

Memory management and address translation

CSE513 · Week 5
Prof. Mehmet Göktürk

A private view of shared memory

How can two processes use the same virtual address without accessing the same data?

First half: address spaces, binding, allocation, fragmentation, segmentation and paging.

Second half: page tables, TLBs, translation cost, protection, sharing and multilevel tables.

By the end, translate an address and explain both the mapping and the permission check.

Memory connects execution to isolation

The CPU fetches instructions and accesses data through addresses. A process must not freely overwrite another process or the kernel.

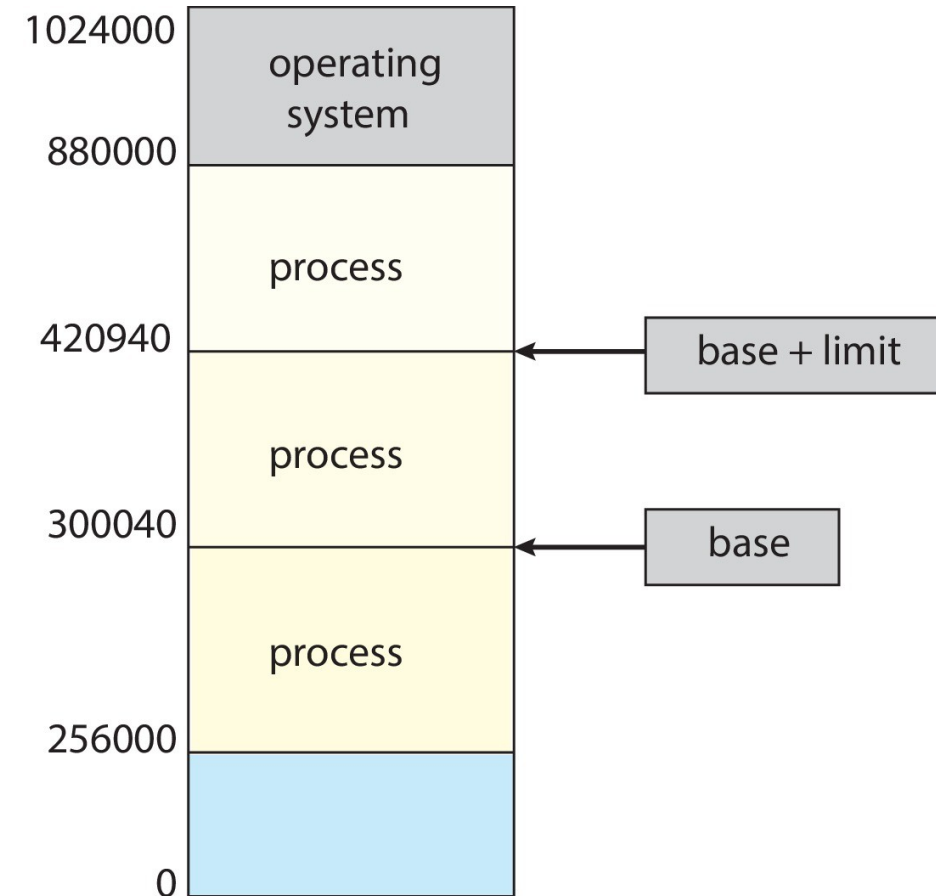
The OS allocates memory and establishes mappings; hardware applies translation and protection during execution.

The address in a program is interpreted in its process context. A numerical address alone does not identify a unique physical byte.

A bounded address space

Base and limit describe a simple contiguous region. Hardware prevents a user process from reaching outside its allowed range.

Keep two models distinct: checking a physical interval versus relocating an offset and checking its length. State which model an exercise uses.



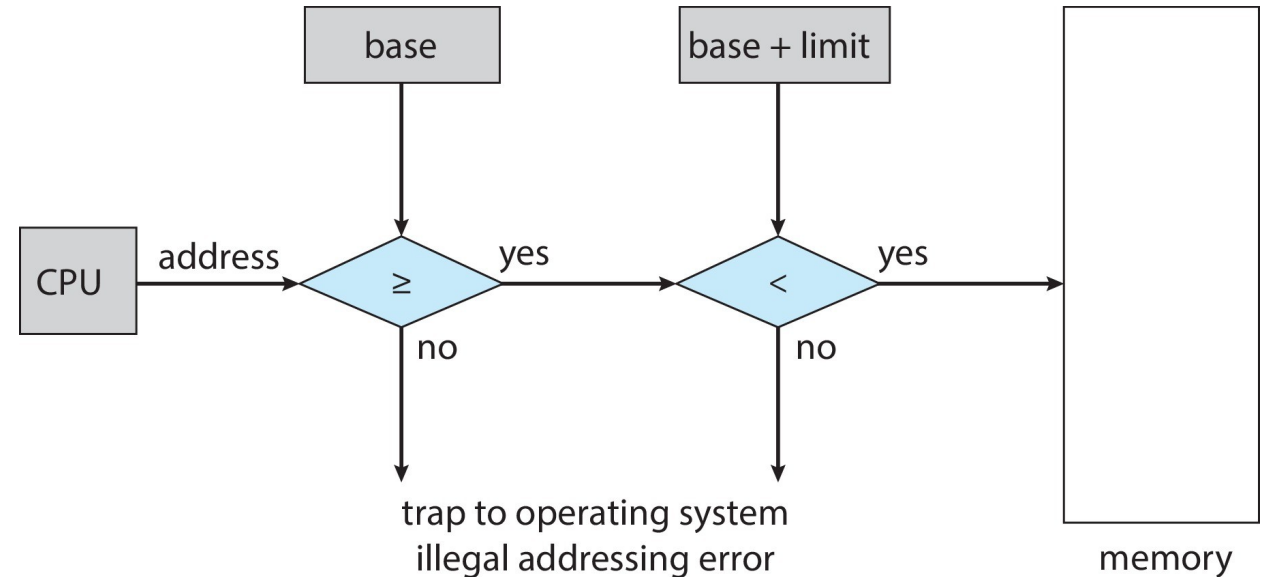
Silberschatz, Galvin & Gagne · OSC 10e · Ch. 9

Checking a physical interval

For this diagram, legal physical addresses satisfy
 $\text{base} \leq \text{address} < \text{base} + \text{limit}$.

Base = 14000, limit = 1000: 14000 and 14999 are legal; 15000 is outside.

Only privileged code may change the protection registers. Otherwise the process could enlarge its own boundary.



Addresses change during program preparation

Source code names variables and functions symbolically. Compilation and linking organize references within an executable.

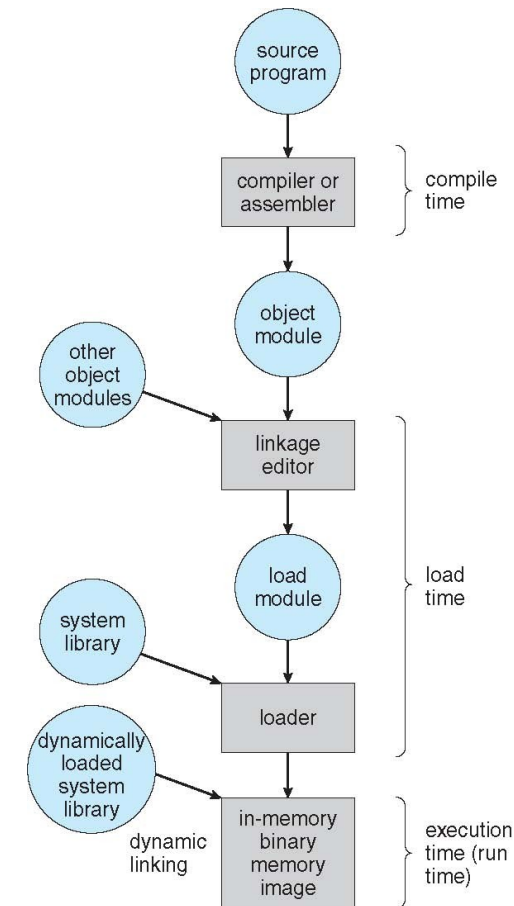
Compile-time binding assumes a location; load-time binding resolves placement when loading; execution-time translation allows later placement changes.

Relocation requires agreement between the executable, loader and hardware. A program's virtual address is not a direct statement of its RAM location.

Compilation, linking and loading

Follow one reference from source code through object modules and the load module. Libraries may enter at different stages.

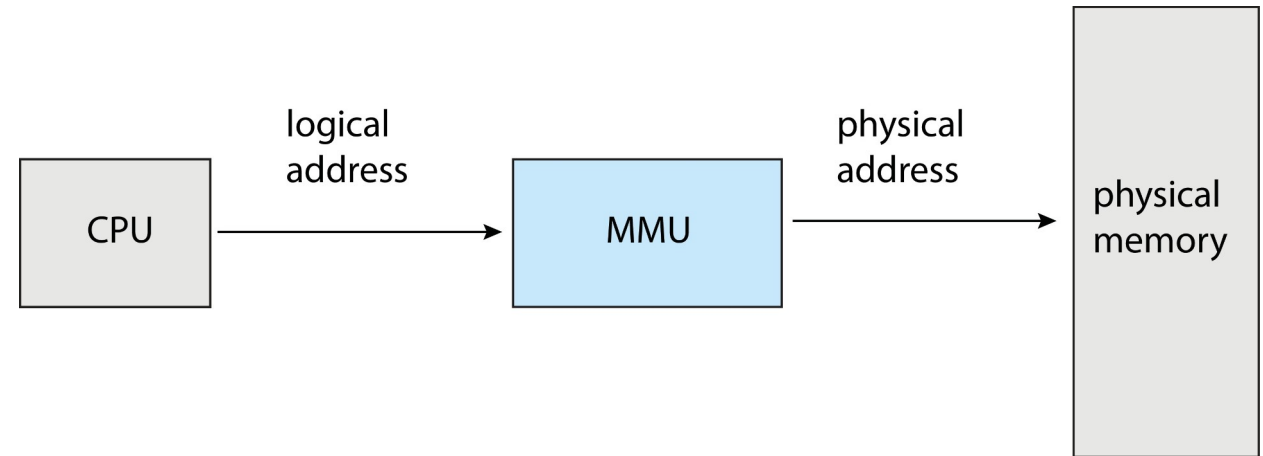
The diagram describes a conceptual toolchain. Runtime address translation is still needed after the program has been loaded.



Logical and physical addresses

A virtual (logical) address is generated in the process's address space. A physical address selects a location after translation.

The memory-management unit performs the mapping. The OS maintains the mapping data and handles exceptional cases.

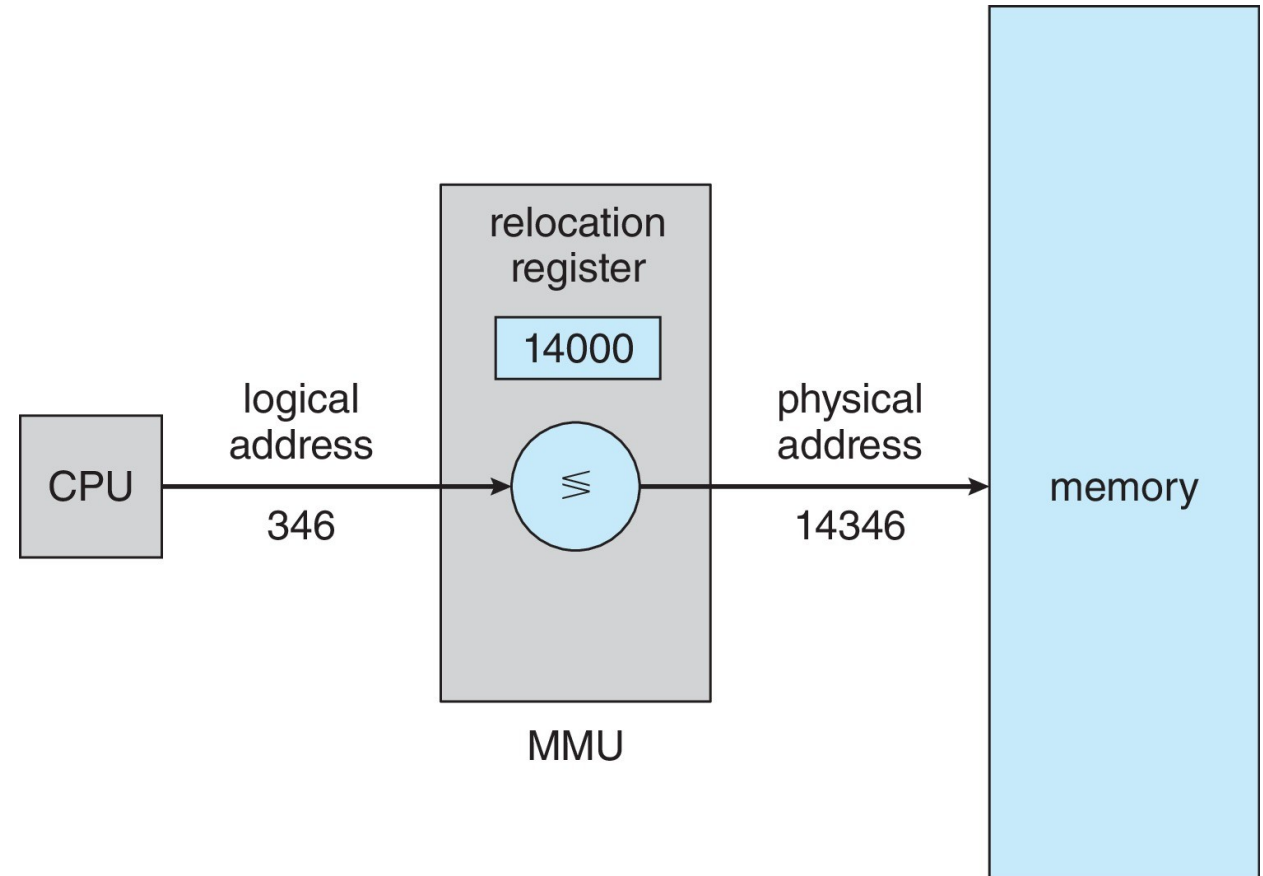


Relocation at execution time

In the simple relocation model, physical address = base + logical offset.

For base 14000 and offset 346, the result is 14346.
The process continues to use offset 346.

Moving the region changes the base, not every address expression in the program.

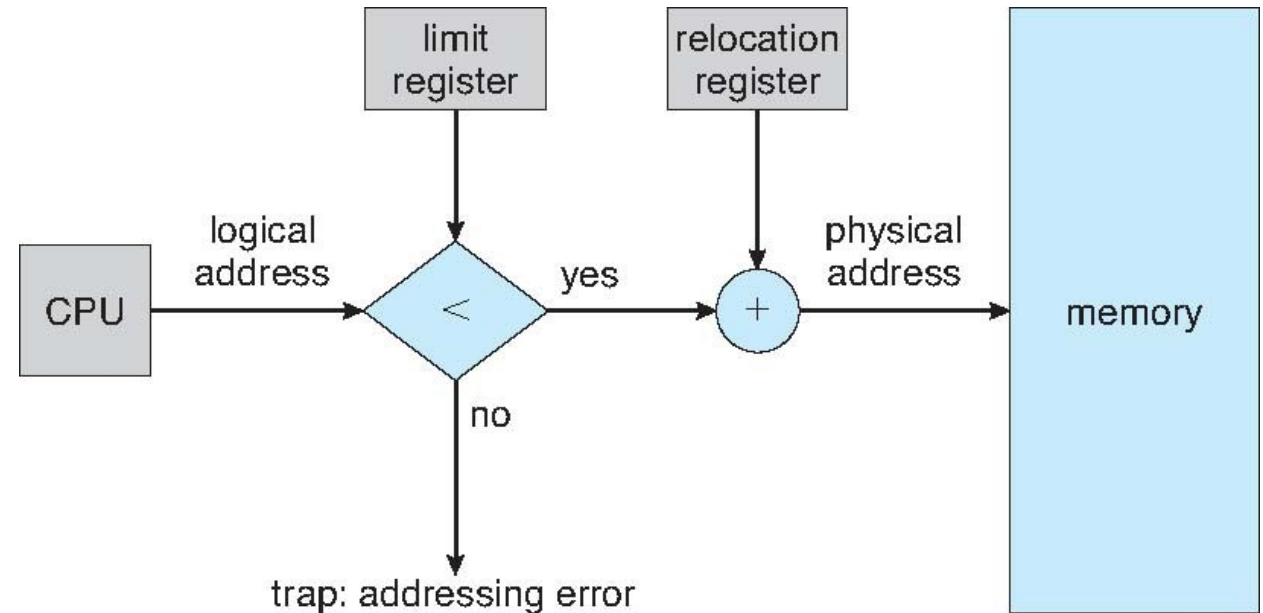


Check the limit before adding the base

For this offset model, require $0 \leq \text{offset} < \text{limit}$.
Only then compute base + offset.

Base 14000 and limit 1000 admit offset 999 $\rightarrow$ 14999. Offset 1000 must trap.

Relocation without a bound would translate an out-of-range reference into someone else's memory.



Dynamic loading and dynamic linking

Dynamic loading delays loading a routine until it is needed. Dynamic linking resolves library references at run time.

They address different questions: which code is loaded, and how references to separately supplied code are connected.

Shared libraries can reduce duplicate code storage. Mapping, permissions and per-process writable state still matter.

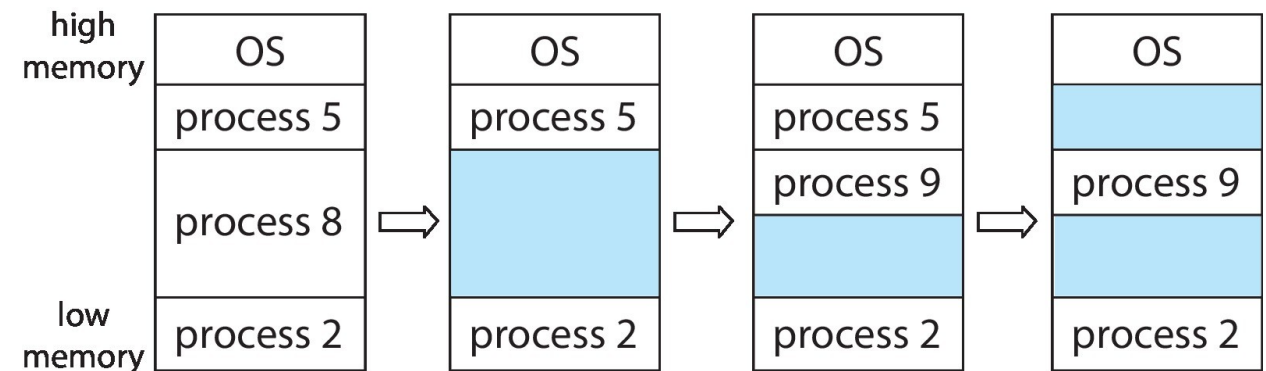
These ideas do not imply that all pages must remain in RAM; demand paging is developed in Week 6.

Contiguous allocation and holes

In this model each process occupies one continuous physical region. Exiting processes leave holes; adjacent free holes can merge.

A request needs one sufficiently large hole. Adding all free bytes is not enough to decide whether it fits.

Observe the changing region boundaries in the original allocation sequence.



Three placement policies

First fit searches in address order and chooses the first sufficient hole.

Best fit chooses the smallest sufficient hole; worst fit chooses the largest.

State the search order, tie rule, splitting and coalescing assumptions before comparing results.

A choice that looks economical for one request can make later requests harder.

Placement exercise: start with the same holes

Holes in address order: 100, 500, 200, 300, 600 KiB. Requests: 212, 417, 112, 426 KiB. Split chosen holes; no deallocation.

Request	First fit	Best fit	Worst fit
212	500 $\rightarrow$ 288	300 $\rightarrow$ 88	600 $\rightarrow$ 388
417	600 $\rightarrow$ 183	500 $\rightarrow$ 83	500 $\rightarrow$ 83
112	288 $\rightarrow$ 176	200 $\rightarrow$ 88	388 $\rightarrow$ 276
426	Cannot fit	600 $\rightarrow$ 174	Cannot fit

External and internal fragmentation

External fragmentation: enough total free space exists, but no single hole can satisfy the contiguous request.

Internal fragmentation: a reserved allocation unit contains bytes not needed by that allocation.

After first fit in the exercise, 959 KiB remains free, but the largest hole is only 300 KiB. The 426 KiB request fails.

The distinctions depend on the allocation unit and model; describe what is being wasted.

Compaction and its costs

Compaction moves allocated regions together to consolidate free space. It requires movable addresses and safe coordination with references.

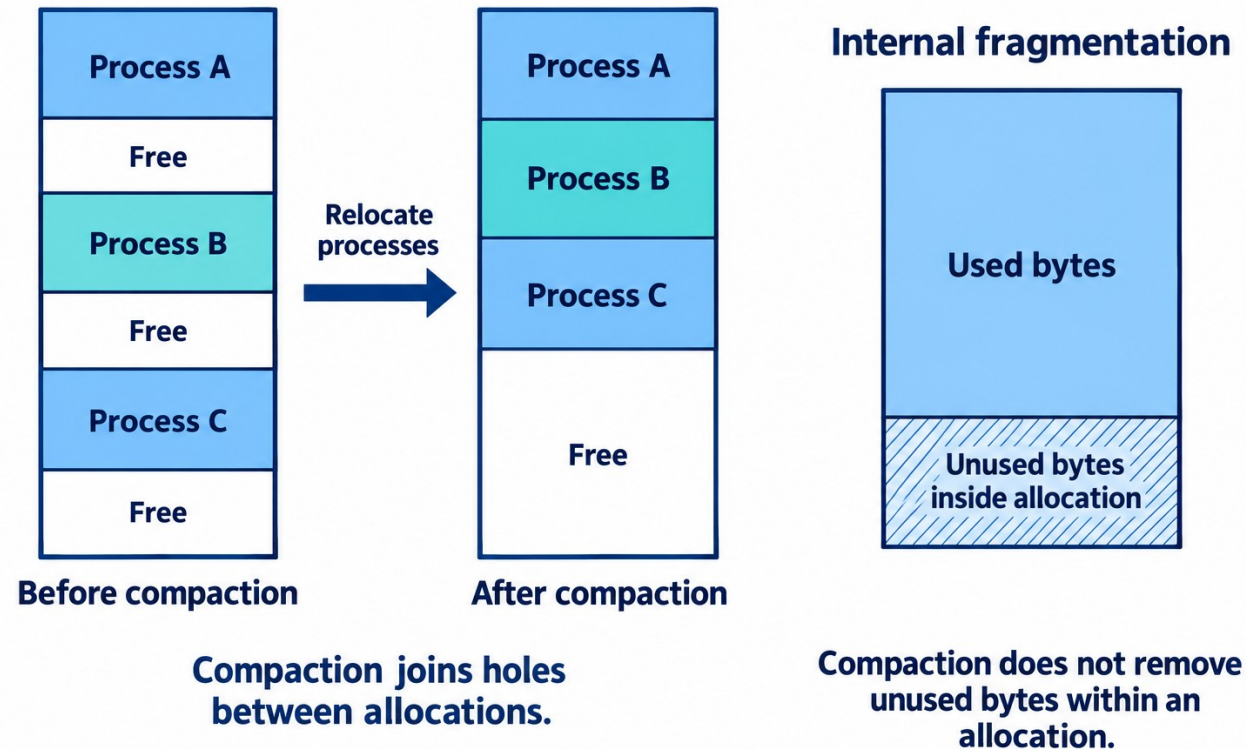
Copying data costs time. Pending I/O and pointers understood outside the relocation mechanism complicate movement.

Paging changes the allocation unit: a process can use separated frames instead of requiring one contiguous physical region.

This removes the contiguous-process placement problem; it does not make every kind of fragmentation disappear.

Fragmentation and compaction

External and internal fragmentation



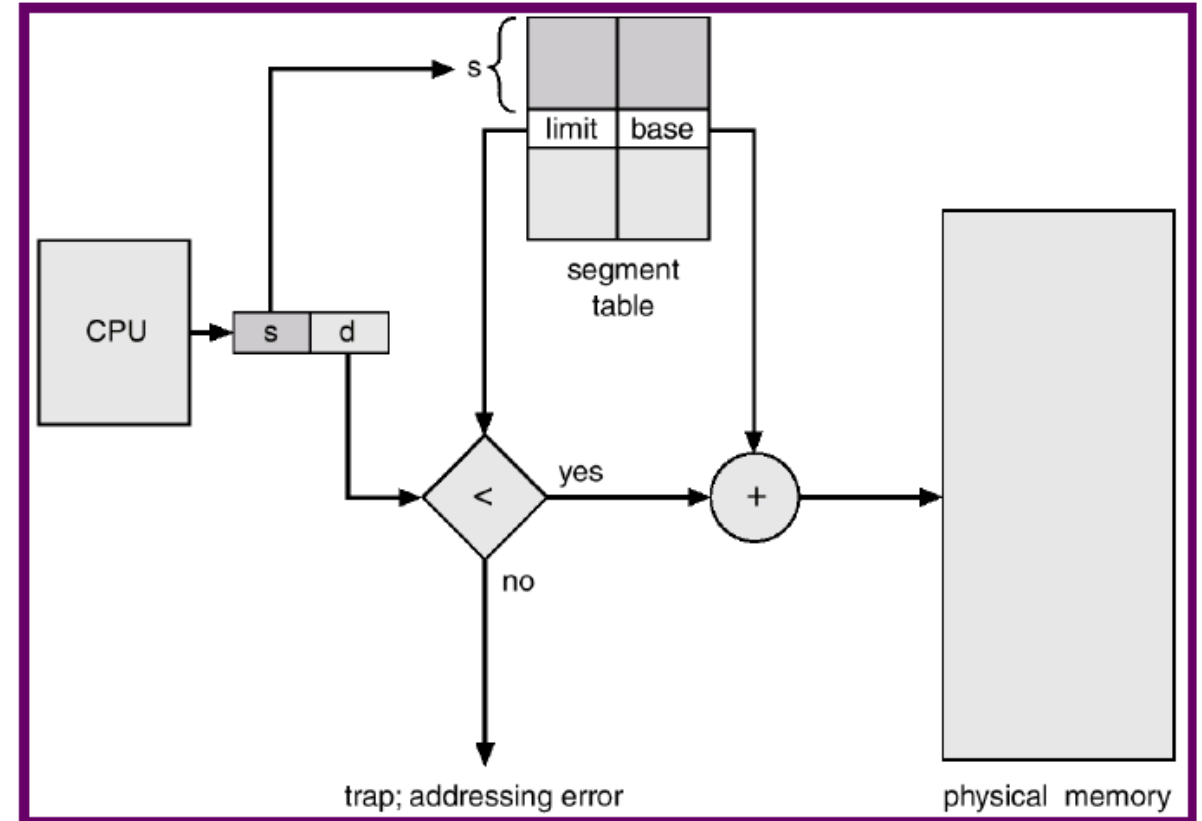
Schematic, not to scale. Compaction relocates allocations; it does not remove unused space inside an allocation.

Segmentation expresses logical regions

A segment is a variable-sized logical region, such as code, a stack or a table. An address is (segment number, offset).

A segment-table entry supplies a base, length and permissions. Check the offset against the length, then add the base.

This explains a useful alternative to fixed-sized pages. The historical hardware model is not a claim about every current Linux process.



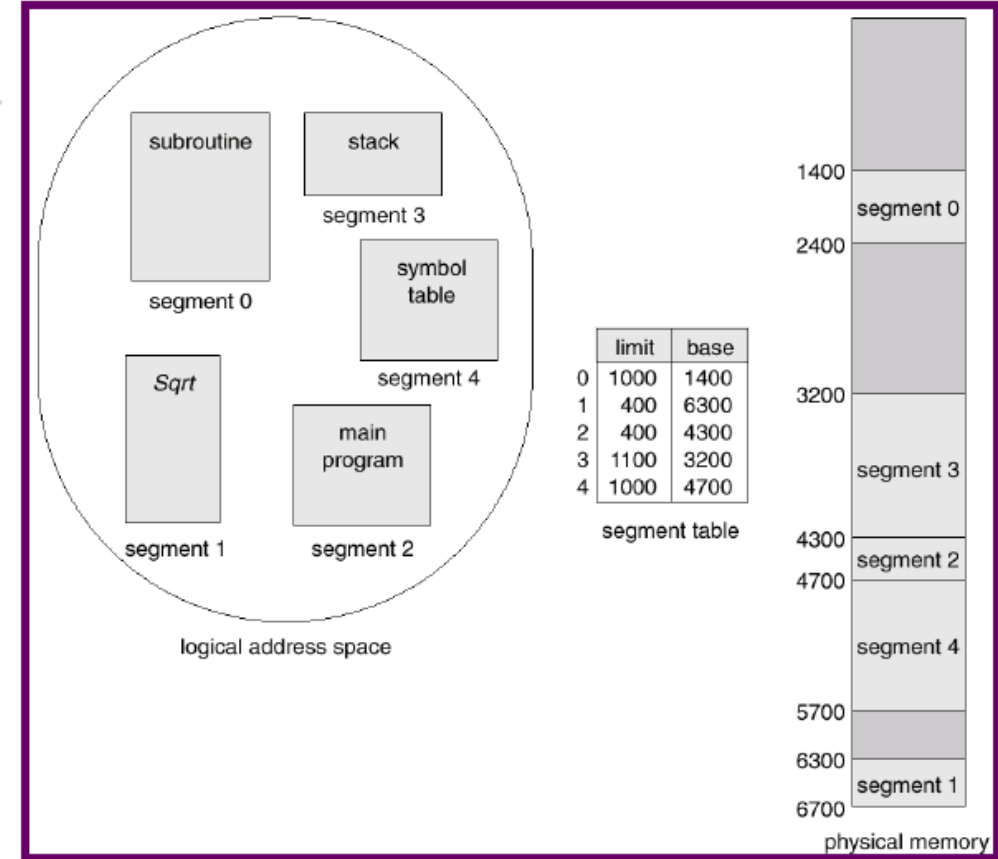
Göktürk · BIL601 · adapted OSC segmentation figure

A segment-table calculation

Segment 0: base 1400, length 1000. Segment 1:
base 6300, length 400.

$(0, 430) \rightarrow 1830$; $(1, 399) \rightarrow 6699$. $(1, 400)$ is
illegal even if physical byte 6700 exists.

Variable lengths match logical regions but can
produce external fragmentation. Paging uses fixed-
sized allocation units.



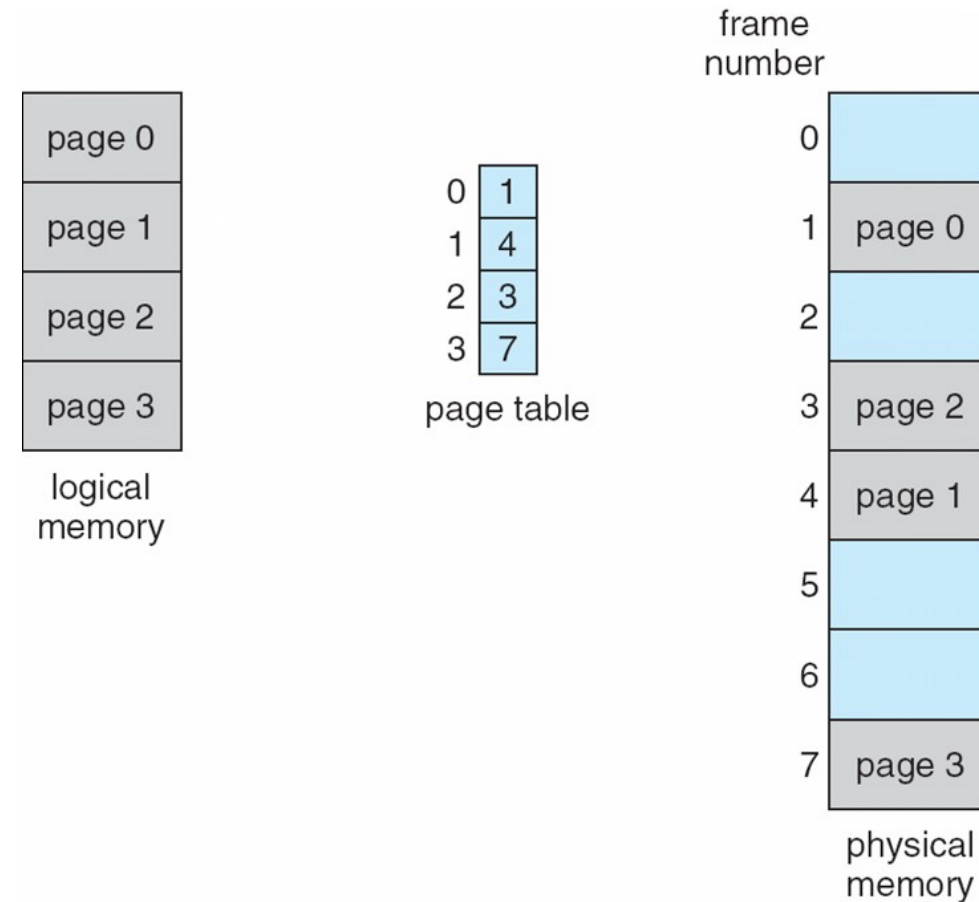
Göktürk · BIL601 · adapted OSC segmentation figure

Pages and frames

Divide the virtual address space into pages and physical memory into frames of the same size.

A page table records which frame backs each mapped virtual page. Consecutive virtual pages need not occupy consecutive frames.

Within a page, byte order and the offset are preserved.

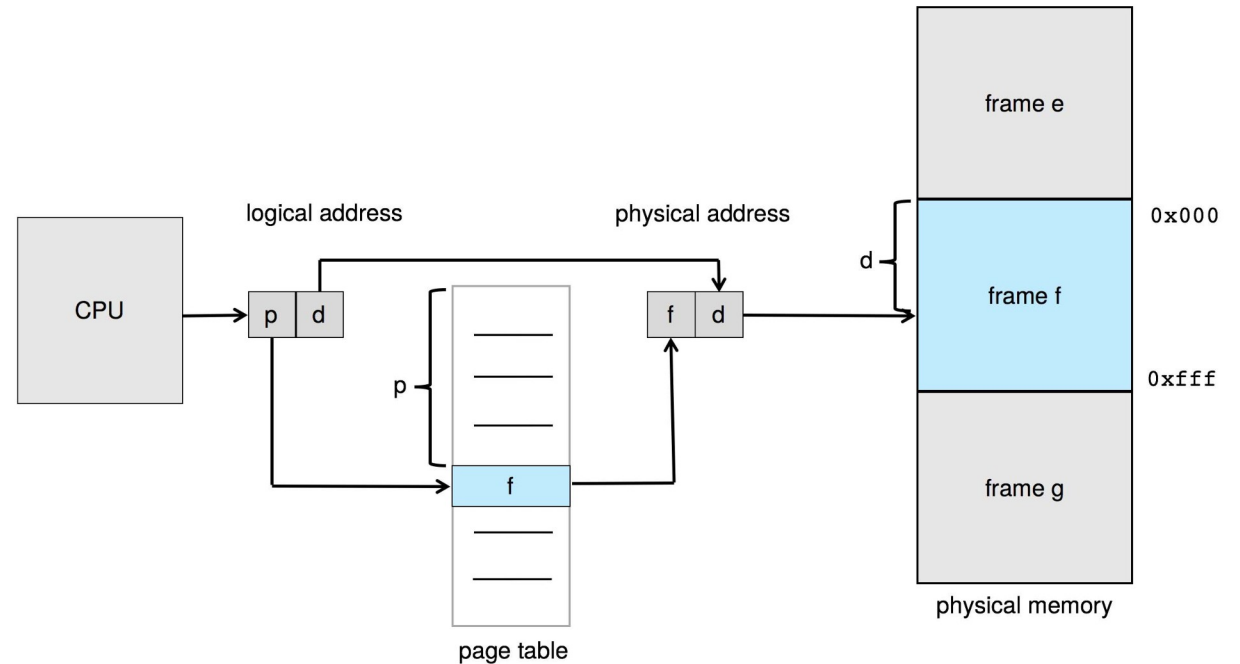


The address translation path

Split virtual address A into page number p and offset d . Look up p to obtain frame f .

For page size P : $p = \text{floor}(A/P)$, $d = A \bmod P$,
physical address = $f \times P + d$.

Validate the mapping and access permissions
before using the result.



With a 32-bit virtual address and 4 KiB pages, the offset uses 12 bits and the virtual page number uses 20 bits.

4 KiB = 4096 bytes = 2^{12} bytes. The low 12 bits survive translation unchanged.

The number of frame-number bits depends on the physical address width. It need not equal the number of virtual-page bits.

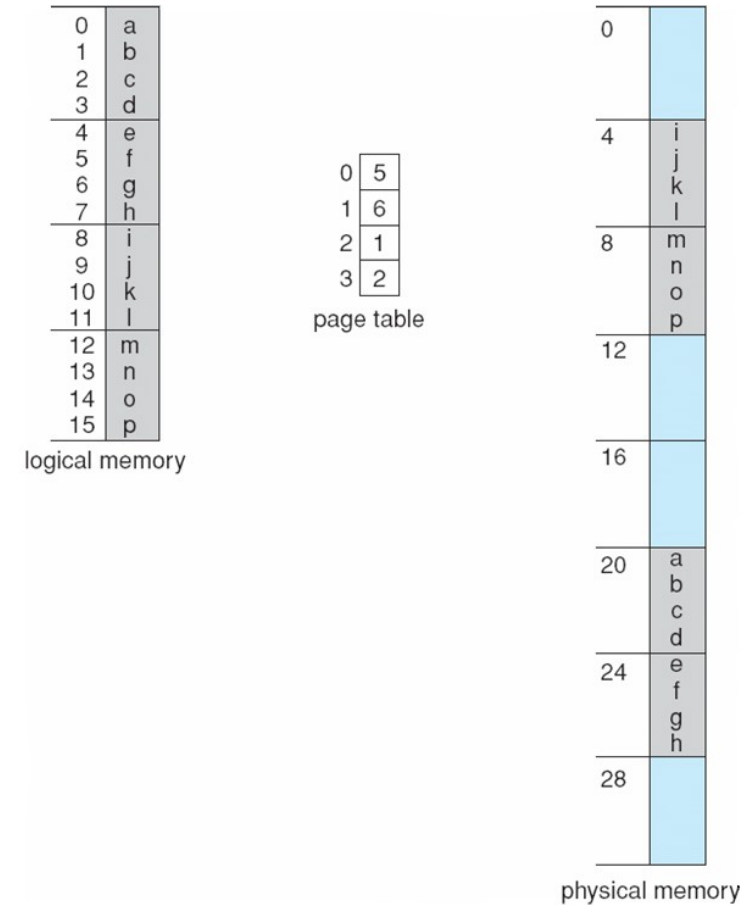
Write addresses in bytes and label KiB, MiB and hexadecimal notation explicitly.

A complete miniature paging example

The source uses 4-byte pages, a 16-byte virtual space and 32 bytes of physical memory.

Page table: $0 \rightarrow 5$, $1 \rightarrow 6$, $2 \rightarrow 1$, $3 \rightarrow 2$. Virtual address 6 is page 1, offset 2.

Its physical address is $6 \times 4 + 2 = 26$. The tiny sizes make every byte visible.



Translation practice with 4 KiB pages

Use this table for a read at virtual address 0x1234. Page 1 maps to frame 9.

Virtual page	Frame	Mapping / permission
0	3	Mapped, read/write
1	9	Mapped, read-only
2	—	Unmapped
3	1	Mapped, read/write

Work through 0x1234

$0x1234 = 4660$ decimal. Divide by 4096: page number = 1; offset = 564 = $0x234$.

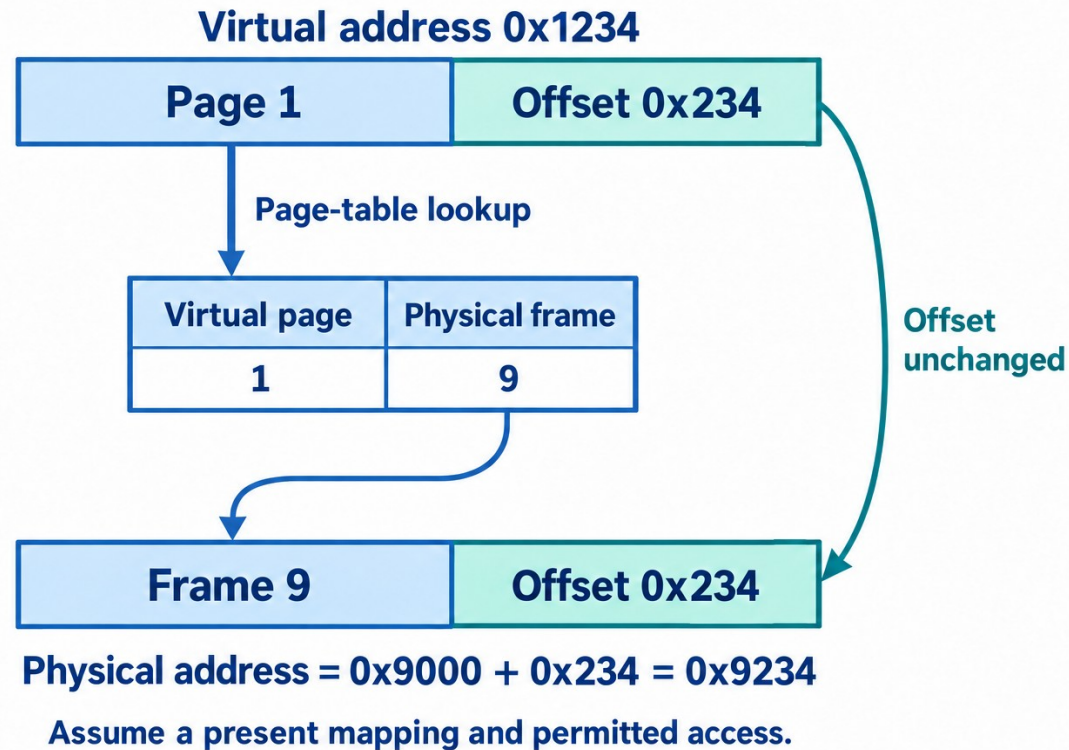
Page 1 maps to frame 9. Its physical base is $9 \times 4096 = 36864 = 0x9000$.

Physical address = $36864 + 564 = 37428 = 0x9234$.

A read is permitted. A write to the same virtual address is rejected by the read-only permission.

The page changes; the offset stays

Translating 0x1234 with 4 KiB pages



Page 1 maps to frame 9. With 4 KiB pages, offset 0x234 gives physical address 0x9234.

Page boundaries are discontinuities

Virtual 0x0FFF → frame 3, offset 4095 → physical 0x3FFF.

The next virtual byte, 0x1000, is page 1, offset 0 → physical 0x9000.

Virtual adjacency does not imply physical adjacency across a page boundary. Within one page the offset remains unchanged.

A multi-byte access crossing a boundary may require two translations and checks.

Reject invalid references explicitly

Virtual 0x2000 names unmapped page 2 in our table. There is no physical result to invent.

A read-only mapping can translate a read but reject a write. Mapping existence and permitted access are separate checks.

A valid virtual page that is temporarily nonresident is another situation. Week 6 explains how a fault handler may make it resident.

Our simulator labels UNMAPPED and PROTECTION separately; it does not simulate demand paging.

The final page wastes space

Page size = 2048 bytes. Process allocation = 72766 bytes.

$72766 = 35 \times 2048 + 1086$, so 36 pages are required.

Reserved space = 73728 bytes. Internal fragmentation = $2048 - 1086 = 962$ bytes.

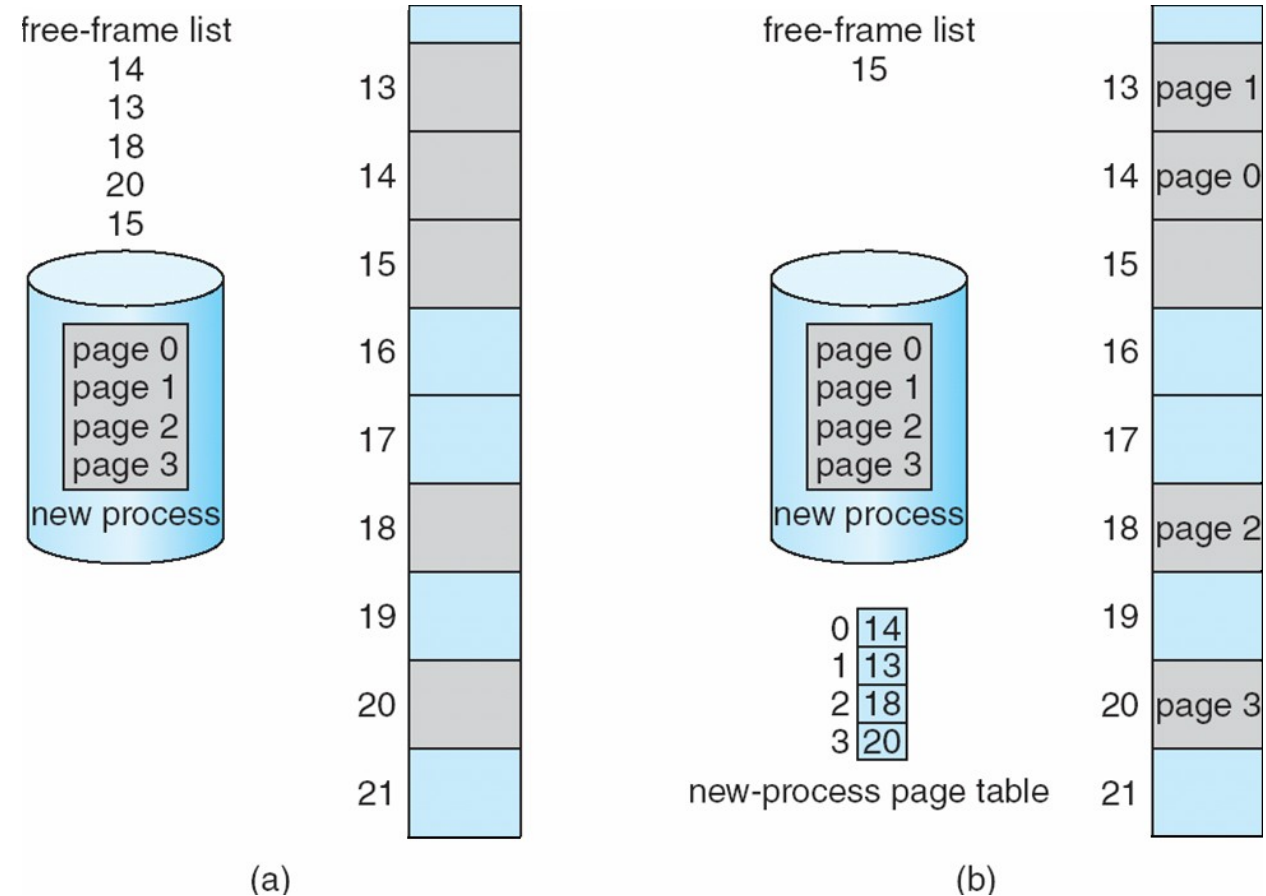
The familiar half-page average assumes a suitable distribution of final-page occupancy; it is not a law for every workload.

Free frames and allocation

The OS selects free physical frames and records their assignment. The process sees an ordered virtual space even when the frames are scattered.

Frame availability is system-wide bookkeeping; a process's page table describes its own mappings.

A larger page can reduce table entries but increase unused space in partially occupied pages.



Half-time check: explain the mapping

Why can two processes both use virtual address 0x1234? Their mappings can select different physical frames.

Why does offset 0x234 stay unchanged? Translation changes the page base, not the byte's position within the page.

Why can a 426 KiB allocation fail with 959 KiB free? The contiguous model requires one sufficiently large hole.

Next: how do we store and cache all these mappings efficiently?

Page tables are data in memory

The page-table base register identifies the current table in a simple one-level model. A page-table entry stores a frame and metadata.

Without a translation cache, one data access requires a PTE read and then the requested memory access.

A page table is not a second copy of the program. It records how to interpret addresses and what accesses are permitted.

The address of a page-table entry

Assume a linear physical page table, PTBR = 0x8000 and each PTE occupies 8 bytes.

For virtual page 1, PTE address = $0x8000 + 1 \times 8 = 0x8008$.

That is where the mapping entry is stored; it is not the physical address of the requested data.

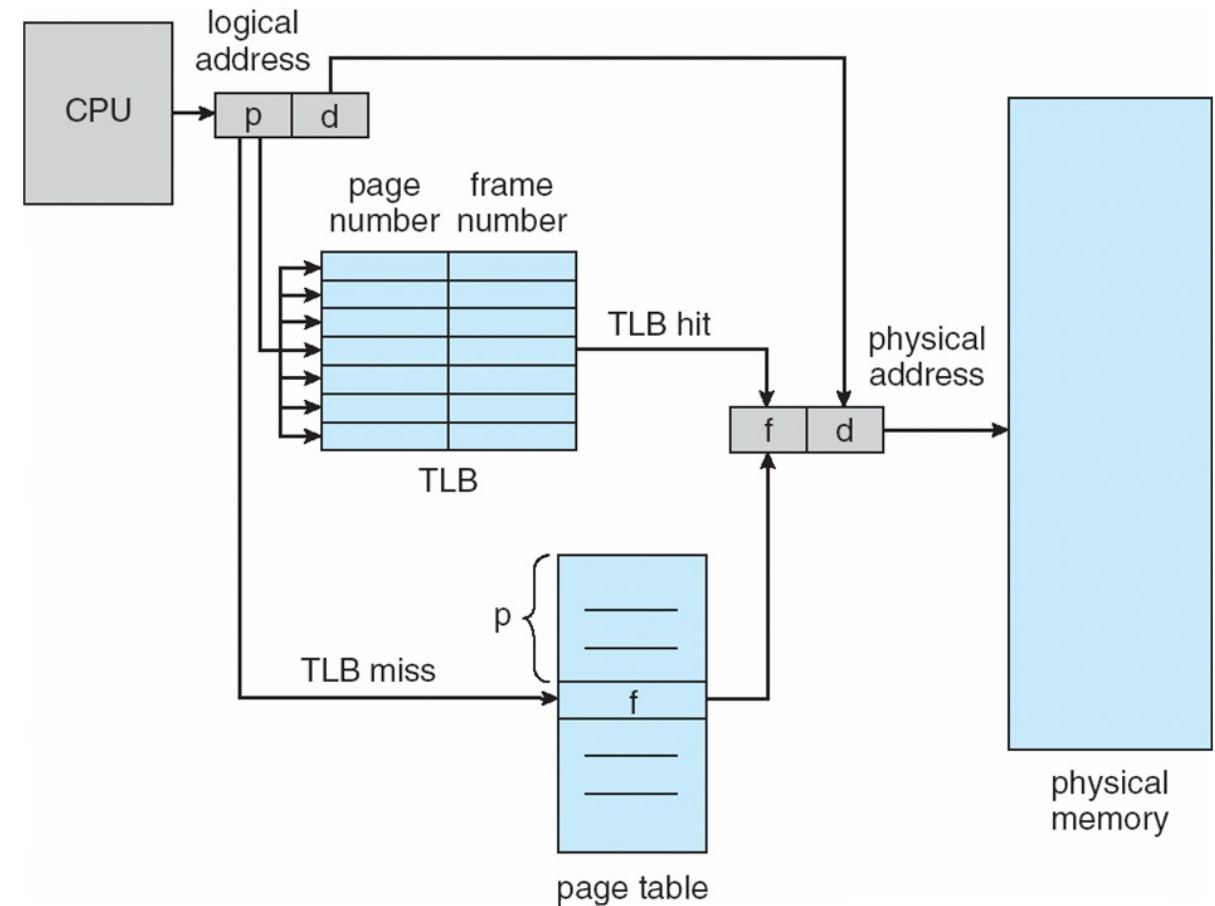
If the entry names frame 9 and offset is 0x234, the data address is 0x9234. Keep the two addresses separate.

The TLB caches translations

The translation lookaside buffer stores recently useful address translations and associated metadata. It does not hold the program's data bytes.

A hit can supply the frame and permissions without a page-table walk. A miss requires looking up the mapping.

Data caches and TLBs solve different problems, even though both exploit locality.



A TLB miss is not a page fault

A missing cached translation can be recovered from a valid, resident page-table mapping. No disk access is required in that case.

A page fault is an exception associated with the attempted access, for example a nonresident page or a protection violation on relevant architectures.

Whether hardware or software handles a TLB miss depends on the architecture. Do not assume every miss invokes the OS.

Record translation-cache behavior separately from residency and permissions.

Trace a small TLB

Assume two fully associative entries, initially empty, LRU replacement, and all pages mapped and resident.
Reference pages 0, 1, 0, 2, 1.

Page	Hit / miss	TLB after access (old → new)
0	Miss	0
1	Miss	0, 1
0	Hit	1, 0
2	Miss; evict 1	0, 2
1	Miss; evict 0	2, 1

Interpreting the TLB trace

There is one hit and four misses. Hit ratio = $1/5 = 20\%$.

All four misses are resolved from resident mappings in this example. The number of demand-paging faults is zero.

Sequential accesses within a page can reuse one translation. Crossing into a new page may need another entry.

The result depends on the reference sequence, cache capacity, initial contents and replacement assumptions.

Effective access time: state the model

Let memory access $m = 100$ ns, TLB lookup $t = 1$ ns, and hit ratio $h = 0.95$. Assume serial lookup, one-level tables and no other caches.

Hit cost = $t + m = 101$ ns. Miss cost = $t + 2m = 201$ ns.

$$\text{EAT} = h(t + m) + (1 - h)(t + 2m) = 106 \text{ ns.}$$

This is a teaching model, not a measured latency prediction for a modern CPU.

Compare hit ratios under one model

Keep $t = 1$ ns and $m = 100$ ns. $EAT = 201 - 100h$ ns.

Hit ratio h	Effective access time	Extra vs. 100 ns baseline
0.80	121 ns	21%
0.95	106 ns	6%
0.99	102 ns	2%
1.00	101 ns	1%

Context switches and cached mappings

The same virtual page number can mean different frames in different address spaces. A TLB must distinguish those contexts.

Address-space identifiers can tag entries. Otherwise relevant old translations must be invalidated when changing address spaces.

Changing permissions or mappings also requires a consistency protocol for cached translations. The details depend on hardware and OS.

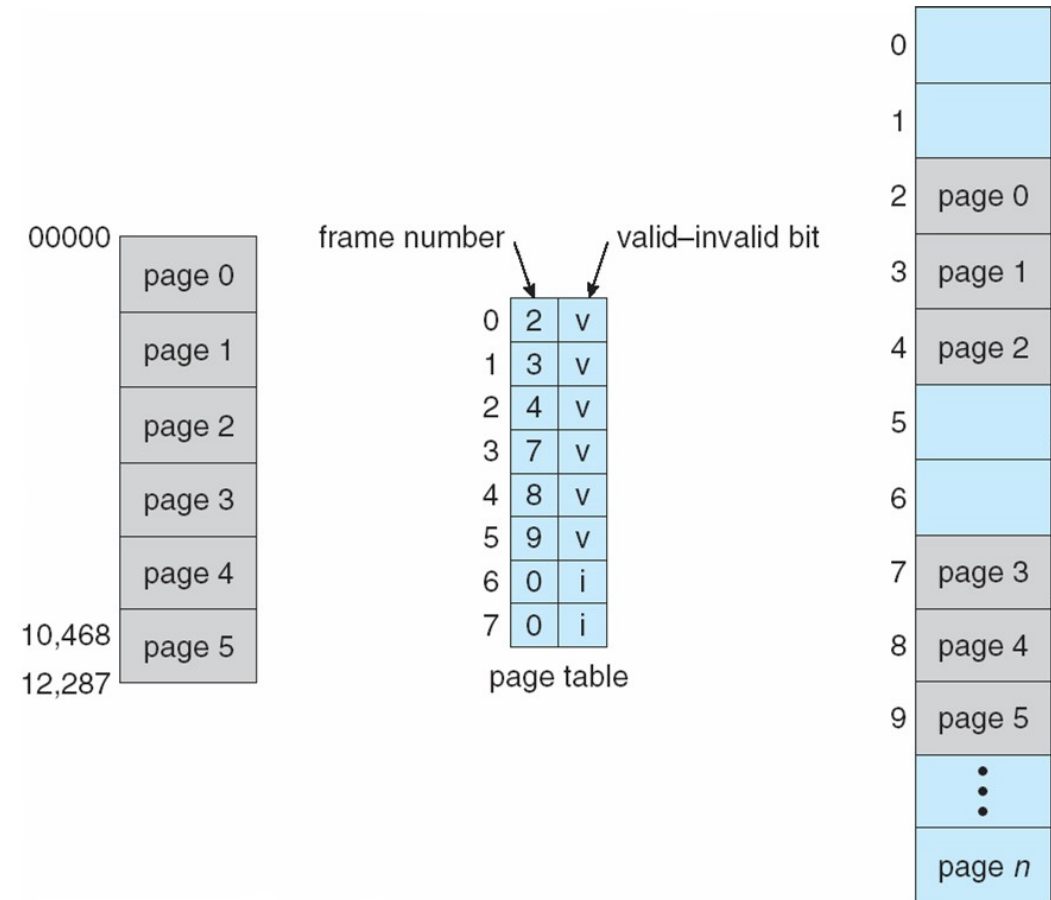
A context switch is not simply a change of instruction pointer.

Mapping validity and permissions

The figure's valid/invalid bit describes membership in a process's legal logical address space. Read the model before interpreting a bit.

A resident/nonresident distinction asks a different question. Real entries contain architecture-specific state.

Read, write, execute and user/kernel restrictions can reject an access even when a frame number is available.

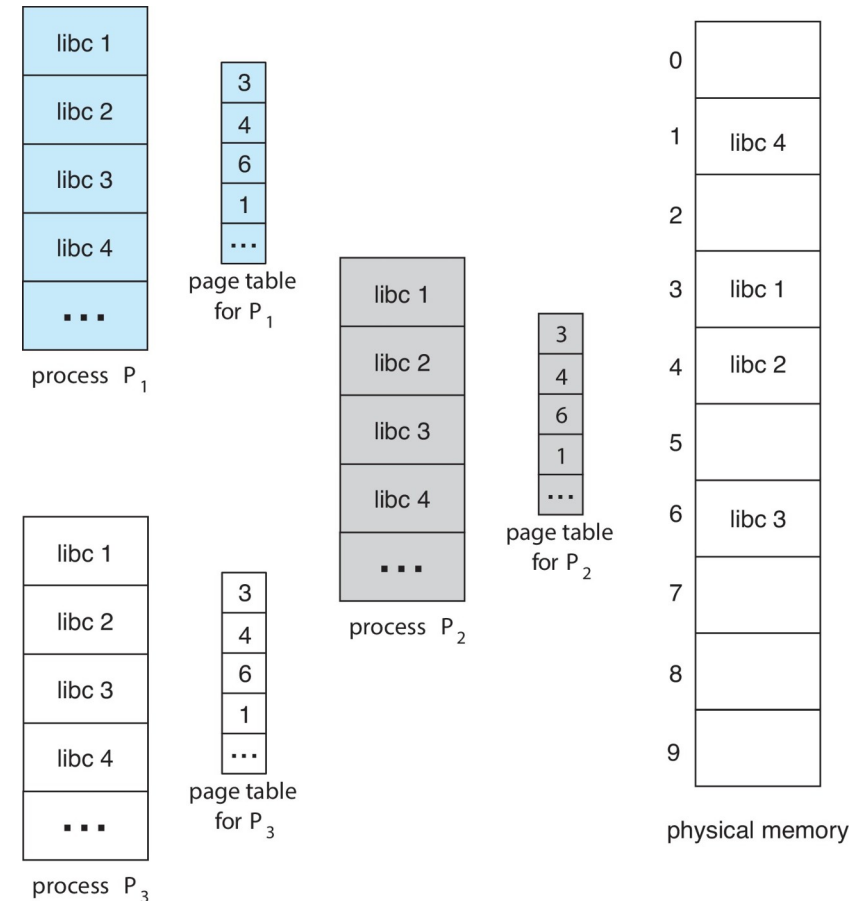


Shared code, private data

Different processes may map shared read-only code frames while retaining private writable data.

Sharing is implemented through mappings. It does not require the processes to share their entire address space.

Writable shared memory also needs a synchronization protocol; address translation does not prevent a data race.



Sharing does not require equal virtual addresses

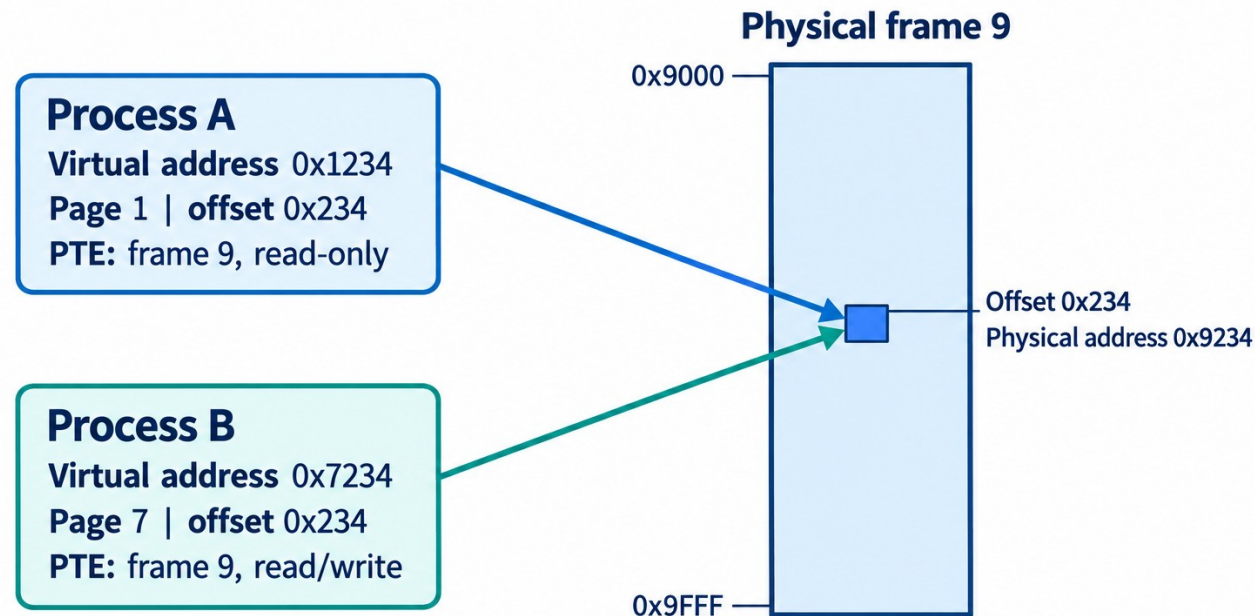
Process A maps virtual page 1 to frame 9; process B maps virtual page 7 to the same frame. Page size is 4096.

A reads 0x1234; B reads 0x7234. Both reach frame 9 at offset 0x234, hence physical 0x9234.

Equal virtual addresses can reach different frames; different virtual addresses can reach the same frame.

Permissions belong to the relevant mapping. A shared frame can have different access permissions through different mappings.

Two virtual addresses, one physical byte



4 KiB pages. Permissions belong to each mapping.

A write through A is rejected; a write through B is permitted.

Both mappings reach the same byte. The access check uses the permissions of the mapping used by the process.

How large is a linear page table?

A 32-bit byte address space with 4 KiB pages contains 2^{20} virtual pages.

At 4 bytes per PTE, a full linear table needs 2^{22} bytes = 4 MiB per address space.

This includes entries for unmapped ranges. A process using only a few distant regions can still require a large linear table.

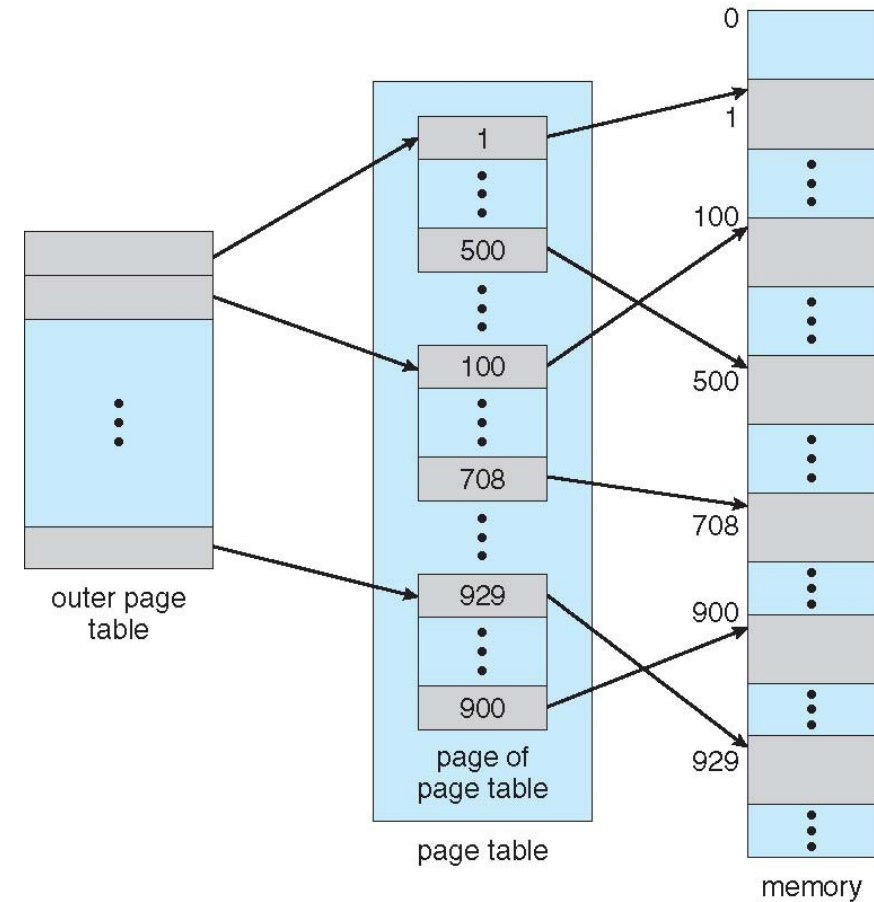
The key question is how to represent sparse mappings without allocating every possible entry.

Hierarchical page tables

Divide the page-table structure into smaller tables.
An outer entry points to a lower-level table when that region is used.

An unused region can lack the lower-level table entirely. Sparse mappings can therefore save table memory.

A miss may need more dependent memory reads;
the TLB reduces how often that walk is required.

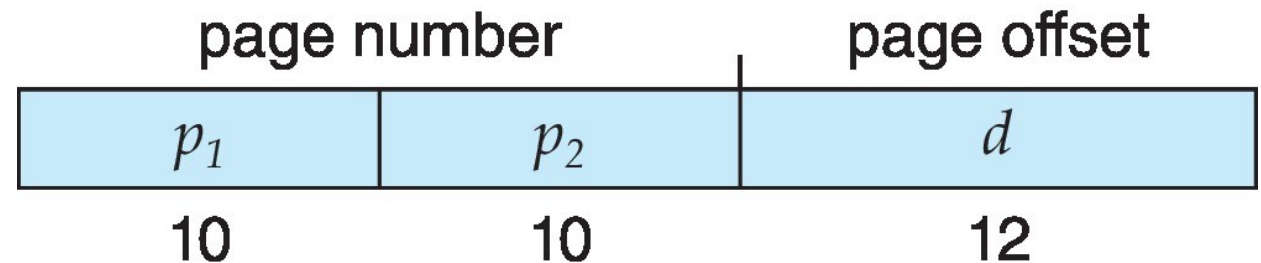


A 10 / 10 / 12 address split

Assume 32-bit addresses, 4 KiB pages, and 4-byte entries in both levels. Each table page holds 1024 entries.

The upper 10 bits select an outer entry; the next 10 select an inner entry; the low 12 are the byte offset.

These bit widths follow from the stated sizes. They are not universal constants.

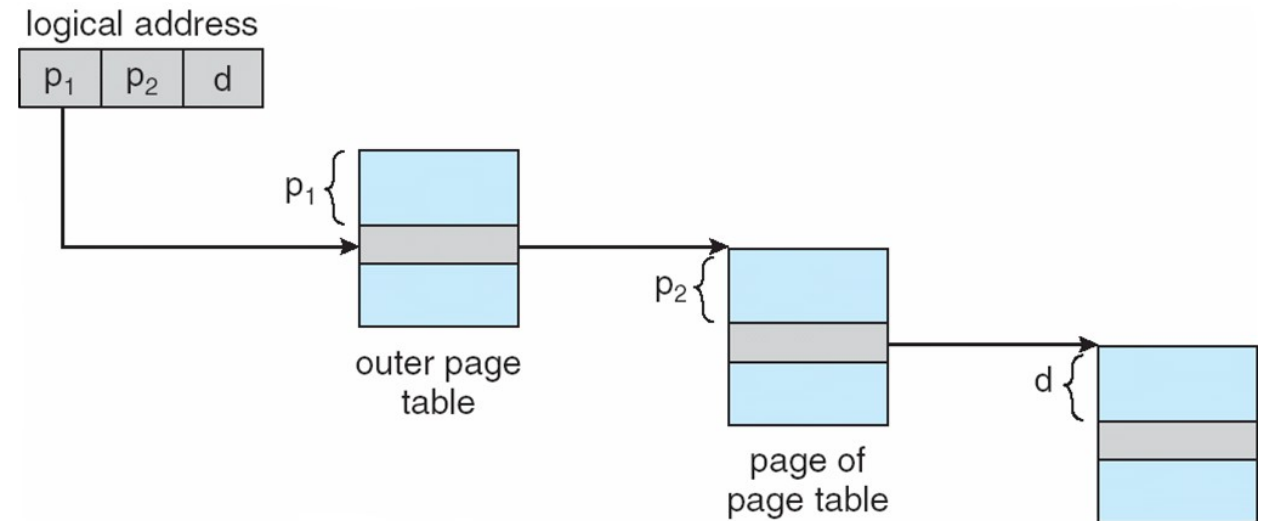


Walking two levels

Use the outer index to select a pointer to an inner table; use the inner index to select the final frame.

Only after finding a permitted leaf mapping do we combine the frame with the offset.

Intermediate table addresses and the final data address play different roles. Label each arrow before calculating.



Two-level numerical walk

Virtual address 0x00403004 has outer index 1, inner index 3 and offset 4. Assume 4-byte entries.

Step	Calculation	Result
Outer entry	$0x1000 + 1 \times 4$	$0x1004 \rightarrow$ inner base $0x5000$
Inner entry	$0x5000 + 3 \times 4$	$0x500C \rightarrow$ frame $0xA$
Data byte	$0xA \times 4096 + 4$	$0xA004$

Sparse-table memory savings

Map pages in only three distinct outer-index regions. Assume one 4 KiB root and one 4 KiB leaf table per used region.

Table storage = $(1 + 3) \times 4 \text{ KiB} = 16 \text{ KiB}$, compared with a full 4 MiB linear table.

If every outer region needs a leaf table, the hierarchy needs 1025 table pages, slightly more than the linear table.

Savings depend on sparsity. A hierarchy is not automatically smaller for every mapping pattern.

The cost of a multilevel walk

With k table levels, an uncached translation can require k PTE accesses followed by one data access.

For $k = 2$ and $m = 100$ ns, a miss costs 301 ns including a serial 1 ns TLB lookup.

With $h = 0.95$, $EAT = 0.95 \times 101 + 0.05 \times 301 = 111$ ns.

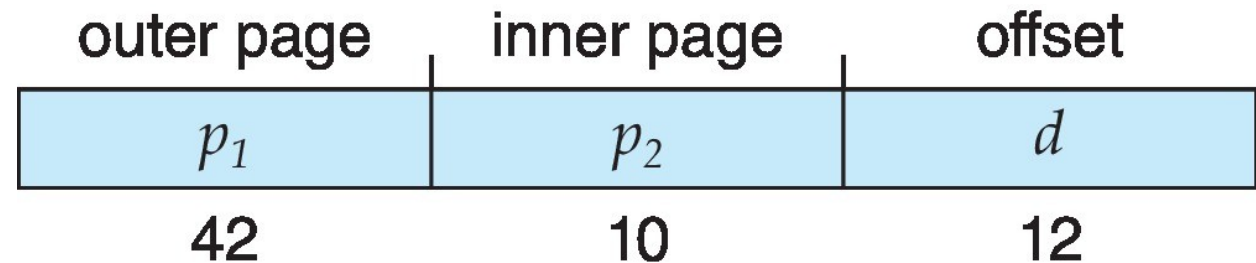
Real CPUs use data caches, walk caches and overlap. This calculation isolates the mechanism.

More address bits need careful accounting

A hypothetical full 64-bit virtual space with 4 KiB pages has 2^{52} possible page entries. A flat table is impractical.

The figure is an arithmetic model of a large address space, not a specification for every 64-bit CPU.

Implemented virtual and physical widths can be smaller than the register width.

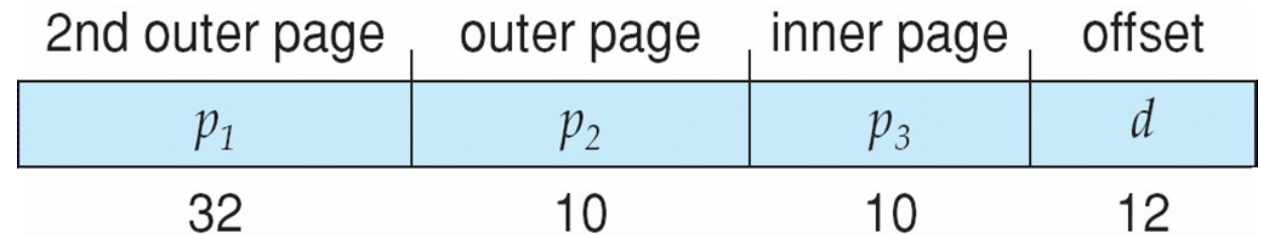


Adding another table level

Each index selects the next table; the last entry selects the data frame. Read the diagram from the outermost index inward.

Extra levels allow large unused ranges to remain unallocated. They also increase uncached walk depth.

A large-page leaf may terminate a walk earlier on architectures that support it.

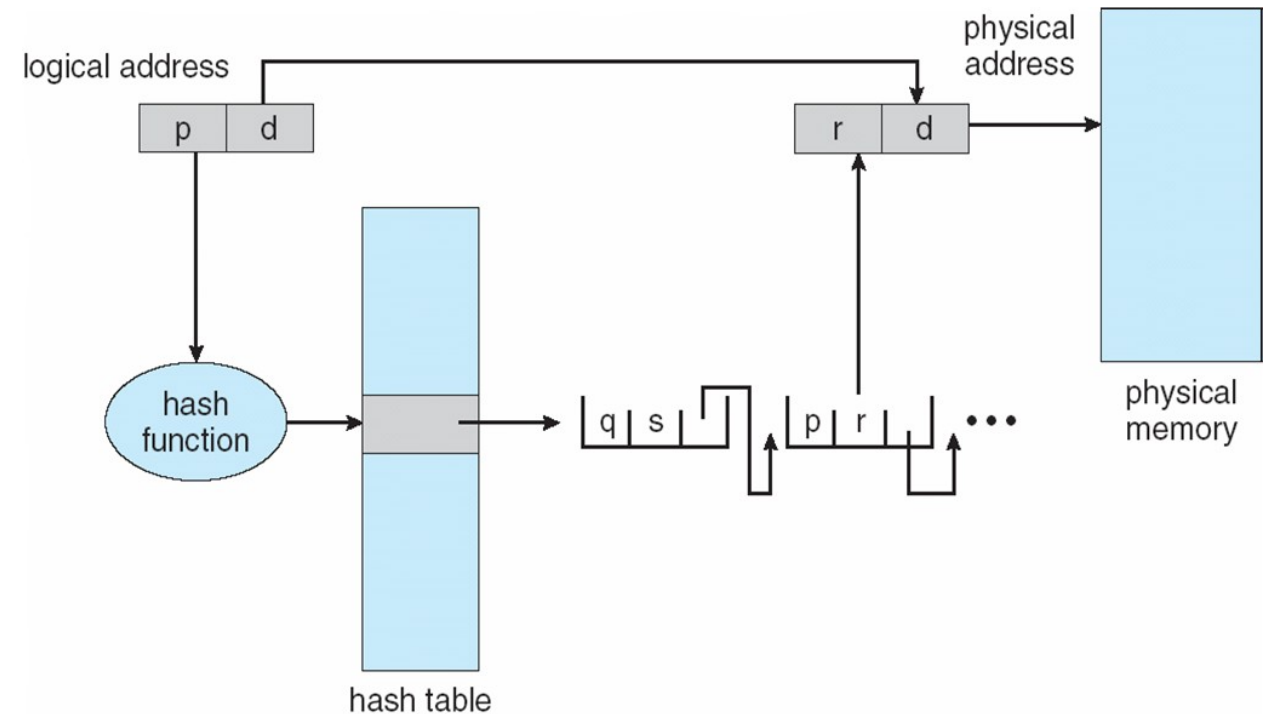


Hashed page tables

Hash the virtual page number to find a candidate chain. Compare the key before accepting a mapping; collisions are possible.

The key must distinguish address spaces when a structure is shared across processes.

Hashing is another representation trade-off. It is not the hierarchy used by every operating system.

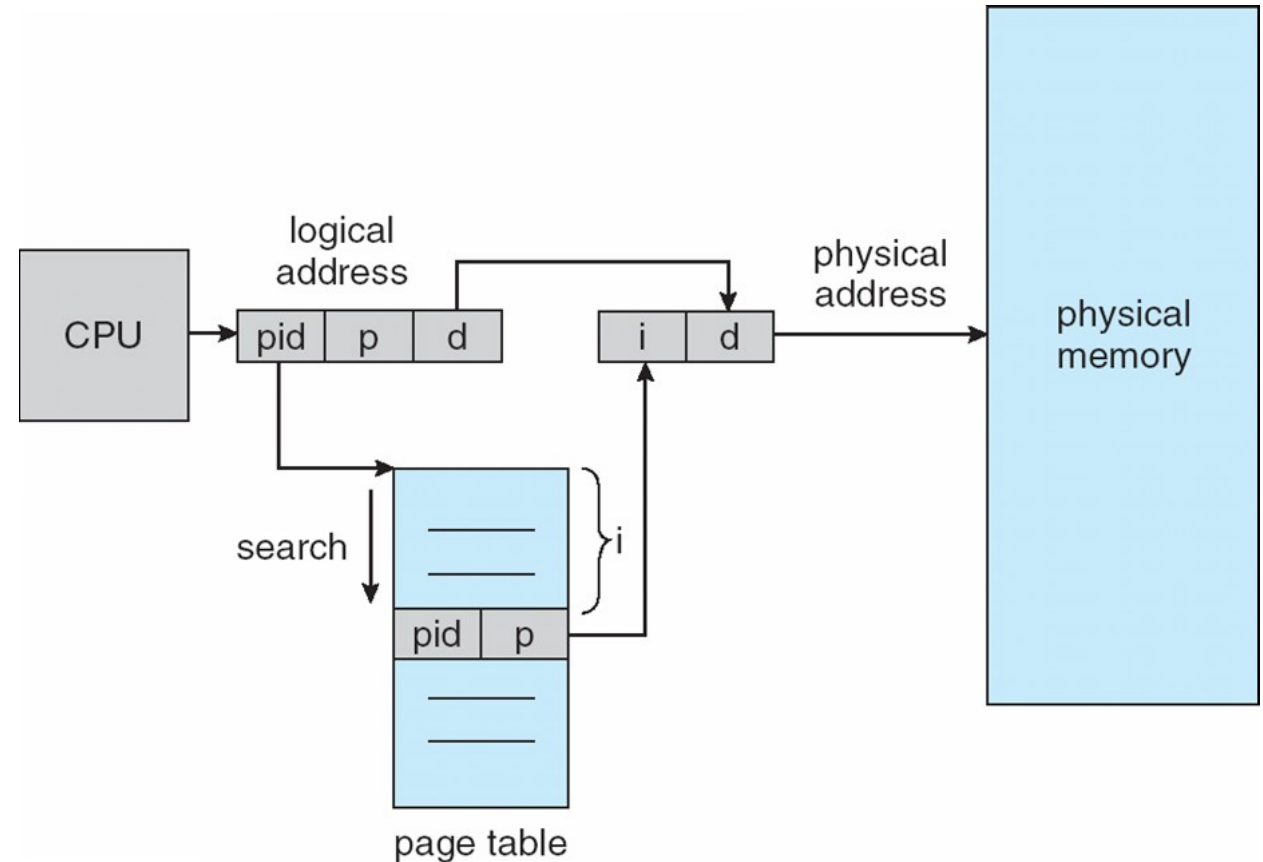


Inverted page tables

An inverted table organizes entries around physical frames, including information identifying the owning virtual mapping.

This can reduce storage tied to the virtual address range, but finding a mapping requires an efficient search mechanism.

Shared frames complicate a simple one-entry/one-owner model. TLBs can still accelerate repeated translations.



Page size is a trade-off

Larger pages can reduce table entries and increase the bytes covered by a fixed number of TLB entries.

Smaller pages can reduce unused space and provide finer allocation/protection granularity.

For a toy TLB with 64 entries: 4 KiB pages cover 256 KiB; 2 MiB pages cover 128 MiB.

This nominal reach is not a guarantee of hits. Associativity, page-size support and access patterns matter.

Linux represents a hierarchy named PGD, P4D, PUD, PMD and PTE. Levels unused by a supported architecture can be folded.

Therefore “Linux always performs five physical lookups” is not a valid conclusion from the five software names.

The address width, enabled paging mode, page size and leaf type determine the relevant hardware walk.

Use architecture documentation and the actual machine configuration when making implementation claims.

Historical architecture examples

IA-32 examples combine segmentation and paging. They help separate logical, linear and physical addresses.

Keep legacy Solaris and ARM examples identified as historical case studies. Do not present their page sizes or table levels as universal today.

Intel 64 / x86-64 should not be confused with the separate IA-64 architecture. The source terminology needs care.

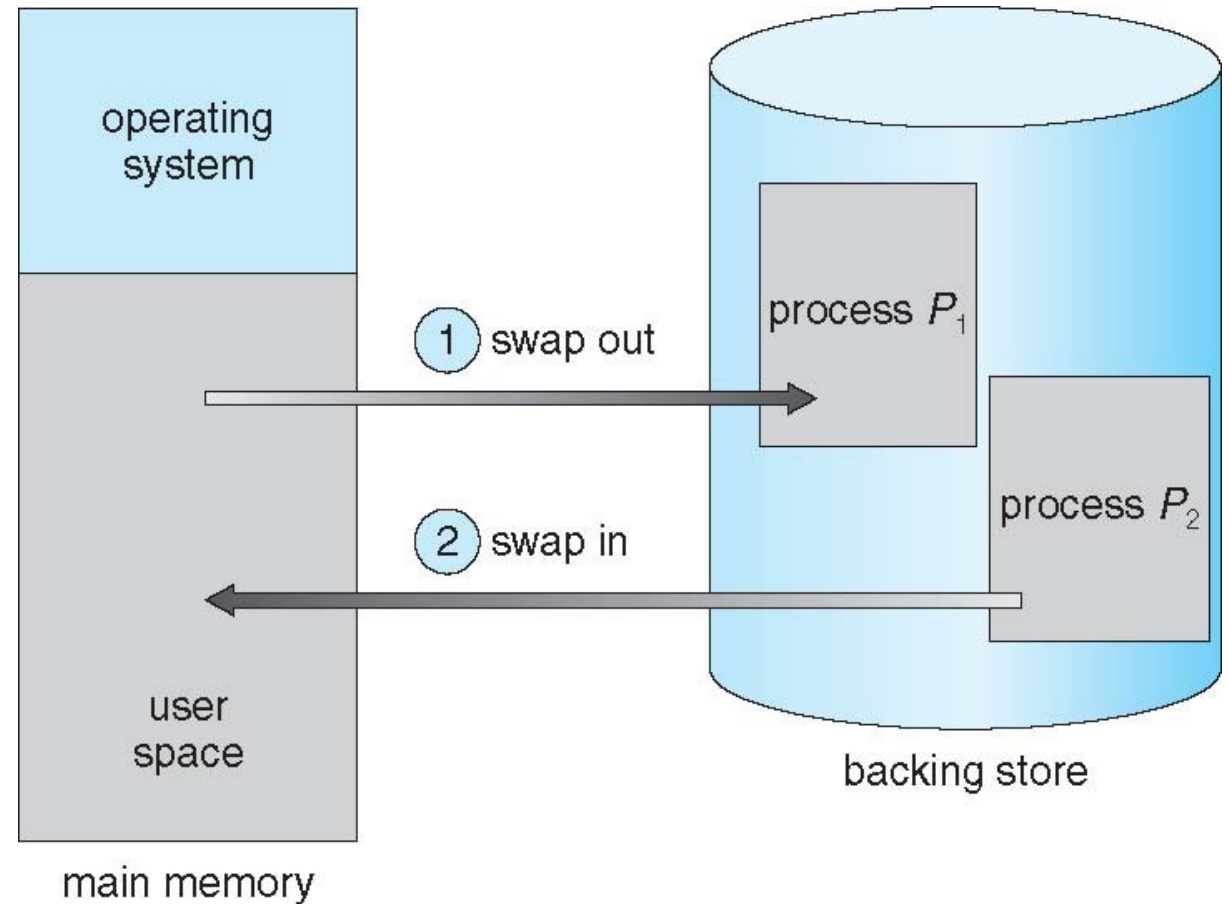
The foundations here are mapping, permissions, sparse tables and caching; architecture details illustrate those ideas.

Whole-process swapping

The historical model transfers a process image between memory and backing store. Address binding determines whether it can return at another location.

Moving an entire image has a cost proportional to transferred bytes, plus device and software overhead.

Do not equate this model with every contemporary use of the word swap.

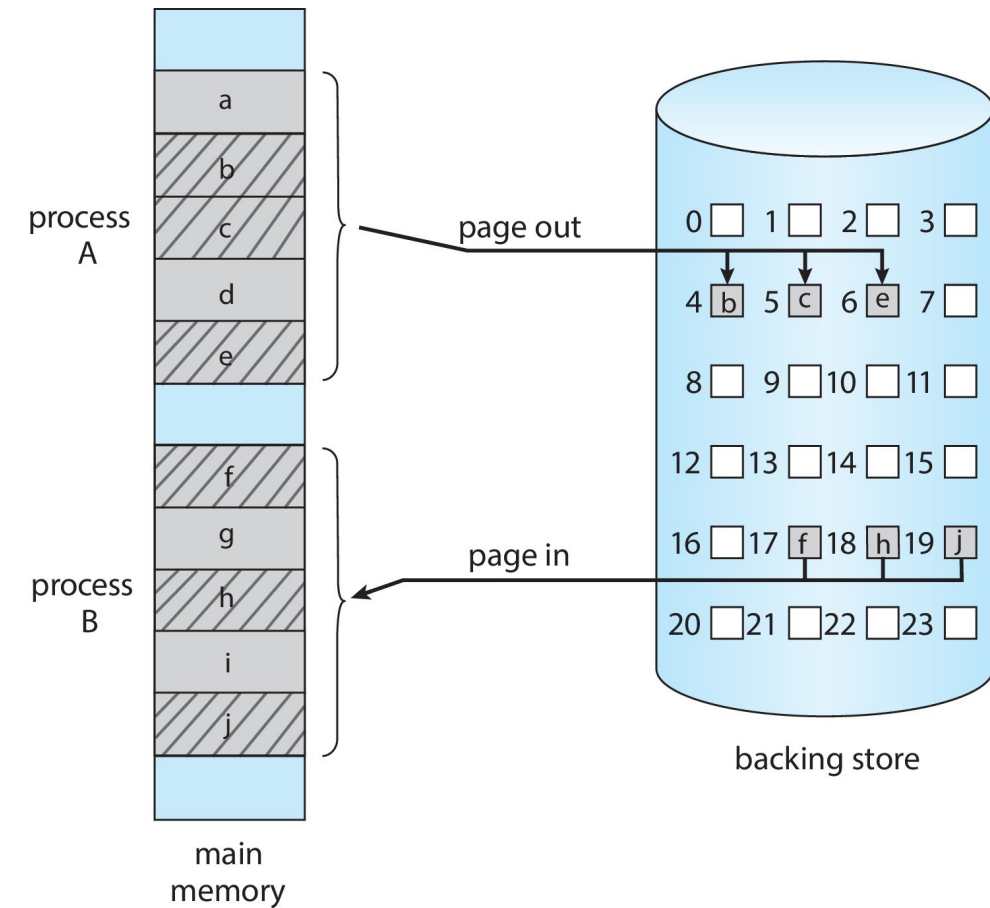


Page-level movement and Week 6

Page-level backing allows some virtual pages to be absent from physical memory while others remain resident.

Address translation identifies the requested page and its state. A fault handler may need to supply a frame before retrying.

Week 6 develops demand paging, replacement, copy-on-write, locality and thrashing. Today's tables and checks are the foundation.



Inspect a process on Ubuntu

Use the supplied `address-view.c` program to print its PID, page size and addresses of static, automatic and allocated objects.

It prints its own `/proc/self/maps` on Linux, so the addresses and map belong to the same process.

Predict which region contains each address, then inspect permissions and mapping boundaries.

This is optional practice. No new graded submission or deadline is introduced.

Compile and observe

Run twice and compare addresses and mapping ranges. Record the machine, compiler and page size.

```
cc -std=c11 -Wall -Wextra -Werror \
-O2 address-view.c -o address-view
./address-view
./address-view
uname -a
cc --version
```

Do not infer a physical frame number from a printed pointer or /proc maps.

Read /proc maps with the right question

A maps line reports a virtual address range and access permissions, with optional file information. The end address is exclusive.

r, w and x describe access permissions; p and s distinguish private and shared mapping types.

A mapping is not evidence that every page is resident or backed by a unique physical frame.

Avoid assuming every malloc allocation appears in a region labeled [heap]; allocator choices and allocation size can differ.

A reproducible translation simulator

translate.c uses the four-page table from the lecture, 4 KiB pages, a two-entry LRU TLB and explicit permissions.

It prints page number, offset, hit/miss and either the physical result or a named rejection.

Read 0x1234, then write it: a cached translation must still enforce read-only protection.

This is a model, not a tool for reading the host's physical page tables.

Prediction, observation and explanation

Before running, hand-compute 0x0FFF, 0x1000, 0x1234 and 0x2000 using the stated table.

Predict TLB contents after pages 0, 1, 0, 3, 1. Then compare every printed step with your trace.

Change one frame mapping in the source and rerun from an empty TLB. Explain which address bits change.

AI assistance is permitted. Verify arithmetic and challenge claims against the model and observed output.

Integrated review problem

Assume 4 KiB pages: A page 1 $\rightarrow$ frame 9 read-only; B page 7 $\rightarrow$ frame 9 read/write. Both mappings are resident.

1. Where does A's read at 0x1234 go? 2. Can A write there?

3. Where does B's write at 0x7234 go? 4. Does A missing in its TLB imply a disk read?

5. What additional reasoning is needed if B and another process update shared data concurrently?

1. A reads physical 0x9234. 2. A's write is rejected by its mapping permission.
3. B's permitted write reaches the same physical byte, 0x9234. Read-only for A does not make the frame immutable to everyone.
4. No disk read follows merely from a TLB miss; this example has a resident mapping.
5. Synchronization is still needed for shared updates. Mapping permissions and race freedom are different obligations.

What to explain without memorizing a diagram

Derive page number, offset and physical address from the stated sizes. Keep PTE addresses separate from data addresses.

Explain why isolation and sharing can coexist, and why a TLB hit does not bypass permissions.

Calculate table storage and access cost under explicit assumptions; identify where real hardware adds complexity.

Distinguish virtual-space validity, residency and cached translation. Carry that distinction into Week 6.

Sources and further study

Adapted from Silberschatz, Galvin & Gagne, Operating System Concepts, 10th edition, Chapter 9.
Original textbook figures retained with credit.

Selected segmentation figures and teaching examples draw on Mehmet Göktürk's historical BIL601 memory-management lecture.

Complementary reading: OSTEP, Paging: Introduction; Paging: Faster Translations; Paging: Smaller Tables.

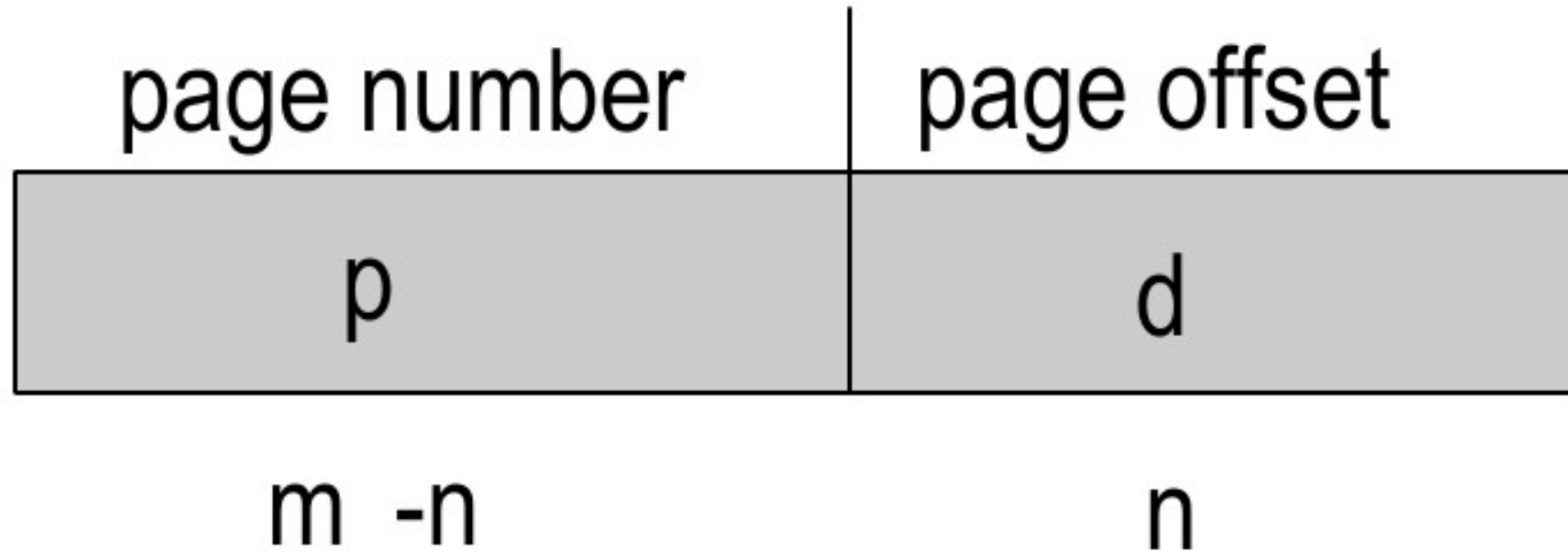
Linux evidence: kernel page-table documentation, `proc_pid_maps(5)` and `sysconf(3)`. Direct links are in the illustrated reading.

The following figures retain the remaining Chapter 9 illustrations for comparison and revision.

Read Intel and ARM diagrams as the historical models identified here. Their specific widths, page sizes and modes are not universal contemporary Linux properties.

Use the main lecture for worked calculations, then trace the corresponding fields and paths in these original figures.

Page-number and offset fields



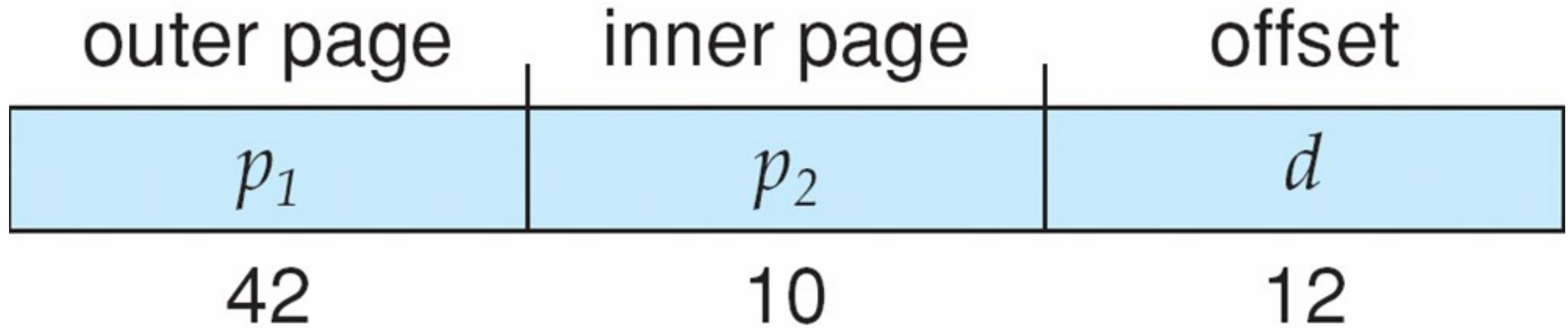
For a 2^m -byte address space and 2^n -byte pages, p has $m-n$ bits and d has n bits.

Associative translation lookup

Page #	Frame #

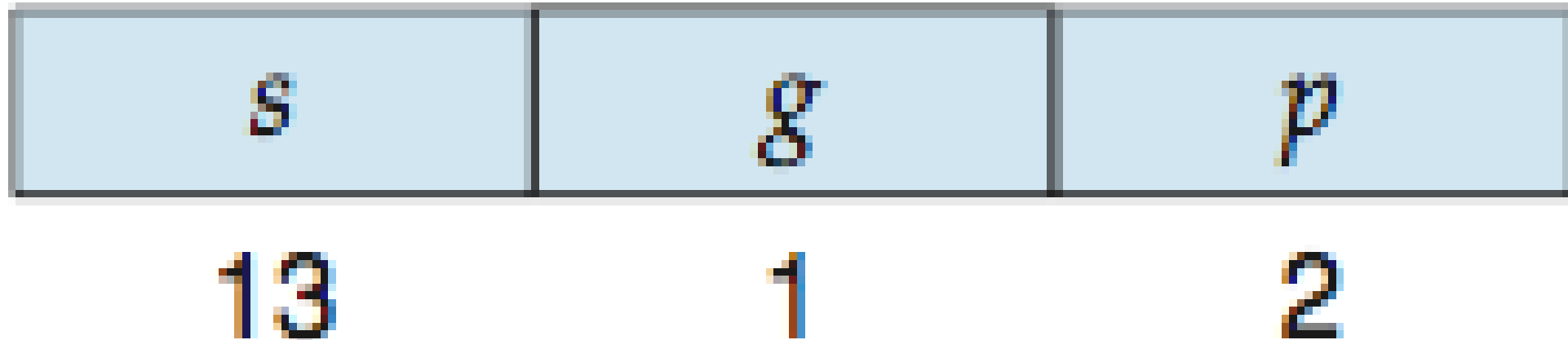
Search page numbers in parallel. A matching entry supplies a frame number; the offset passes through unchanged.

A two-level split for a 64-bit model



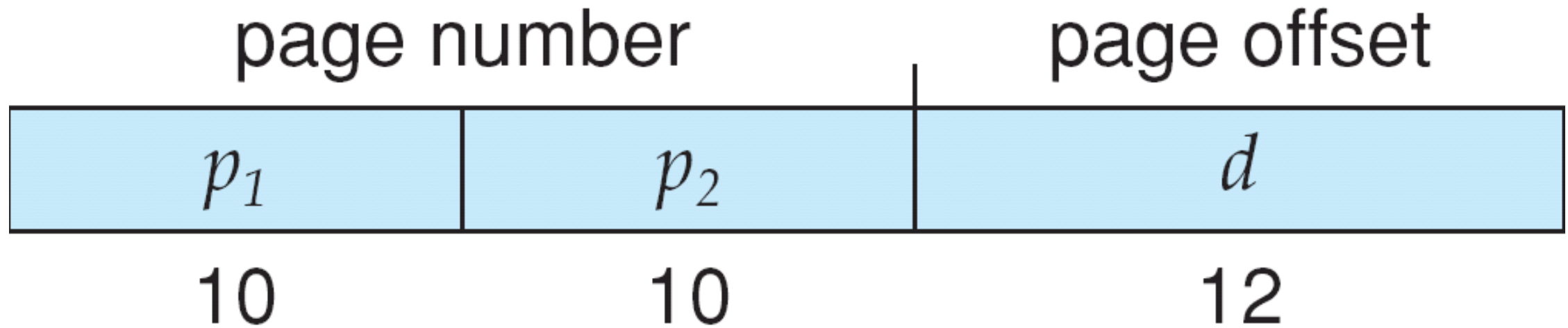
The 42 / 10 / 12 split leaves an impractically large outer table. Compare it with the three-level example in the main lecture.

IA-32 segment selector



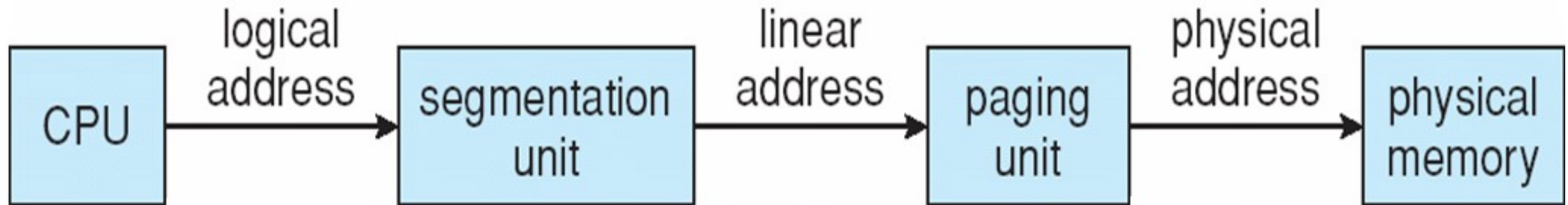
Historical protected-mode model: the selector has a 13-bit index, a table-selection bit and two privilege bits.

IA-32 page-address fields



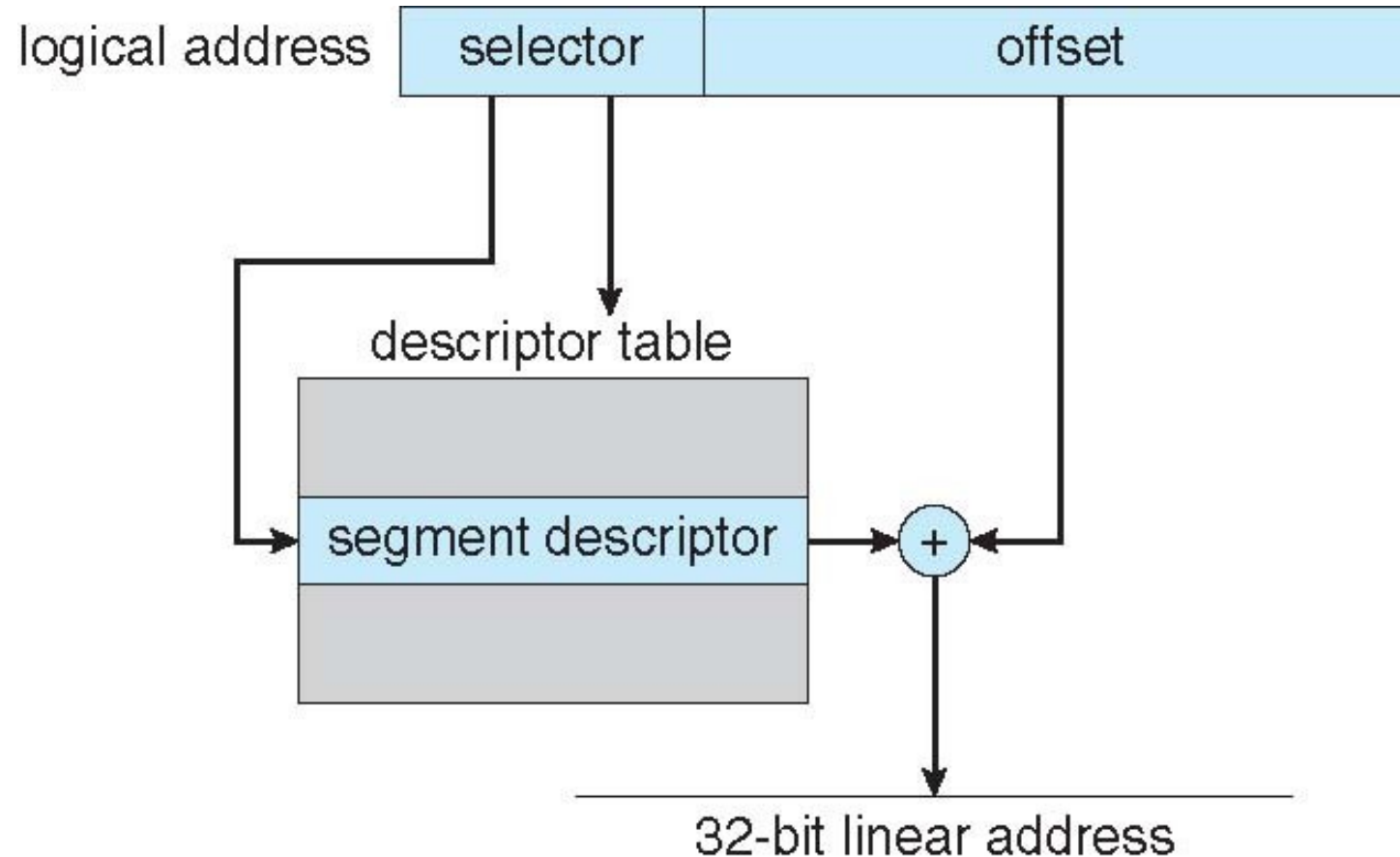
Historical two-level model with 4 KiB pages: two 10-bit indices and a 12-bit byte offset.

IA-32 translation stages



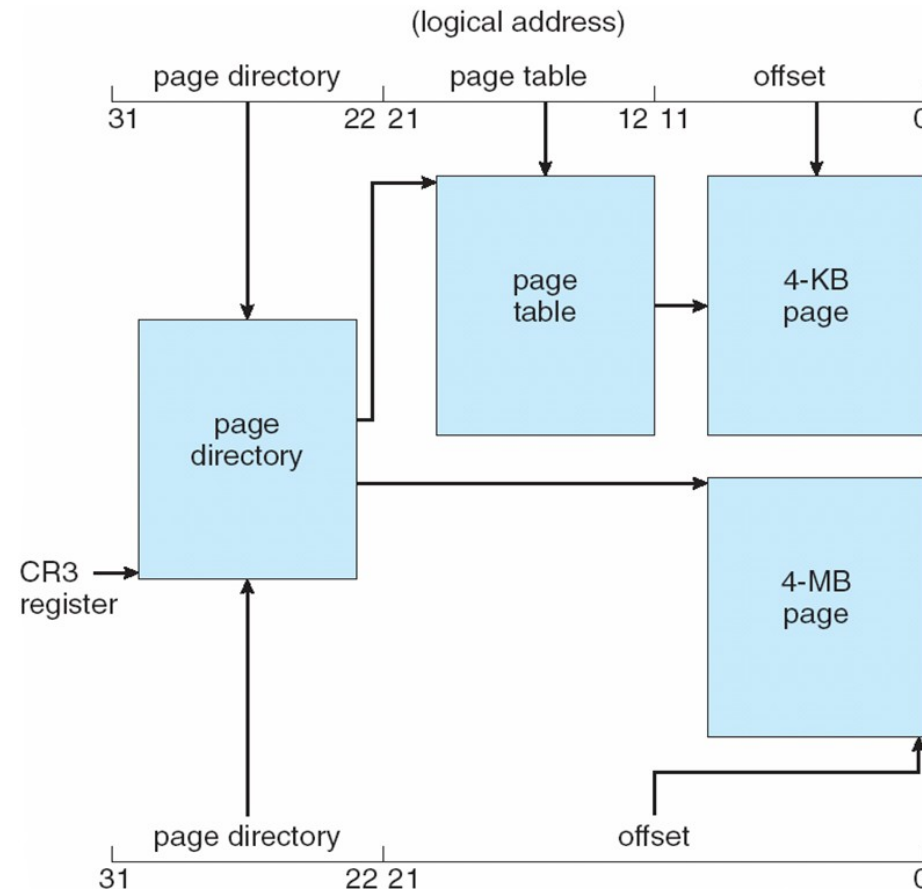
Segmentation produces a linear address. Paging then translates that linear address to a physical address.

IA-32 segmentation



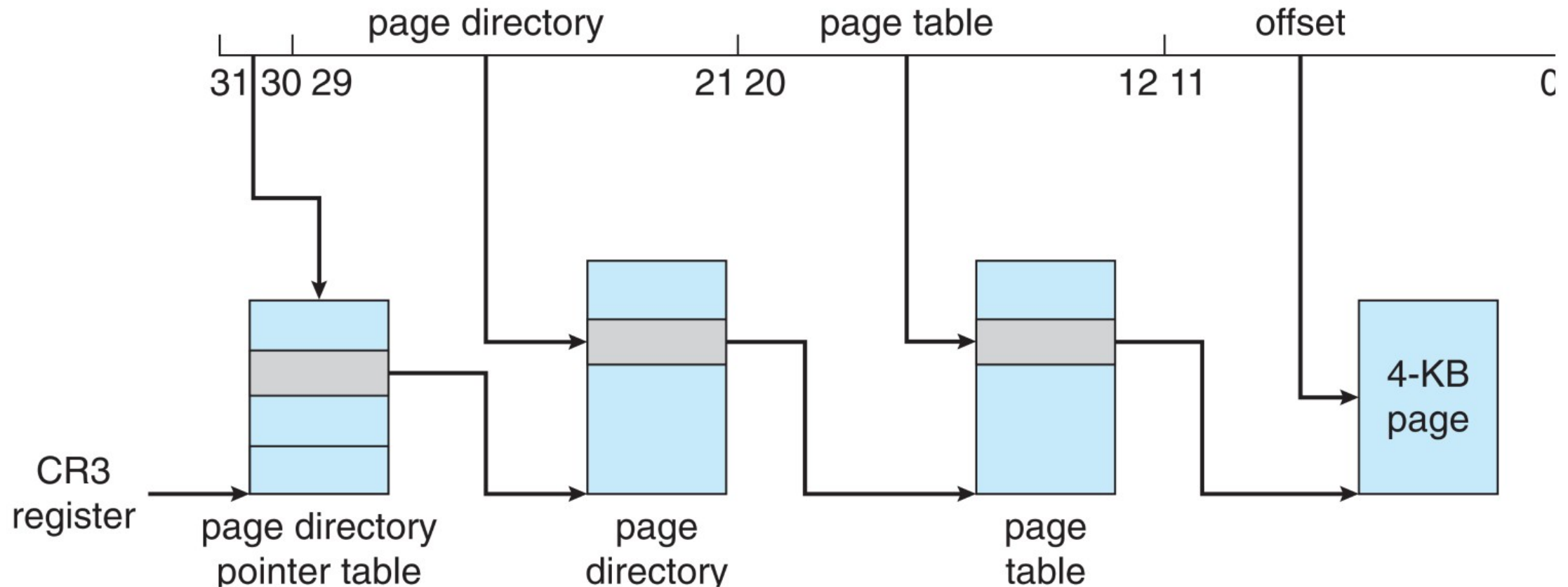
The selector identifies a descriptor. Its base and the offset form the linear address, subject to access and bounds checks.

IA-32 paging alternatives



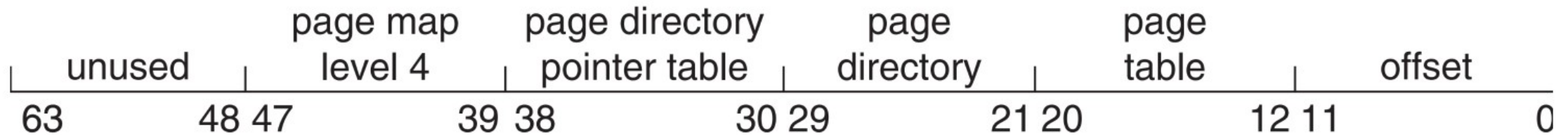
Historical diagram: a directory entry can lead to a page table or map a larger page directly. Follow each path separately.

IA-32 physical address extension



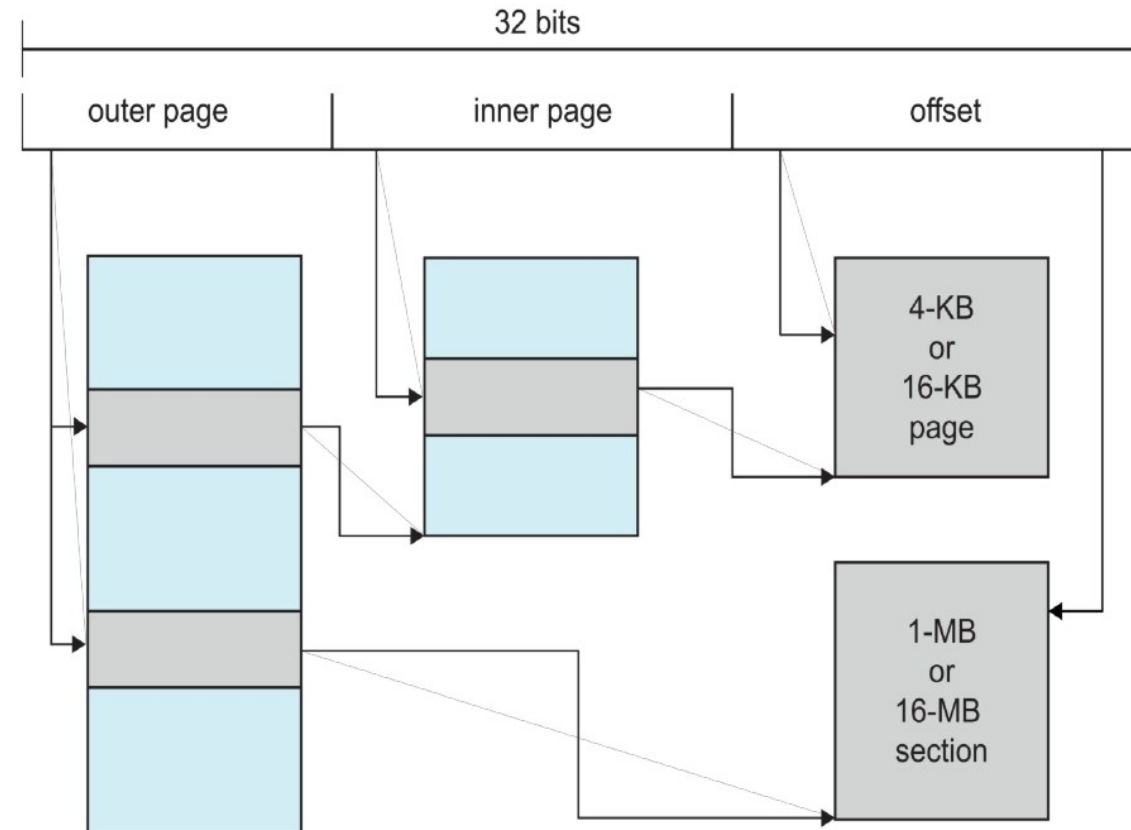
PAE adds a directory-pointer level and wider entries. Offset: bits 11–0. A process still uses a 32-bit linear address.

A four-level x86-64 model



Historical 48-bit virtual-address example. The upper bits must satisfy canonical-address rules; this is not a universal current limit.

Historical ARM page and section mappings



Compare a direct section mapping with a two-level page mapping. The page sizes shown belong to this historical ARM model.