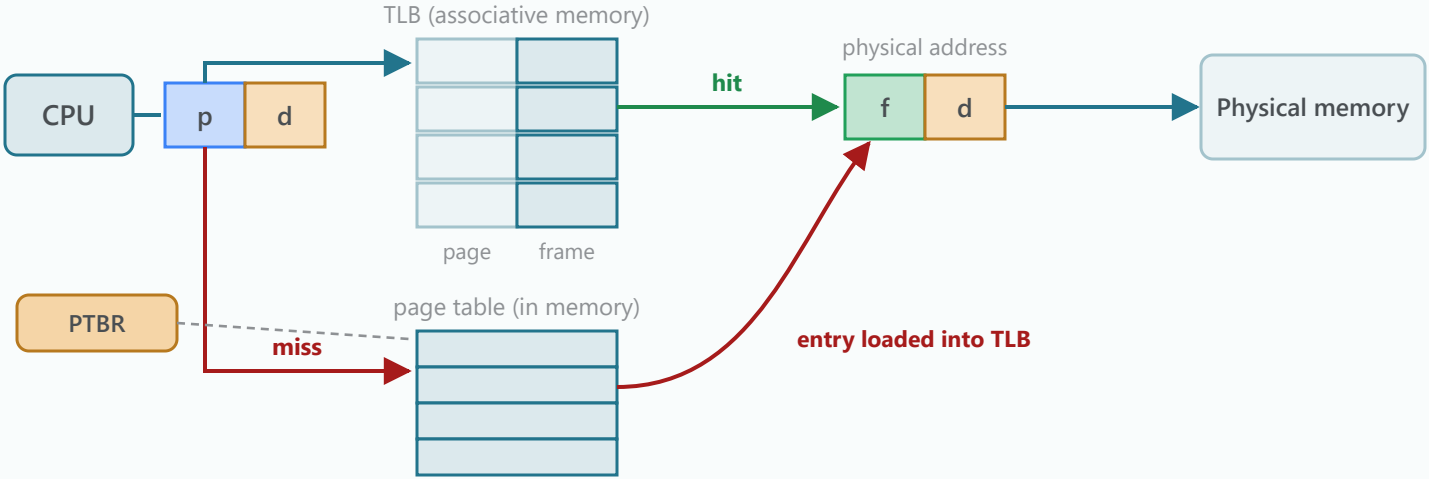
Protection is at page granularity: permissions are kept separately in each PTE and checked with the TLB entry on every access.

Shared Pages

- **Reentrant** code that never modifies itself can be shared: if three users run the same editor, its code frames are in memory once; only the data pages differ.
- Shared libraries (libc) are shared this way. When the owning process ends, the frame can't be freed at once: a **reference count** is kept.
- `fork()` : parent and child share all pages with **copy-on-write** (COW); pages are marked read-only and copied on the first write.
- **Memory-mapped files** (`mmap`) and shared memory (`shm_open`): two processes see the same frames and communicate without copying.



Where Is the Page Table? The TLB



- The table is too large to keep in registers; it lives in memory and the **PTBR** (x86: `CR3`, ARM: `TTBR0/1`) points to its start. But then every access becomes **two memory accesses**.
- **TLB** (translation look-aside buffer): a small, fast, associative (searched in parallel) cache inside the MMU; typically 64 L1 entries + 1–3 thousand L2 entries.
- If entries are tagged with an **ASID**/PCID, the TLB needn't be flushed on a context switch.

TLB Hits and Misses

access sequence (16 bits, 4 KiB pages)

0x0010	0x2004	0x0020	0x5010	0x9000	0x2008	0xB004	0x0030
p = 0	p = 2	p = 0	p = 5	p = 9	p = 2	p = 11	p = 0

TLB (4 entries, LRU; oldest to newest, left to right)

The page table is Tanenbaum's table from the MMU example (0→2, 2→6, 5→3, 9→5, 11→7). The TLB starts empty.

Step 1 / 9

TLB Hits and Misses

access sequence (16 bits, 4 KiB pages)

0x0010	0x2004	0x0020	0x5010	0x9000	0x2008	0xB004	0x0030
p = 0	p = 2	p = 0	p = 5	p = 9	p = 2	p = 11	p = 0
miss							

TLB (4 entries, LRU; oldest to newest, left to right)

0 → 2			

physical

2·4096 + 16 = 0x2010

0x0010: page 0 absent → **miss**; read from the page table and written to the TLB.

TLB Hits and Misses

access sequence (16 bits, 4 KiB pages)

0x0010	0x2004	0x0020	0x5010	0x9000	0x2008	0xB004	0x0030
p = 0	p = 2	p = 0	p = 5	p = 9	p = 2	p = 11	p = 0
miss	**miss**						

TLB (4 entries, LRU; oldest to newest, left to right)

0 → 2	2 → 6		

physical

6·4096 + 4 = 0x6004

0x2004: page 2 absent → **miss**; read from the page table and written to the TLB.

TLB Hits and Misses

access sequence (16 bits, 4 KiB pages)

0x0010	0x2004	0x0020	0x5010	0x9000	0x2008	0xB004	0x0030
p = 0	p = 2	p = 0	p = 5	p = 9	p = 2	p = 11	p = 0
miss	**miss**	**hit**					

TLB (4 entries, LRU; oldest to newest, left to right)

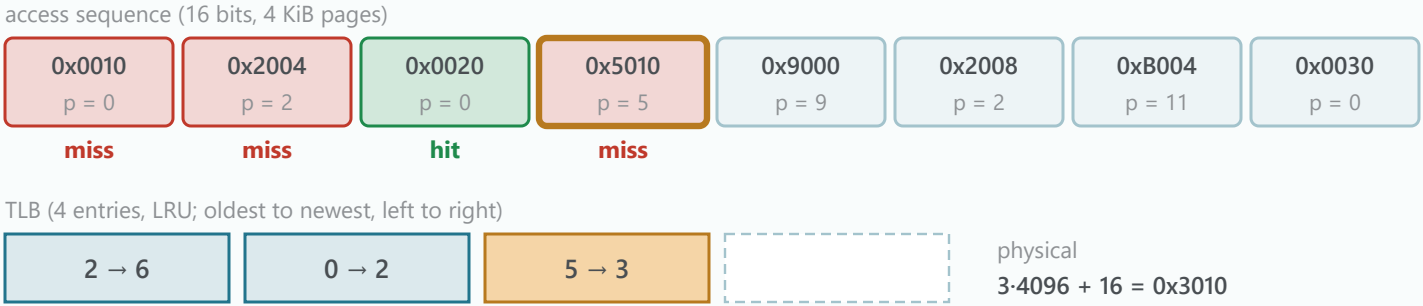
2 → 6	0 → 2		

physical

2·4096 + 32 = 0x2020

0x0020: page 0 is in the TLB → **hit**; no page-table access.

TLB Hits and Misses



0x5010: page 5 absent → **miss**; read from the page table and written to the TLB.

TLB Hits and Misses

access sequence (16 bits, 4 KiB pages)

0x0010	0x2004	0x0020	0x5010	0x9000	0x2008	0xB004	0x0030
p = 0	p = 2	p = 0	p = 5	p = 9	p = 2	p = 11	p = 0
miss	**miss**	**hit**	**miss**	**miss**			

TLB (4 entries, LRU; oldest to newest, left to right)

2 → 6	0 → 2	5 → 3	9 → 5

physical

5·4096 + 0 = 0x5000

0x9000: page 9 absent → **miss**; read from the page table and written to the TLB.

TLB Hits and Misses

access sequence (16 bits, 4 KiB pages)

0x0010	0x2004	0x0020	0x5010	0x9000	0x2008	0xB004	0x0030
p = 0	p = 2	p = 0	p = 5	p = 9	p = 2	p = 11	p = 0
miss	**miss**	**hit**	**miss**	**miss**	**hit**		

TLB (4 entries, LRU; oldest to newest, left to right)

0 → 2	5 → 3	9 → 5	2 → 6

physical

6·4096 + 8 = 0x6008

0x2008: page 2 is in the TLB → **hit**; no page-table access.

TLB Hits and Misses

access sequence (16 bits, 4 KiB pages)

0x0010	0x2004	0x0020	0x5010	0x9000	0x2008	0xB004	0x0030
p = 0	p = 2	p = 0	p = 5	p = 9	p = 2	p = 11	p = 0
miss	**miss**	**hit**	**miss**	**miss**	**hit**	**miss**	

TLB (4 entries, LRU; oldest to newest, left to right)

5 → 3	9 → 5	2 → 6	11 → 7

physical

7·4096 + 4 = 0x7004

evicted: page 0

0xB004: page 11 absent → **miss**; read from the page table and written to the TLB; it's full, so the least recently used page, 0, is evicted.

TLB Hits and Misses

access sequence (16 bits, 4 KiB pages)

0x0010	0x2004	0x0020	0x5010	0x9000	0x2008	0xB004	0x0030
p = 0	p = 2	p = 0	p = 5	p = 9	p = 2	p = 11	p = 0
miss	**miss**	**hit**	**miss**	**miss**	**hit**	**miss**	**miss**

TLB (4 entries, LRU; oldest to newest, left to right)

9 → 5	2 → 6	11 → 7	0 → 2

physical

2·4096 + 48 = 0x2030

evicted: page 5

0x0030: page 0 absent → **miss**; read from the page table and written to the TLB; it's full, so the least recently used page, 5, is evicted. Total: 2 hits / 8 accesses. In real programs locality pushes the ratio above 99%.

Effective Access Time

TLB lookup time **ε = 20 ns**, memory access **t = 100 ns**, hit ratio **α**.
EAT = (t + ε)·α + (2t + ε)·(1 − α) = 2t + ε − t·α

The formula: one memory access on a hit, two on a miss (table + data).

Effective Access Time

TLB lookup time **ε = 20 ns**, memory access **t = 100 ns**, hit ratio **α**.
EAT = (t + ε)·α + (2t + ε)·(1 − α) = 2t + ε − t·α

The formula: one memory access on a hit, two on a miss (table + data).

Hit: 20 + 100 = **120 ns**; miss: 20 + 100 + 100 = **220 ns**

The time for each case.

Effective Access Time

TLB lookup time **ε = 20 ns**, memory access **t = 100 ns**, hit ratio **α**.
EAT = (t + ε)·α + (2t + ε)·(1 − α) = 2t + ε − t·α

The formula: one memory access on a hit, two on a miss (table + data).

Hit: 20 + 100 = **120 ns**; miss: 20 + 100 + 100 = **220 ns**

The time for each case.

α = 80%: 120 × 0.80 + 220 × 0.20 = **140 ns** (40% slowdown)

At 80% hits the average is 140 ns: good compared with 200 ns without a TLB, but still 40% slower than memory.

Effective Access Time

TLB lookup time **ε = 20 ns**, memory access **t = 100 ns**, hit ratio **α**.
EAT = (t + ε)·α + (2t + ε)·(1 − α) = 2t + ε − t·α

The formula: one memory access on a hit, two on a miss (table + data).

Hit: 20 + 100 = **120 ns**; miss: 20 + 100 + 100 = **220 ns**

The time for each case.

α = 80%: 120 × 0.80 + 220 × 0.20 = **140 ns** (40% slowdown)

At 80% hits the average is 140 ns: good compared with 200 ns without a TLB, but still 40% slower than memory.

α = 98%: 120 × 0.98 + 220 × 0.02 = **122 ns**

At 98% hits, 122 ns. In real workloads the ratio is usually above 99%.

Effective Access Time

TLB lookup time **ε = 20 ns**, memory access **t = 100 ns**, hit ratio **α**.
EAT = (t + ε)·α + (2t + ε)·(1 − α) = 2t + ε − t·α

The formula: one memory access on a hit, two on a miss (table + data).

Hit: 20 + 100 = **120 ns**; miss: 20 + 100 + 100 = **220 ns**

The time for each case.

α = 80%: 120 × 0.80 + 220 × 0.20 = **140 ns** (40% slowdown)

At 80% hits the average is 140 ns: good compared with 200 ns without a TLB, but still 40% slower than memory.

α = 98%: 120 × 0.98 + 220 × 0.02 = **122 ns**

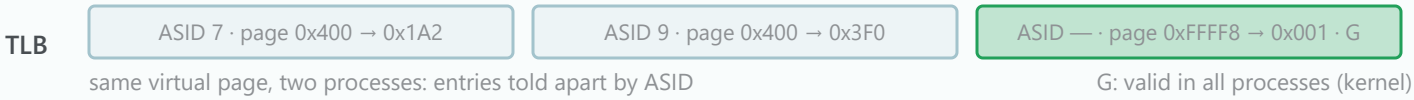
At 98% hits, 122 ns. In real workloads the ratio is usually above 99%.

Four-level table (x86-64): miss = 20 + 4·100 + 100 = **520 ns**; α = 98% → 128 ns, α = 99% → **124 ns**

With multilevel tables a miss is far more expensive; that's why hardware also caches the intermediate levels (a page-walk cache) and the TLB hit ratio matters even more.

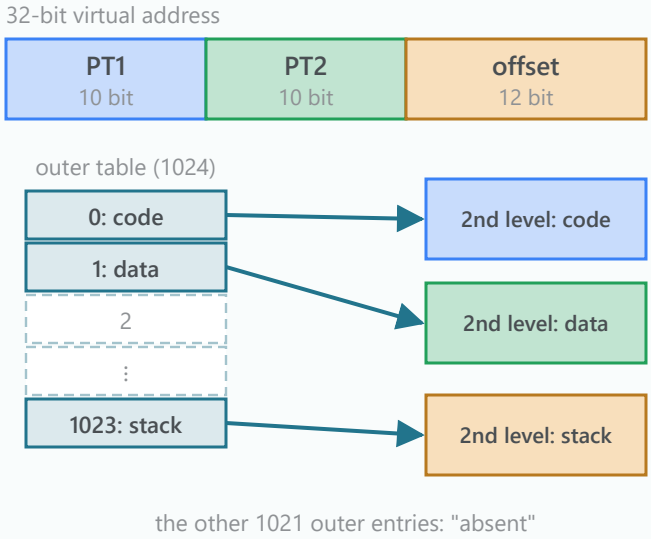
TLB Management

Topic	Description
Hardware-managed	x86-64, ARMv8, RISC-V: on a miss the MMU walks the page table itself (a page walk); the OS only builds the table.
Software-managed	In older RISCs such as MIPS, SPARC, and Alpha, a miss traps to the OS; the kernel finds the entry and writes it to the TLB. Not true of all RISCs: ARMv8 and RISC-V are hardware-managed.
Soft / hard miss	Soft miss: the page is in memory, just not in the TLB (ns). Hard miss: the page is on disk and needs I/O (µs–ms).
ASID / PCID	Entries are tagged with a process ID, so the TLB isn't flushed on a context switch. After Meltdown (2018), PCID reduces the cost of KPTI.
Global entries	Kernel mappings are marked with the G bit and stay valid in all processes.
TLB shootdown	On multicore, when a mapping changes, the other cores' TLBs are invalidated with interrupts (IPIs); newer processors have broadcast instructions (AMD `INVLPGB`).

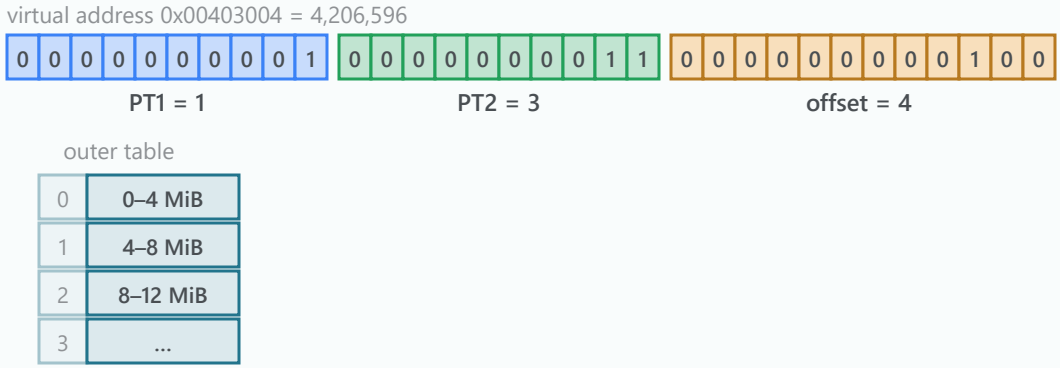


Hierarchical (Two-Level) Page Tables

- A single-level table with 32 bits and 4 KiB pages has 2^{20} entries × 4 B = **4 MiB**, for *every process*, and it must be contiguous.
- Single-level at 64 bits: 2^{52} × 8 B = **32 PiB** (≈ 33.6 million GiB). Even using 48 bits, 2^{36} × 8 B = 512 GiB.
- The solution: **page the page table**. 32-bit x86: 10 bits for the outer table (page directory), 10 bits for the second level, 12 bits of offset. What gets split is the **virtual address**, not the PTE.
- Only regions in use get a second-level table: a process using 4 MiB of code + 4 MiB of data + 4 MiB of stack → 1 + 3 tables = **16 KiB** (instead of 4 MiB).



A Two-Level Translation Example

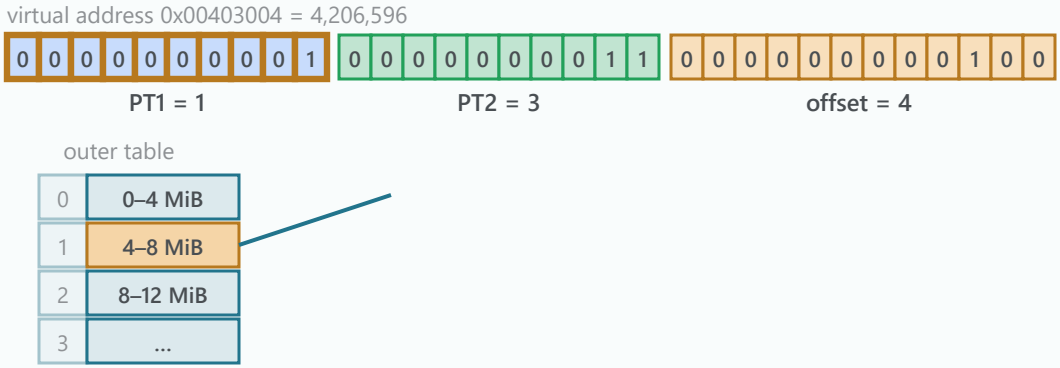


Virtual address **0x00403004** (4,206,596): between 4 MiB and 8 MiB, in the data region (Tanenbaum Figure 3-13).

Step 1 / 4

The frame numbers are illustrative. On 32-bit x86 the outer table's address is in `CR3`; the table at each level fills exactly one 4 KiB page (1024 × 4 B).

A Two-Level Translation Example

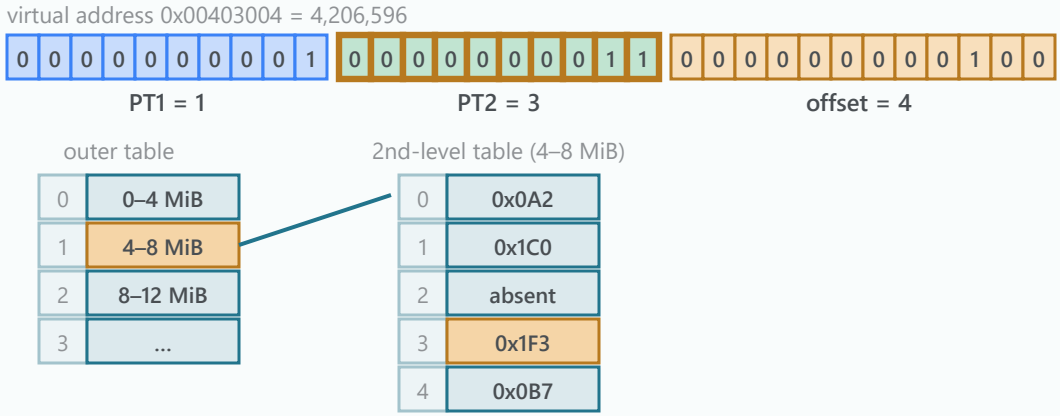


The top 10 bits, **PT1 = 1**: outer-table entry 1 points to the second-level table for the 4–8 MiB region. Each entry covers 4 MiB.

Step 2 / 4

The frame numbers are illustrative. On 32-bit x86 the outer table's address is in `CR3`; the table at each level fills exactly one 4 KiB page (1024 × 4 B).

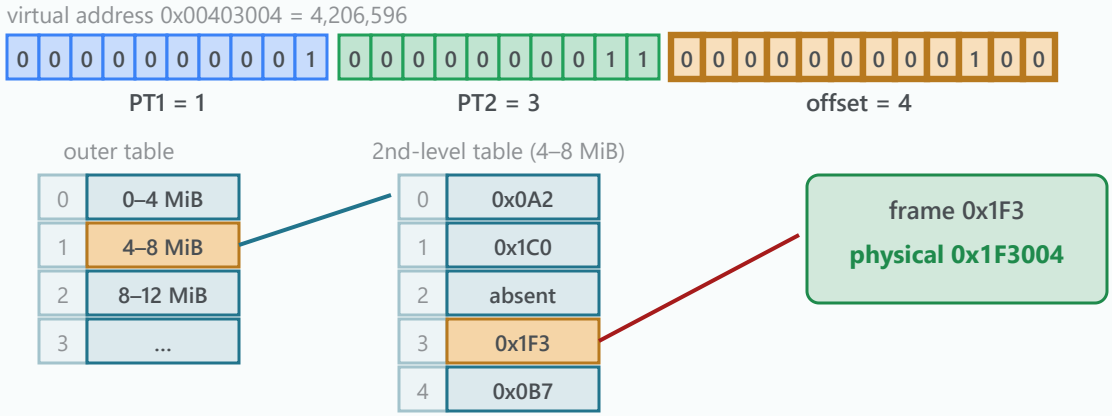
A Two-Level Translation Example



The next 10 bits, **PT2 = 3**: look at entry 3 of that table (the page covering bytes 12,288–16,383). With the present bit 0, it would be a page fault.

The frame numbers are illustrative. On 32-bit x86 the outer table's address is in `CR3`; the table at each level fills exactly one 4 KiB page (1024 × 4 B).

A Two-Level Translation Example



The entry gives the frame number (0x1F3 here); **offset = 4** is added → physical **0x1F3004**. On a miss: 2 table accesses + 1 data access.

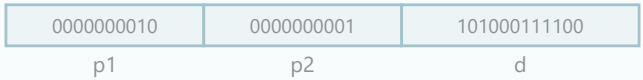
Step 4 / 4

The frame numbers are illustrative. On 32-bit x86 the outer table's address is in `CR3`; the table at each level fills exactly one 4 KiB page (1024 × 4 B).

Two-Level Translation: Code Trace

```
1  /* 32 bits: 10 + 10 + 12 bits */
▶  uint32_t translate(uint32_t va) {
3    uint32_t p1 = va >> 22;
4    uint32_t p2 = (va >> 12) & 0x3FF;
5    uint32_t d  = va & 0xFFF;
6    pte_t e1 = outer[p1];
7    if (!e1.p) page_fault(va);
8    pte_t e2 = e1.table[p2];
9    if (!e2.p) page_fault(va);
10   return (e2.frame << 12) | d;
11 }
```

va = 0x00801A3C



Virtual address 0x00801A3C (just above 8 MiB). Only entry 2 of the outer table is valid; in its table, entry 1 → frame 0x2F.

Step 1 / 9

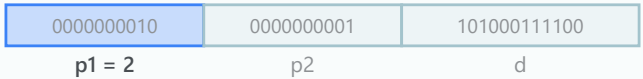
VARIABLES

va = **0x00801A3C**

Two-Level Translation: Code Trace

```
1  /* 32 bits: 10 + 10 + 12 bits */
2  uint32_t translate(uint32_t va) {
▶    uint32_t p1 = va >> 22;
4    uint32_t p2 = (va >> 12) & 0x3FF;
5    uint32_t d  = va & 0xFFF;
6    pte_t e1 = outer[p1];
7    if (!e1.p) page_fault(va);
8    pte_t e2 = e1.table[p2];
9    if (!e2.p) page_fault(va);
10   return (e2.frame << 12) | d;
11 }
```

va = 0x00801A3C



Top 10 bits: 0x00801A3C >> 22 = **2**.

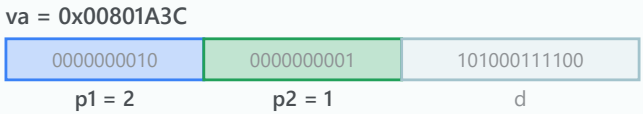
Step 2 / 9

VARIABLES

va = **0x00801A3C** p1 = **2**

Two-Level Translation: Code Trace

```
1  /* 32 bits: 10 + 10 + 12 bits */
2  uint32_t translate(uint32_t va) {
3    uint32_t p1 = va >> 22;
▶    uint32_t p2 = (va >> 12) & 0x3FF;
5    uint32_t d  = va & 0xFFF;
6    pte_t e1 = outer[p1];
7    if (!e1.p) page_fault(va);
8    pte_t e2 = e1.table[p2];
9    if (!e2.p) page_fault(va);
10   return (e2.frame << 12) | d;
11 }
```



Middle 10 bits: (va >> 12) & 0x3FF = 0x801 & 0x3FF = **1**.

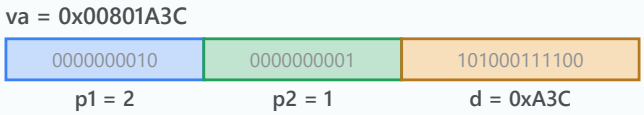
Step 3 / 9

VARIABLES

`va = 0x00801A3C` `p1 = 2` `p2 = 1`

Two-Level Translation: Code Trace

```
1  /* 32 bits: 10 + 10 + 12 bits */
2  uint32_t translate(uint32_t va) {
3    uint32_t p1 = va >> 22;
4    uint32_t p2 = (va >> 12) & 0x3FF;
▶    uint32_t d  = va & 0xFFF;
6    pte_t e1 = outer[p1];
7    if (!e1.p) page_fault(va);
8    pte_t e2 = e1.table[p2];
9    if (!e2.p) page_fault(va);
10   return (e2.frame << 12) | d;
11 }
```



Low 12-bit offset: **0xA3C**; it never changes during translation.

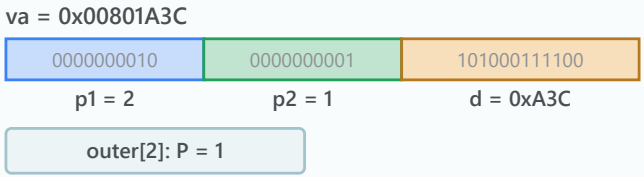
Step 4 / 9

VARIABLES

`va = 0x00801A3C` `p1 = 2` `p2 = 1` `d = 0xA3C`

Two-Level Translation: Code Trace

```
1  /* 32 bits: 10 + 10 + 12 bits */
2  uint32_t translate(uint32_t va) {
3    uint32_t p1 = va >> 22;
4    uint32_t p2 = (va >> 12) & 0x3FF;
5    uint32_t d  = va & 0xFFF;
▶    pte_t e1 = outer[p1];
▶    if (!e1.p) page_fault(va);
8    pte_t e2 = e1.table[p2];
9    if (!e2.p) page_fault(va);
10   return (e2.frame << 12) | d;
11 }
```



Outer-table entry 2 is valid (P = 1): it gives the frame of the second-level table. The first memory access.

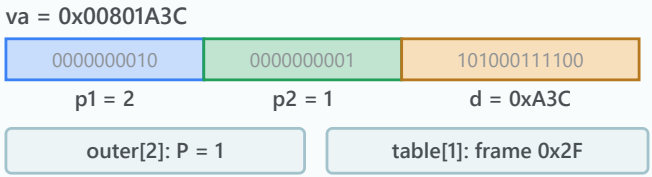
Step 5 / 9

VARIABLES

`va = 0x00801A3C` `p1 = 2` `p2 = 1` `d = 0xA3C` `e1 = P=1`

Two-Level Translation: Code Trace

```
1  /* 32 bits: 10 + 10 + 12 bits */
2  uint32_t translate(uint32_t va) {
3    uint32_t p1 = va >> 22;
4    uint32_t p2 = (va >> 12) & 0x3FF;
5    uint32_t d  = va & 0xFFF;
6    pte_t e1 = outer[p1];
7    if (!e1.p) page_fault(va);
▶    pte_t e2 = e1.table[p2];
▶    if (!e2.p) page_fault(va);
10   return (e2.frame << 12) | d;
11 }
```



Second-level entry 1: frame **0x2F**, P = 1. The second memory access.

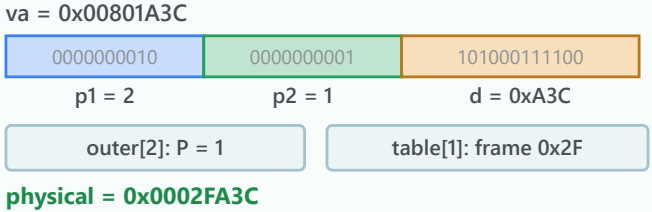
Step 6 / 9

VARIABLES

`va = 0x00801A3C` `p1 = 2` `p2 = 1` `d = 0xA3C` `frame = 0x2F`

Two-Level Translation: Code Trace

```
1  /* 32 bits: 10 + 10 + 12 bits */
2  uint32_t translate(uint32_t va) {
3    uint32_t p1 = va >> 22;
4    uint32_t p2 = (va >> 12) & 0x3FF;
5    uint32_t d  = va & 0xFFF;
6    pte_t e1 = outer[p1];
7    if (!e1.p) page_fault(va);
8    pte_t e2 = e1.table[p2];
9    if (!e2.p) page_fault(va);
▶    return (e2.frame << 12) | d;
11 }
```



0x2F << 12 = 0x2F000; combined with the offset, physical **0x0002FA3C**. The third access is the data.

Step 7 / 9

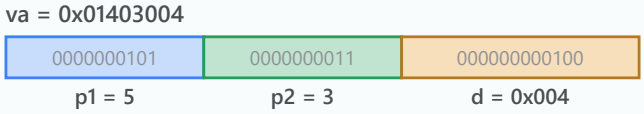
VARIABLES

`va = 0x00801A3C` `frame = 0x2F`

`pa = 0x0002FA3C`

Two-Level Translation: Code Trace

```
1  /* 32 bits: 10 + 10 + 12 bits */
2  uint32_t translate(uint32_t va) {
▶    uint32_t p1 = va >> 22;
▶    uint32_t p2 = (va >> 12) & 0x3FF;
▶    uint32_t d  = va & 0xFFF;
6    pte_t e1 = outer[p1];
7    if (!e1.p) page_fault(va);
8    pte_t e2 = e1.table[p2];
9    if (!e2.p) page_fault(va);
10   return (e2.frame << 12) | d;
11 }
```



A second address, 0x01403004: p1 = **5**, p2 = 3, offset = 0x004.

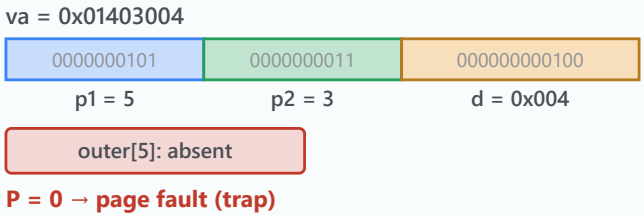
Step 8 / 9

VARIABLES

`va = 0x01403004` `p1 = 5` `p2 = 3` `d = 0x004`

Two-Level Translation: Code Trace

```
1  /* 32 bits: 10 + 10 + 12 bits */
2  uint32_t translate(uint32_t va) {
3    uint32_t p1 = va >> 22;
4    uint32_t p2 = (va >> 12) & 0x3FF;
5    uint32_t d  = va & 0xFFF;
▶    pte_t e1 = outer[p1];
▶    if (!e1.p) page_fault(va);
8    pte_t e2 = e1.table[p2];
9    if (!e2.p) page_fault(va);
10   return (e2.frame << 12) | d;
11 }
```



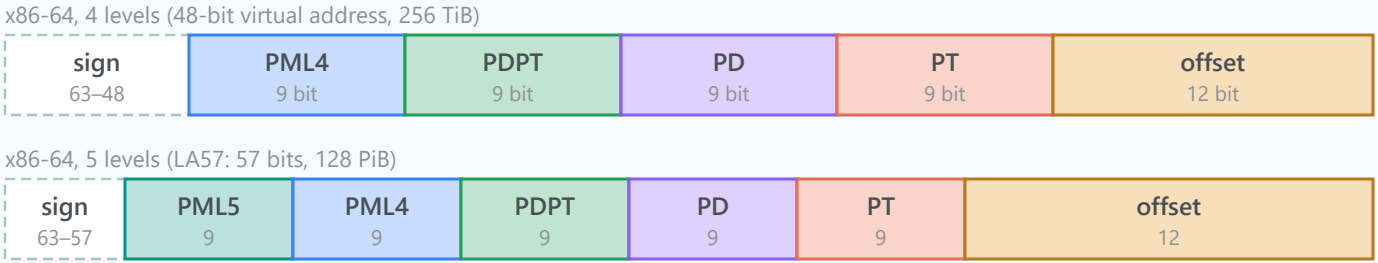
Outer-table entry 5 is invalid (P = 0): there is **no** second-level table at all for the 20–24 MiB region. The MMU traps; no memory was spent on a table.

Step 9 / 9

VARIABLES

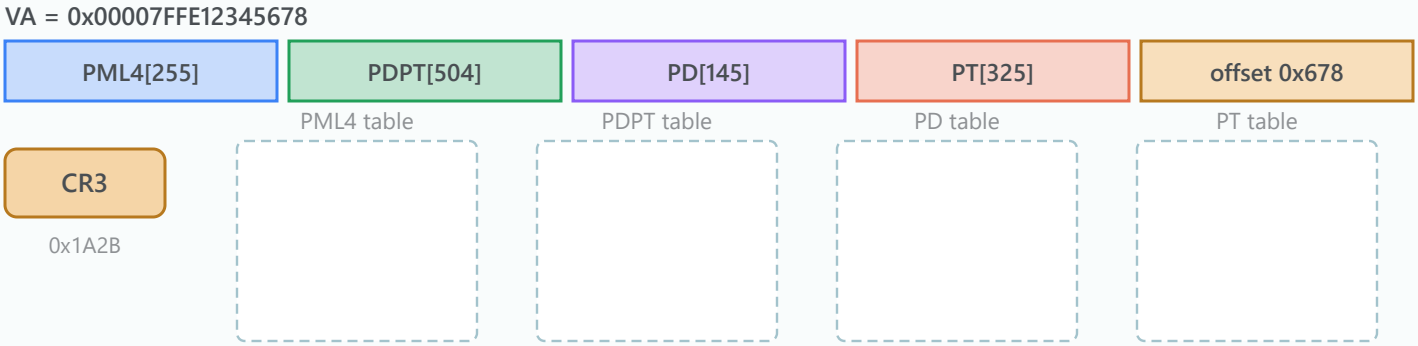
`va = 0x01403004` `p1 = 5` `e1 = P=0` `result = page fault`

x86-64: Four- and Five-Level Paging



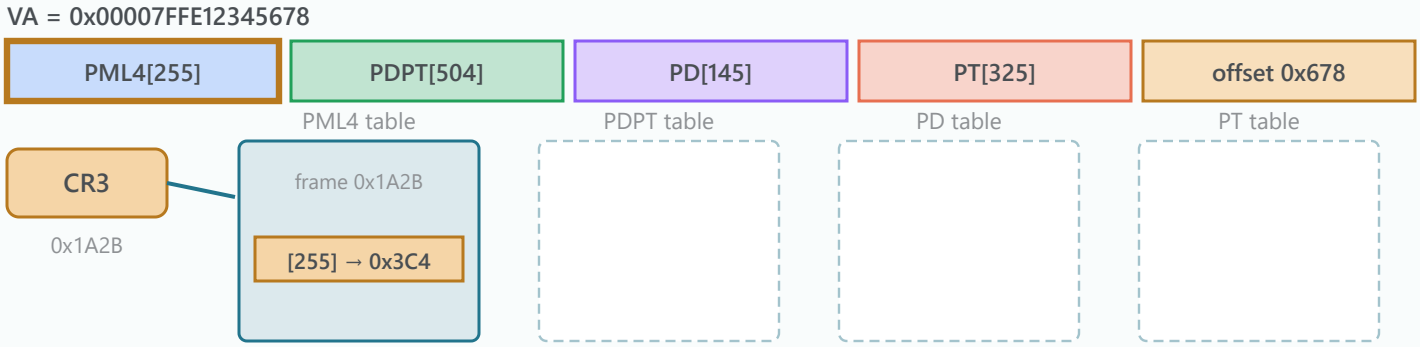
- Each table is 512 entries × 8 B = **4 KiB**, exactly one page; a 9-bit index. 4 × 9 + 12 = **48 bits**.
- The top 16 bits must copy bit 47 (a **canonical address**): the lower half for user space, the upper half for the kernel.
- 5 levels (Intel Ice Lake server, AMD Zen 4 and later) give 57-bit virtual addresses; Linux supports them since 4.14.
- Hierarchical tables work at 64 bits too: since not all 64 address bits are used, today's x86-64 and ARMv8 use exactly such tables.
- If the `PS` bit is 1 in an intermediate entry, the walk stops there: a **2 MiB** huge page at the PD level, **1 GiB** at the PDPT level.

A Four-Level Walk



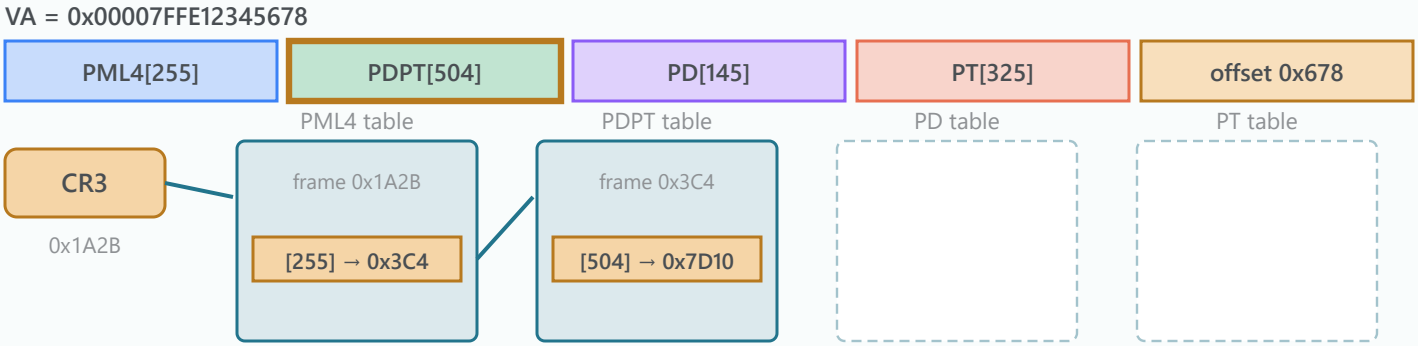
An address near the user stack. Its split (9+9+9+9+12): PML4 = 255, PDPT = 504, PD = 145, PT = 325, offset = 0x678.

A Four-Level Walk



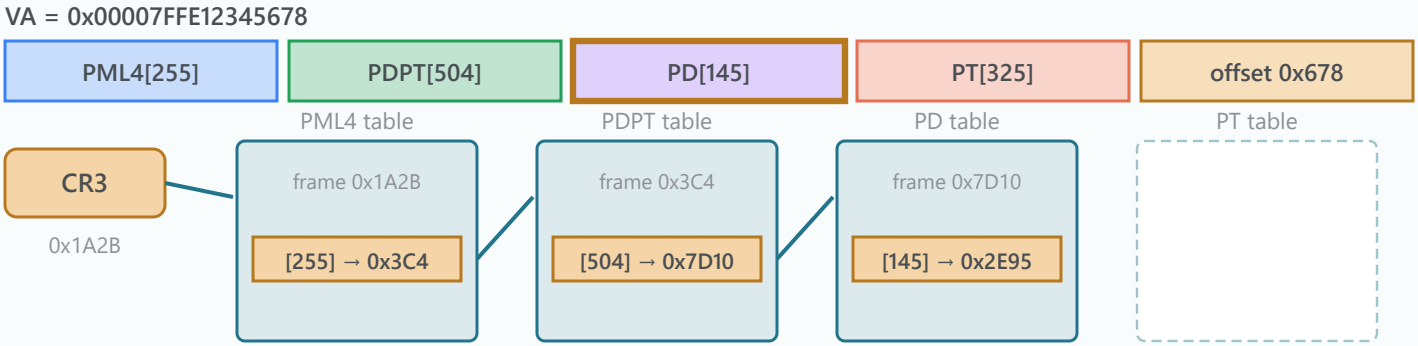
`CR3` gives the frame of the PML4 table; entry 255 is read → the frame of the PDPT table.

A Four-Level Walk



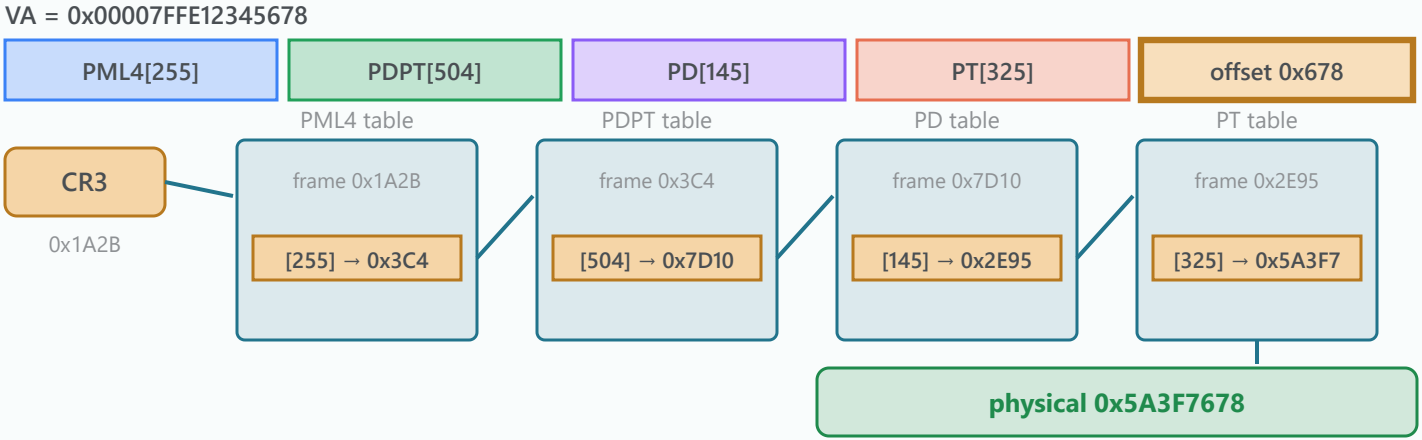
PDPT[504] → the frame of the page directory (PD). Had `PS` = 1 here, a 1 GiB page would have been found.

A Four-Level Walk



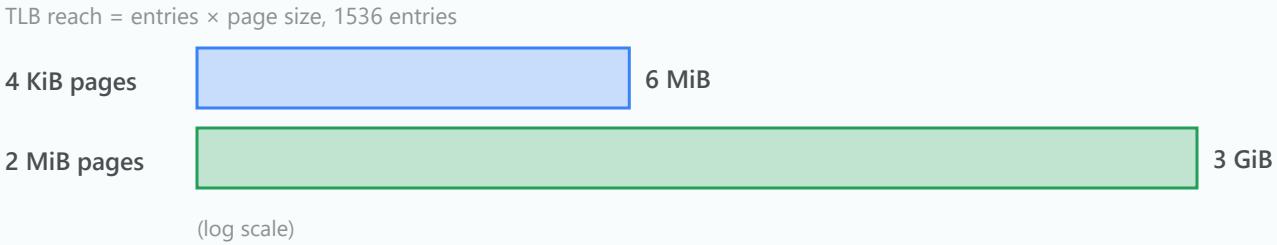
PD[145] → the last-level table. With `PS` = 1, a 2 MiB page.

A Four-Level Walk



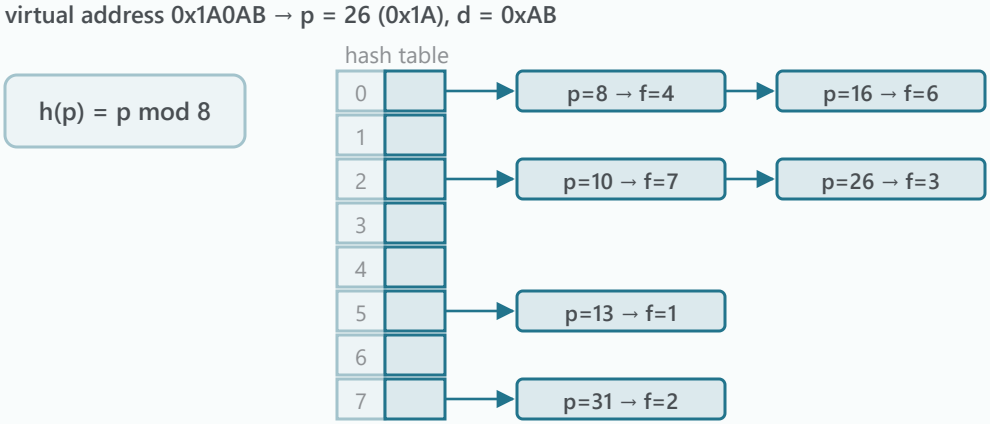
PT[325] gives frame 0x5A3F7; the offset is added. A TLB miss costs **4 memory accesses** + the data access; the frame values are examples.

Huge Pages



- Programs with large working sets (databases, virtual machines, AI) suffer constant TLB misses with 4 KiB pages.
- Mapping 1 GiB with 4 KiB pages takes 262,144 PTEs = a 2 MiB table; with 2 MiB pages, only 512 PD entries (4 KiB).
- Linux: **hugetlbfs** (preallocated 2 MiB / 1 GiB), **transparent huge pages** (THP, merged by the kernel automatically), and in recent versions mTHP (16–64 KiB intermediate sizes). Windows: *large pages*.
- The cost: internal fragmentation, and the need to find 2 MiB of contiguous physical frames (the buddy system and compaction return here).
- ARMv8's *contiguous* bit: 16 adjacent 4 KiB entries can be held in one TLB entry (64 KiB).

Hashed Page Tables



Each bucket is a chain of **(virtual page, frame, next)** entries that hash to the same value. Here, 8 buckets, h(p) = p mod 8.

Step 1 / 4

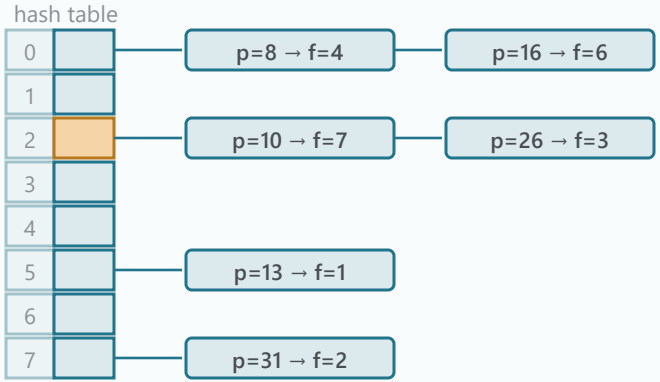
- Used for address spaces larger than 32 bits; the table's size is proportional to the number of pages in use, not to the address space.
- **Clustered** variant: each entry covers several (e.g., 16) consecutive pages; suitable for sparse 64-bit spaces.

Hashed Page Tables

virtual address 0x1A0AB → p = 26 (0x1A), d = 0xAB

h(p) = p mod 8

h(26) = 2



Page number 26 is hashed: bucket **2**.

Step 2 / 4

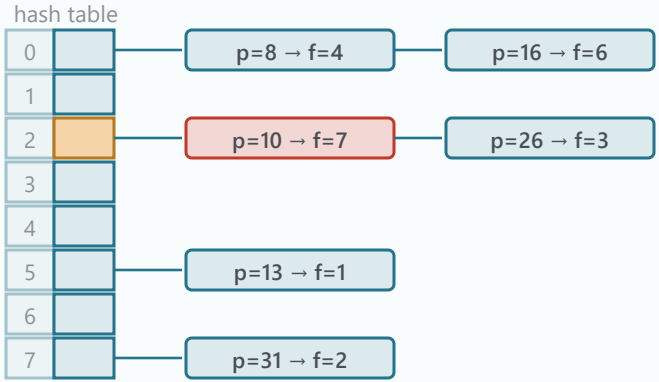
- Used for address spaces larger than 32 bits; the table's size is proportional to the number of pages in use, not to the address space.
- **Clustered** variant: each entry covers several (e.g., 16) consecutive pages; suitable for sparse 64-bit spaces.

Hashed Page Tables

virtual address 0x1A0AB → p = 26 (0x1A), d = 0xAB

h(p) = p mod 8

h(26) = 2



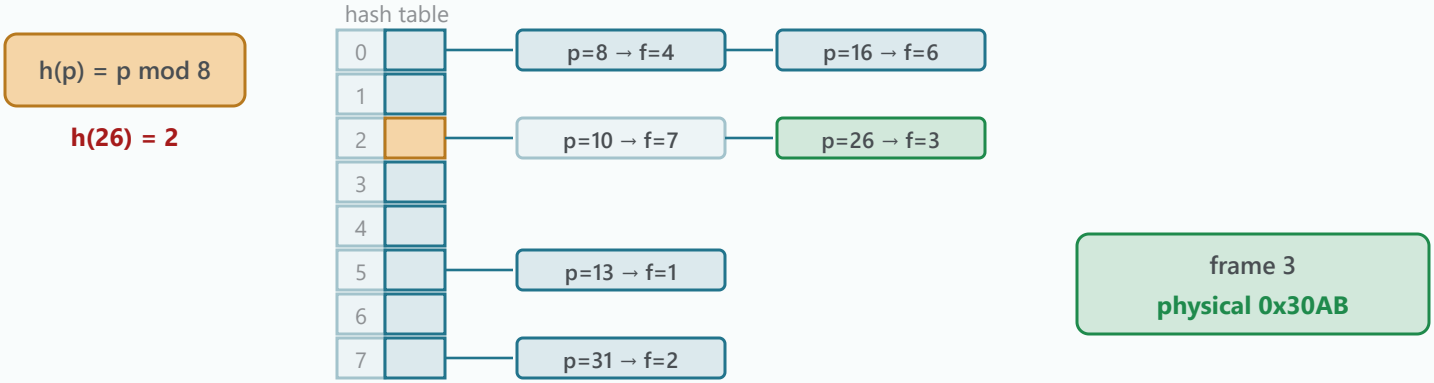
The chain's first entry is p = 10: no match, move to the next.

Step 3 / 4

- Used for address spaces larger than 32 bits; the table's size is proportional to the number of pages in use, not to the address space.
- **Clustered** variant: each entry covers several (e.g., 16) consecutive pages; suitable for sparse 64-bit spaces.

Hashed Page Tables

virtual address 0x1A0AB → p = 26 (0x1A), d = 0xAB

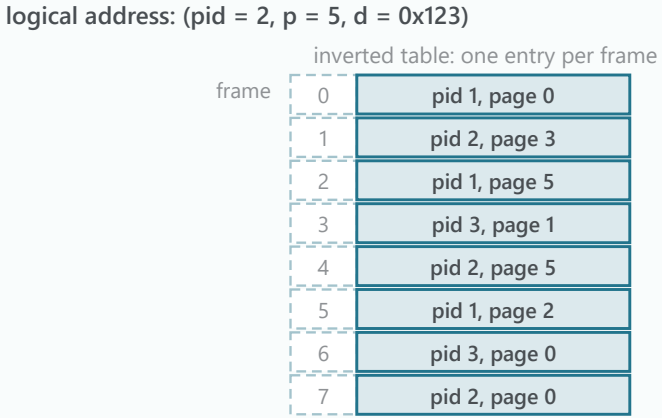


p = 26 matches → frame **3**; physical = 3 × 4096 + 0xAB = **0x30AB**.

Step 4 / 4

- Used for address spaces larger than 32 bits; the table's size is proportional to the number of pages in use, not to the address space.
- **Clustered** variant: each entry covers several (e.g., 16) consecutive pages; suitable for sparse 64-bit spaces.

Inverted Page Tables



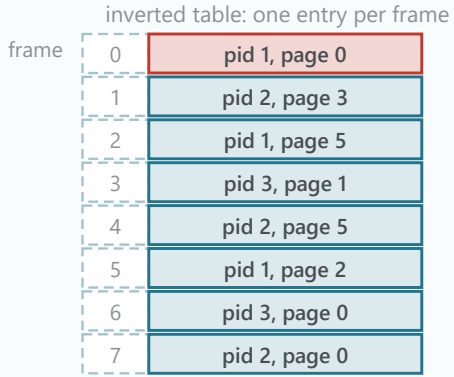
The table has one entry **per physical frame, not per process**: which page of which process is in that frame.

Step 1 / 6

- The table's size is proportional to physical memory, and there is **one** per system: 16 GiB RAM, 4 KiB frames → 4,194,304 entries; 32 MiB at 8 B per entry.
- Drawbacks: shared memory is hard (one virtual page per frame); handling page faults needs separate per-process external tables. Examples: PowerPC/IBM POWER (hashed inverted table), IA-64.

Inverted Page Tables

logical address: (pid = 2, p = 5, d = 0x123)



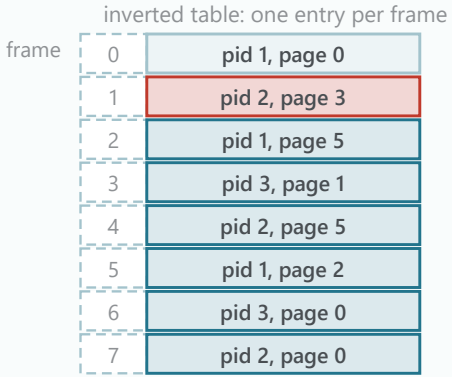
Entry 0: (pid 1, page 0) ≠ (2, 5), next.

Step 2 / 6

- The table's size is proportional to physical memory, and there is **one** per system: 16 GiB RAM, 4 KiB frames → 4,194,304 entries; 32 MiB at 8 B per entry.
- Drawbacks: shared memory is hard (one virtual page per frame); handling page faults needs separate per-process external tables. Examples: PowerPC/IBM POWER (hashed inverted table), IA-64.

Inverted Page Tables

logical address: (pid = 2, p = 5, d = 0x123)



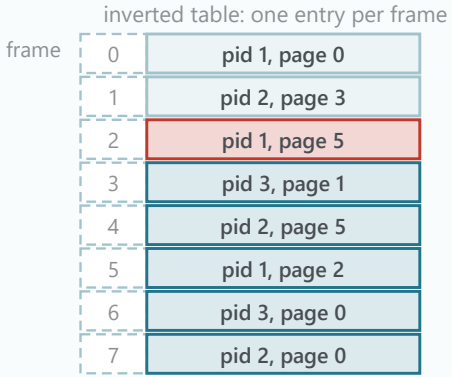
Entry 1: (pid 2, page 3) ≠ (2, 5), next.

Step 3 / 6

- The table's size is proportional to physical memory, and there is **one** per system: 16 GiB RAM, 4 KiB frames → 4,194,304 entries; 32 MiB at 8 B per entry.
- Drawbacks: shared memory is hard (one virtual page per frame); handling page faults needs separate per-process external tables. Examples: PowerPC/IBM POWER (hashed inverted table), IA-64.

Inverted Page Tables

logical address: (pid = 2, p = 5, d = 0x123)



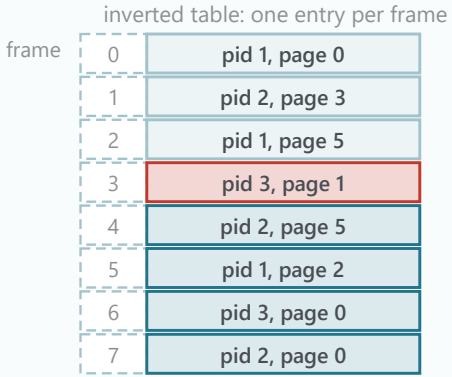
Entry 2: (pid 1, page 5) ≠ (2, 5), next.

Step 4 / 6

- The table's size is proportional to physical memory, and there is **one** per system: 16 GiB RAM, 4 KiB frames → 4,194,304 entries; 32 MiB at 8 B per entry.
- Drawbacks: shared memory is hard (one virtual page per frame); handling page faults needs separate per-process external tables. Examples: PowerPC/IBM POWER (hashed inverted table), IA-64.

Inverted Page Tables

logical address: (pid = 2, p = 5, d = 0x123)

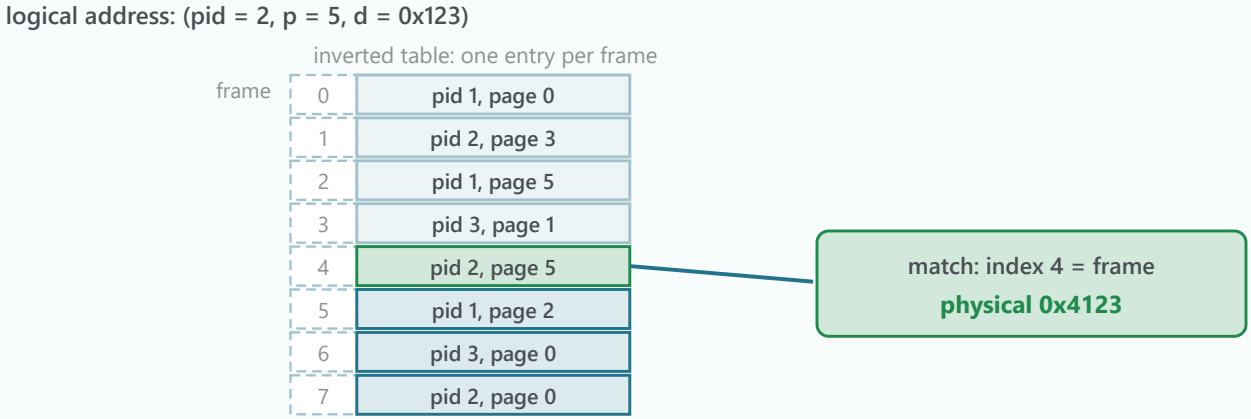


Entry 3: (pid 3, page 1) ≠ (2, 5), next.

Step 5 / 6

- The table's size is proportional to physical memory, and there is **one** per system: 16 GiB RAM, 4 KiB frames → 4,194,304 entries; 32 MiB at 8 B per entry.
- Drawbacks: shared memory is hard (one virtual page per frame); handling page faults needs separate per-process external tables. Examples: PowerPC/IBM POWER (hashed inverted table), IA-64.

Inverted Page Tables



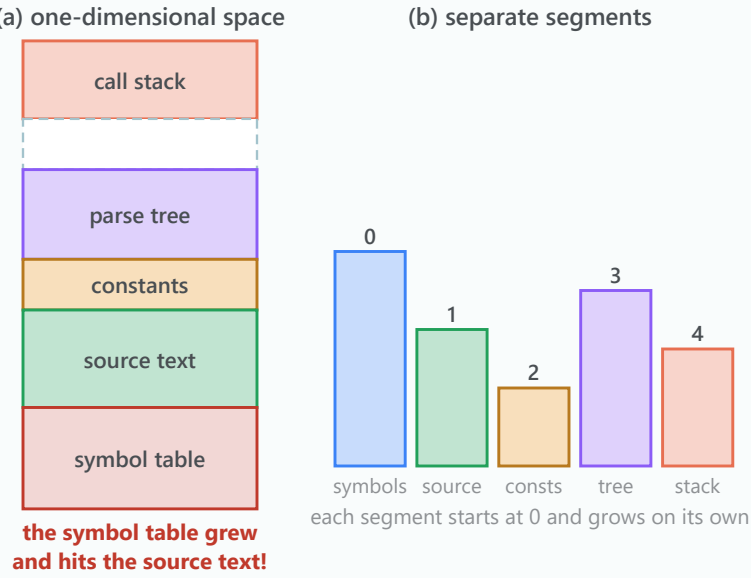
Entry 4 matches: **the index itself is the frame number** → 4 × 4096 + 0x123 = **0x4123**. A linear search is expensive, so real systems use a hash table + TLB.

Step 6 / 6

- The table's size is proportional to physical memory, and there is **one** per system: 16 GiB RAM, 4 KiB frames → 4,194,304 entries; 32 MiB at 8 B per entry.
- Drawbacks: shared memory is hard (one virtual page per frame); handling page faults needs separate per-process external tables. Examples: PowerPC/IBM POWER (hashed inverted table), IA-64.

Segmentation: Why?

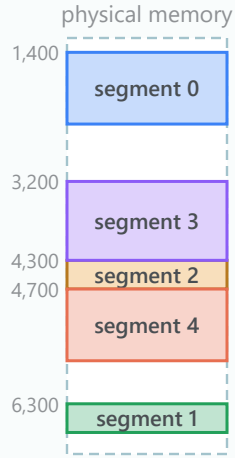
- A compiler builds many tables while compiling: source text, the **symbol table**, constants, the parse tree, the call stack. All of them grow and shrink.
- In a one-dimensional address space, when one table fills up it **collides** with its neighbor, even if the neighbor is empty.
- **Segmentation**: several independent, variable-length address spaces. An address = **(segment number, offset)**.
- It matches the programmer's view: main program, functions, stack, arrays, and symbol table are separate units.



Translation with a Segment Table

segment table

	limit	base
0	1,000	1,400
1	400	6,300
2	400	4,300
3	1,100	3,200
4	1,000	4,700



Each entry holds a segment's **base** (physical start) and **limit** (length); the table's location and size are in the STBR and STLR registers.

Step 1 / 5

Silberschatz's five-segment example. Segmentation is a separate base–limit pair for each segment.

Translation with a Segment Table

segment table

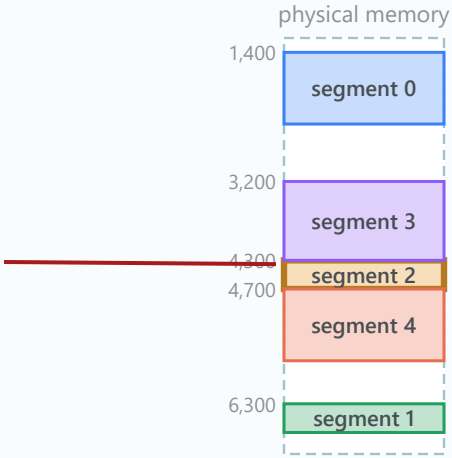
	limit	base
0	1,000	1,400
1	400	6,300
2	400	4,300
3	1,100	3,200
4	1,000	4,700

(s, d) = (2, 53)

d < limit ? 53 < 400

yes → base + d

4,300 + 53 = 4,353



(2, 53): 53 < 400 → 4,300 + 53 = **4,353**.

Step 2 / 5

Silberschatz's five-segment example. Segmentation is a separate base–limit pair for each segment.

Translation with a Segment Table

segment table

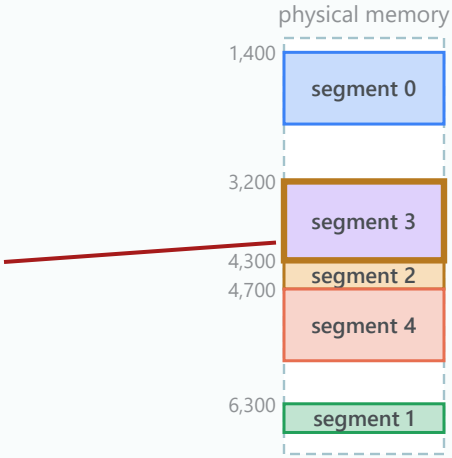
	limit	base
0	1,000	1,400
1	400	6,300
2	400	4,300
3	1,100	3,200
4	1,000	4,700

(s, d) = (3, 852)

d < limit ? 852 < 1,100

yes → base + d

3,200 + 852 = 4,052



(3, 852): 852 < 1,100 → 3,200 + 852 = **4,052**.

Step 3 / 5

Silberschatz's five-segment example. Segmentation is a separate base–limit pair for each segment.

Translation with a Segment Table

segment table

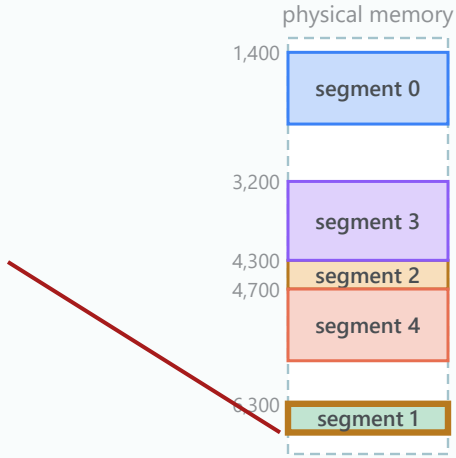
	limit	base
0	1,000	1,400
1	400	6,300
2	400	4,300
3	1,100	3,200
4	1,000	4,700

(s, d) = (1, 399)

d < limit ? 399 < 400

yes → base + d

6,300 + 399 = 6,699



(1, 399): 399 < 400 → 6,300 + 399 = **6,699**.

Silberschatz's five-segment example. Segmentation is a separate base–limit pair for each segment.

Translation with a Segment Table

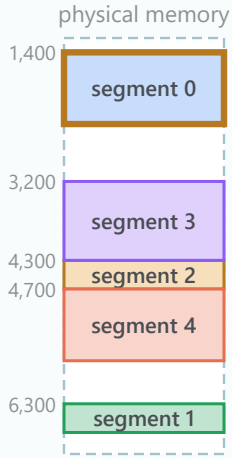
segment table

	limit	base
0	1,000	1,400
1	400	6,300
2	400	4,300
3	1,100	3,200
4	1,000	4,700

(s, d) = (0, 1,222)

d < limit ? 1,222 < 1,000

no → trap (addressing error)



(0, 1,222): segment 0 is only 1,000 bytes long → **trap**. Without the check, address 2,622, outside the segment, would be accessed.

Silberschatz's five-segment example. Segmentation is a separate base–limit pair for each segment.

Segment Bits and a Downward-Growing Stack

segment	base	size	grows	protection
00 code	24,576	3,072	+	r-x
01 heap	40,960	2,048	+	rw-
11 stack	32,768	2,048	–	rw-

segment 00 → code

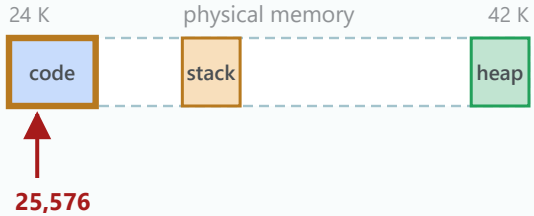
offset < size ? 1,000 < 3,072

yes → base + offset

24,576 + 1,000 = 25,576

virtual address 1,000





1,000: top bits **00** → code. 1,000 < 3,072 → 24,576 + 1,000 = **25,576**.

Step 1 / 4

A 14-bit virtual address (16 KiB): the top two bits select one of four segments, the other 12 give up to 4 KiB each. The **grows** bit enables the stack, the **protection** bits enable sharing: an `r-x` code segment can map to the same frames in many processes; writing to it is a protection fault.

Segment Bits and a Downward-Growing Stack

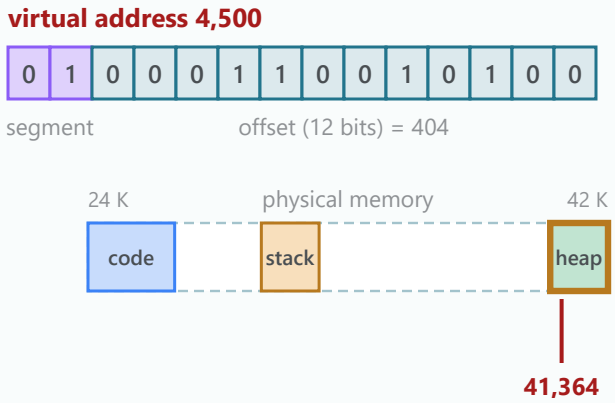
segment	base	size	grows	protection
00 code	24,576	3,072	+	r-x
01 heap	40,960	2,048	+	rw-
11 stack	32,768	2,048	–	rw-

segment 01 → heap

offset < size ? 404 < 2,048

yes → base + offset

40,960 + 404 = 41,364



4,500: **01** → heap. Offset 4,500 − 4,096 = 404 → 40,960 + 404 = **41,364** (not address + base = 45,460).

Step 2 / 4

A 14-bit virtual address (16 KiB): the top two bits select one of four segments, the other 12 give up to 4 KiB each. The **grows** bit enables the stack, the **protection** bits enable sharing: an `r-x` code segment can map to the same frames in many processes; writing to it is a protection fault.

Segment Bits and a Downward-Growing Stack

segment	base	size	grows	protection
00 code	24,576	3,072	+	r-x
01 heap	40,960	2,048	+	rw-
11 stack	32,768	2,048	–	rw-

segment 11 → stack

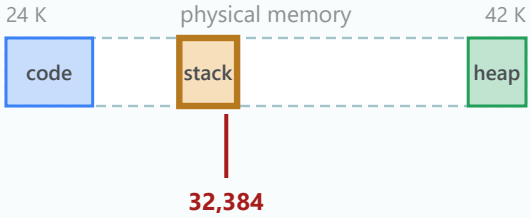
grows down: negative offset = 3,712 − 4,096 = −384

|−384| ≤ 2,048 → valid

32,768 − 384 = 32,384

virtual address 16,000





16,000: **11** → stack. It grows down, so the maximum segment size (4,096) is subtracted from the offset: −384 → **32,384**.

A 14-bit virtual address (16 KiB): the top two bits select one of four segments, the other 12 give up to 4 KiB each. The **grows** bit enables the stack, the **protection** bits enable sharing: an `r-x` code segment can map to the same frames in many processes; writing to it is a protection fault.

Segment Bits and a Downward-Growing Stack

segment	base	size	grows	protection
00 code	24,576	3,072	+	r-x
01 heap	40,960	2,048	+	rw-
11 stack	32,768	2,048	–	rw-

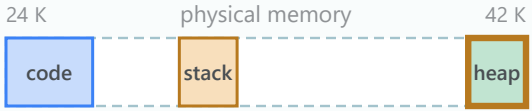
segment 01 → heap

offset < size ? 2,904 < 2,048

no → trap (protection fault)

virtual address 7,000





7,000: **01** → heap, but offset 2,904 ≥ 2,048: the hardware **traps**; the OS kills the process with `SIGSEGV`.

Step 4 / 4

A 14-bit virtual address (16 KiB): the top two bits select one of four segments, the other 12 give up to 4 KiB each. The **grows** bit enables the stack, the **protection** bits enable sharing: an `r-x` code segment can map to the same frames in many processes; writing to it is a protection fault.

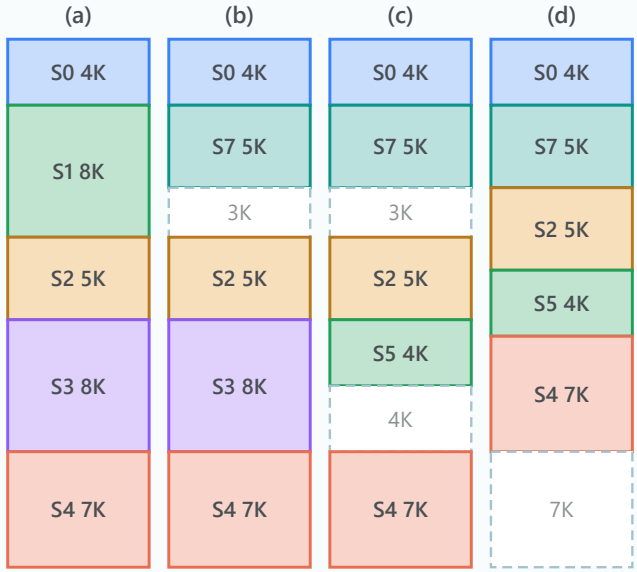
Pros and Cons of Segmentation

Advantages

- Growing and shrinking tables don't affect each other.
- Segments can be compiled separately; changing one function doesn't change the others' addresses.
- A library can be placed in a segment and **shared**.
- Per-segment protection: code `x`, constants `r`, data `rw`.

Disadvantages

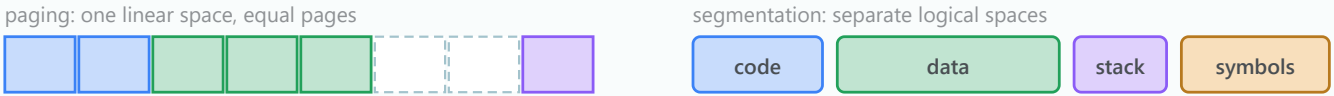
- Variable length → **external fragmentation** (checkerboarding), compaction.
- A segment must be entirely in memory; the fix is to page the segments (MULTICS, Intel 386).



(b)–(c) external fragmentation (checkerboarding), (d) compaction

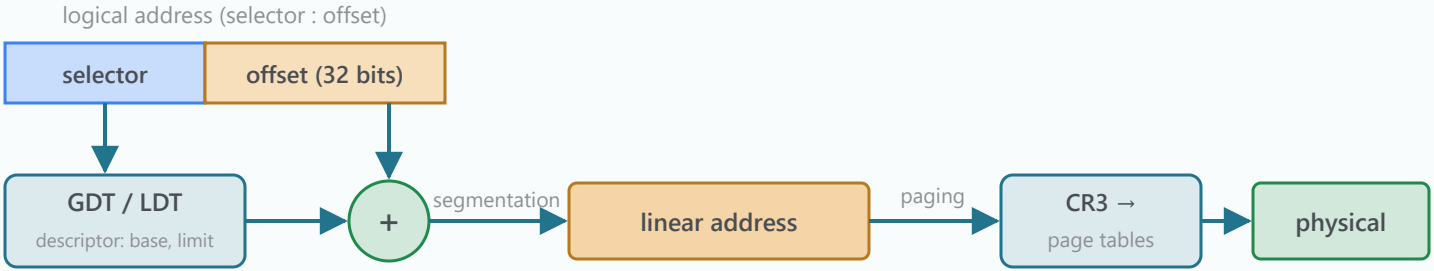
Paging versus Segmentation

Question	Paging	Segmentation
Must the programmer be aware of the technique?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed physical memory?	Yes	Yes
Can code and data be separated and protected separately?	Yes, via page bits*	Yes
Are tables whose size fluctuates easily accommodated?	No	Yes
Is sharing functions between users easy?	With shared pages*	Yes
Fragmentation	Internal (last page)	External
Why was it invented?	A large linear address space beyond physical memory	To split programs and data into independent logical spaces; sharing, protection



Adapted from Tanenbaum Figure 3-32. * Tanenbaum says "no" for paging here; modern R/W/NX bits and shared pages do the same job at page level.

x86: Segmentation + Paging, and the Flat Model



- 8086 real mode: physical = segment × 16 + offset (20 bits, 1 MiB); no protection.
- 80386 protected mode: a 16-bit selector (13-bit index, table select, RPL) picks an 8-byte descriptor from the GDT/LDT; base + offset = the **linear address**, followed by two-level paging. The hardware counterpart of the MULTICS idea.
- Even on 32 bits, Linux and Windows used a **flat model**, giving every segment base 0 and limit 4 GiB: segmentation effectively off, protection via pages.
- **x86-64 long mode**: the CS, DS, ES, SS bases are treated as 0 by hardware and limits aren't checked. Only the **FS and GS** bases are used: thread-local storage (TLS) in user space, per-CPU data in the kernel.

A Note on ARMv8-A and RISC-V

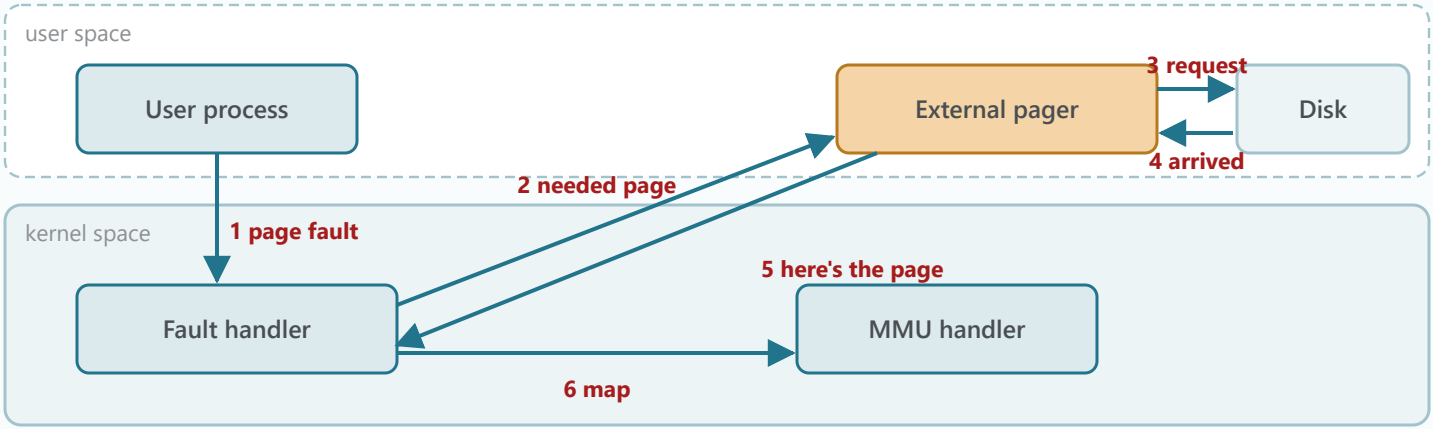
	ARMv8-A (AArch64)	RISC-V
Page (granule)	4, 16, or 64 KiB; Apple Silicon and some Android 15+ devices use 16 KiB	4 KiB (+ 2 MiB, 1 GiB, 512 GiB huge pages)
Virtual address	48 bits (4 levels with 4 KiB, L0–L3); 52 bits with LVA/LPA2	Sv39 (3 levels), Sv48 (4), Sv57 (5)
Table base	`TTBR0_EL1` lower half (user), `TTBR1_EL1` upper half (kernel)	`satp` (mode + ASID + root frame)
Process tag	ASID 8/16 bit	ASID (up to 16 bits)
Protection	AP bits, UXN/PXN (no-execute), PAN (≈ SMAP)	R/W/X/U bits, SUM
TLB	Hardware walk; A/D bits optionally by hardware (v8.1); contiguous bit	Hardware walk (most implementations); `sfence.vma`



Sv39: 3 levels × 512 entries; satp holds the root table's frame and the ASID

Neither architecture has segmentation: a flat virtual address, multilevel page tables, a TLB, and ASIDs. The same ideas as x86-64, different names.

Separating Policy and Mechanism



- Paging involves the OS at four points: process creation (the table is set up), context switch (the table is activated, TLB), page faults, and process exit (frames and swap space are released).
- The **mechanism** is in the kernel: the machine-dependent *MMU handler* and the machine-independent *page-fault handler*. The **policy** (which page, when, where) lives in a user-space *external pager* (Mach).
- Flexibility and modularity are gained; the price is crossing the user/kernel boundary many times.
- Today's equivalents: Linux `userfaultfd` (live migration of virtual machines, CRIU), and user-space pagers in microkernels (seL4, Fuchsia).

Summary

Technique	Idea	Pros	Cons
Base–limit	One contiguous region + MMU	Simple, fast	External fragmentation, hard to share
Variable partitions	Choosing holes with first/best fit	No internal fragmentation	50-percent rule, compaction
Paging	Fixed page ↔ frame, page table	No external fragmentation; sharing, protection	Table size, double access
TLB	Translation cache, ASID	EAT ≈ memory time	Context switches, shootdowns
Hierarchical table	Page the table (x86-64: 4/5 levels)	Sparse spaces are cheap	Many accesses on a miss
Hashed / inverted	One entry per used page / frame	Small table	Lookup, sharing is hard
Segmentation	(segment, offset), segment table	Logical division, protection	External fragmentation; flat model on x86-64

Today's general-purpose systems: a flat virtual address space + multilevel paging + TLB + huge pages; the buddy system for physical frames, slab (SLUB) for kernel objects.

Exercises

1. The loop `for (i = 0; i < 1000; i++) a[i] = 0;` compiles to the four instructions below. With a linear page table and no TLB, how many memory accesses occur in total? With a TLB? (`a` starts at virtual 40,000, `int` is 4 B, pages are 4 KiB, the code fits in one page.)

```
loop:  movl $0,(%edi,%eax,4)  /* a[i] = 0 */
       incl %eax              /* i++      */
       cmpl $1000,%eax
       jne  loop
```

2. With the segment registers from the "Segment Bits" slide, where do virtual addresses 13,000 and 14,500 go?
3. An allocator with 16 B headers does `malloc(1000)` three times from a 4,096 B heap. What is the free chunk's size? Can a fourth `malloc(1040)` be satisfied?
4. What happens if `free(p + 4)` is called?

▶ **Answers**

SUMMARY

Review Questions

1. With base = 300,040 and limit = 120,900, what happens on an access to address 420,940?
2. Holes 100, 500, 200, 300, 600 K; requests 212, 417, 112, 426 K. Which algorithm places them all?
3. With 32-bit addresses and 4 KiB pages, what are p and d for `0x12345678`?
4. In Tanenbaum's table (5 → 3), what physical address does virtual 20,500 translate to?
5. What is the EAT for a single-level table with ε = 20 ns, t = 100 ns, α = 90%?
6. Why can hierarchical page tables be used on 64-bit systems too? How many levels does x86-64 use?
7. What is left of segmentation in x86-64 long mode?
8. An empty direct-mapped cache with 4 sets and 64 B lines sees accesses to addresses 0, 128, 256, 0. How many hits?

▶ **Answers**

References

- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 (Chapter 9: Main Memory).
- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed. (2014) / 5th ed. (2023), Pearson (Chapter 3: Memory Management, §3.1–3.3, 3.5–3.7).
- R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org: "Address Spaces", "Address Translation", "Segmentation", "Free-Space Management", "Paging", "TLBs", "Smaller Tables".
- D. E. Knuth, *The Art of Computer Programming*, Vol. 1, §2.5 Dynamic Storage Allocation (the 50-percent rule).
- Intel 64 and IA-32 Architectures Software Developer's Manual, Vol. 3A, Chapters 4–5 (Paging, Protection); Arm Architecture Reference Manual for A-profile (VMSAv8-64); RISC-V Privileged Specification (Sv39/48/57).
- U. Drepper, "How To Write Shared Libraries", 2011; Linux kernel documentation: `mm/`, "Transparent Hugepage Support", `userfaultfd`.

The End

Chapter 8: Memory Management

Previous: Chapter 7: Deadlocks · Next: Chapter 9: Virtual Memory