Administrative

- MP6 due on Monday
  - A few hints today
- Grading – will give your highest MP grade extra 3%
- Marco will be lecturing on Friday
  - I will not have office hours
- Slides posted night before, didn't work
  - Will post updated slides this afternoon

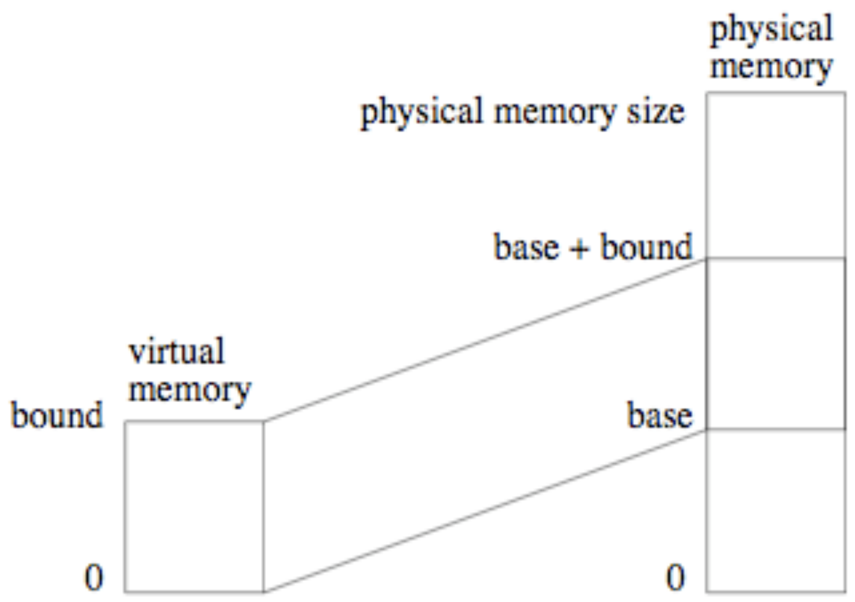
MP6 hints

- Think carefully about your “jobs” data structure
  - Go through all of the ops in your design while deciding what to put in “jobs” struct
- Check return value to sem_wait
  - I actually got this part wrong in my sol ☺
- Ok to serialize all access to internal structures
  - Calls to “system” need to run in parallel if possible

Base and bounds

- Load each process into contiguous regions of phys. mem

```
If(virt addr > bound)
    trap to kern
} else {
    phys addr = virt addr +
                base
}
```



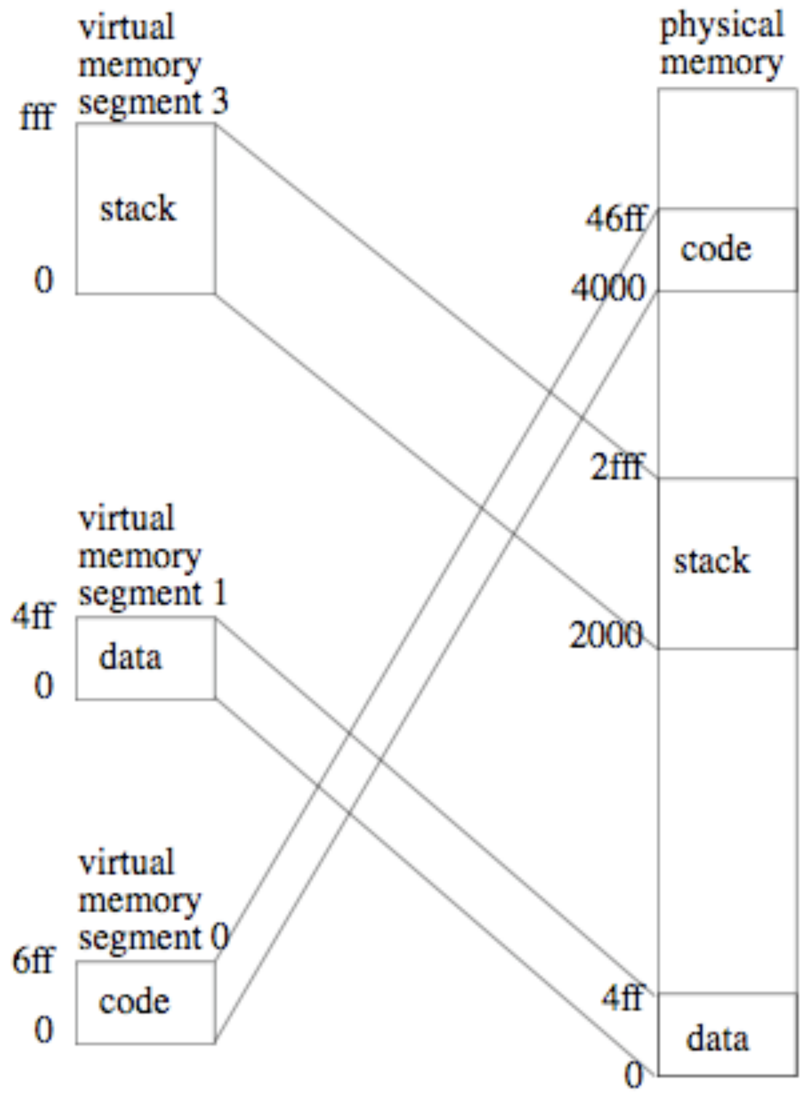
Segmentation

- Segment: region of contiguous memory
- Segmentation: generalized base and bounds (support for multiple segments at once)

Segmentation

- Segment specified lots of different ways
- What are the advantages over base and bounds?

segment #	base	bound	description
0	4000	700	code segment
1	0	500	data segment
2	unused		
3	2000	1000	stack segment



Problem with segmentation and B&B

- What was the key abstraction not supported well by segmentation and by B&B?
  - How could you support this using B&B and segmentation?

- Note: x86 used to support segmentation, now effectively deprecated with x86-64

Paging

- Allocate physical memory in terms of fixed-size chunks
  - Fixed unit makes it easier to allocate
  - Any free physical page can store any virtual page
- Virtual address
  - Virtual page # (high bits of address)
  - Offset (low bits of address, e.g., bits 11-0 for 4k page)

Translation table

Virtual page #	Physical page #
0	10
1	15
2	20
3	invalid
…	invalid
1048575	invalid

Translation process

```
If(virtual page is invalid or non-resident or
   protected) {
    trap to OS fault handler
} else {
    physical page # = pageTable[virtpage#]
                          .physPageNum
}
```

- What must change on a context switch?
- Each virtual page can be in physical memory or swapped out to disk (called paged)

Paging

- How does the processor know that a virtual page is not in memory?

- Like segments, pages can have different protections
  - Read, write, execute

Valid vs resident

- Resident means a virtual page is in memory.
  - NOT an error for a program to access non-resident page
- Valid means that a virtual page is legal for the program to access
  - E.g., not part of the address space

Valid vs resident

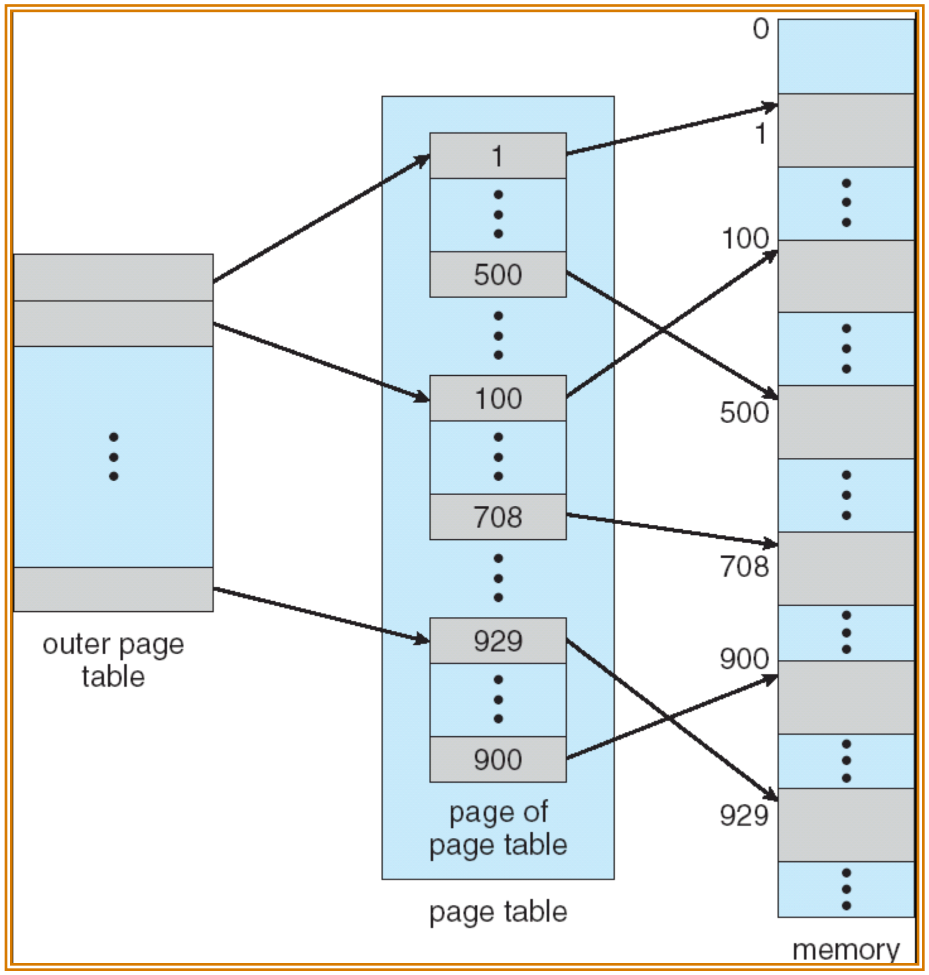
- Who makes a page resident/non-resident?
- Who makes a virtual page valid/ invalid?
- Why would a process want one if its virtual pages to be invalid?

Multi-level translation

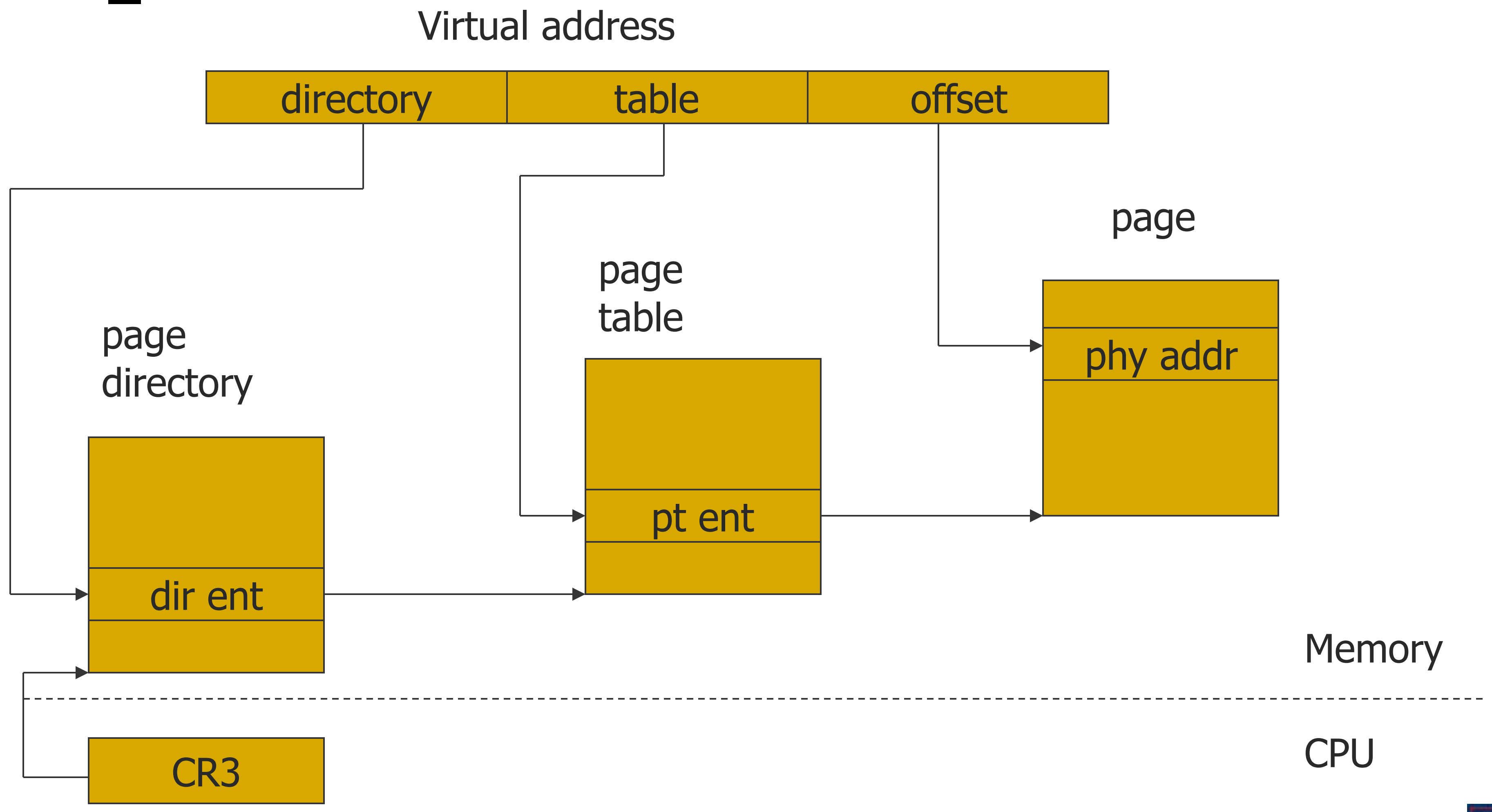
- Standard page table is a simple array
  - Might take huge amounts of memory for sparse address space.
    - 32 bit address space (4KB pages): 2^{20} * 4 = 4 MB
    - 64 bit address space (4KB pages): 2^{52} * 8 = 32 PB!
  - Multi-level translation changes this into a tree

- E.g., two-level page table on 32 bit machine
  - Level 1 – virtual address bits 31-22 index
  - Level 2 – virtual address bits 21-12 index
  - Offset: bits 11-0 (4KB page)

Two-Level Page-Table



x86 Page tables (32 bit)



From Intel manual

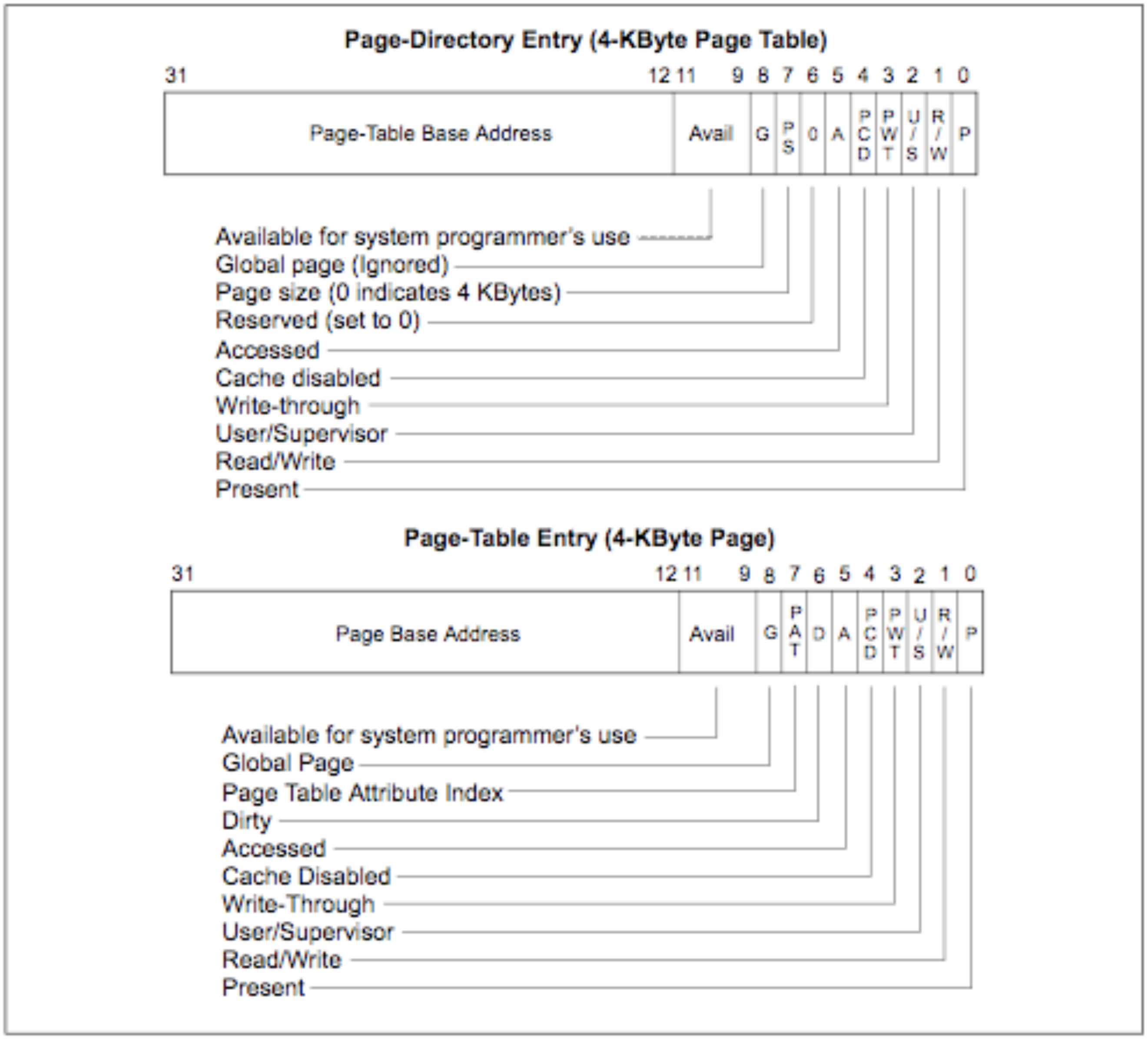


Figure 3-14. Format of Page-Directory and Page-Table Entries for 4-KByte Pages and 32-Bit Physical Addresses

Problem (from Tanenbaum)

- A computer with a 32-bit address uses a two-level page table. Virtual addresses split into a 9-bit top-level page table field, an 11-bit second-level page table field, and an offset. How large are the pages and how many are there in the address space?

Problem (from Tanenbaum)

- Offset is 12 bits
  - Page size 2^12 or 4KB
- Virtual pages = (2^32 / 2^12) = 2^20
- Note: driven by number of bits in offset
  - Independent of size of top and 2nd level

Problem

- Assume single-level page table
- Page table entry
  - Top 20 bits for physical address
  - Bottom 12 for permissions, etc.
  - Just like x86 page table entries
- Write a function, translate, that converts a virtual address to a physical address

Return the physical address

```
ulong translate(ulong va, pte_t *pt) {







}
```

Return the physical address

```
ulong translate(ulong va, pte_t *pt) {
    int virtPageNo = va >> 12;
    // a pte for each virtual page
    pte_t pte = pt[virtPageNo];
    ulong pa = (pte & 0xfffff000) |
                    (va & 0x00000fff);
    return pa;
}
```

Discussion

- How can paging be made faster?
  - Mapping must be done for every reference
    - 2 level page table, 3 memory ops per each load/store

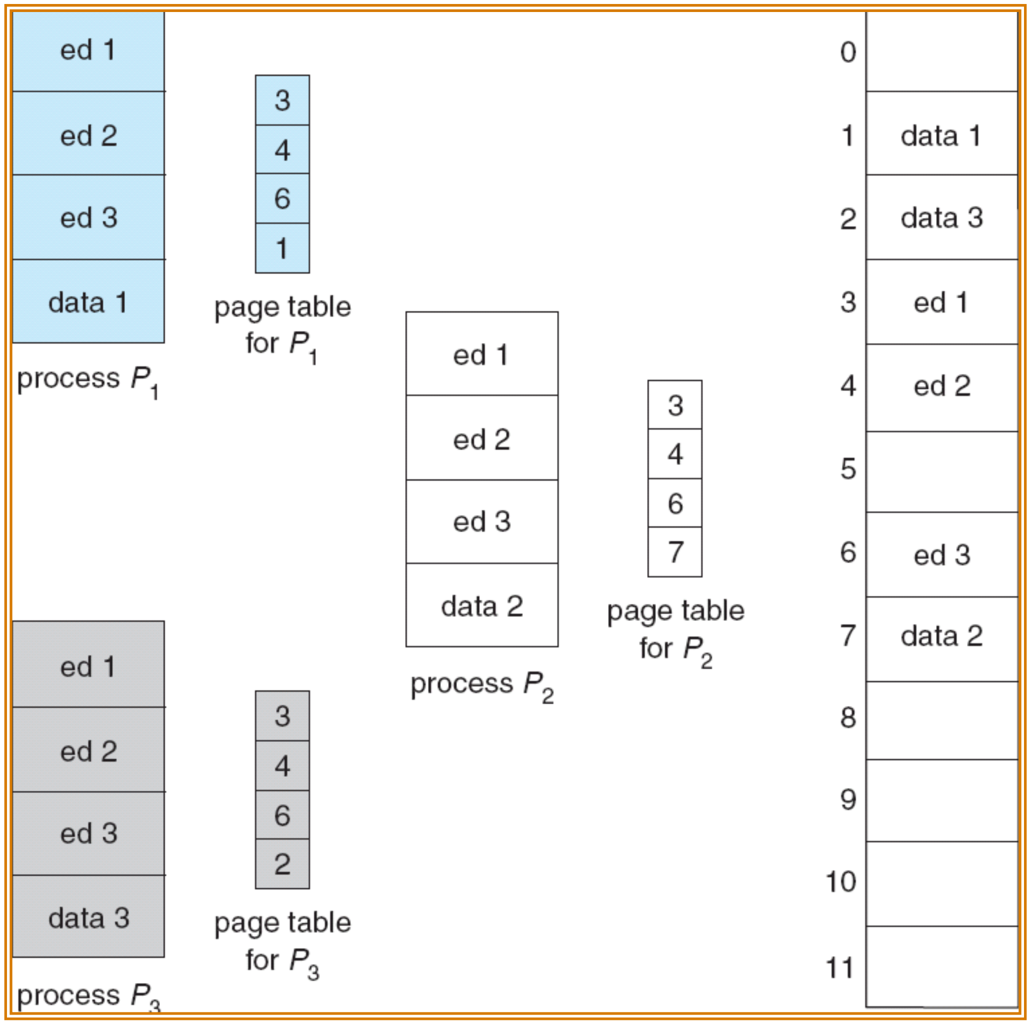
Paging - Caching the Page Table

- Cache page table entries in registers
  - Called a translation lookaside buffer
    - I.e., TLB
- Keep page table in memory
  - Location given by a *page table base register*
- Page table base register changed at context switch time

Sharing Pages

- Shared code
  - One copy of read-only code shared (e.g., libraries) among processes (e.g., text editors, compilers, web browsers).
- Private code and data
  - Each process keeps a separate copy of the code and data

Shared Pages


ed 1
ed 2
ed 3
data 1
process P_1
3
4
6
1
page table for P_1
ed 1
ed 2
ed 3
data 2
process P_2
3
4
6
7
page table for P_2
ed 1
ed 2
ed 3
data 3
process P_3
3
4
6
2
page table for P_3
0
1 data 1
2 data 3
3 ed 1
4 ed 2
5
6 ed 3
7 data 2
8
9
10
11