Memory Management II

Operating Systems

Oleg Goldshmidt

ogoldshmidt@computer.org

Lecture 5

Contiguous Allocation

- both the OS (kernel) and user processes need to reside in memory
- the main memory is usually divided into 2 partitions — for the OS and for user processes
- do we place the OS in low or high memory?
  - the interrupt memory is often in low memory, OS is commonly there as well
  - assume OS is in low memory, the other case is not significantly different
- OS must be protected from the user processes
- user processes must be protected from each other

Single Partition Allocation

- add a limit register to the relocation register
- the relocation register contains the value of the lowest physical address accessible by the process
- the limit register contains the value of the highest logical address accessible by the process
- the dispatcher loads the relocation and limit registers as a part of a context switch
- `if (virt < limit) phys = reloc + virt; else goto segfault;`
- can be used to change the OS size dynamically
  - e.g., a driver and its buffers are not used — why keep them in memory?

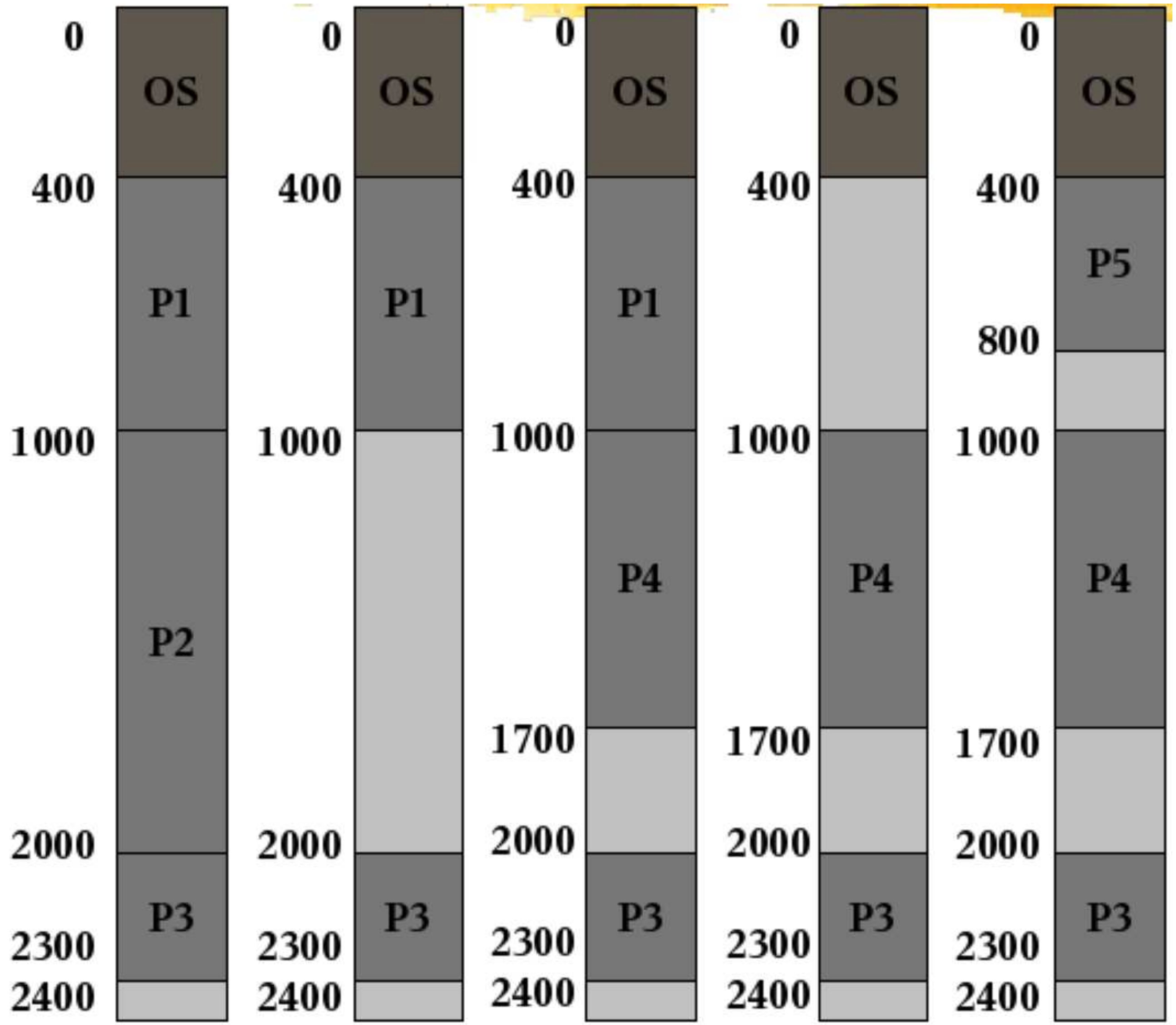
Multiple Partition Allocation

- how to allocate memory to various processes?
- multiple fixed size partitions (early IBM OS/360)
  - divide the memory into fixed partitions in advance, load a process from the input queue into a free partition
  - degree of multiprogramming is predefined
  - size of process is predefined
  - requires careful tuning dependent on workload
- generalization: multiple variable size partitions
  - OS keeps a table of allocated and free space
  - for each process we allocate just enough space

Long Term Scheduling

P1	600K	10
P2	1000K	5
P3	300K	20
P4	700K	8
P5	400K	15

- RR scheduling, $\Delta t = 1$
- $t = 14$: P_2 terminates
- $t = 28$: P_1 terminates



Allocation Algorithms I

- special case of dynamic resource allocation
- given a set of holes, satisfy a request of size n
- first-fit
  - allocate (a part of) the first hole that is big enough
  - start either at the beginning of the free list or where the previous search stopped
- best-fit
  - allocate the smallest hole that is big enough
  - must search the entire list (or keep it ordered)
  - produces the smallest leftover hole

Allocation Algorithms II

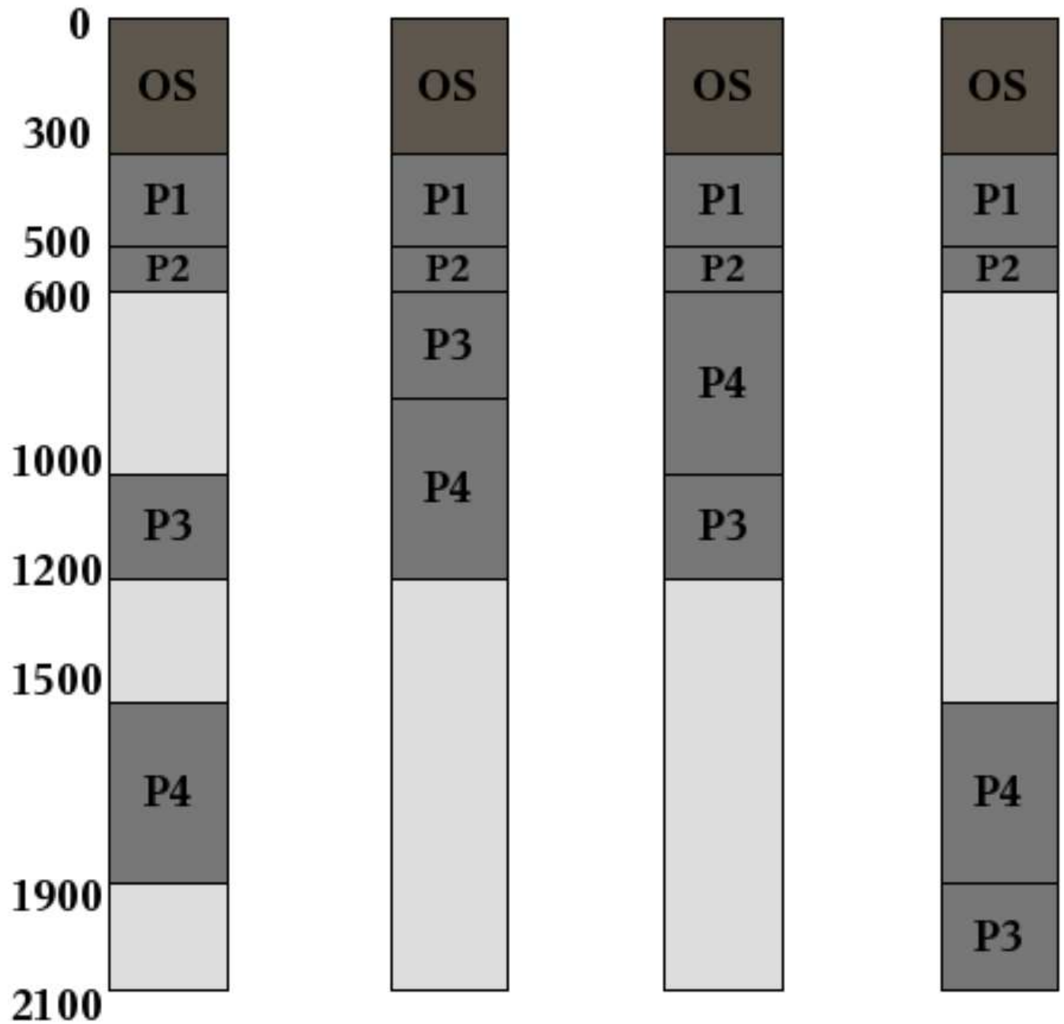
- worst-fit
  - allocate the largest hole
  - must search the entire list (or keep it ordered)
  - produces the largest leftover hole
- first-fit and best-fit usually yield better utilization
- best-fit does not win clearly on utilization (especially if scheduling is flexible)
- first-fit is the fastest

Fragmentation

- external fragmentation: enough memory exists to satisfy a request, but it is not contiguous
  - exists for any allocation scheme
  - exists whether we allocate the low or the high end of a hole
  - often reaches 50% (for first-fit) — “50 per cent rule”
- internal fragmentation: more memory is allocated to a process than is really needed
  - the overhead of keeping track of many very small holes is not justified — better allocate a slighly larger partition

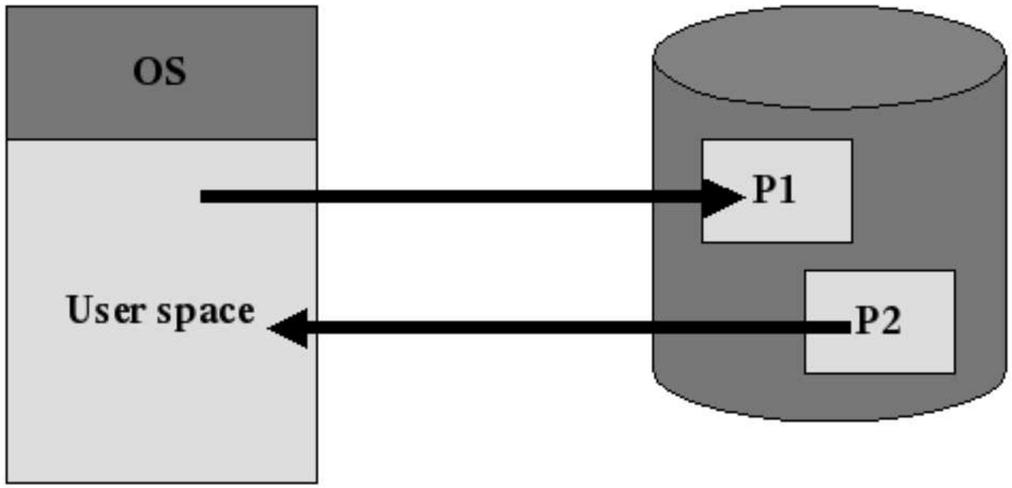
Compaction

- reshuffle memory to make a single large free hole
- only possible if relocation is dynamic and done at runtime
- move program and data, change the base register
- optimization is difficult



Swapping

- RR: swap out the process that has just finished its quantum
- priority scheduling: swap out low priority processes (roll-in, roll-out)
- for runtime binding the new memory space may be different
- may help with compaction



Swapping Tradeoffs I

- backing store — a (fast) disk
  - commonly a “swap partition” allocated at install time
  - may be a “swap file” (but extra head seeks)
  - can be attached (“mounted”) on demand
  - must be large enough — rule of thumb $2 \times RAM$
- much slower than RAM — high context switch overhead
  - the time quantum must be long enough
  - transfer time is proportional to the amount of memory used
  - useful to know how much memory the process is using (as opposed to how much it might use)
  - allocate and free memory dynamically

Swapping Tradeoffs II

- process being swapped must be completely idle
- we might want to swap out a process waiting for I/O
  - but what if I/O may access the user's buffers asynchronously? *(this will be discussed later in the course)*
  - swap in a different process — I/O might corrupt its memory
- solutions to the I/O problem
  - never swap blocked processes
  - only use OS buffers (incurs extra copy)

Freeing Memory

- modern OS do not actually release unused memory until another process makes a request
- typical memory usage reported by the OS when probed is close to 100%

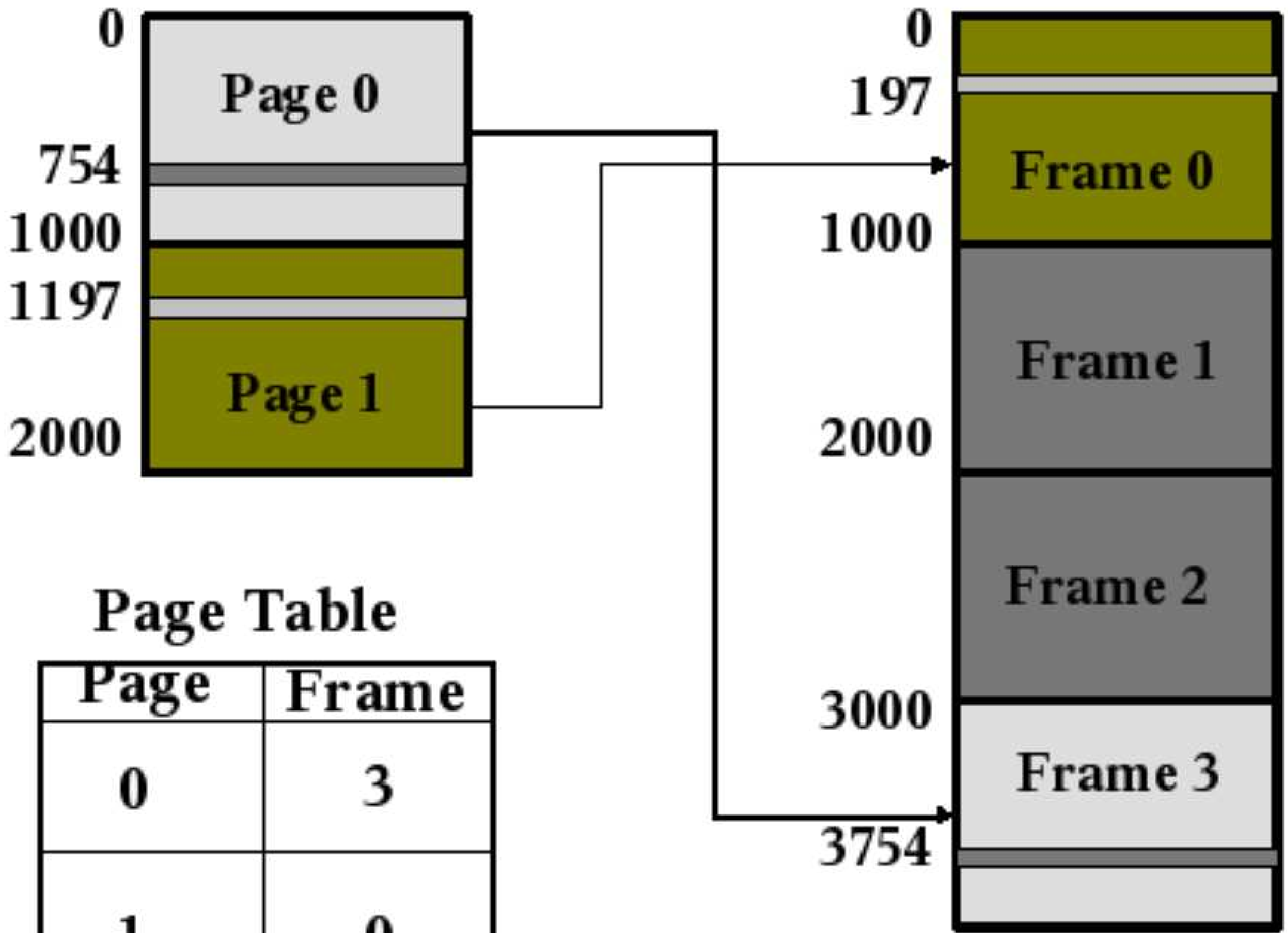
```
# free
                total        used        free
Mem:           511356      503920        7436
----------------------------------------------
               shared     buffers      cached
                    0       12032      228328
----------------------------------------------
-/+ buffers/cache:         263560      247796
Swap:         1050832      144624      906208
```

(output format modified to fit the slide)

Paging

- contiguous memory allocation suffers from fragmentation
- solution: allow the logical address space of a process be non-contiguous
- allocate memory in (relatively small) chunks — pages
- physical memory is divided into fixed-sized blocks — frames
- logical memory consists of blocks of the same size — pages
- pages are loaded into available frames from the backing store (also divided into chunks of the same size)
- any page can be loaded into any frame

Paging: Basics



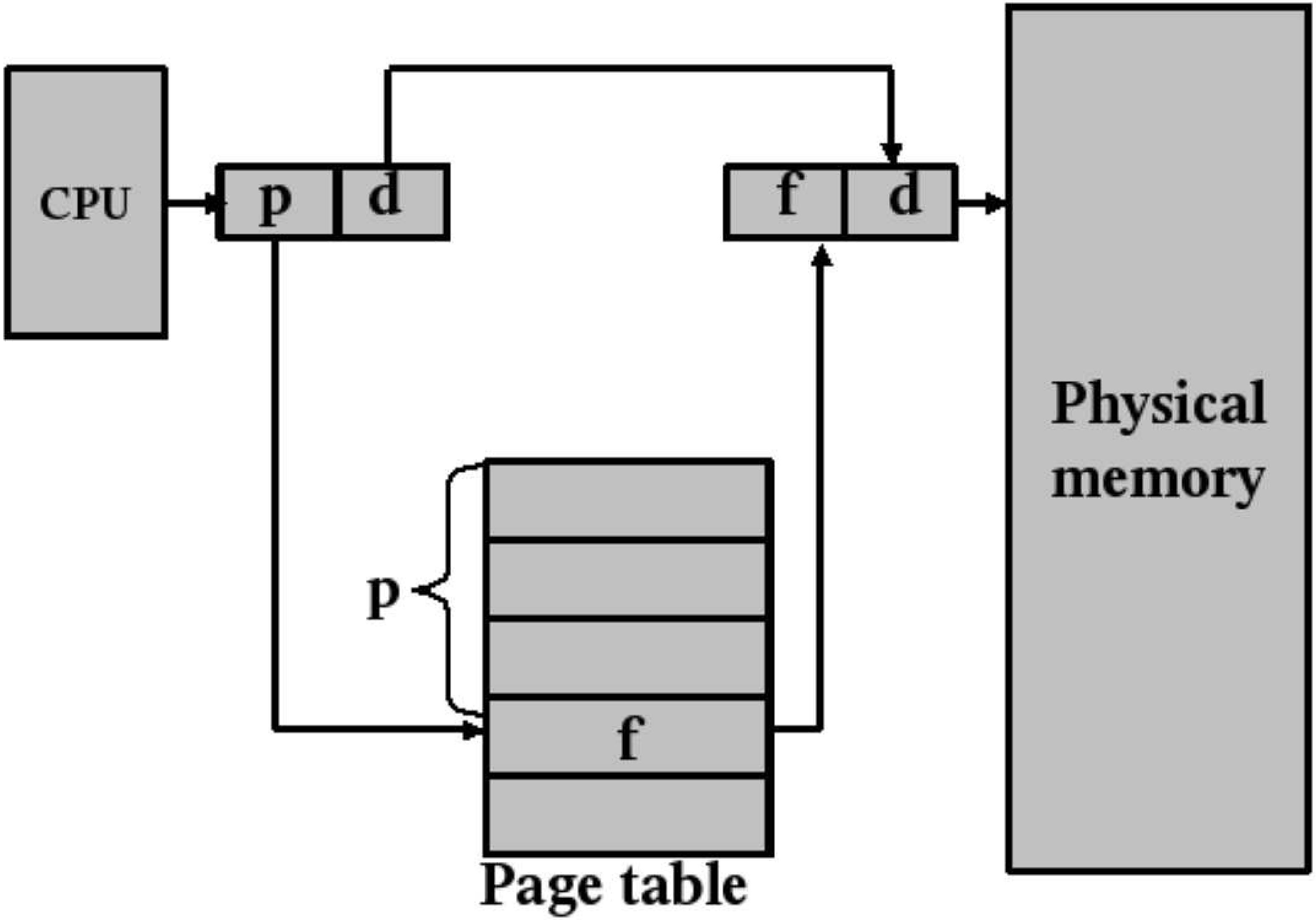
Page Table

Page	Frame
0	3
1	0

Page Tables

- logical addresses are divided into page number and page offset
- frame size (and page size) is determined by hardware — normally a power of 2 of addressing units (bytes or words)
- e.g., page size is 2^n, logical address space size is 2^m
  - $m - n$ high order bits of the logical address designate the page number
  - n low order bits form the page offset
- page table contains the base address of each page in physical memory
  - `phys = base(pagenum) + offset`

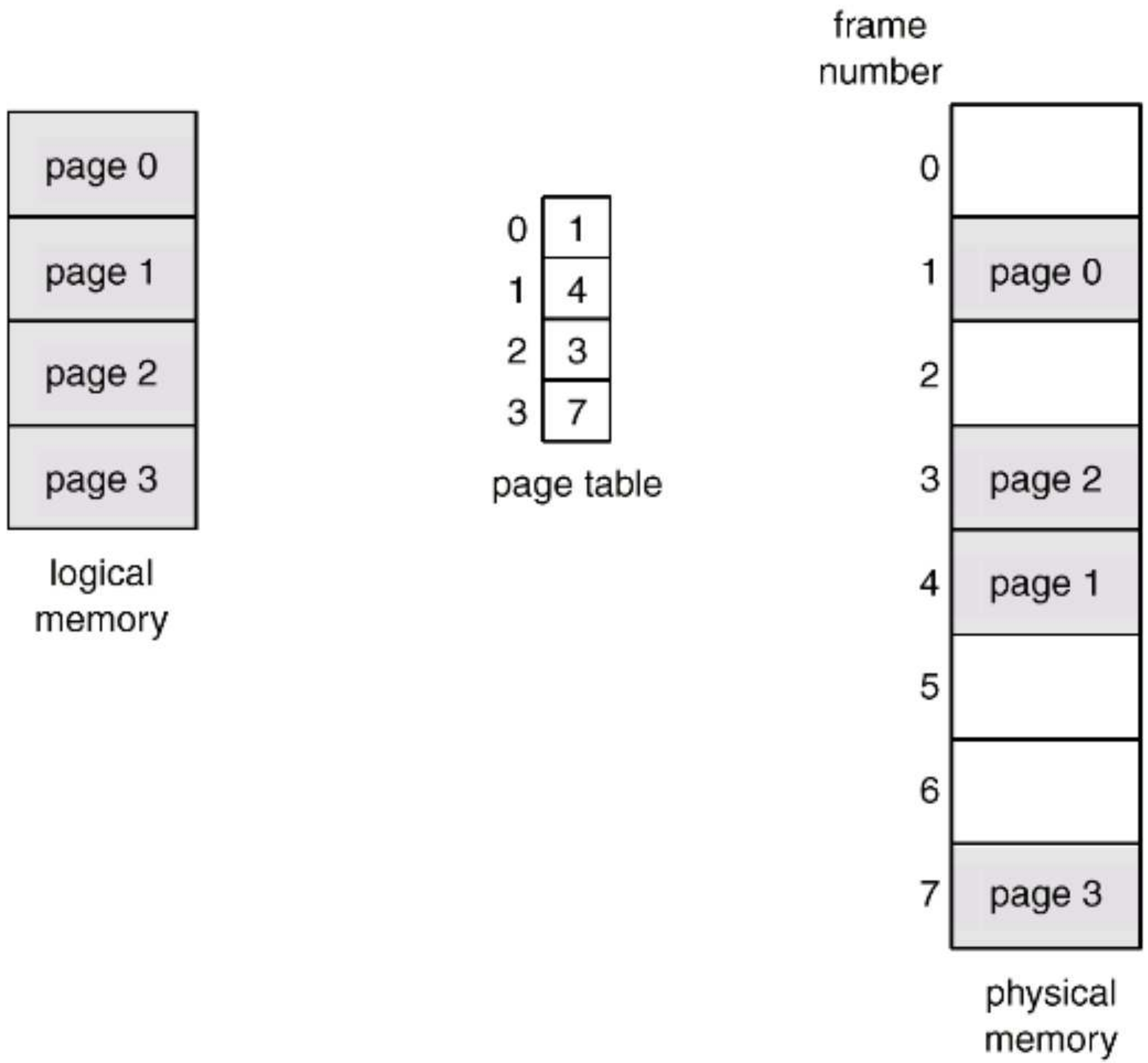
Paging: Hardware Support



Paging and Fragmentation

- paging is a form of dynamic relocation
  - page table is essentially a table of relocation registers — one per frame
- no external fragmentation
- but we still have internal fragmentation because we allocate a page even if the process needs less (usually the last page)
- on average, 1/2 page per process is wasted
- reducing page size may help, but overhead increases

Paging: Example



Transparent Memory Management

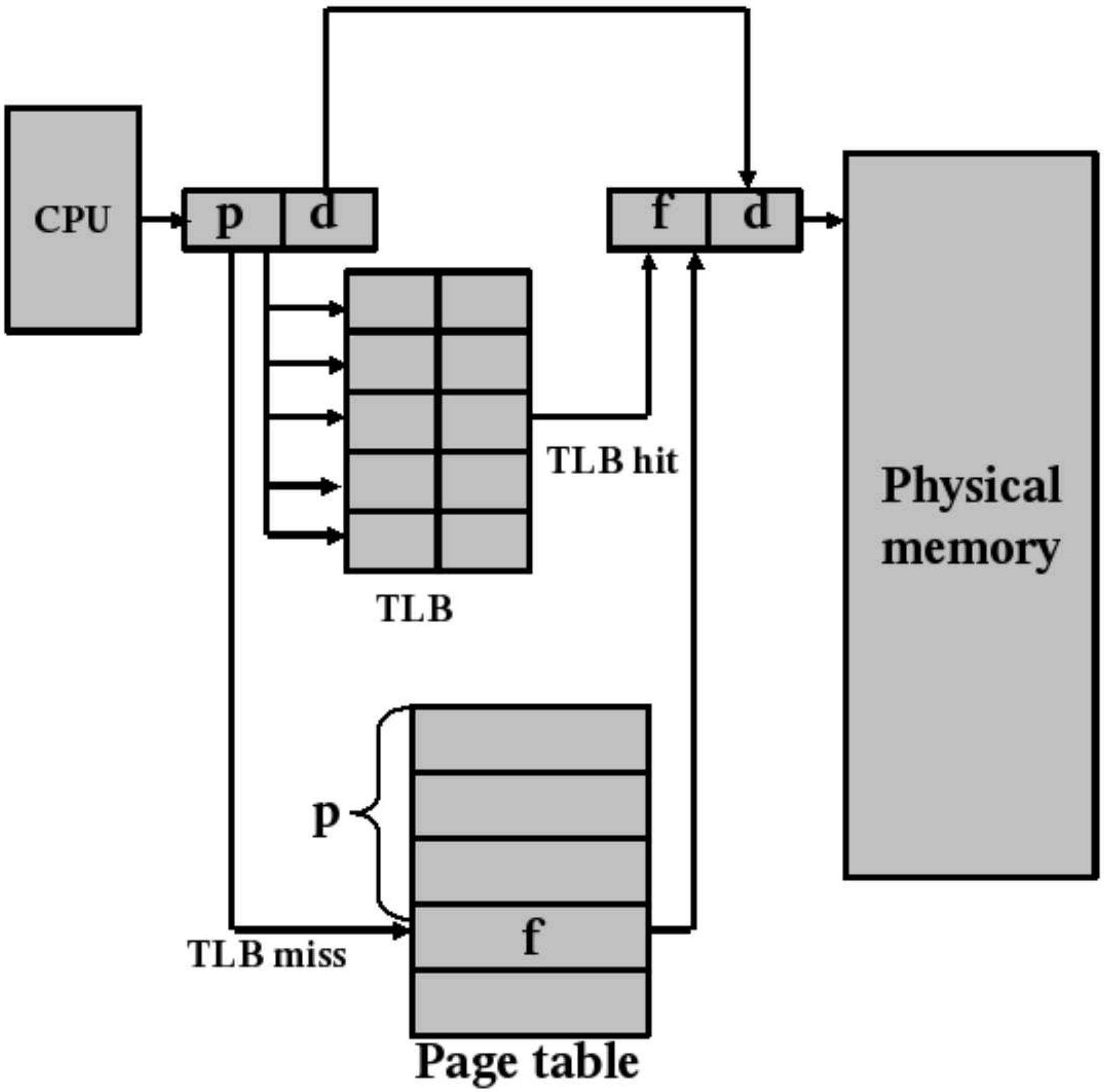
- user sees contiguous logical memory
- the mappings are hidden
- user can only access memory that is in the page table of the process
- frame table: what frames are allocated to which page of which process?
- OS maintains the mappings per process
  - paging increases the context switch overhead

Paging: Implementation

- a set of dedicated relocation registers
  - fast
  - feasible only when the page table is small
  - PDP-11: 16-bit addressing, 8K pages — 8 base registers
- keep page table in main memory, its address in a register
  - reload the page table base register (PTBR) only during context switch
  - two memory access for each request (`pte` and offset)
  - very inefficient!

Translation Lookaside Buffer (TLB)

- paging cache
  - fast
  - expensive
  - small
- associative registers — parallel search
- flushed during context switch
- “tagging” may allow flushing only some entries



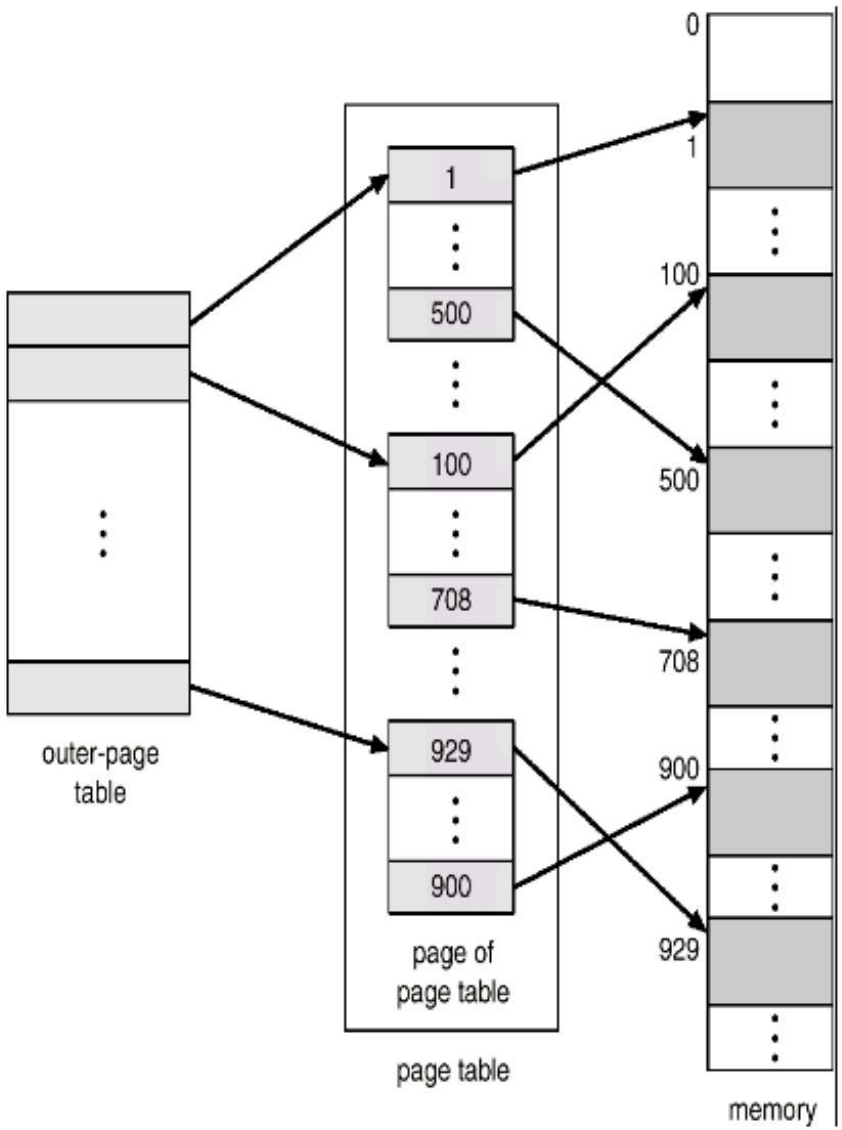
Multilevel Page Tables

- very large logical address spaces
  - e.g., 32-bit architecture with $4\,\mathrm{K}$ pages
  - $2^{32}/2^{12} = 2^{20} > 1,000,000$ page table entries
  - with 4 bytes per entry need $4\,\mathrm{M}$ for the page table
- do not allocate contiguosly: paged page table

p1	p2	d
10	10	12

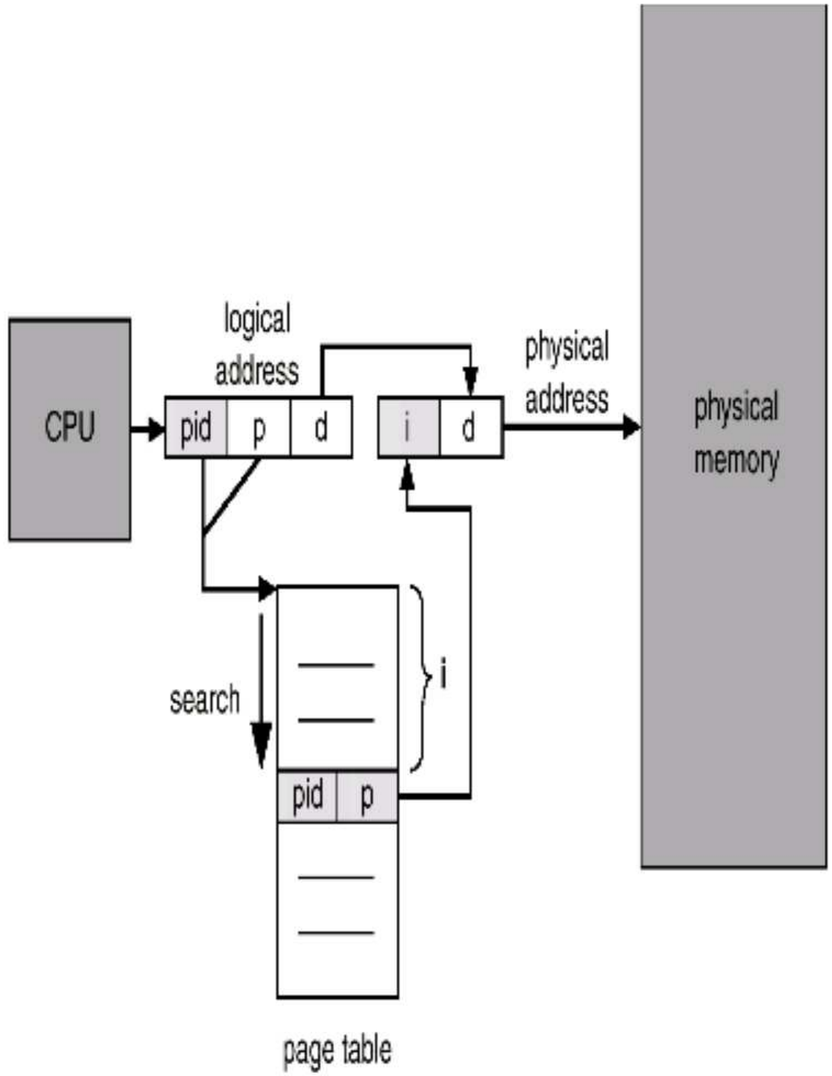
- for 64-bit architectures 2 levels are not enough
- n-level page tables — n memory accesses per request
  - a TLB with a high hit rate is essential!

Two-Level Page Table



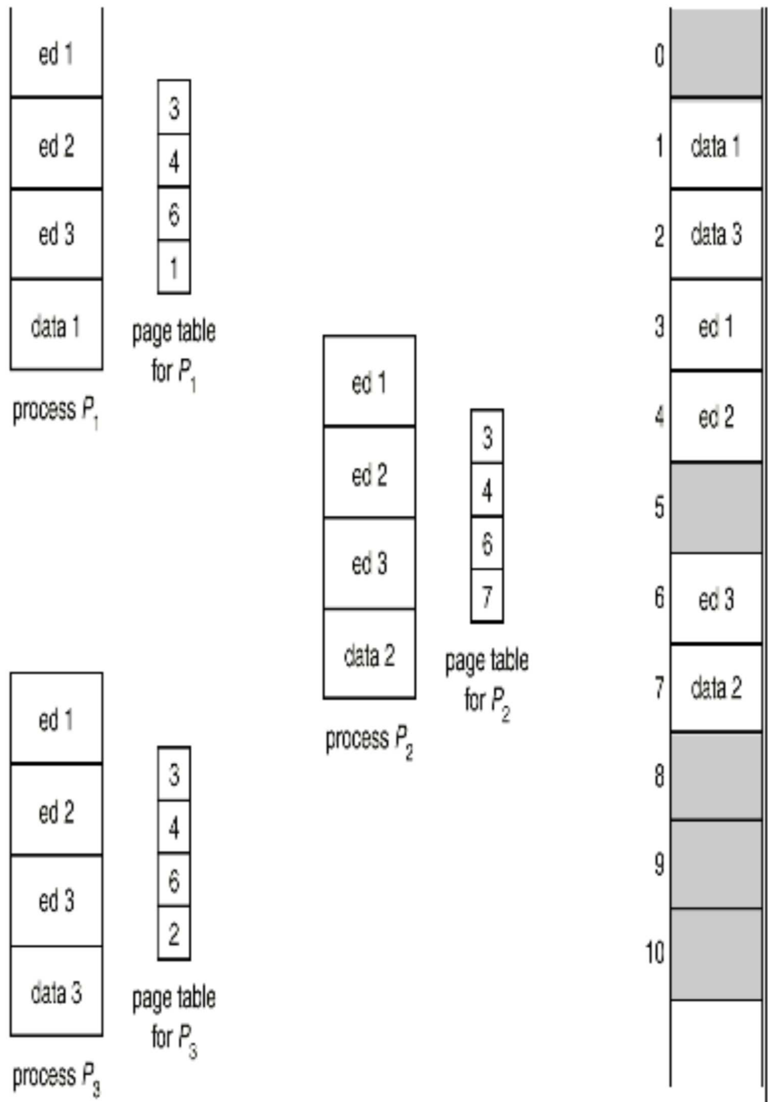
Inverted Page Table

- $4\,\mathrm{M}$ per process is too much
- map each frame to virtual address and `pid`
- logical address: $\langle pid, page, offset\rangle$
- slow search (sorted by physical, lookup by virtual)
- hashing adds a memory access
- cacheing helps



Shared Pages

- reentrant code — never changes during execution, can be shared
- only one copy in physical memory, mapped into different processes' virtual memory
- code must be correct, OS must enforce read-only
- systems with inverted page tables have a problem: more than one virtual address per physical address

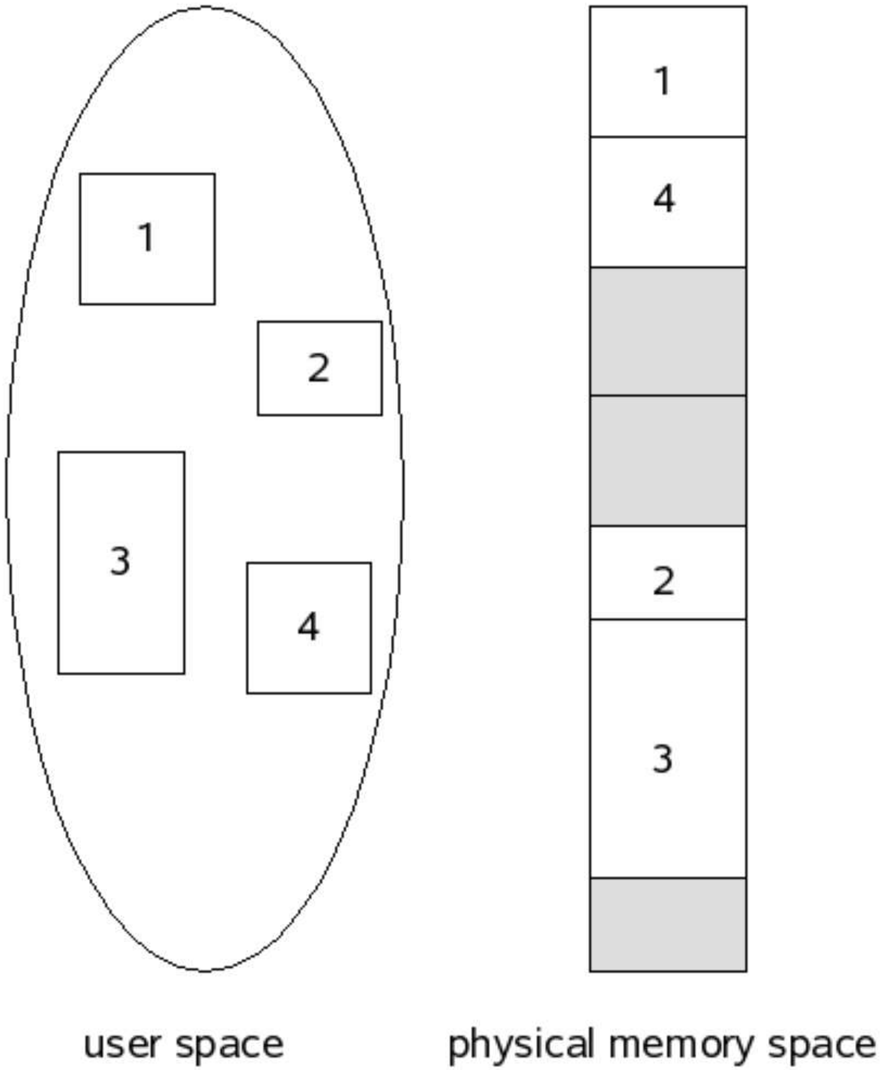


Segmentation

- paging separates the user's view of memory from reality
- segmentation supports the user's view of memory
- users do not see memory as a linear array of bytes, but rather as a collection of different functional segments:
  - main program
  - functions
  - stack
  - symbol table
  - global variables
  - etc.
- addressing: segment ID and offset
  - compare with paging: single address that is interpreted by the OS

Segmentation: Software Support

- user writes code using the logical structure of the language and the program environment
- compiler automatically divides the object code into segments reflecting the structure of the original program
- loader will assign segment numbers

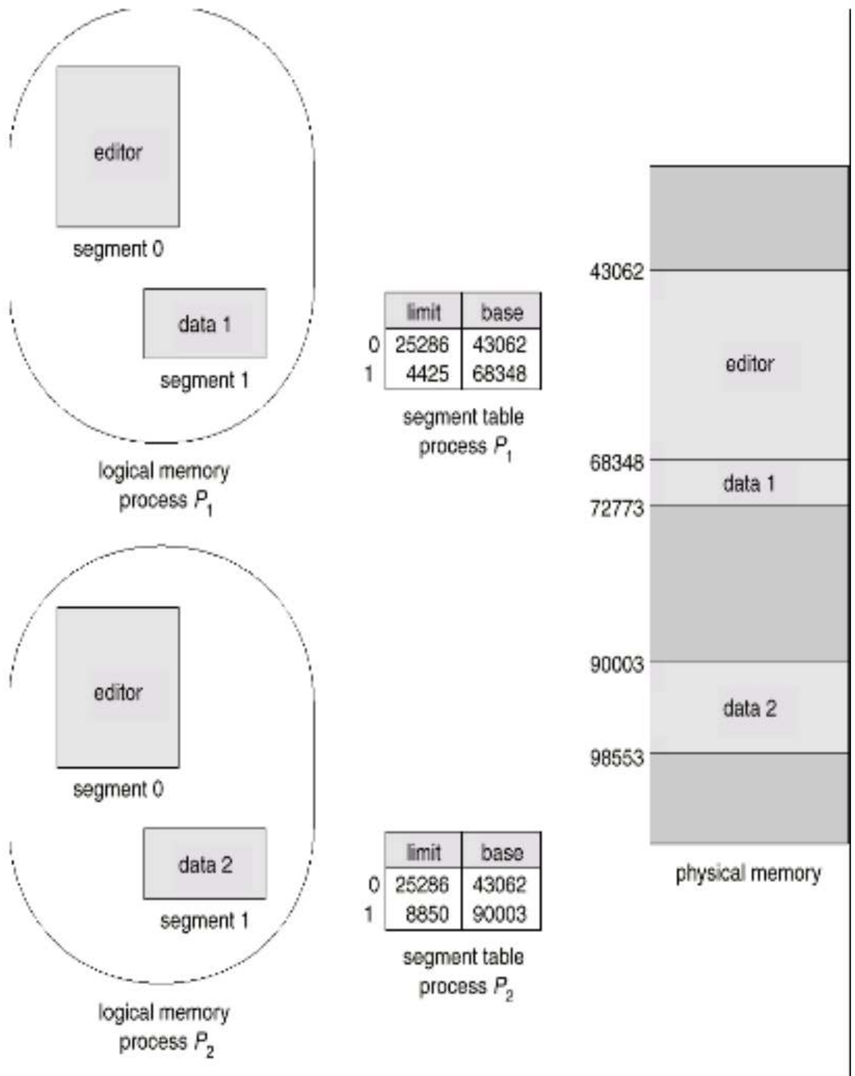


Segmentation: Hardware Support

- segment table maps segment number and offset to a linear physical address
- each table entry has a segment base and a segment limit
- if offset is larger than limit we trap
- basically an array of base-limit register pairs
- implementation
  - fast registers for small number of segments
  - in memory for large number of segments, `STBR` (base) and `STLR` (length) registers
  - multiple memory accesses and caching similar to paging

Shared Segments

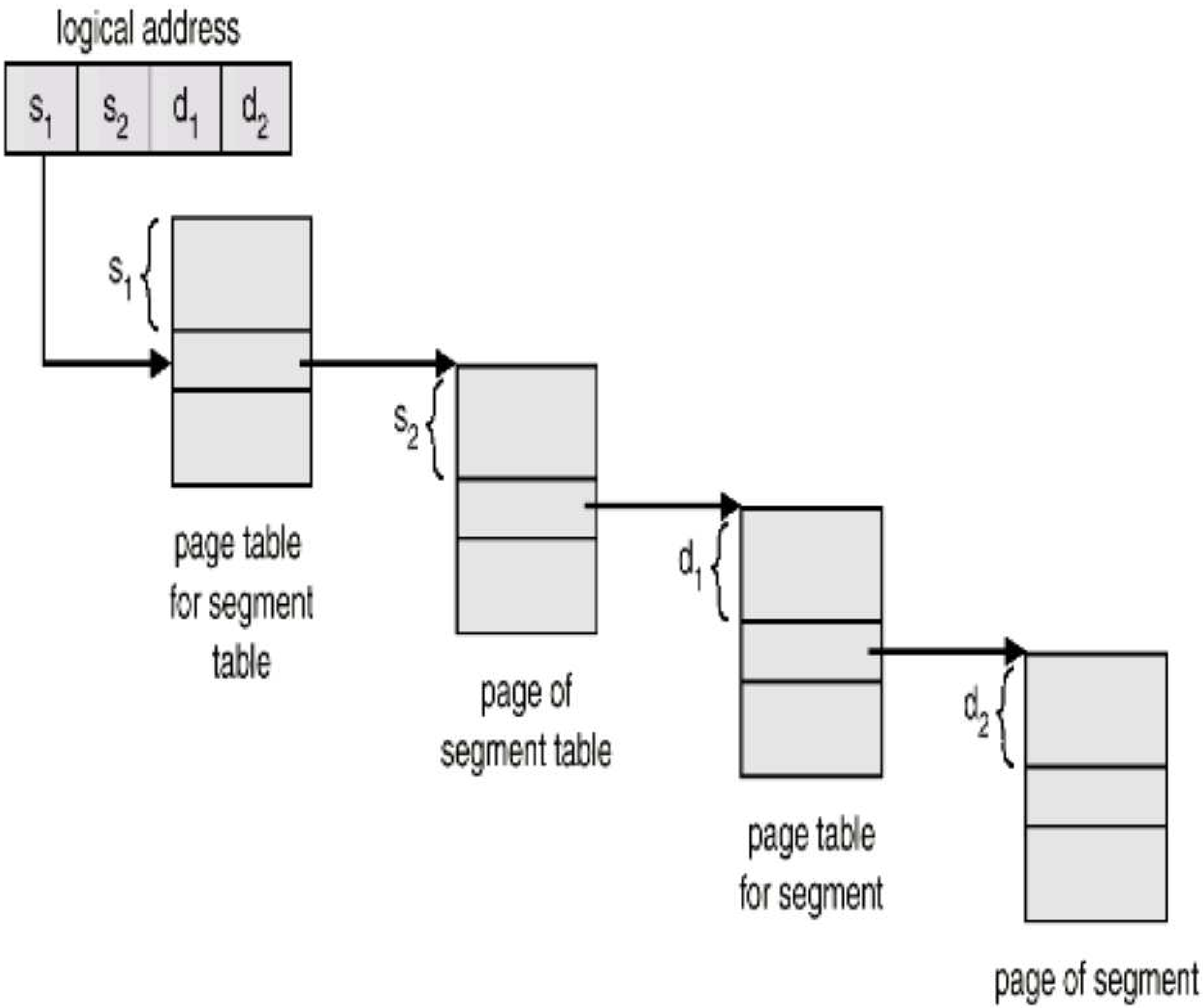
- segmentation is related to usage patterns, semantics
- protection is simplified
- parts of memory can be shared
- code segments refer to themselves: `jmp ADDR`
- shared code segments must have the same number, or use relative addressing w.r.t. program counter or segment number register



Fragmentation Revisited

- long-term scheduler must allocate memory for all the segments of the process
- while pages are of fixed size, segments are of variable size — similar to the variable-sized partition scheme (best-fit, first-fit)
- external fragmentation is possible, depending on average partition size
  - segment per process — variable-sized partitioning
  - segment per byte — no fragmentation but 100% overhead
  - small fixed-sized segments — paging

Segmentation and Paging



Intel i386 Memory Management

- 2 partitions per process
- $8\,\mathrm{K}$ segments
- LDT and GDT
- 6 segment registers
- 16-bit selector
  - 13 bits for segment, 1 bit for global/local, 2 bits for protection
- 32-bit address, 2-level page table
- swapping: 1 invalid bit, 31 bits for disk location