Sheets for MIEIC's SO

based on teaching material supplied by A. Tanenbaum for book: Modern Operating Systems, ed...

Chap 4: Memory Management

Chapter 4

Memory Management

Basics
Swapping
Virtual memory
Paging
Segmentation

Memory Management

Ideally, programmers want memory that is

- large
- fast
- non volatile

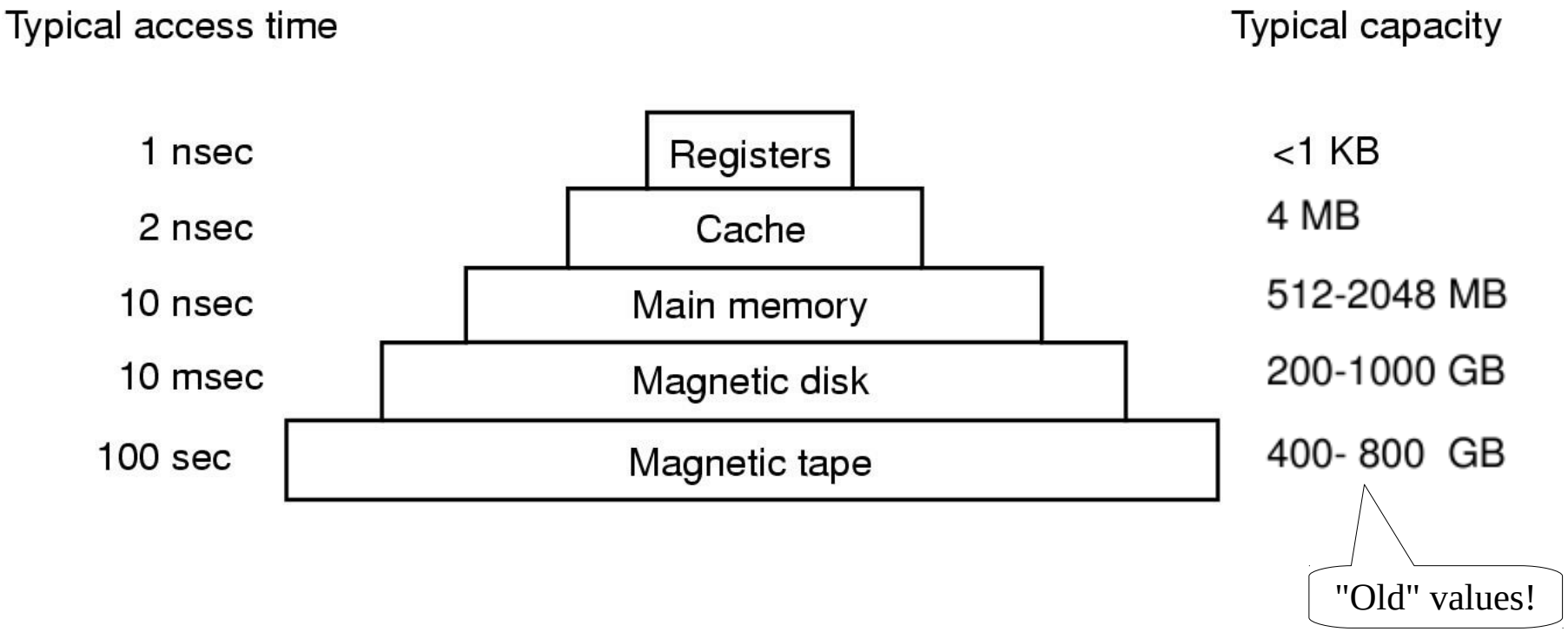
But in reality, there is available

- small amount of fast, expensive, volatile (**cache**) memory
- some medium-speed, medium price, (**main**) volatile memory
- gigabytes of slow, cheap, non-volatile (**disk**) storage

forming a Memory hierarchy [FIG]
handled by the *Memory manager*

Computer hardware: memory

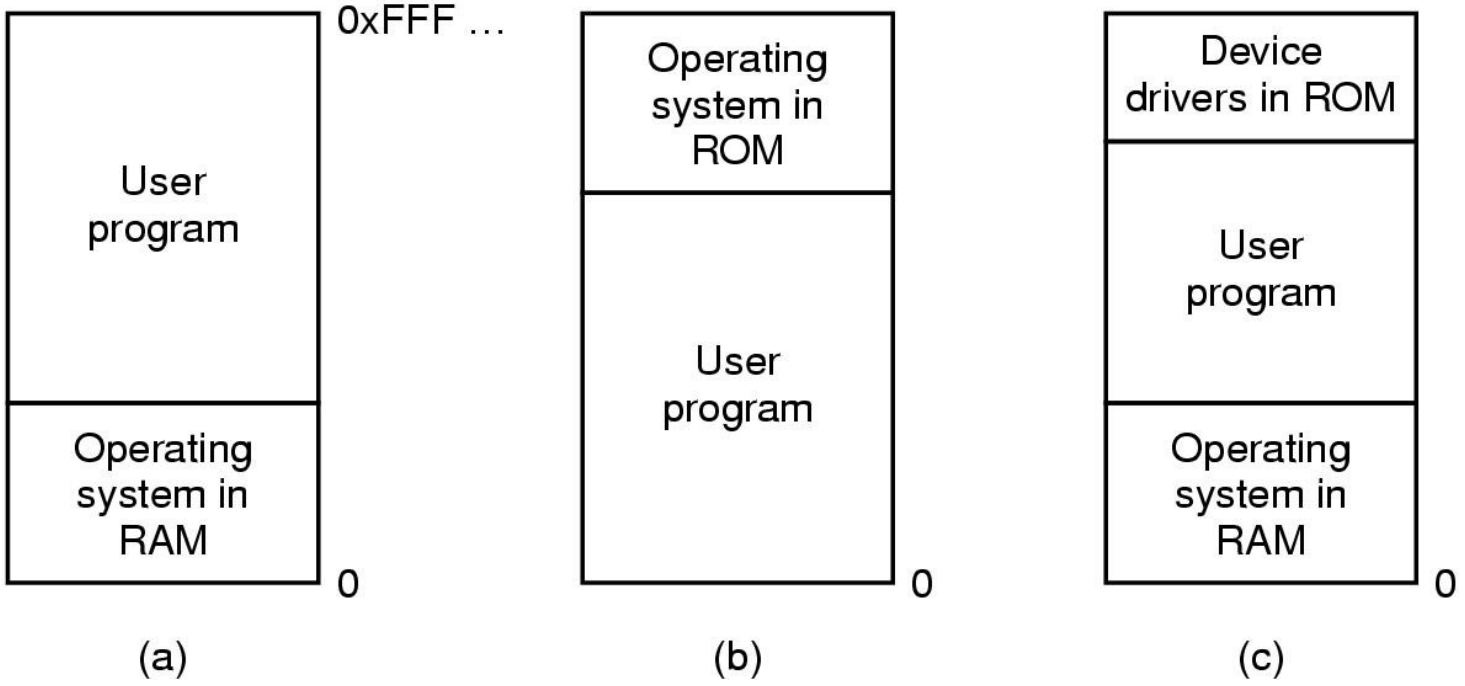
(*from Introductory chapter*)



Typical memory hierarchy

numbers shown are rough (obsolete?) approximations

Basic Memory Management
Monoprogramming

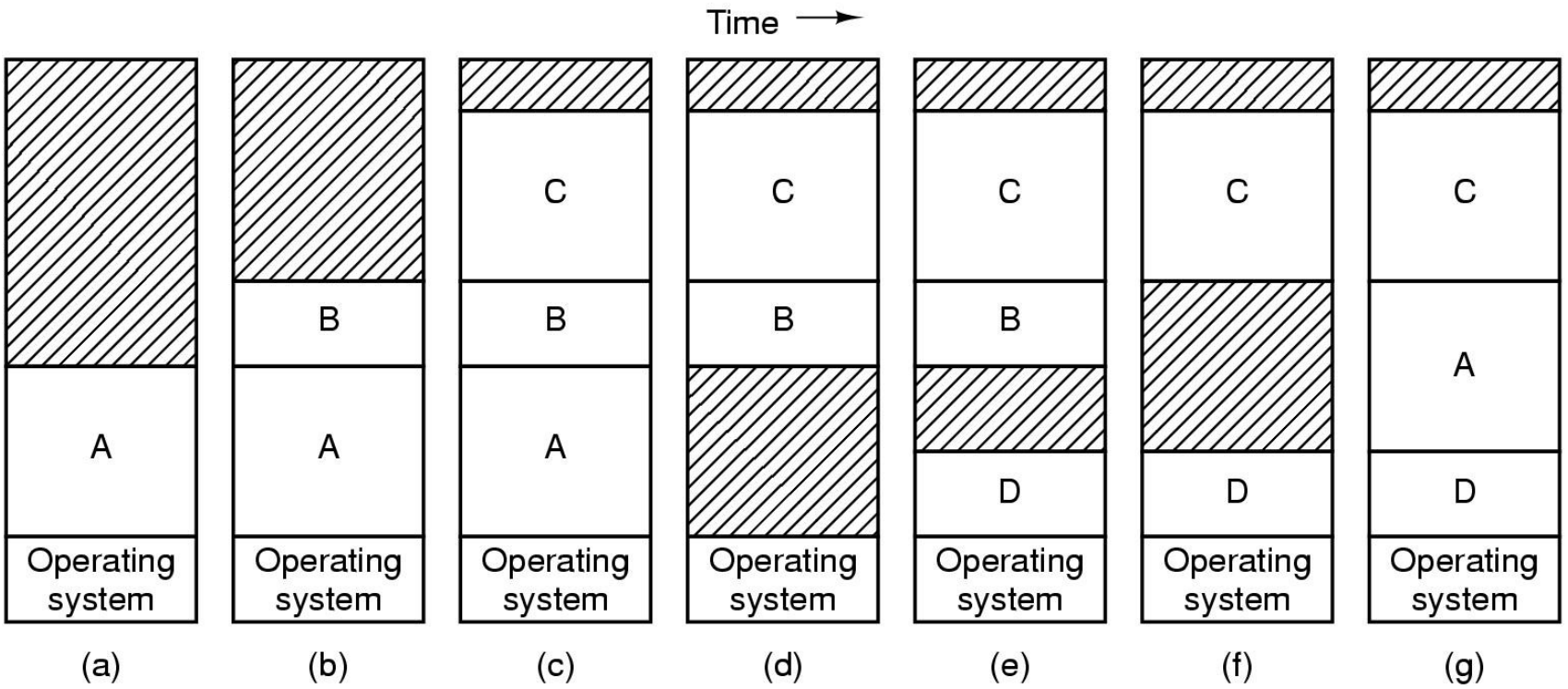


Three simple ways of organizing memory

an operating system with a single user process

c) IBM PC running MSDOS (system ROM = BIOS)

Multiprogramming: variable memory partitions



Variable memory partitions

- (a) to (e): memory allocation changes as processes evolve (initiate, terminate, are swapped out...)
- memory holes (shaded regions) are formed
  → external fragmentation [*see segmentation, ahead*]

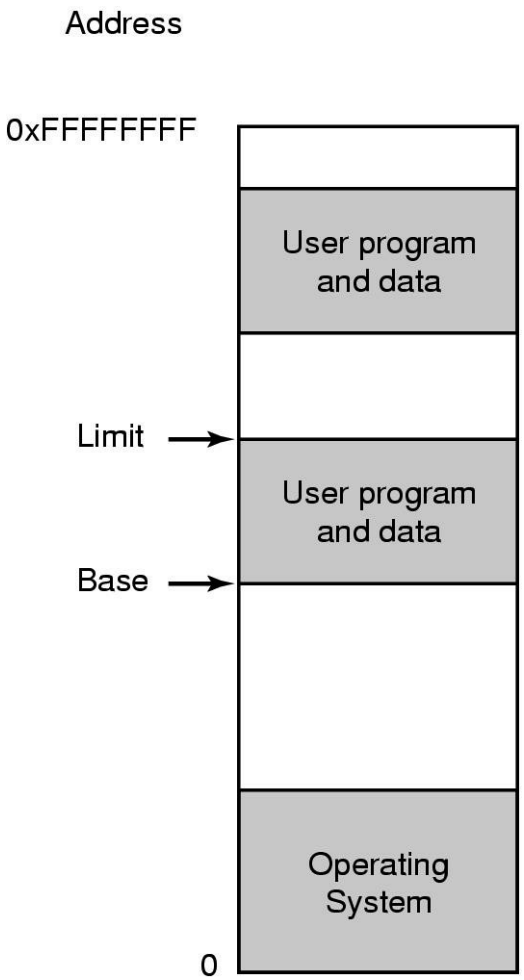
Multiprogramming: Relocation and Protection

Problem: where will program be loaded in memory?

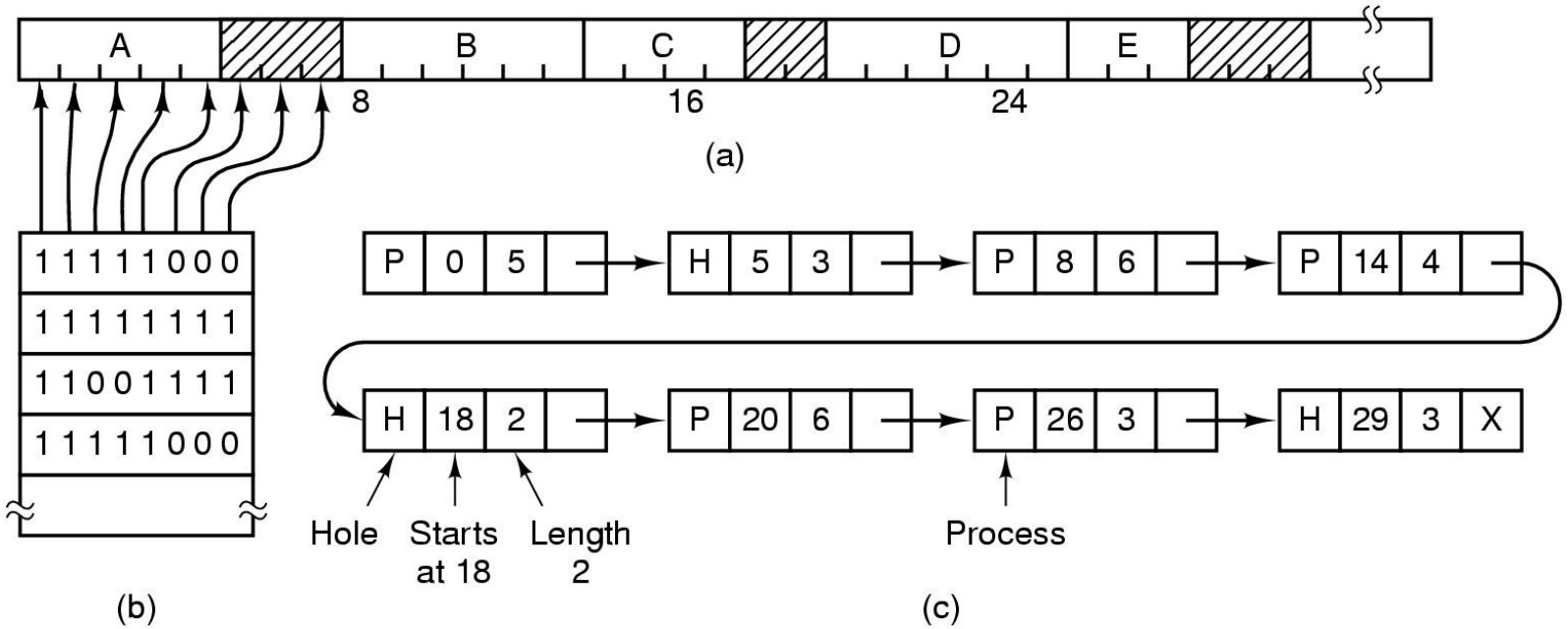
- location of variables and code routines cannot be absolute addresses
- a program must be kept out of other processes' partitions

Solution: use base and limit values

- (relative) address locations added to base value will map to physical address
- (relative) address locations larger than limit value is an error



Multiprogramming: Accounting and Fitting (memory management!)



Accounting methods:

a) Part of memory with 5 processes, 3 holes
(tick marks show allocation units;
shaded regions are free)

b) Corresponding bit map

c) Same information as a linked list

Fitting algorithms:

first fit : simple & quick

best fit : marginally better, but slow

worst fit : also not so good

Multiprogramming: more Fitting

(as said) Memory allocation changes with time as processes

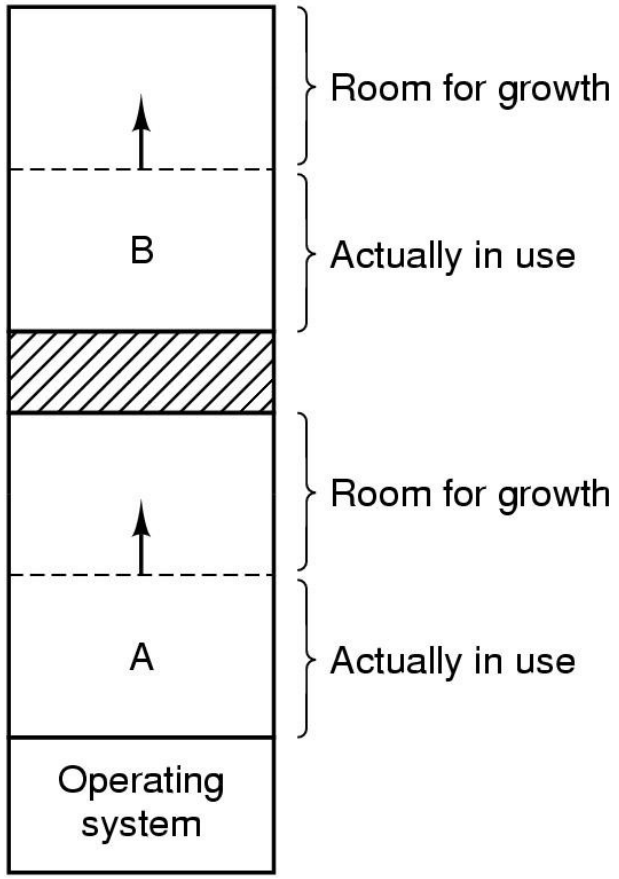
- come into memory, demand and liberate memory, leave memory...

→ Huge problem in multiuser, interactive systems!

If process do not "fit" in memory, Oper. Sys. can

- refuse or stop running it (*humm*...)
- reallocate all or some processes
- use *virtual* memory
  - swap processes between disk & memory (wholly or partially)

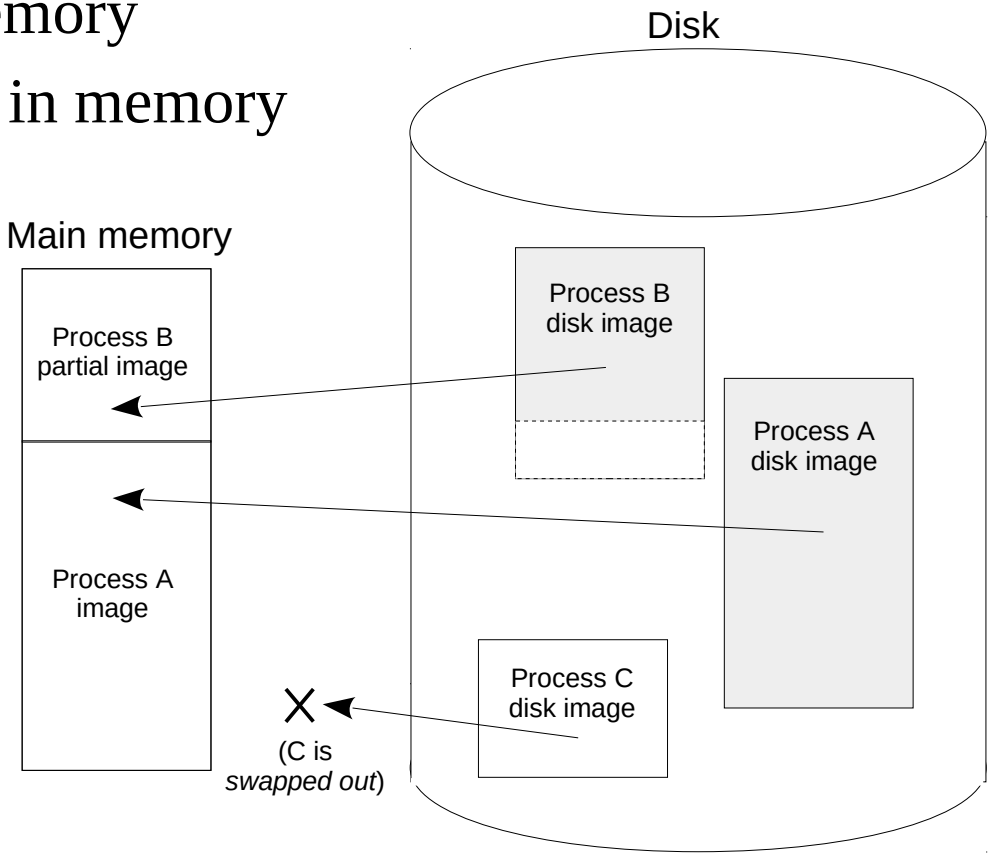
→ (needs much more) memory management!



Multiprogramming: swapping!

Virtual memory: program in computer memory and image on disk

- processes's images
  - are all in memory
  - are partially in memory

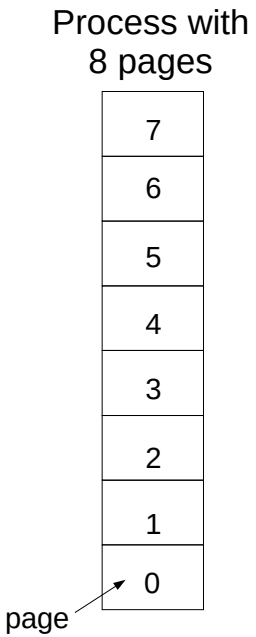


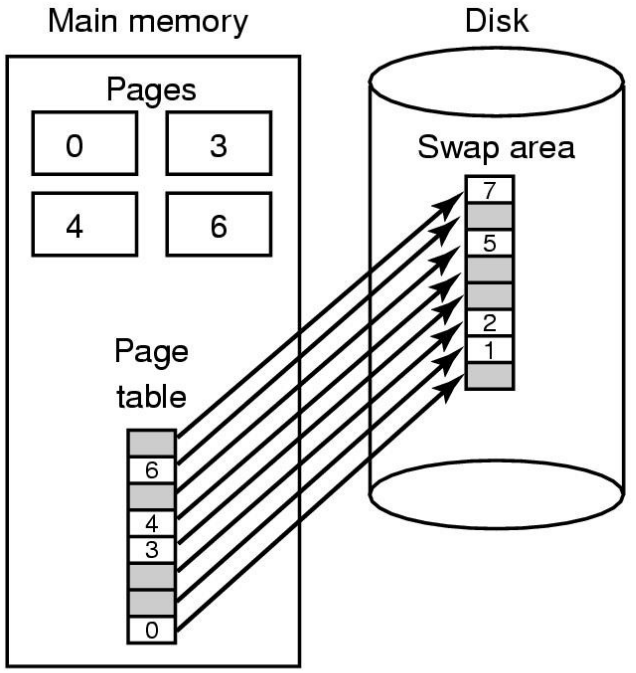
Virtual Memory with paging

Paging:

e.g. of 4096 bytes

- processes' address space is divided into pages
  - e.g. a 4 MiB ($4*2^{20}$ B) process takes 1024 pages of size 4KiB
- in general, some processes will be in main memory, others will not:
    - will be on disk (swapped)





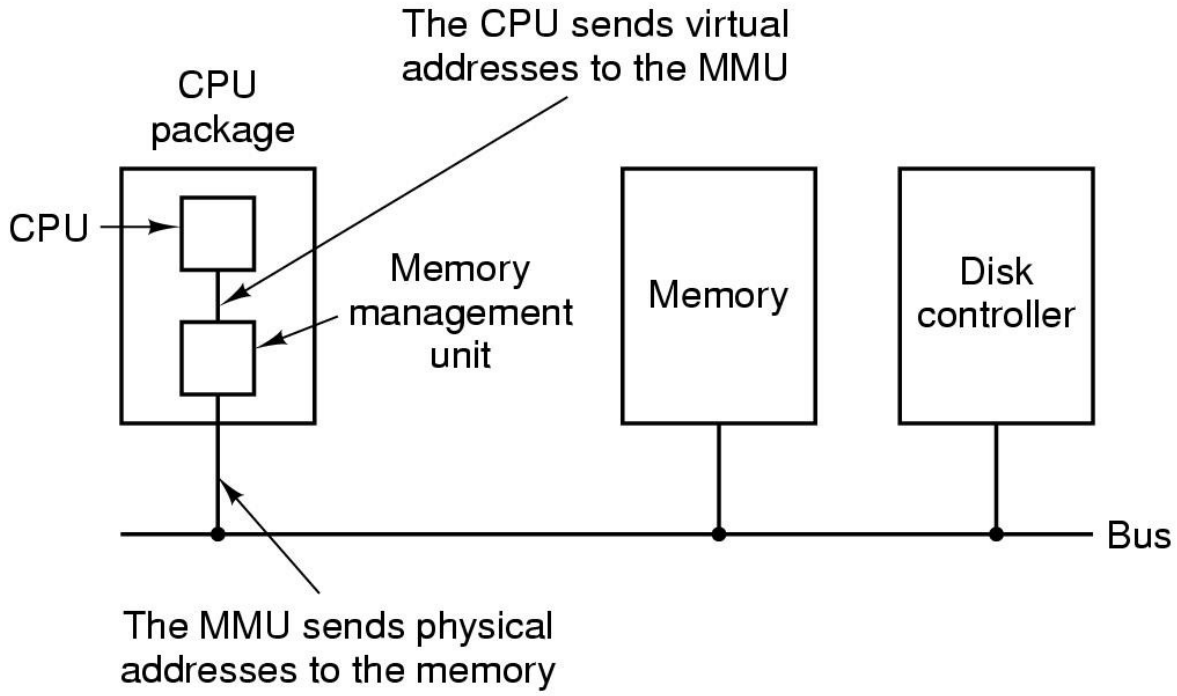
Paging (2): the Memory Management Unit

Memory Management Unit (MMU):

- position: "near" CPU
- function: conversion of Virtual and Physical addresses

seen by process

in real memory



Paging (3): Memory Management Unit function

Virtual – Physical addrs

(series of pages – series of frames)

Typical page (frame) size: 4096 B

Example: compiler generates

```
MOV REG, 4111
```

Virtual address:

```
4111 = 1*4096 + 15 (bytes)
```

4111 / 4096 = 1 (page no.)

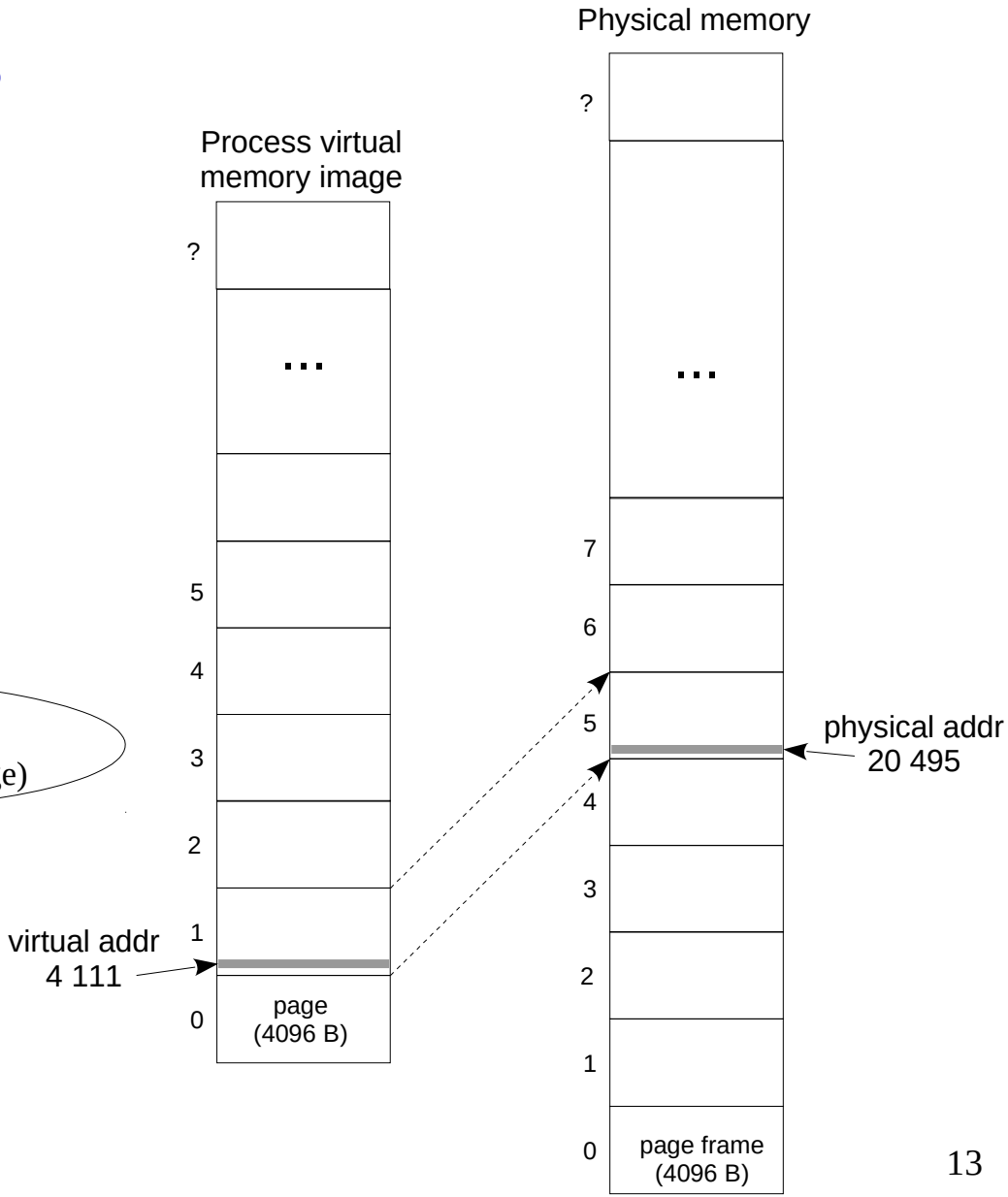
4111 % 4096 = 15 (offset in page)

Physical address:

addr (frame of page 1) + 15 B

```
20495 = 5*4096 + 15 (bytes)
```

page	frame
0	...
1	5
...	...



Paging (4): Memory translation & Page table

Virtual – Physical addrs

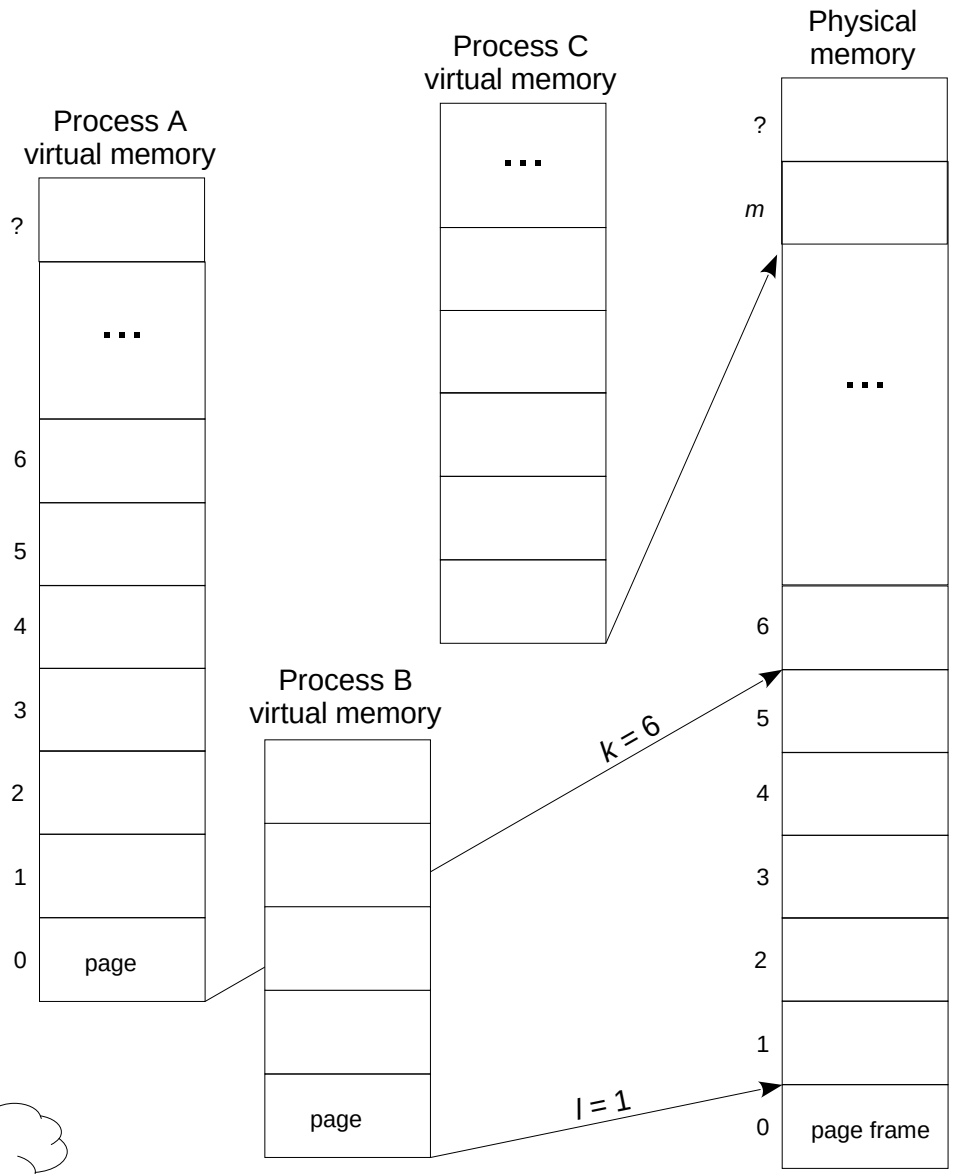
A Page (mapping) Table is used for all pages:

page	frame
0	...
1	5
...	...

But why not just map Page 0 of a process to its physical addr?

Process page 0	Physical addr
A	*k**4096
B	*l**4096
C	*m**4096
...	...

(*k*, *l*, *m* – positive integers)

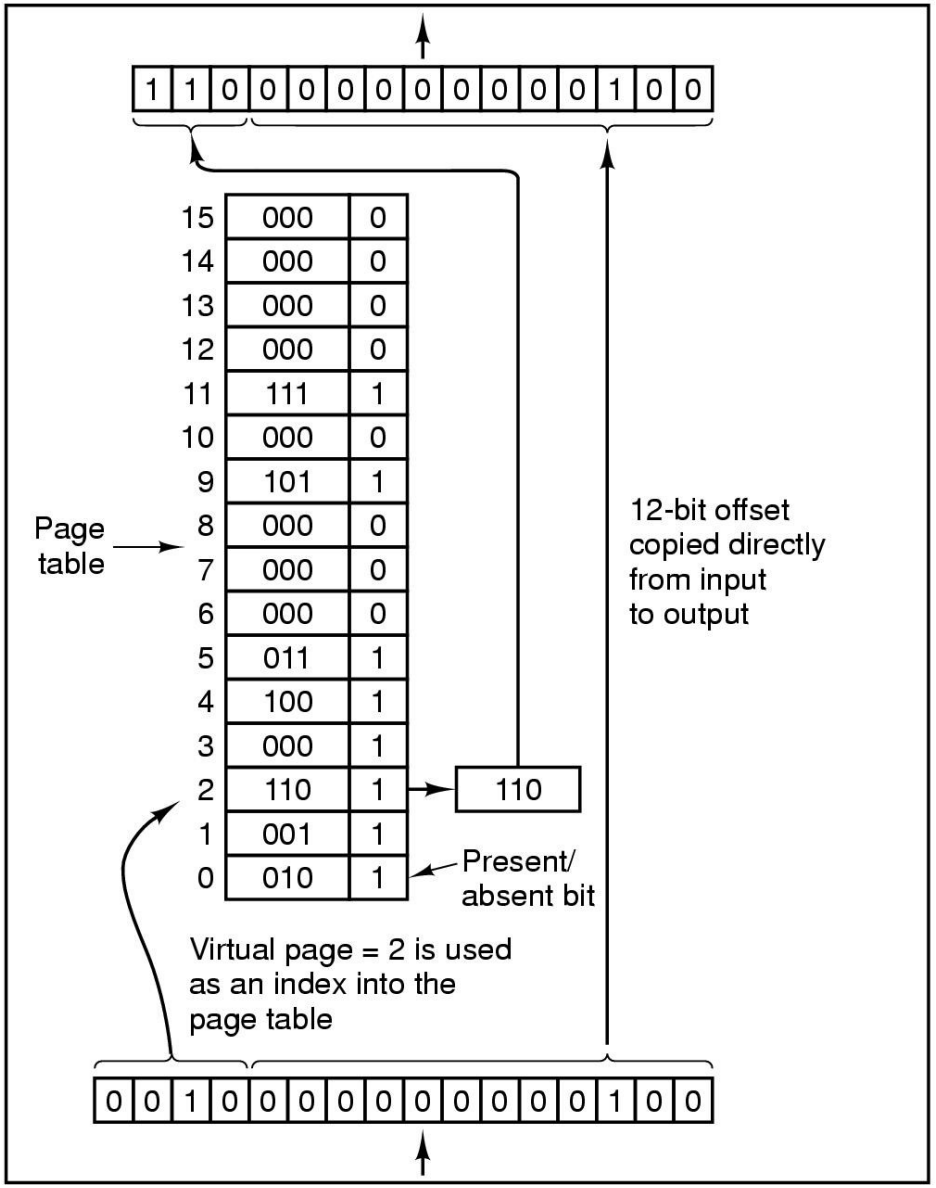


FE4

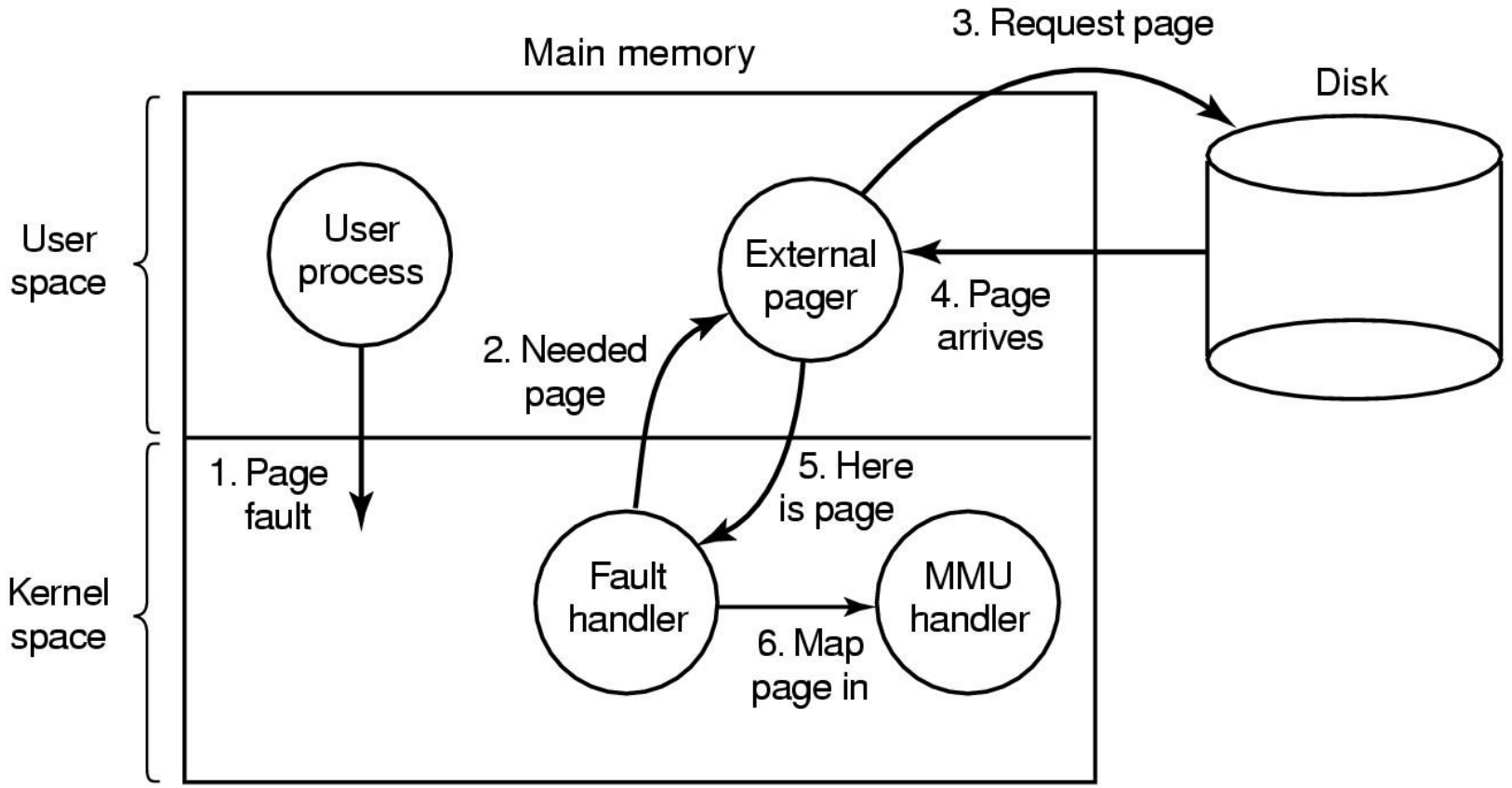
Paging (5): Page Table, address translation in binary

Example of MMU operation:

- 16 pages
- 8 page frames
- (page/frame: 4 KiB)
- 8196 --> 24580
  - 8196 / 4096 = 2
    - 2 --> 6 (Pg Tbl)
    - 6 * 4096 = 24576
  - 8196 % 4096 = 4 (offset)
    - 24576 + 4 = 24580



Paging (6): page fault processing

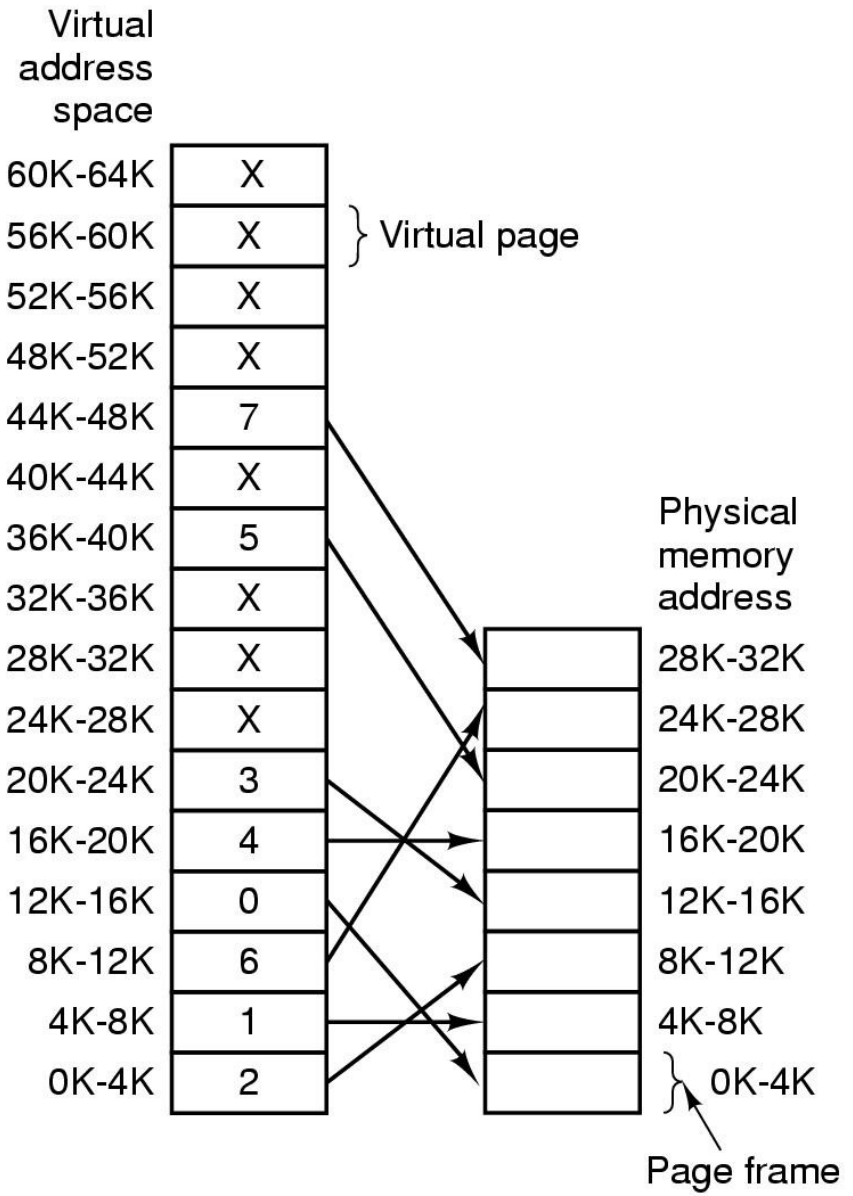


The general procedure for processing a page in fault.

Paging (7): page fault handling

Page Table and page fault processing

- X represents pages that are not in memory (are in disk)
- if a page with X is accessed, a page fault (interrupt) traps to the Oper. Syst. (1.):
  - a little used page frame is picked, written to disk if necessary, and X marked in its Page Table entry (2.)
  - referenced page is brought from disk, put in page frame just made available and Page Table entry is updated to point to this frame (3. 4. 5. 6.)
  - memory instruction is repeated



Paging (8): large page tables

Prob: with 32 b virtual addresses, and 4 KiB pages, there will be 1 (binary) million pages (*for each process*)!

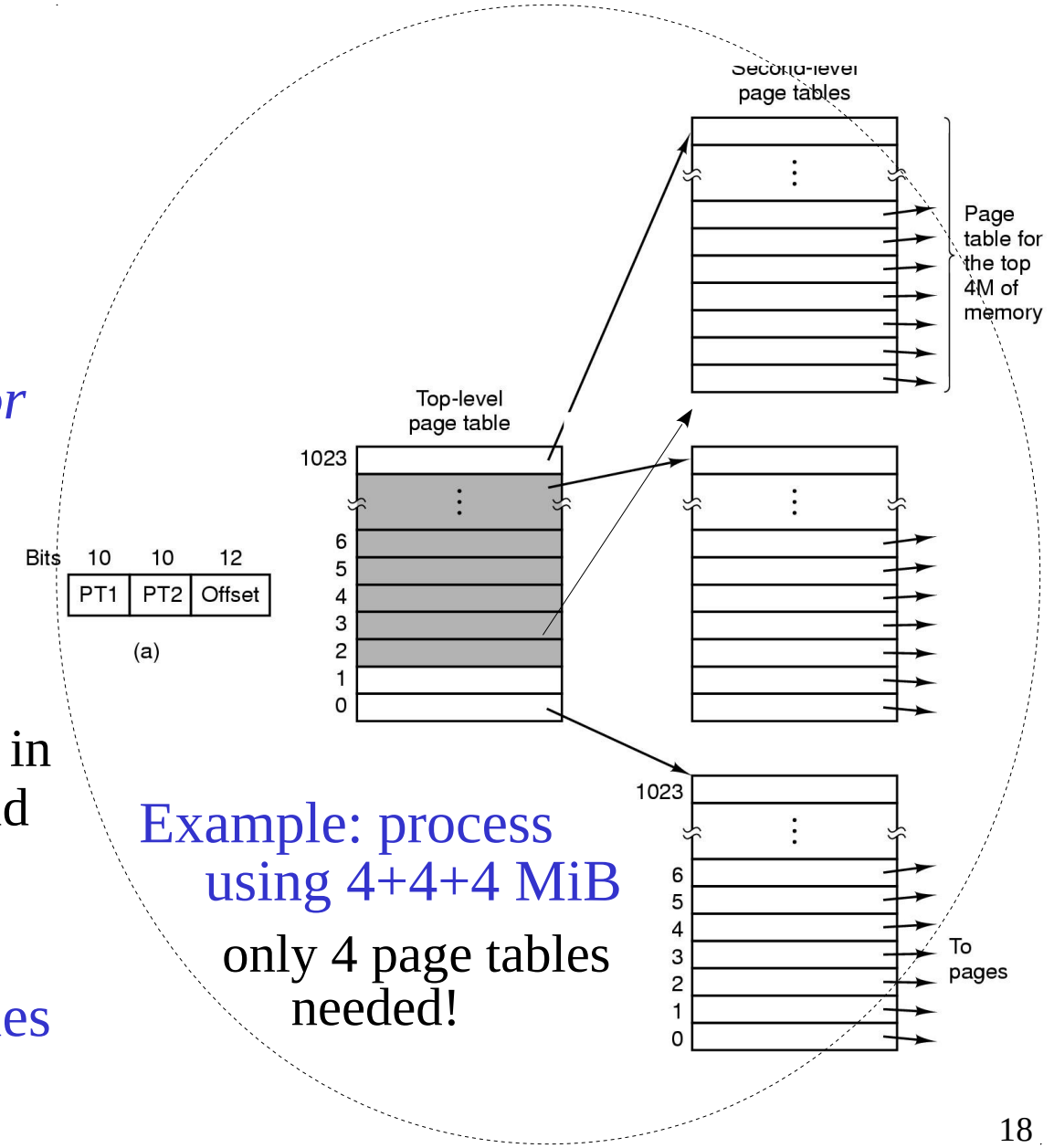
32 – 12 = 20 b :

2^{20} = 1 Mebi pages

Sol: multi-level page tables

- virtual address is divided in more than one table field
- each field refers to a different page table

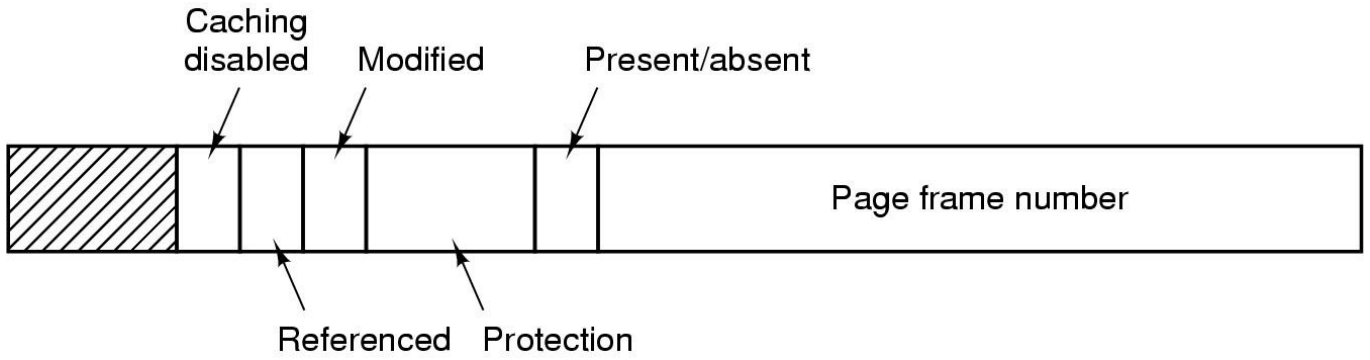
E.g. Linux uses 3-level tables



Example: process using 4+4+4 MiB

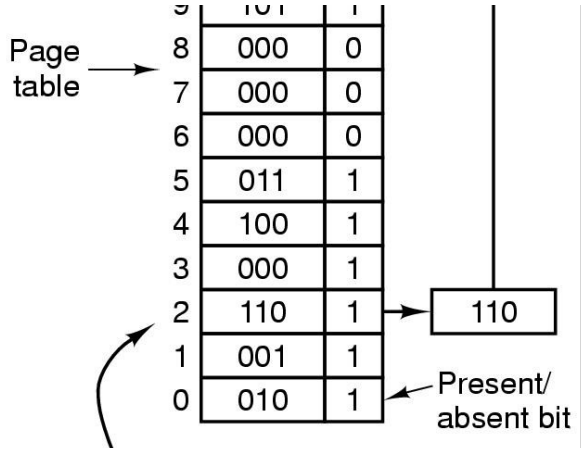
only 4 page tables needed!

Paging (7): page table entry



Typical page table entry:

- Page frame number: *n* bits
  (e.g. *n* = 18 w/ 1GiB of physical memory)
- Present/absent : 1b (page hit or page fault?)
- Protection: 3b (RWX)
- Modified: 1b (dirty bit)
- Referenced: 1b (for page replacement algorithms)
- Caching disabled: 1b (for I/O memory mapped devices)



Disk Page addresses are not here:
(other Oper. Syst. tables...)

Why 18 bits, when 30 bits are necessary for addressing all memory?...

Page Replacement Algorithms (1)

Page fault forces decision

- which page to remove (making room for incoming one)
- modified page must first be saved!

Better not to choose an often used page

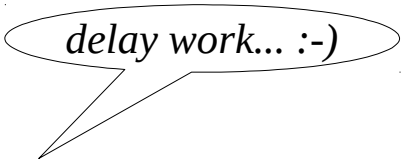
- will probably need to be brought back in soon
- concept of program's *working set*!

Replace page needed at the farthest point in future

- **optimal** but **unrealizable**
- estimate by logging page use on previous runs of process
  - impractical! [Why?]

Page Replacement Algorithms (2)

Not Recently Used Page



- each page entry has Reference bit and Modified bit
- R = M = 0 -> removed first; ...; R = M = 1 -> removed last

FIFO

- maintain a linked list of all pages in order they came into memory
  - *variant*: give old page a second chance (clock algorithm)

Least Recently Used

- maintain linked list of pages, most recently used at front
- update this list for every memory reference !
  - *variant*: periodic measure of reference count (LFrequentlyU)

Working Set based

- locality of references...

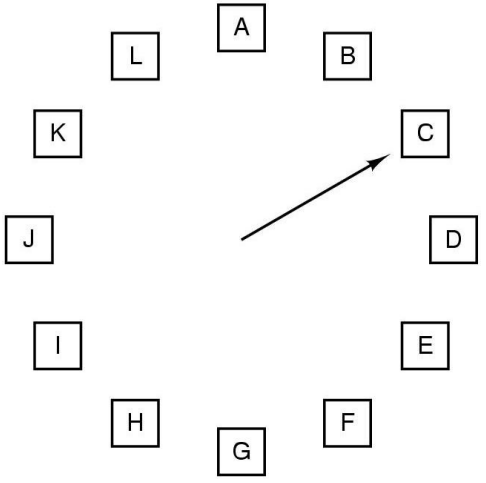
Page Replacement Algorithms (3)

Examples:

- NRU, Not Recently Used Page
  - Reference, Modified bits are set by Memory Management Unit;
  - Periodically (e.g. each clock tick), a Page Table Daemon clears R

Eviction Podium	R	M
1st	0	0
2nd	0	1
3rd	1	0
4th	1	1

- Clock page replacement (FIFO, 2nd chance)
  - R & M bits are set by MMU;
  - Pages are kept in circular buffer, ordered by age;
  - When page fault occurs, page pointed to by clock hand is inspected:
    - if R=0, page is substituted by new one
    - if R=1, R is cleared and hand is advanced



Paging: page size

Page sizes: 512B - 64KiB

Advantages of smaller size

- less *internal* fragmentation
- better fit for various data structures, code sections
- less unused program in memory

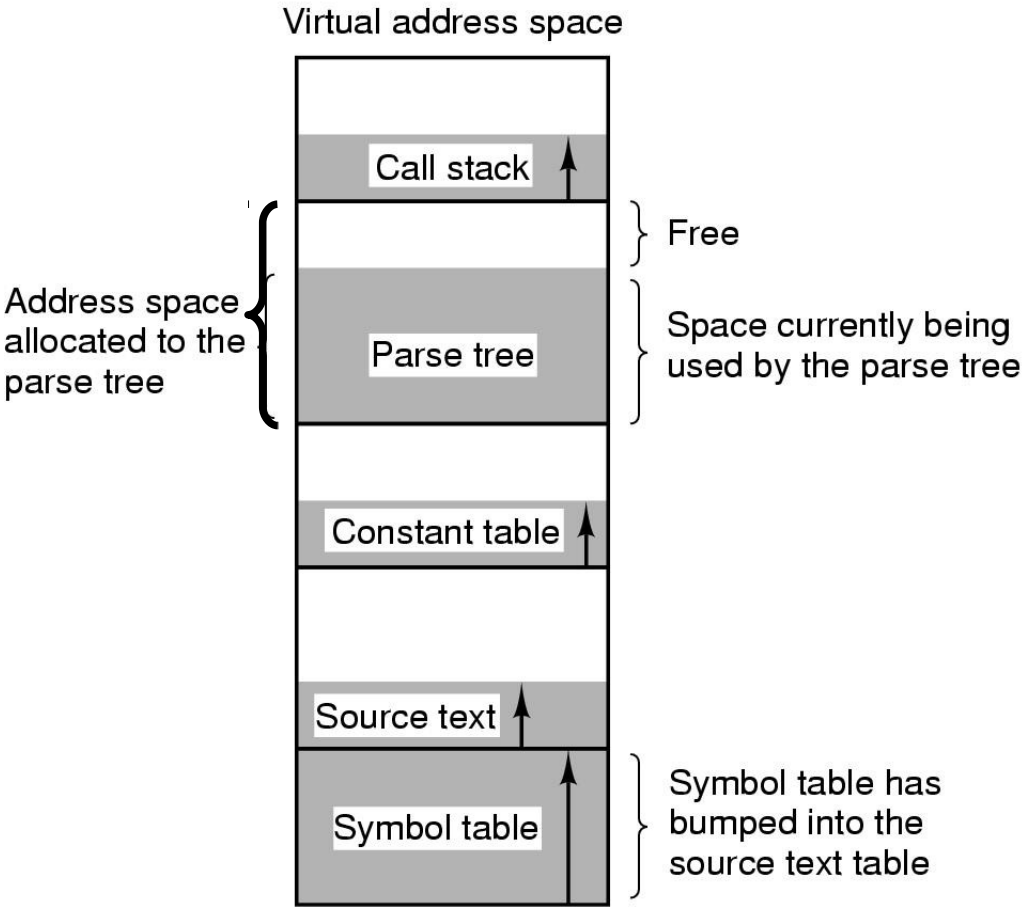
Disadvantages of smaller size

- programs need many pages, larger page tables

Typical size

- 4 KiB
- (Linux, MsWindows, 32-b, 64-b architectures)

Virtual Memory: Segmentation (1)

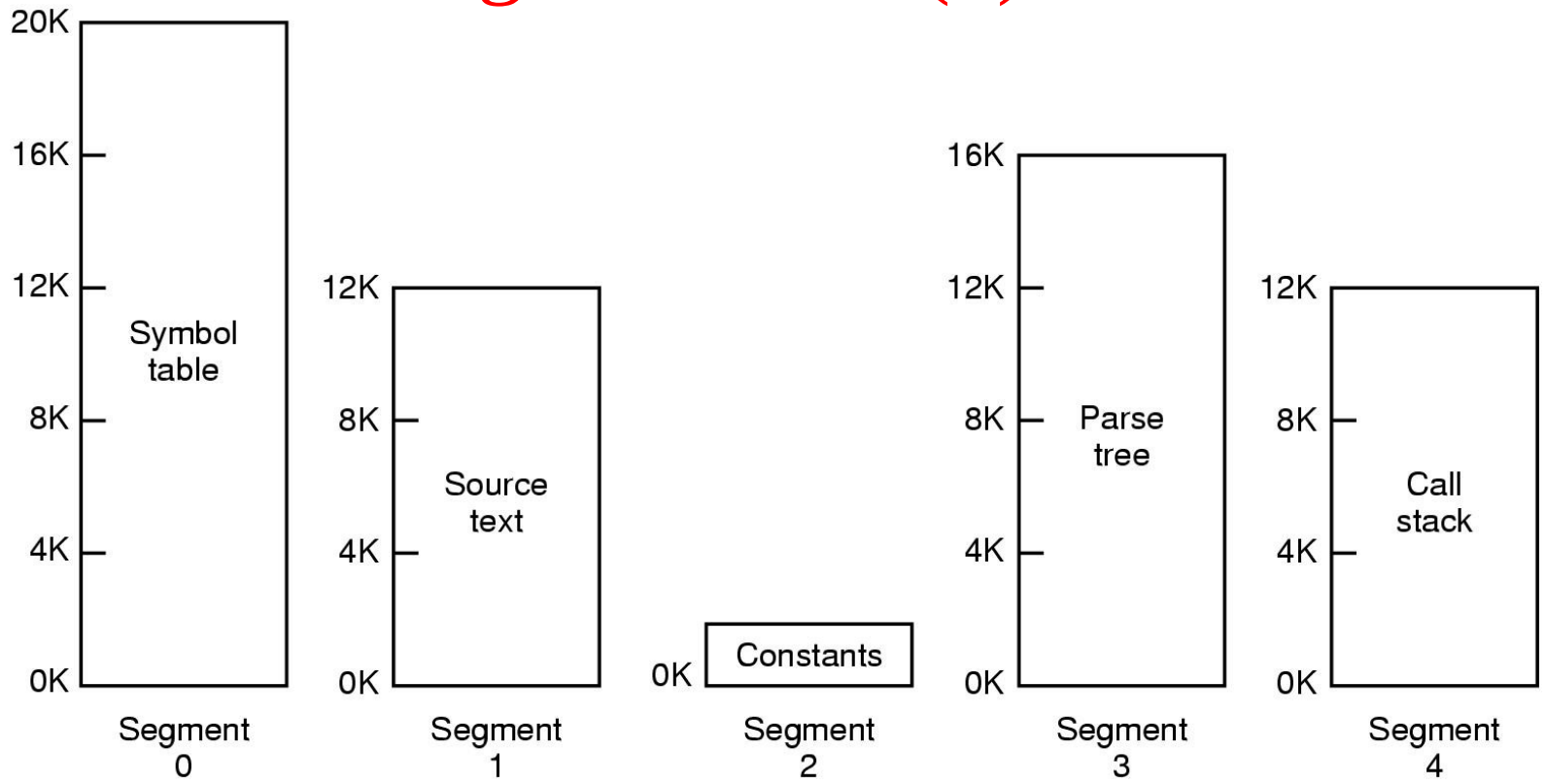


Example:

compiler runs in a one-dimensional address space

- growing tables may bump into another
- address format : *number*

Segmentation (2)

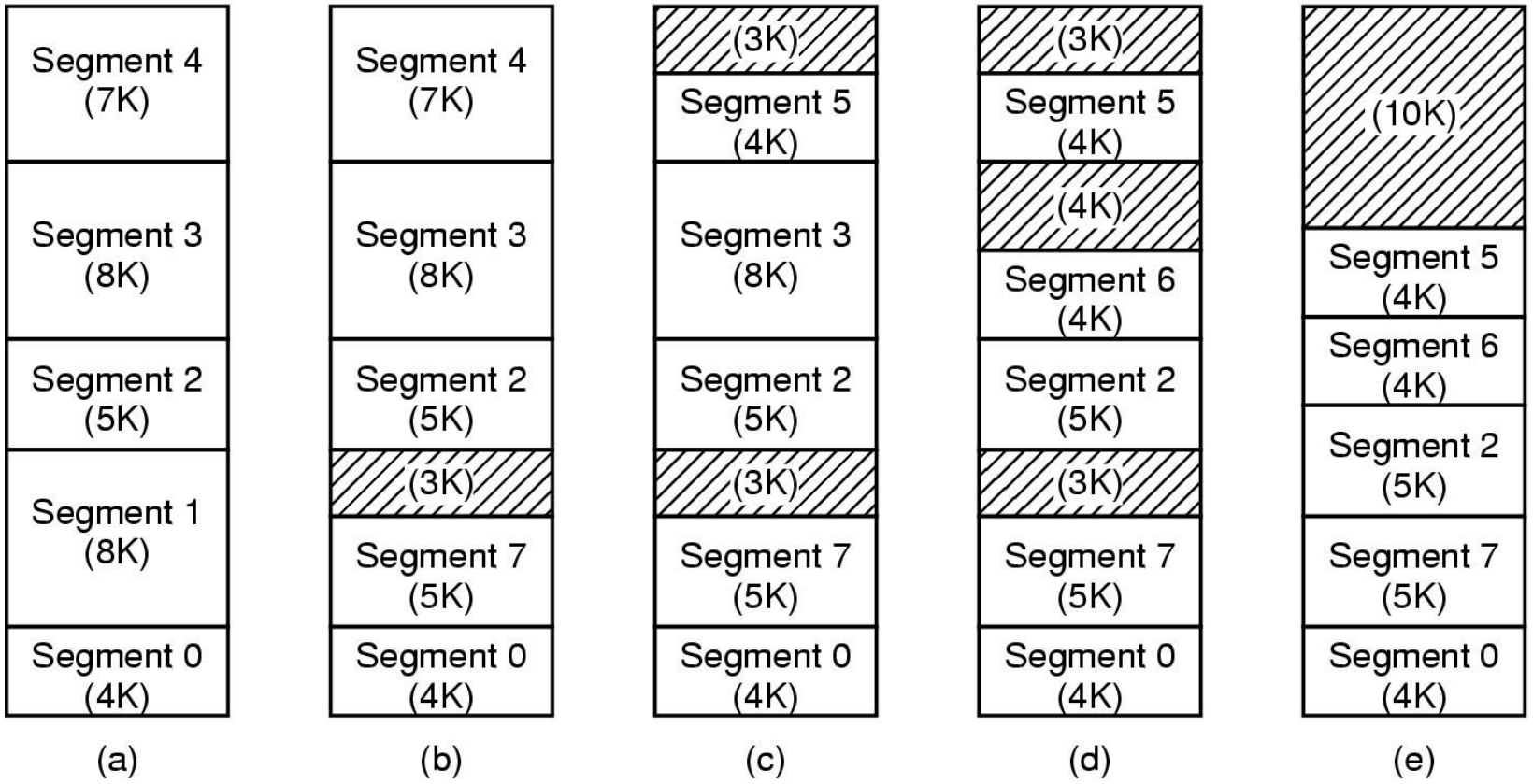


Example:

compiler runs in a multi-dimensional address space

- each table grows or shrinks, easily and independently
- each segment may be protected differently (R, W, X...)
- address format : *segment:offset*

Segmentation (3): external fragmentation



Segments have variable sizes:

- external fragmentation!

Segmentation *versus* Paging

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Note: there is also segmentation *with* paging!