Announcements

- Assignment 4 is online for preview only; not officially handed out.
- Assignment 2: Hand out Wed, due Mon.
- How did you do on your GREs?
- Any assignment 1's?

Chapter 9: Memory Management

- Linking, Loading.
- Address Binding.
- Contiguous Allocation.
- Paging.
- Segmentation.
- Segmentation with Paging.
- Swapping.

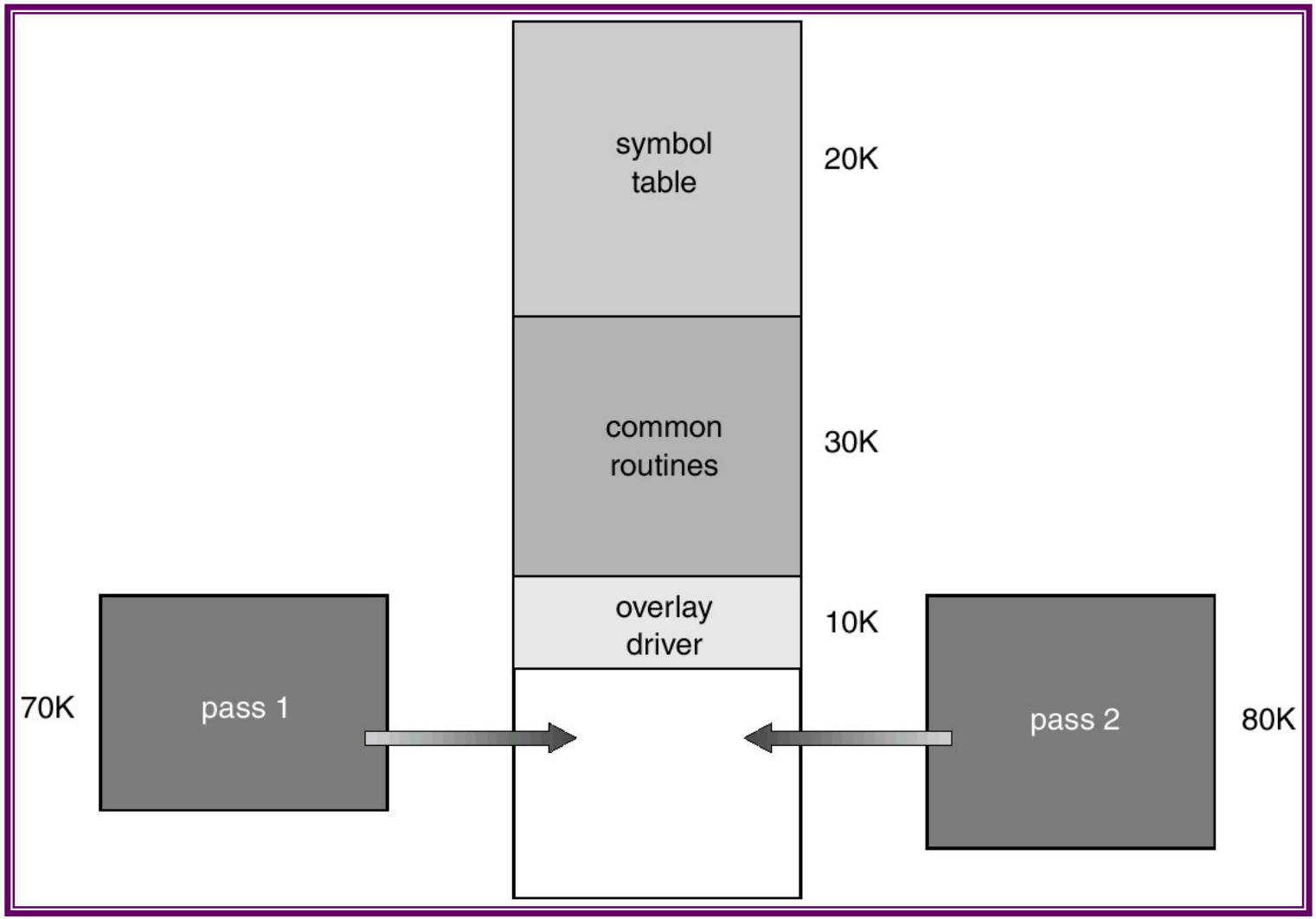
Linking, Loading, Binding

- Program must be brought into memory and placed within a process for OS to run it.
- Link: find & merge the parts.
  - Each part: routines it exports, routines it needs.
  - Linking resolves external references.
  - Static: before code runs.
  - Dynamic: after code runs.
- Load: bring in program part from disk into memory.
  - “Static”: completed before code runs.
  - Dynamic: additional parts brought in while process runs.
- Address binding: translate symbolic memory references into numbers.
- Link, load, binding: all must happen before a program part runs.
- Same concepts for JVM: classpath, class loader, etc.

Dynamic Loading/Overlays

- OS provides system calls to load. Program uses them for explicit management of process memory.
- Program doesn't load routine until it is called. Linking statically completed earlier; parts are known.
- Better memory utilization; unused routines never loaded.
- Useful when large amounts of code are needed to handle infrequently occurring cases.
- Once loaded:
  - Dynamic loading: it stays.
  - Overlays: user can unload and replace with other code.
    - Necessary if not enough RAM to run program.
    - Common for programs with locality (next slide).
- Used in early PC OSs, still used in simple OSs (game engine loading next part of game).

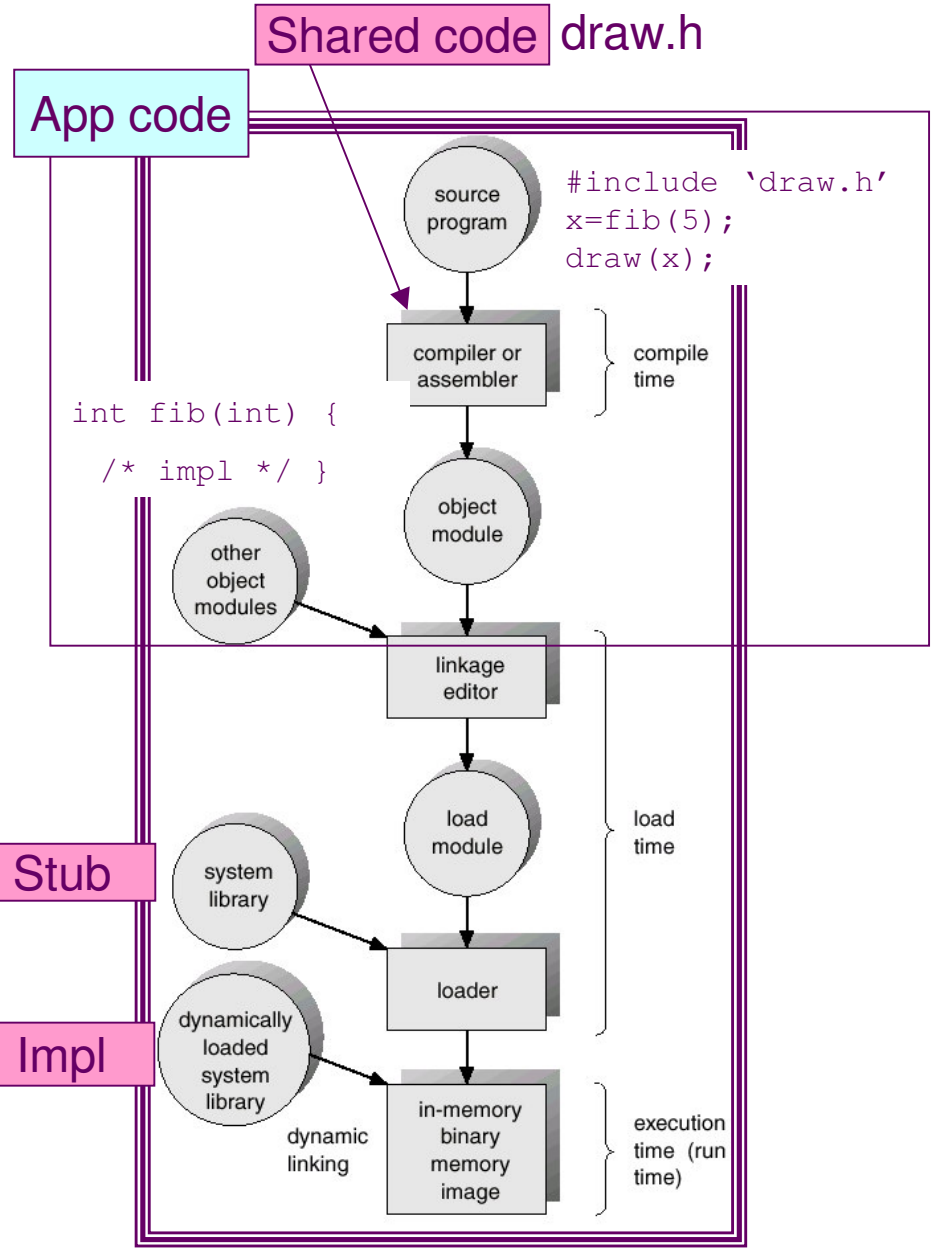
Overlay Example



Dynamic Linking

- OS handles linking and loading dynamically.
- Small piece of code (*stub*) locates the appropriate library routine when called:
  - If already loaded in memory by other process, use it from that part of RAM.
  - Else (dynamically) load into memory.
- Stub replaces itself with the address of the routine, so that calls to the stub will call the real routine. (Self-modifying code or indirection table.)
- Dynamic linking is particularly useful for libraries shared across many programs.
- Windows .DLL, Unix .so.

Dynamic Linking Example



libdraw.a

```
void draw(int x) {
 static void *drawImpl=null;
 if (drawImpl==null)
  /* find and load */
 (*drawImpl)(x); }
```

```
void draw(int x) {
 /* draw x on screen */ }
```

libdraw.so

Address Binding

- OS must *bind* all program addresses before running it. Classic:
  - Source:

```
        /* Initialization */
label: /* Loop body */
        jmp    label
```

  - Compiler creates relocateable address:
    - Assumes program first instruction has address 0.
    - Assume "Initialization" section can be represented using 14 bytes of instructions.
    - Then "jmp label" becomes "LOAD PC, 14".

Program Counter

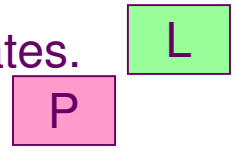
  - Loader *binds* (i.e. computes physical addresses):
    - Assume program stored in RAM at address 100.
    - Then "LOAD PC, 14" becomes "LOAD PC, 114".
- Compiler/Loader map from one *address space* to another. Address space is range of addresses allocated to a process.

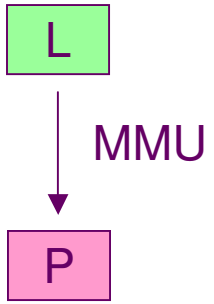
Timing of Address Binding

- **Compile time**:
  - Memory location known a priori.
  - Code with physical addresses can be generated.
  - Must recompile code if starting address changes.
  - MS-DOS .COM executables.

- **Load time**:
  - Compiler generates *relocatable* code (see previous slide).
  - Loader binds to RAM address before running.
  - Process stays in same part of RAM until it terminates.

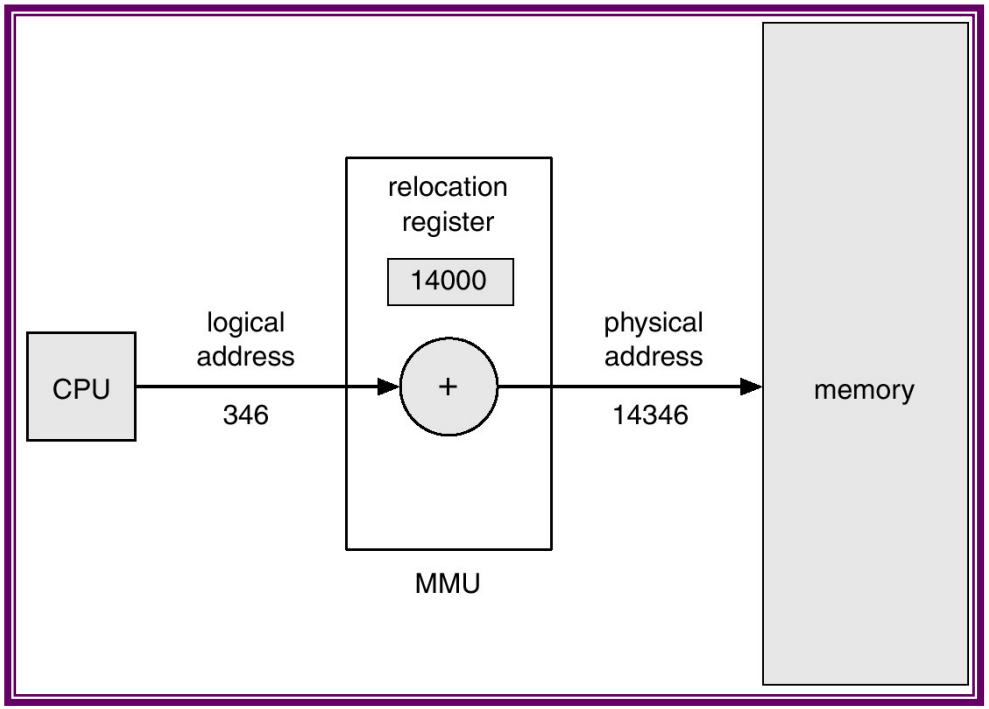
- **Execution time**:
  - Modern technique. Need move processes around physical memory to optimize its use. Think bin-packing.
  - Binding (mapping) done when process issues instruction.
  - Need hardware support for address map.
  - Logical/virtual address: what the CPU generates.
  - Physical address: RAM address accessed.





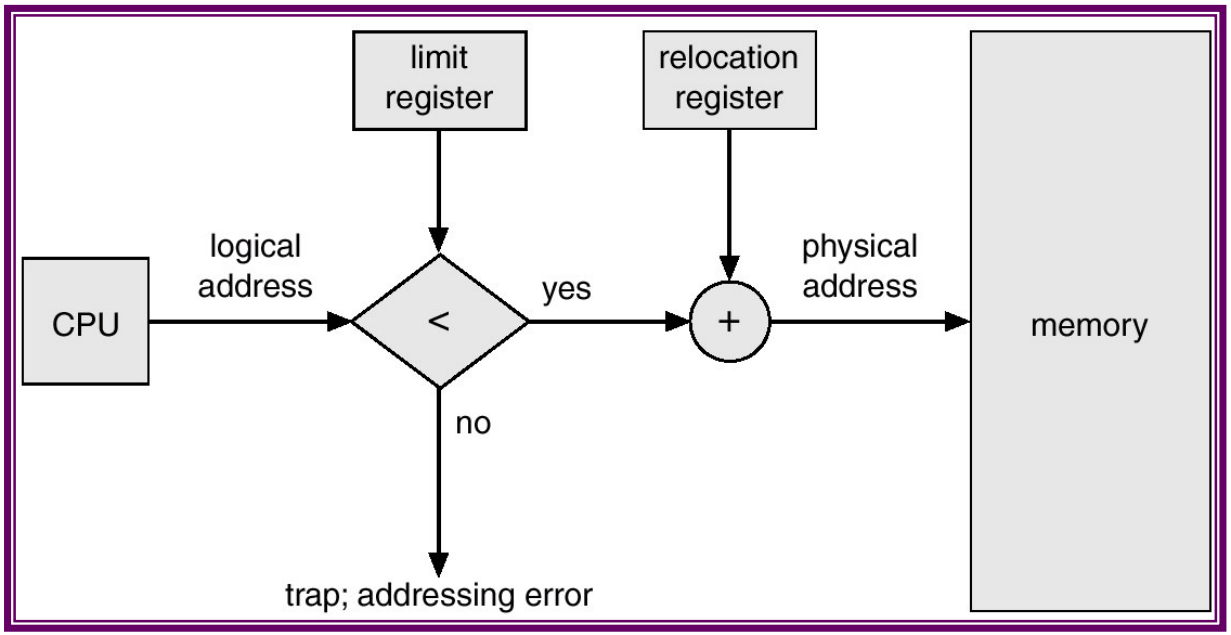
Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address.
- Simple example: the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- More interesting/flexible maps: paging, segmentation (this lecture), virtual memory (next lecture).



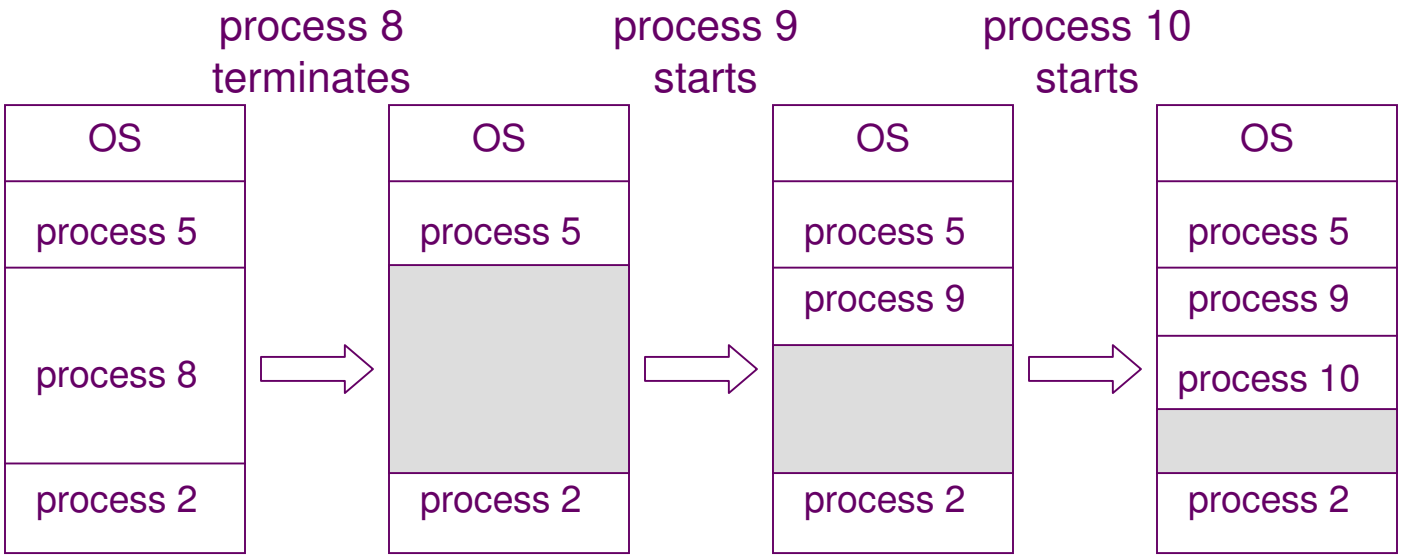
Contiguous Allocation

- Main memory usually into two partitions:
  - Resident OS, in low memory with interrupt vector.
  - User processes in high memory.
- Allocation of each partition:
  - Relocation-register: where partition starts.
  - Limit-register: max logical address (implies partition end).
  - Process must use RAM within partition only. Protect user processes from each other, OS from them.



Contiguous Allocation (Cont.)

- Allocation:
  - *Hole*: block of available memory; holes of various size scattered throughout memory.
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it.
  - OS maintains information about allocated partitions, holes.



Dynamic Storage-Allocation Problem

How to satisfy a request of given size from a list of free holes.

- **First-fit**: Allocate the *first* hole that is big enough. Fast.
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- **Worst-fit**: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole leaving lots of space for other processes.

First-fit and best-fit better than worst-fit in terms of speed and storage utilization.

External Fragmentation

- Problem: total memory space exists to satisfy a request, but it is not contiguous.

- Solution: shuffle memory contents to place all free memory together in one large hole (*compaction*).
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time.
  - I/O problem: what if process is waiting for I/O and it has given buffer memory address to I/O controller?
    - Latch job in memory while it is involved in I/O.
    - Do I/O only into OS buffers. Must copy into process address space for process to use (overhead).

Paging (Non-Contiguous Allocation)

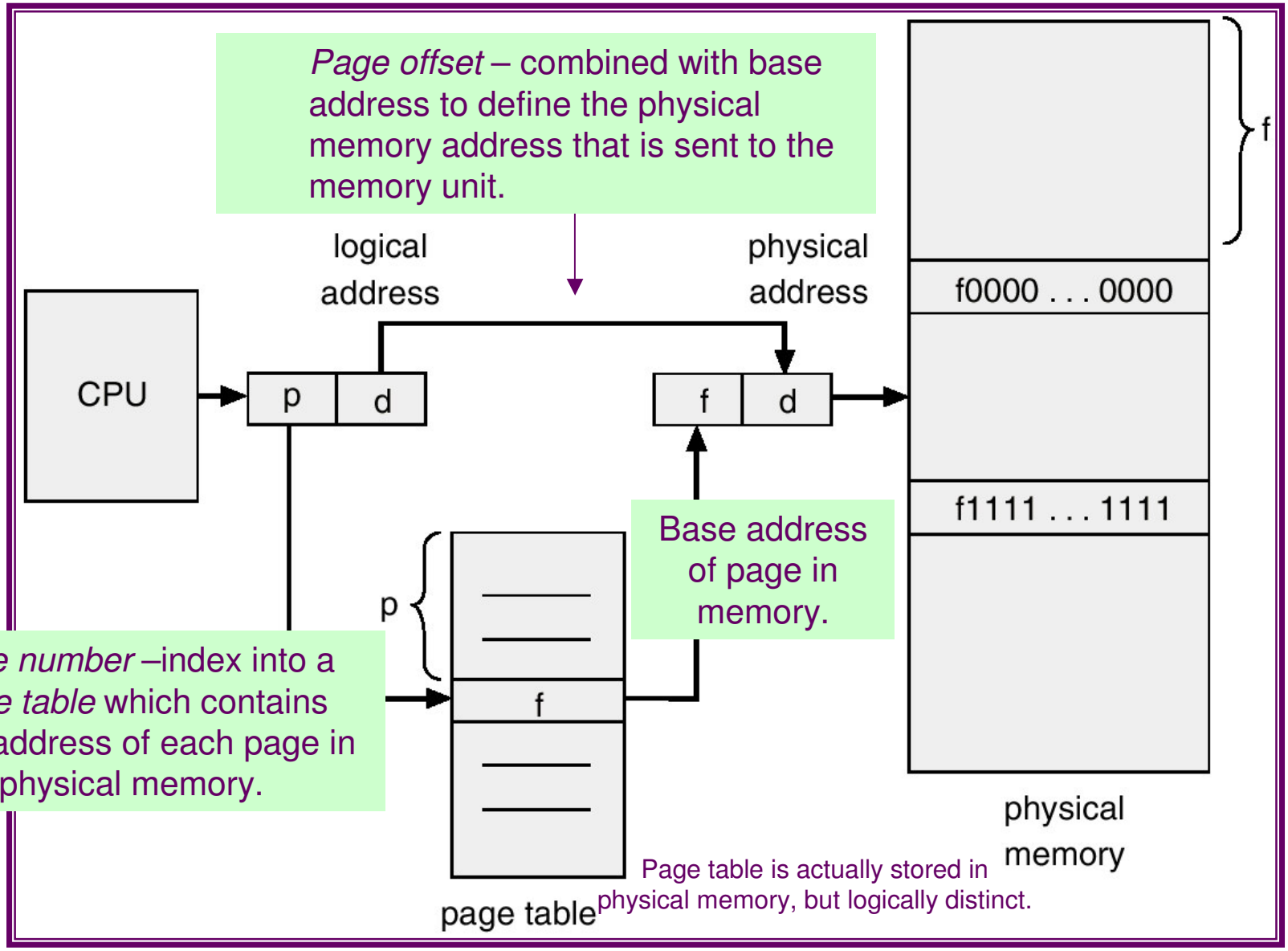
- Physical address space of a process can be noncontiguous even though logical is contiguous; process is allocated physical memory wherever the latter is available.
- Divide physical memory into fixed-sized blocks called **frames**.
- Divide logical memory into blocks of same size called **pages**.
- To run a program of size *n* pages, find *n* free frames.
- Set up a page table to translate logical to physical addresses. Different table for each process.
- **Internal Fragmentation**: if process needs 100 bytes total, it is given 8192. 8092 are wasted. Want small pages.

- Fragmentation summary: fragmentation = wasted memory.
  - Internal Fragmentation: waste *within* allocated memory.
  - External Fragmentation: waste *between* allocated memory.

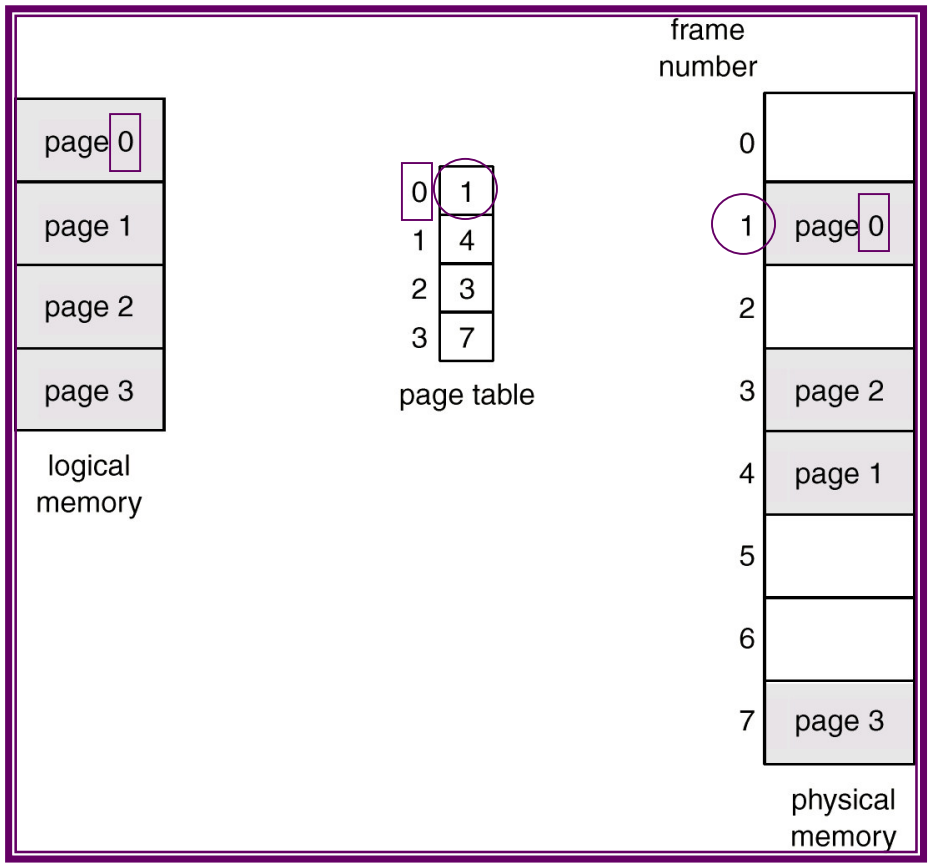
Frame/page size

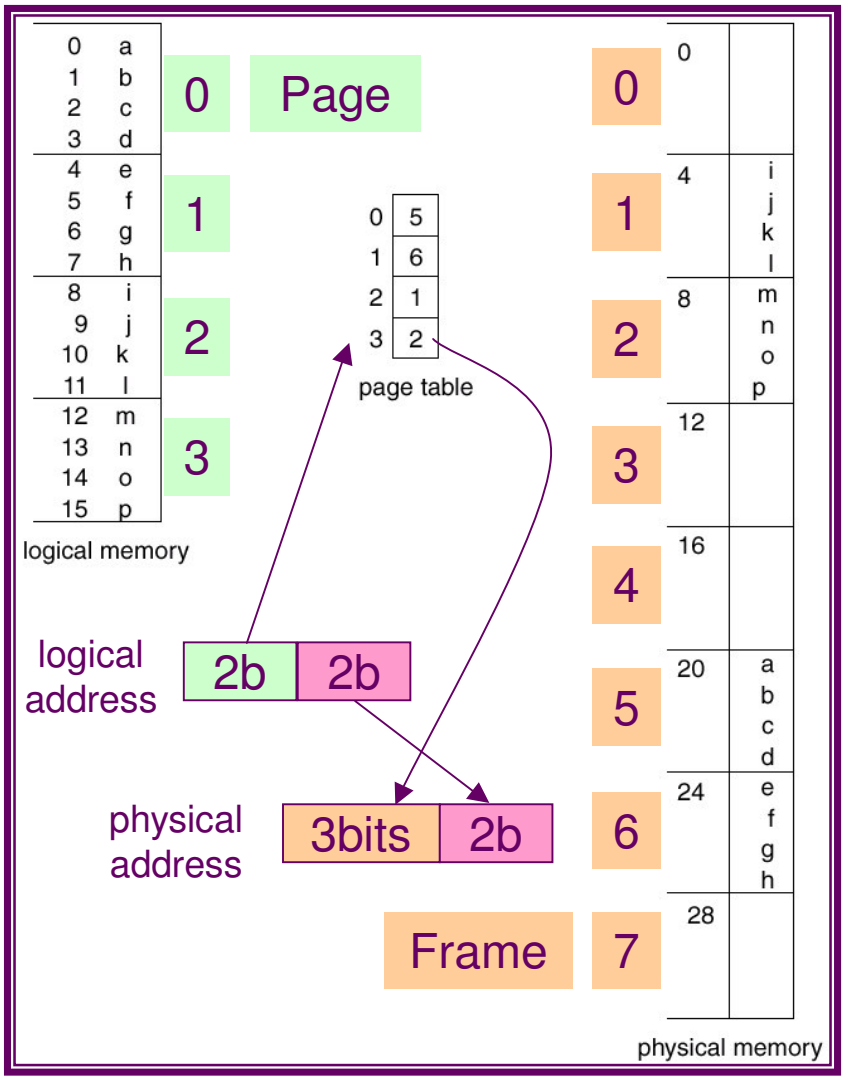
- If frame/page size s=3, the *logical* addresses
  - 0, 1, 2 go into *page* 0:
    - 0 goes to first byte within page (*offset* d=0).
    - 1 goes to second byte within page (offset 1).
    - 2 goes to third byte within page (offset 2).
  - 3, 4, 5 go into page 1:
    - 3 goes to offset 0.
    - 4 goes to offset 1.
    - 5 goes to offset 2.
  - n, n+1, n+2 go into page n div 3:
    - n+d goes to offset $d = n$ mod s.
- $p = n$ div s.
- $d = n$ mod s.
- If $s=2^k$, div and mod are bit masks:
- And if $m = f \times 2^k + d$ then concatenation:
- Think x10 or /10 in decimal.

Address Translation Scheme



Paging Examples

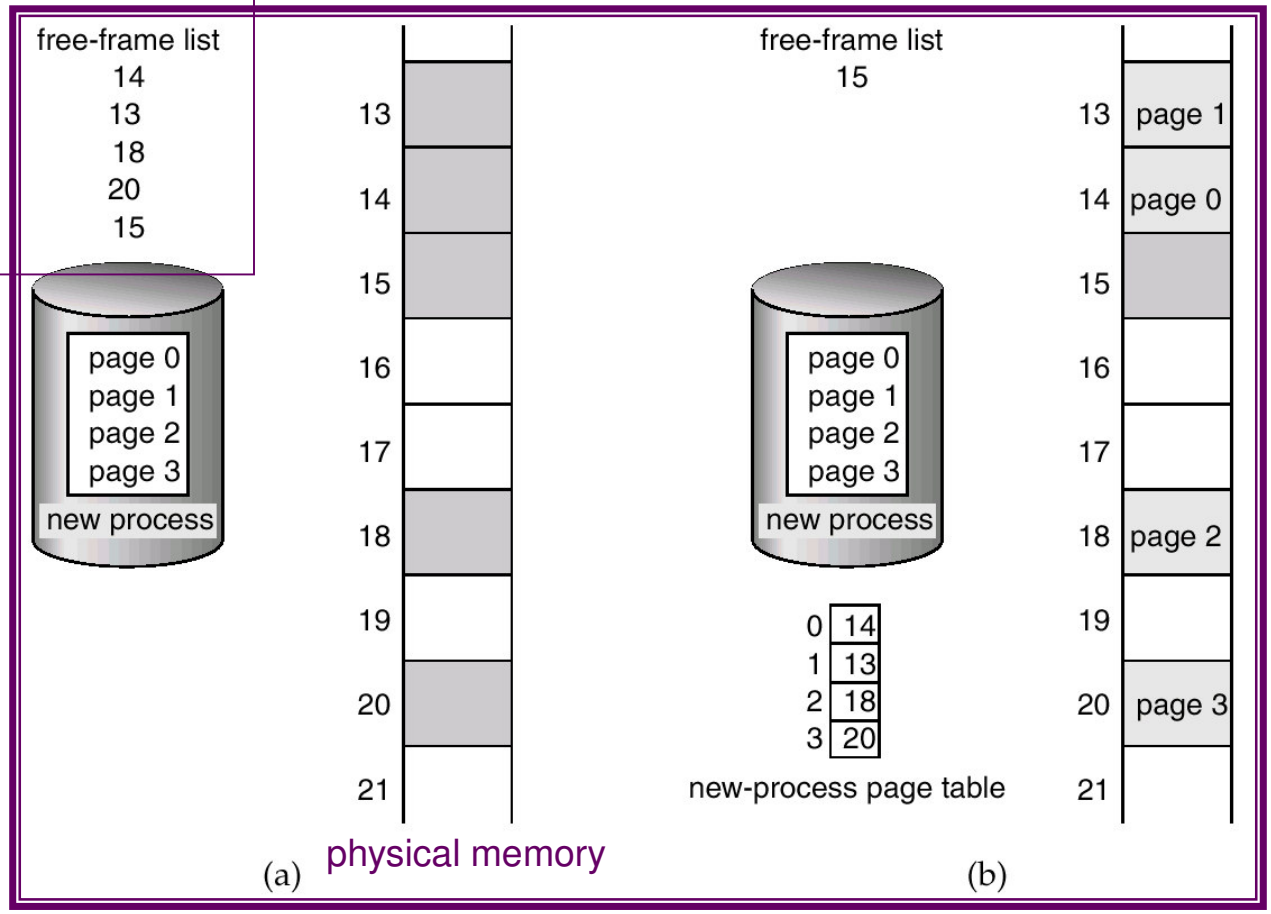




Free Frames

Or frame table instead:

- one entry per frame.
- frame allocated flag.
- process & page.

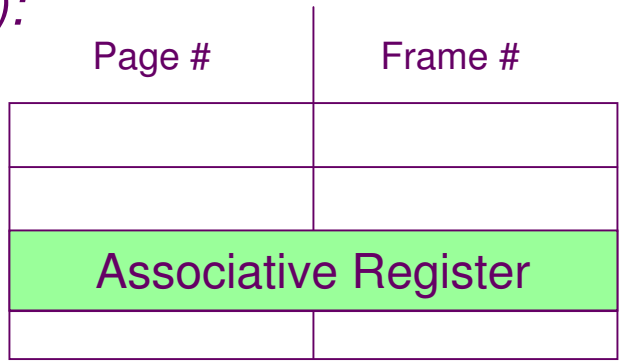


Before allocation

After allocation

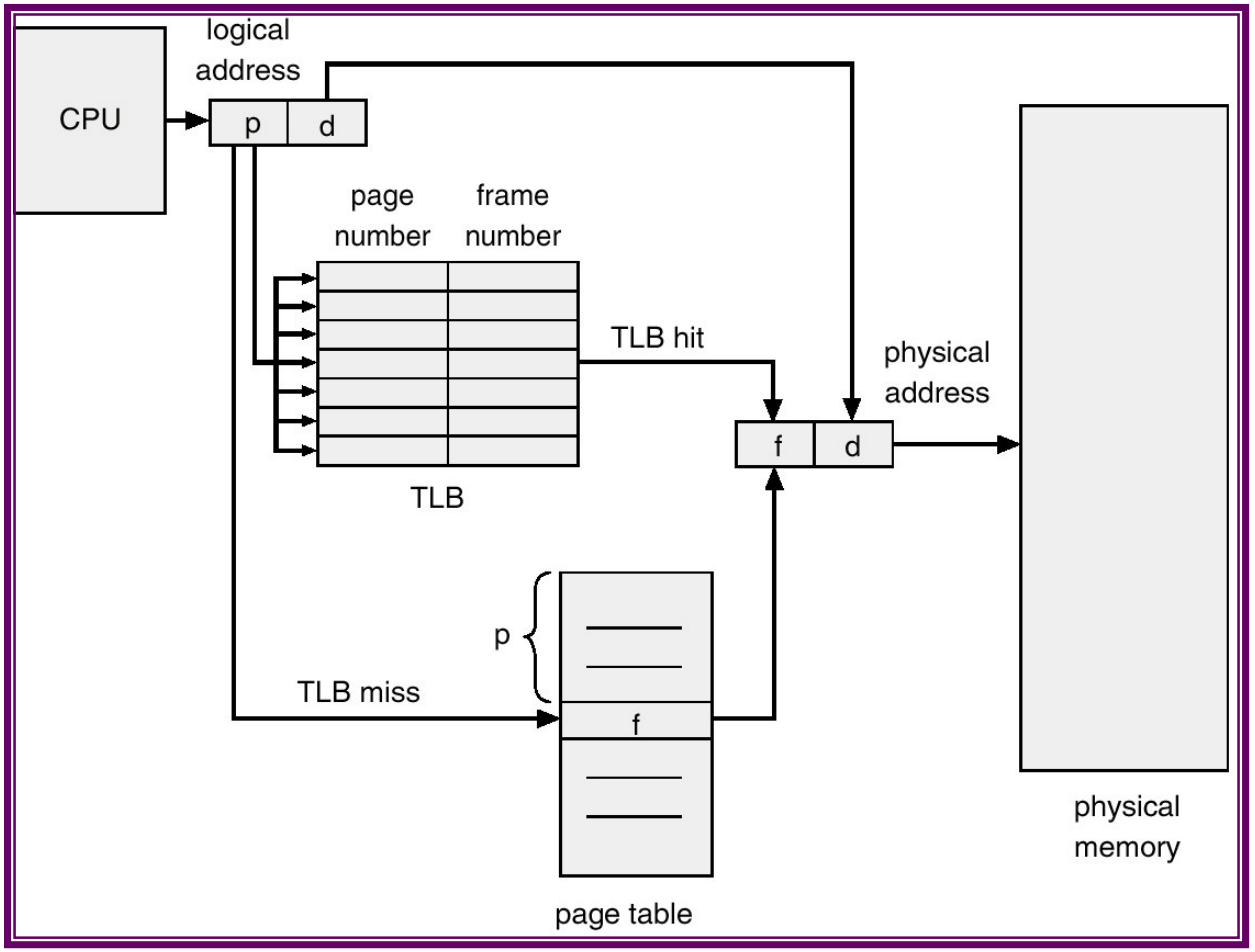
Implementation of Page Table

- Page table is kept in main memory:
  - *Page-table base register (*PTBR): table start.
  - *Page-table length register* (PTLR): table size.
- Problem: every data/instruction access requires two memory accesses (page table, data/instruction).
- Solution: *associative memory* or *translation look-aside buffers (TLBs):*



  - Fast-lookup hardware cache. All associative registers searched at the same time.
  - Need store either process ID (ASID: address space ID) or flush with context switch.

Paging Hardware With TLB



Effective Access Time

- Hit ratio: frequency that page number is found in TLB.
  - More associative registers, higher ratio but also price.
- Assume:
  - Hit ratio is f.
  - TLB Lookup is t microseconds.
  - Memory cycle time is m microseconds.
- Compute Effective Access Time (EAT):

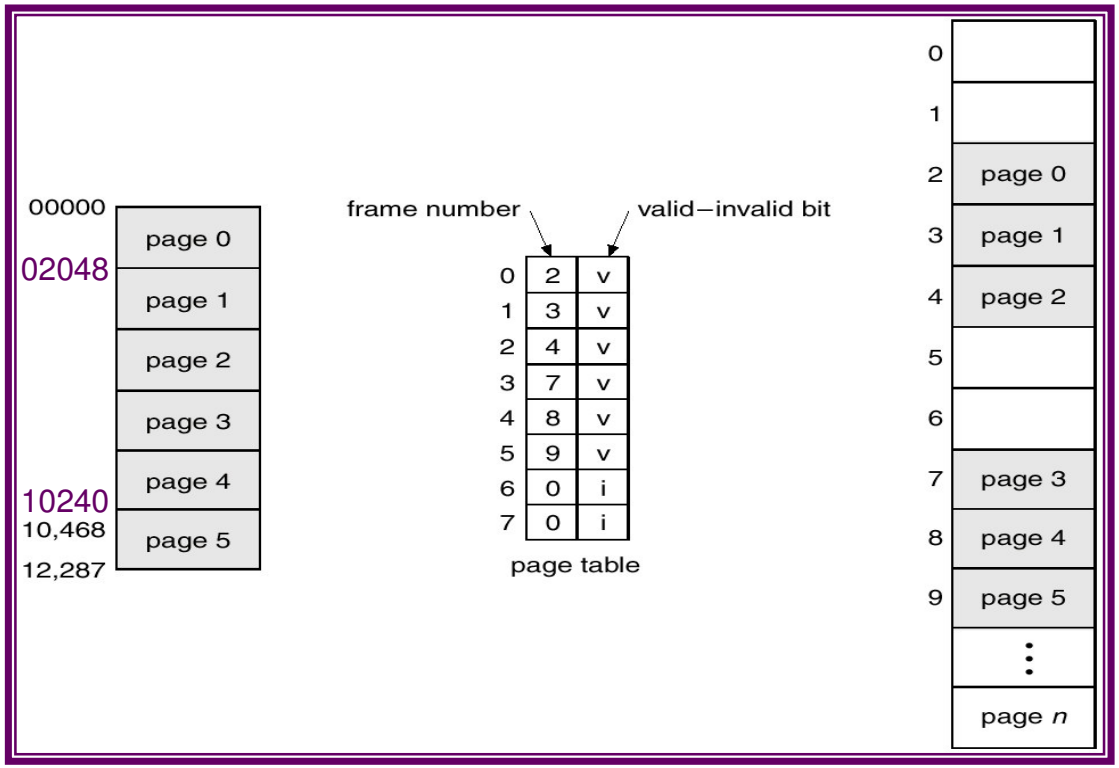
$$EAT = \underbrace{(m + t)\, f}_{\text{TLB hit}} + \underbrace{(2m + t)(1 - f)}_{\text{TLB miss}} = mf + tf + 2m + t - 2mf - tf$$

$$= m\,(2 - f) + t$$

Memory Protection

- Associating protection bits with each page:
  - Read/Write/Execute: protect code pages.
  - Valid/Invalid: instead of PTLR, all tables have same length and mark entries in use. HW needs same length to aid context switch.
- Still can't trap illegal accesses within last page (internal fragmentation).

If process size is 10469, process can access 10469 to 12287 (end of page) though it's probably a bug.



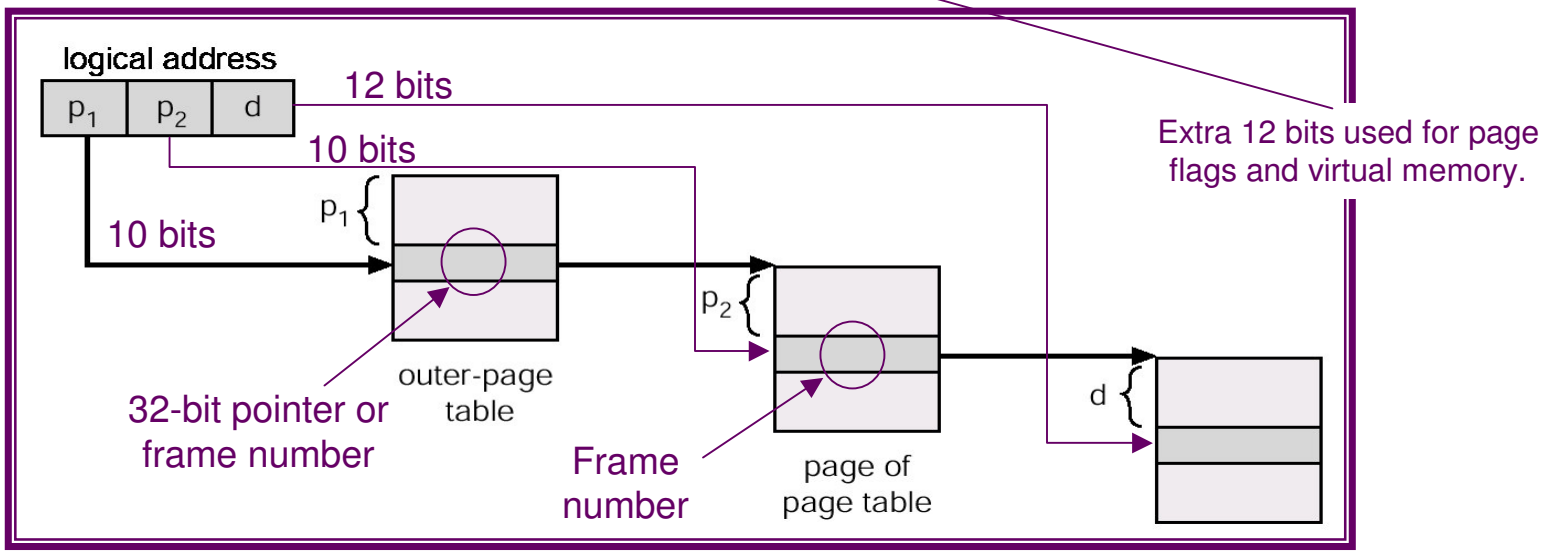
Page Table Structure

- So far, a single flat table of variable (PTLR) or constant size (valid/invalid bits).
- But physical memory size = frame size * frame count:
  - If memory large (32-bit address) frame size small (4KB=2^{12}) then frame count large ($2^{20}=2^{32}/2^{12}$ ~ 1 million).
  - Larger pages? Too much internal fragmentation so no.
  - So if we want a process to be able to use all RAM, then 1 million entries per page table.
  - Many processes too!
- Solutions:
  - Hierarchical Paging.
  - Hashed Page Tables.
  - Inverted Page Tables.

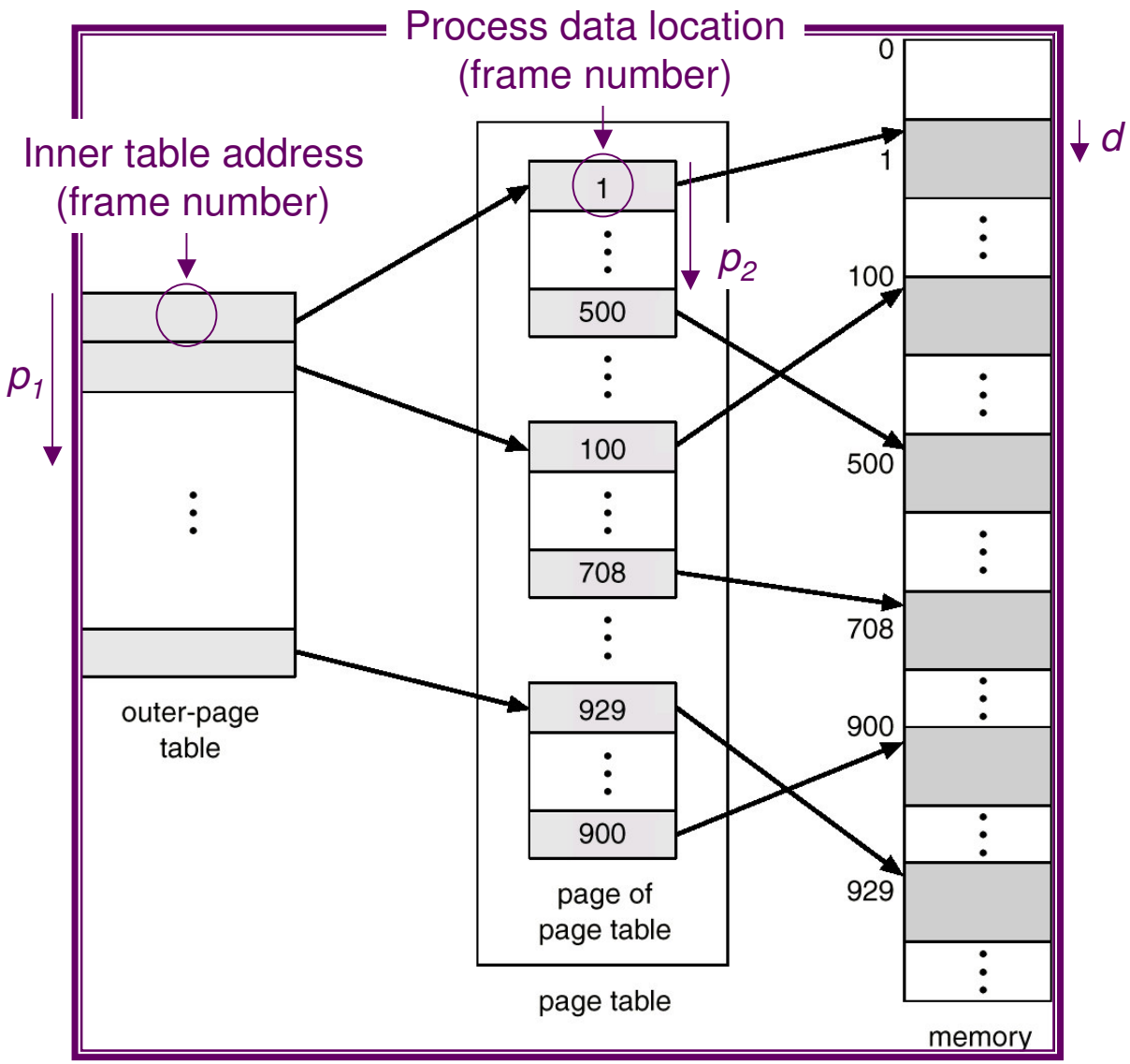
Hierarchical Page Tables

- Break up the logical address space into multiple page tables.
- A logical address (on 32-bit machine with 4KB = 2^{12} page size) is divided into:
  - Page number *p*: 20 bits. (Frame number also needs 20 bits.)
  - Page offset *d*: 12 bits. (10 used for word-addressable memory.)
- The *page table is paged* so the page number is further split:
  - Outer page number p_1: 10 bits.
  - Inner page number p_2 : 10 bits.
  - Same size: outer, inner tables come from same OS memory pool and table size = frame size:

2^{10} entries x (20 bit per frame number) <= 2^{10} x 32 bits = 2^{12} bytes.



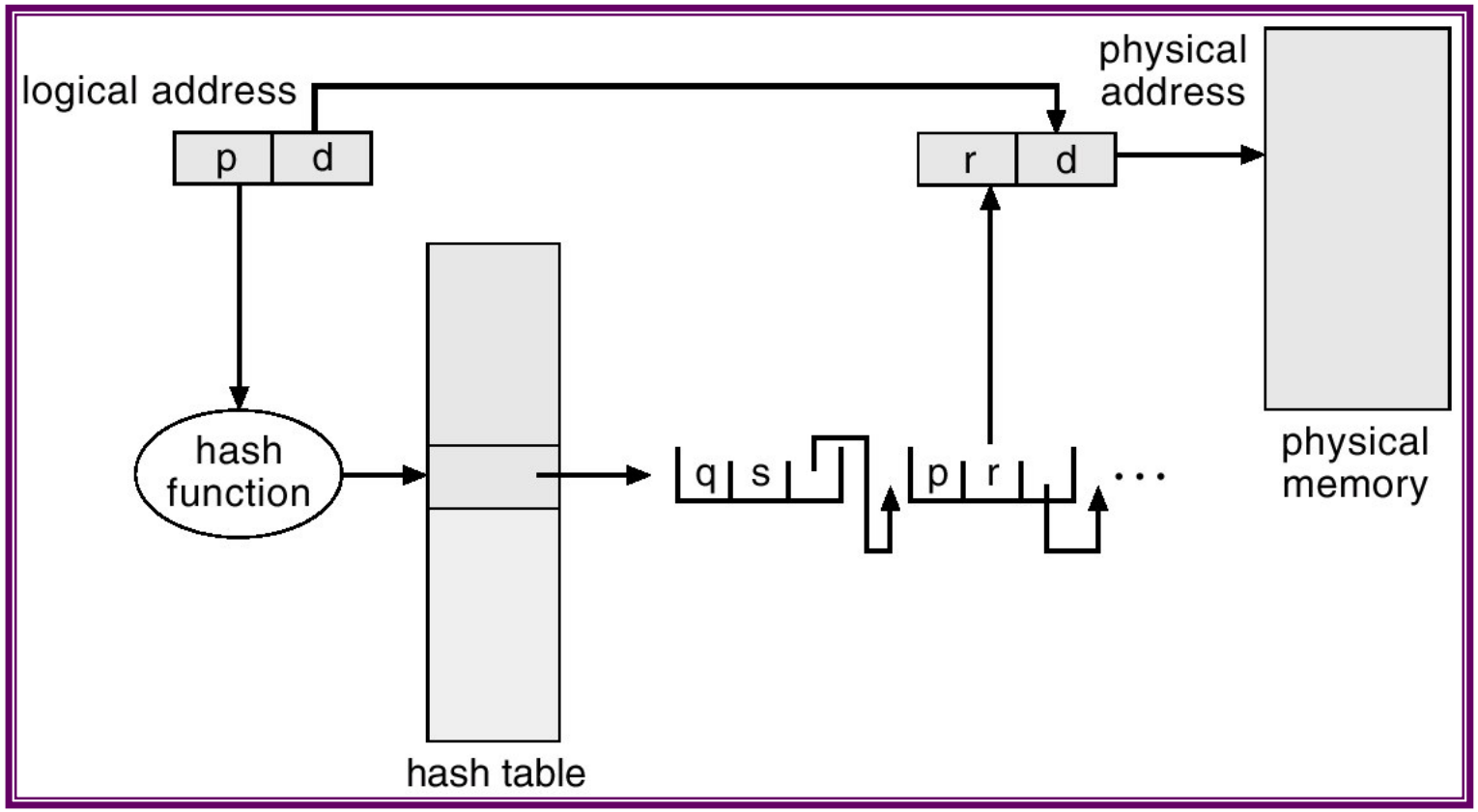
Two-Level Page-Table Scheme



Hashed Page Tables

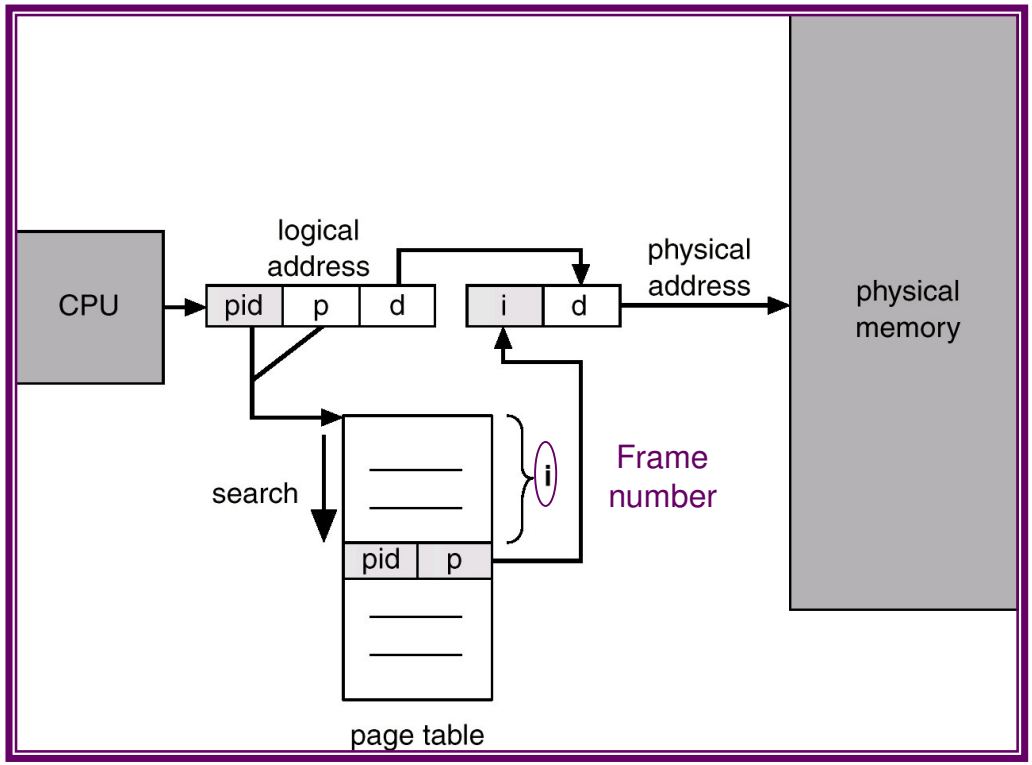
Common for 64-bit CPUs. Hashtable:

- Page number hashed into a page table which contains chain of elements hashing to the same location.
- Search for exact match to page number. If match is found, frame is extracted; if not, access is invalid.



Inverted Page Table

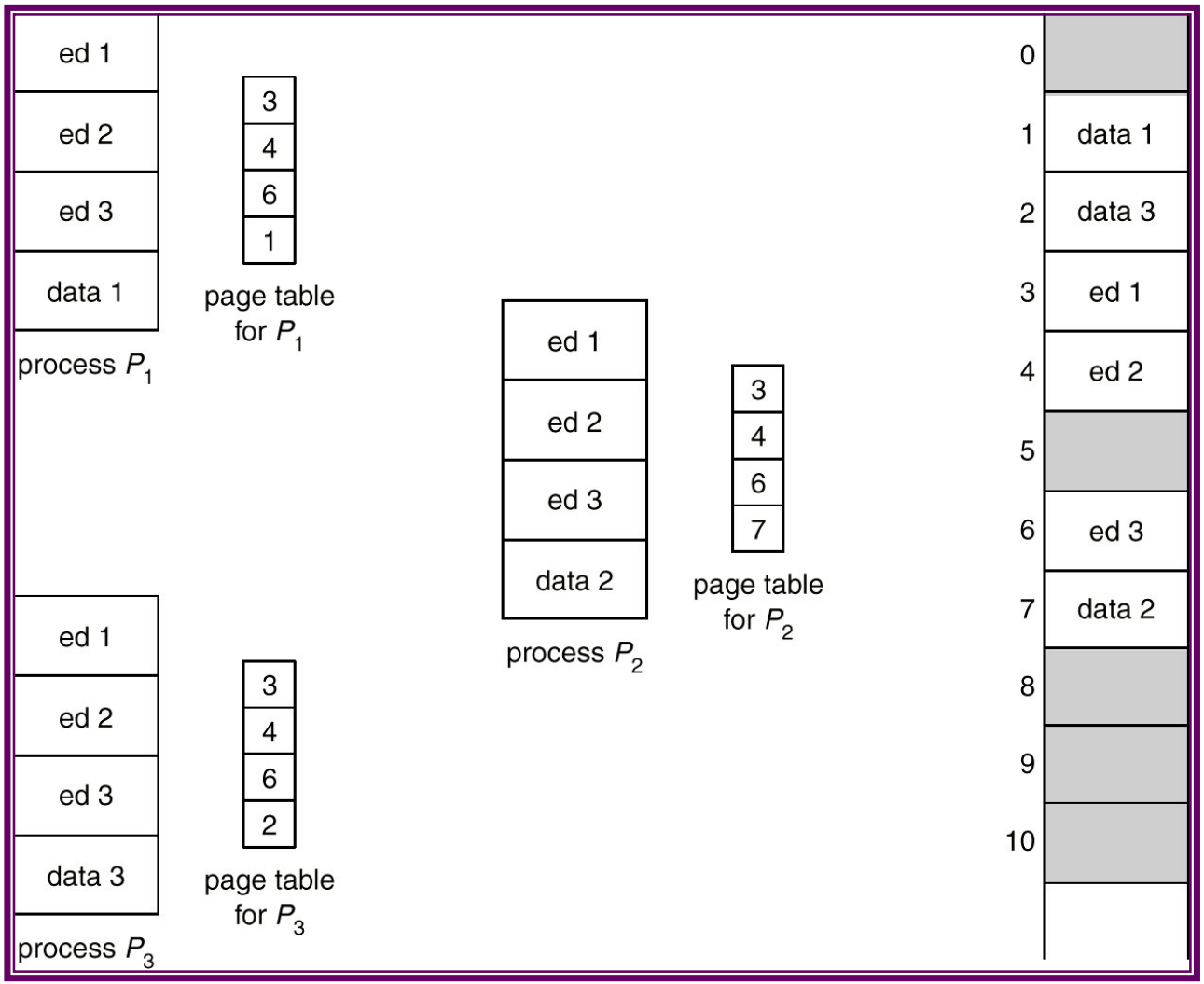
- One entry per frame with logical address of page stored in that frame, and ID of process that owns that page.
- No per-process page tables, but increases time to find frame given page: must search the table.
- Uncommon: has trouble with shared memory (see next slide) where one frame is used by 2 or more processes.



Shared Pages

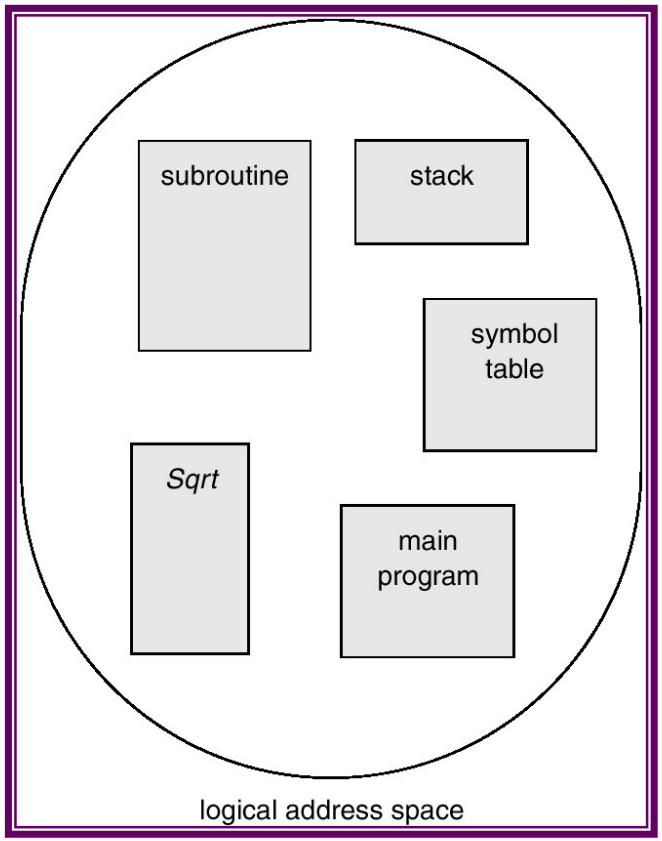
- Shared code: one copy of read-only (reentrant, non-self modifying) code shared among processes:
  - Examples: editor, window system.
  - OS forces read-only aspect: pages marked so.
  - Recall dynamic linking.
  - Shared code logical address:
    - If it has absolute branches, it must appear in same location in the logical address space of all processes.
    - If not, it can go anywhere in the space of each process.
- Shared data: used for interprocess communication.
- Private code and data:
  - Each process keeps separate copy of
    - Its own private code.
    - Data used by private or shared code.
  - Those pages can go anywhere in the logical address space.

Shared Pages Example



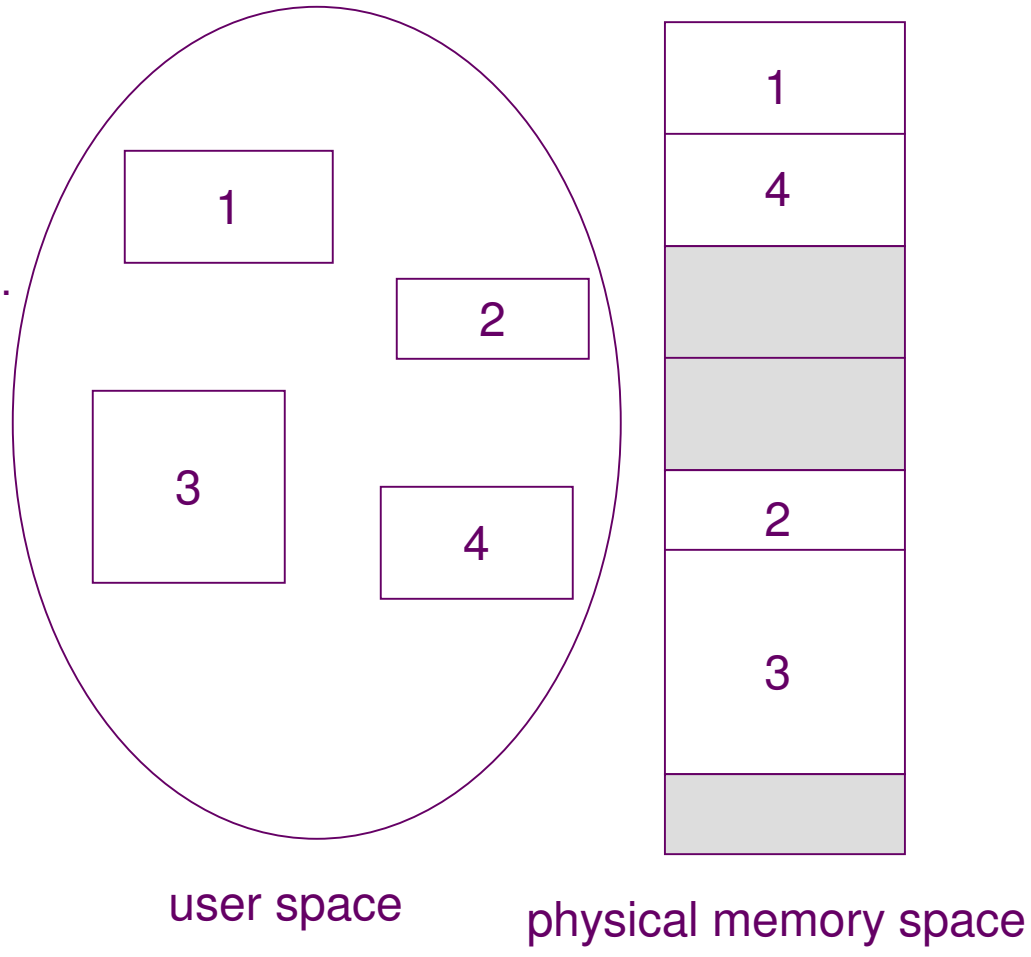
Segmentation

- Memory-management scheme that supports user's view of memory.
- A program is a collection of segments. A segment is a logical unit such as:
  - main program,
  - procedure,
  - function,
  - method,
  - object,
  - local variables, global variables,
  - common block,
  - stack,
  - symbol table, arrays.



Logical View of Segmentation

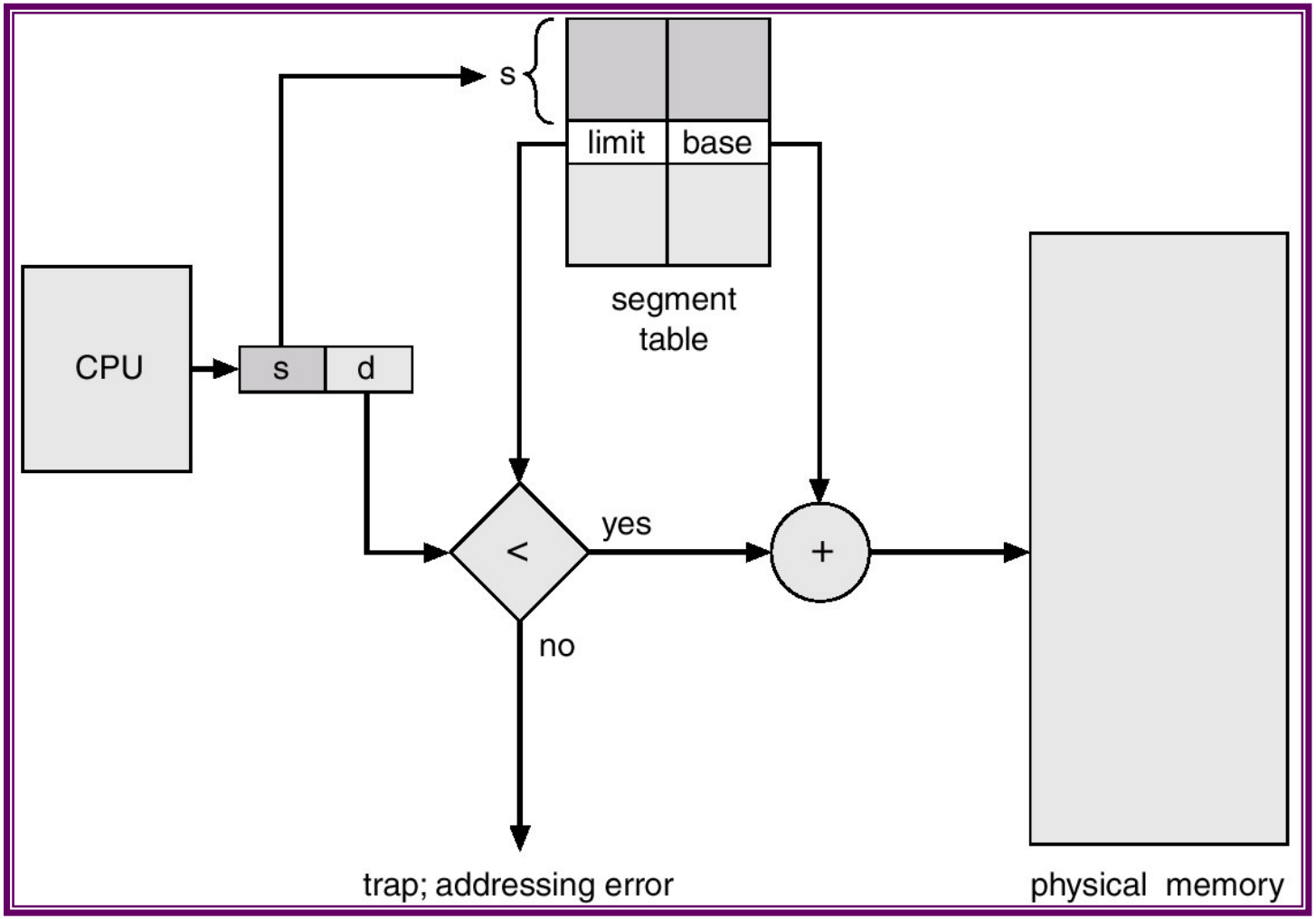
- Paging and prior techniques: logical address in range [0,*n*] and all are used, e.g. [0,*c*] for code, [*c*+1,n] for data.
- Segmentation: address split into:
  - Left part: segment number.
  - Right part: offset.
- Segmentation example:
  - Code: address 0 0 to 0 *c*.
  - Data: 1 0 to 1 *d* where *d*=*n*-*c*-1.
  - 0 *c* to 1 0 are never used.



Segmentation Architecture

- Segmentation data structures:
  - Segment-number: index into segment table (below).
  - Offset: as in paging.
- Segment table entry:
  - Base address: physical address where segment starts.
  - Limit: length of the segment.
- Registers:
  - *Segment-table base register (STBR):* table start.
  - *Segment-table length register (STLR)*: table size.
- Mix of contiguous and non-contiguous:
  - Like contiguous, we effectively have relocation & limit registers, but different value for each segment, not each process.
  - Like paging (non-contiguous), logical address goes through table lookup and one process has multiple pieces.

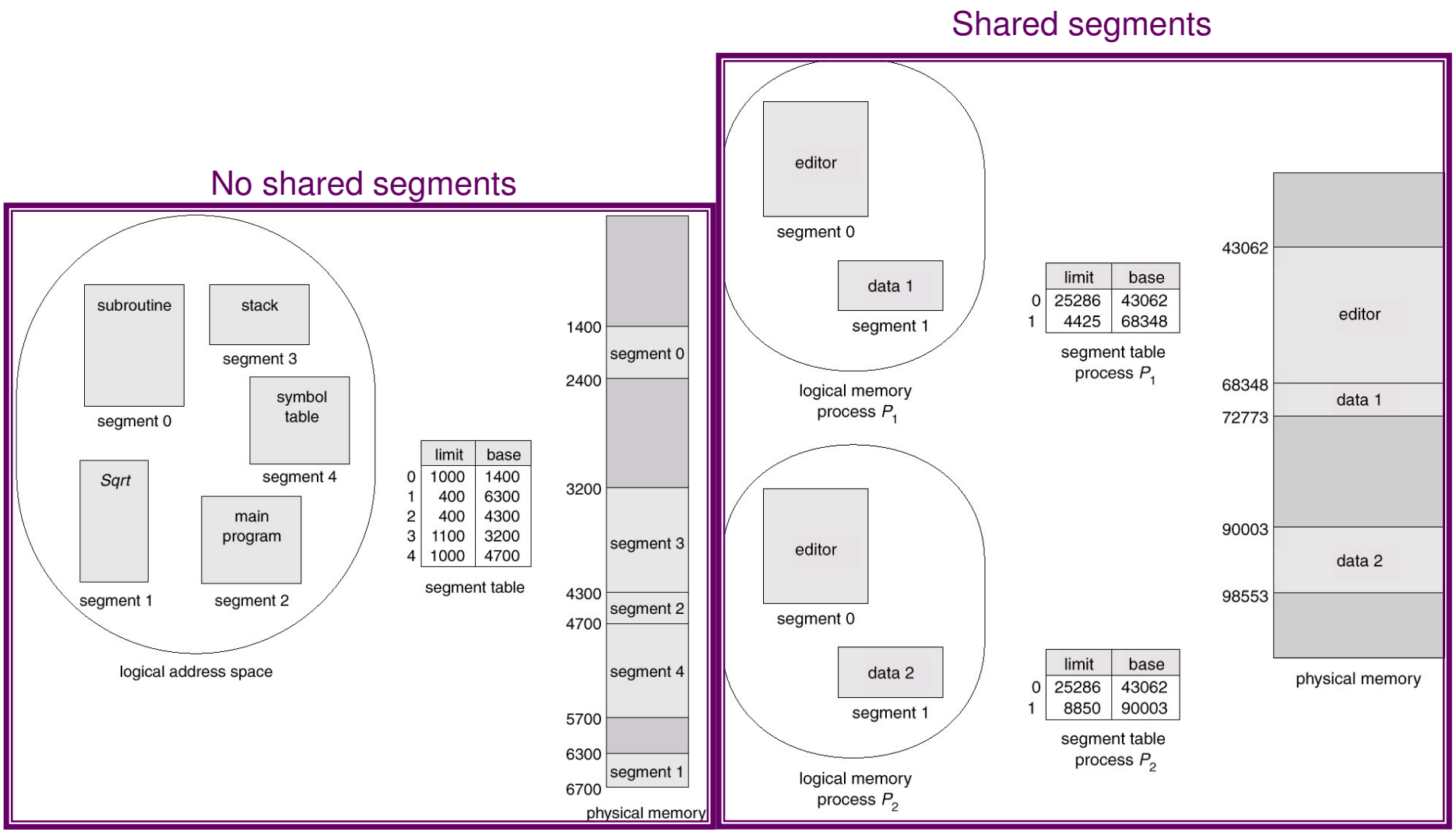
Segmentation Hardware



Segmentation Architecture (Cont.)

- Code relocation: change segment table entry.
- Sharing segment: same issues as shared pages but makes more sense to user than paging since segments are logical units.
- Protection. With each entry in segment table associate:
  - Valid/Invalid: like paging.
  - Read/Write/Execute: make more sense to user and less waste than paging:
    - Paged process has 3 bytes: 1 executable, 1 read, 1 R/W.
    - 8KB pages: wastes 3*8191 bytes! If no protection, all bytes go in one page, waste is 8189 total.
- Allocation: same issues as contiguous allocation (dynamic storage-allocation problem).

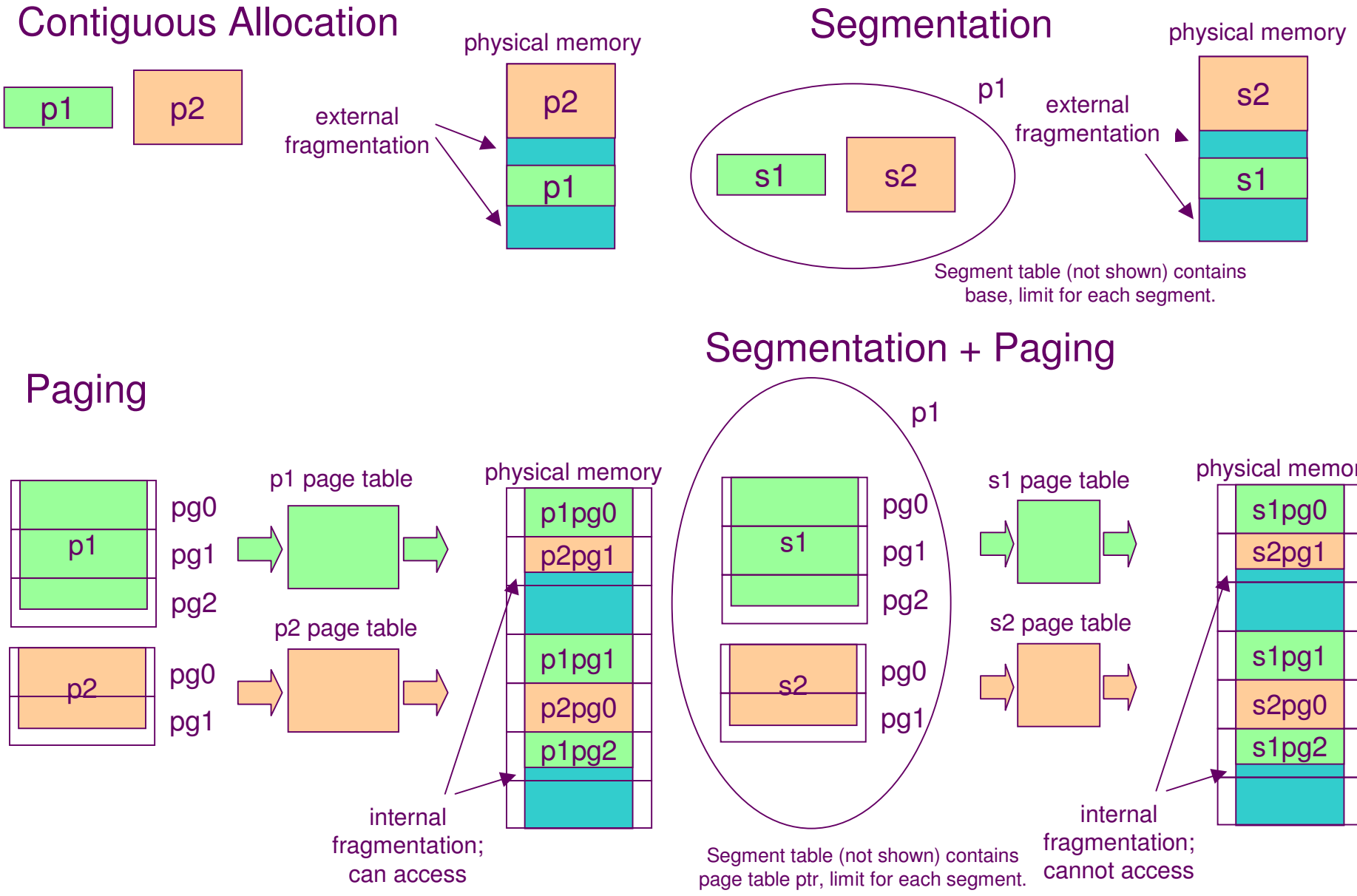
Examples of Segmentation



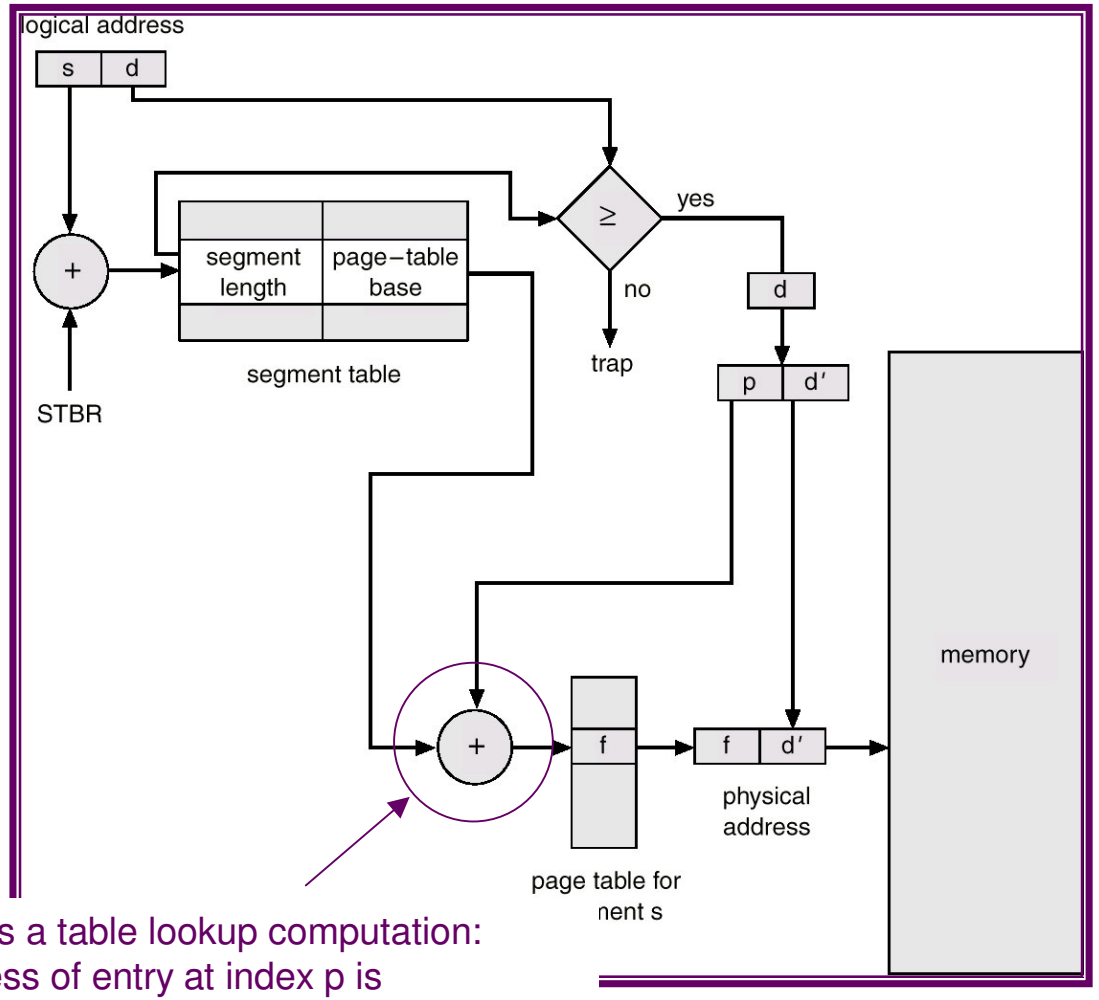
Segmentation with Paging

- Solve allocation issue by paging the segments. Intuition:
  - Pure paging solves problem of fitting a set of processes into memory. Each process if variable sized sequence of addresses.
  - Segment is a variable sized sequence of addresses too! Hence, treat segment as if it was a process: add one page table per segment, and page its contents into physical memory, which is divided into frames.
  - Segment table maps each each segment to a page table like OS maps each process to a page table.
  - One segment table per process, one page table per segment.
- Segment-table entry contains the base address of a *page table* for this segment.
- So like paging but
  - Memory matches user view (protection, sharing).
  - No memory access beyond true end of process on last page.
- Intel 386 and later.
- Key part of assignment 2.

Segmentation with Paging Intuition



Segmentation with Paging Example



This is a table lookup computation: address of entry at index p is (page-table base + p). Conceptually, page-table base points to the table, and p is the entry index, as with PTBR and p.

Swapping

- A process can be *swapped* temporarily out of memory to a *backing store*, and then come back into memory to continue execution.
- Backing store: fast disk large enough to accommodate copies of all memory images for all users; must provide quick, direct access to these memory images (not via complex filesystem structures).
- *Roll out, roll in*: swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.
- Major part of swap time is transfer time; total transfer time is directly proportional to the *amount* of memory swapped.
- Can also swap OS data structures, e.g. unused page tables. Segment table entry:
  - If page table in memory, its address.
  - If page table on disk, the disk block index.

- How about using disk to store only part of a process, not all of it, while the rest is in memory and runs? Virtual memory (next lecture).