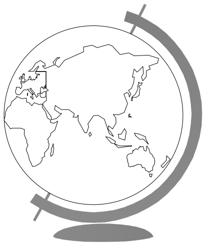
Operating Systems

Memory Management

(Chapter 8: 8.1-8.6)

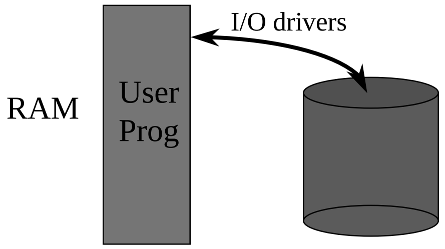
Overview

• Provide Services	(done)
– processes	(done)
– files	(after memory management)
• Manage Devices	
– processor	(done)
– memory	(next!)
– disk	(done after files)



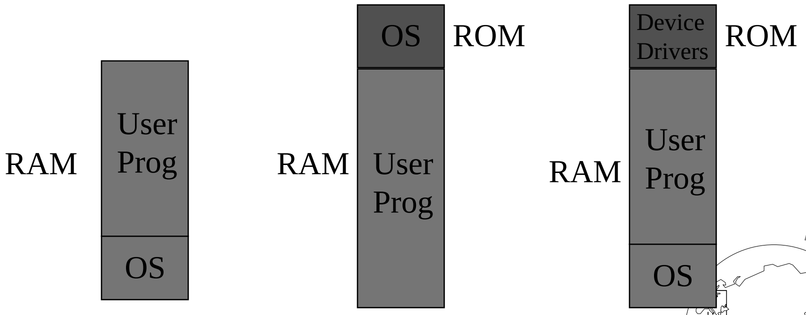
Simple Memory Management

- One process in memory, using it all
  - – each program needs I/O drivers
  - – until 1960



Simple Memory Management

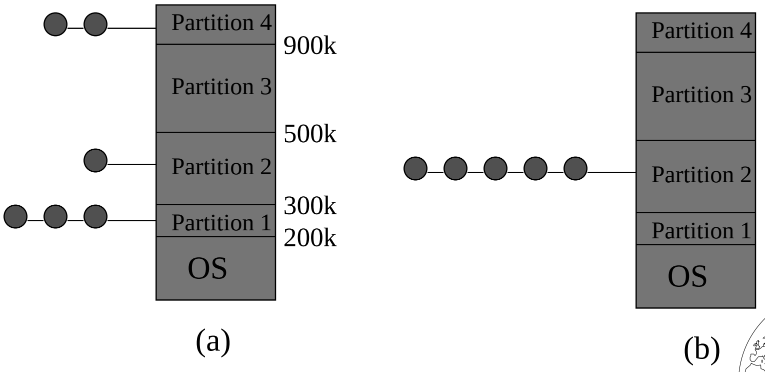
- Small, protected OS, drivers
  - – DOS



- “Mono-programming” -- No multiprocessing!
  - Early efforts used “Swapping”, but slooooow

Multiprocessing w/Fixed Partitions

Simple!

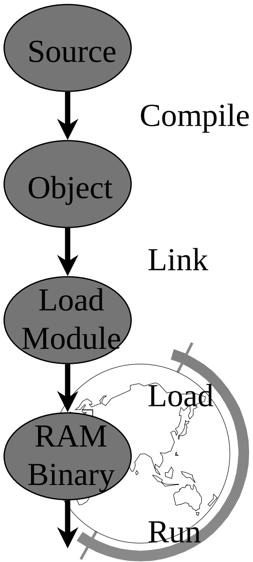


- Unequal queues
- Waste large partition
- Skip small jobs

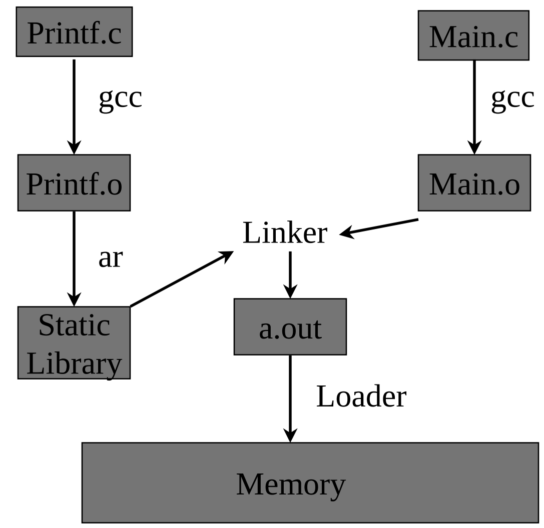
Hey, processes can be in different memory locations!

Address Binding

- Compile Time
  - maybe absolute binding (`.com`)
- Link Time
  - dynamic or static libraries
- Load Time
  - relocatable code
- Run Time
  - relocatable memory segments
  - overlays
  - paging



Normal Linking and Loading



Printf.c

gcc

Printf.o

ar

Static Library

Main.c

gcc

Main.o

Linker

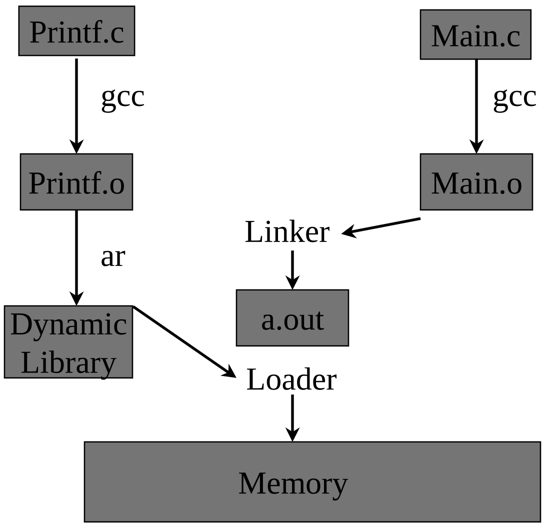
a.out

Loader

Memory

X Window code:
- 500K minimum
- 450K libraries

Load Time Dynamic Linking



Printf.c

gcc

Printf.o

ar

Dynamic Library

Main.c

gcc

Main.o

Linker

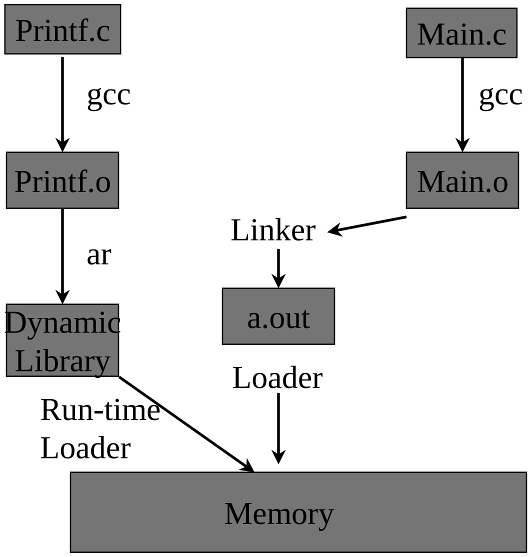
a.out

Loader

Memory

- Save disk space.
- Libraries move?
- Moving code?
- Library versions?
- Load time still the same.

Run-Time Dynamic Linking



Save disk space.
Startup fast.
Might not need all.

Memory Linking Performance Comparisons

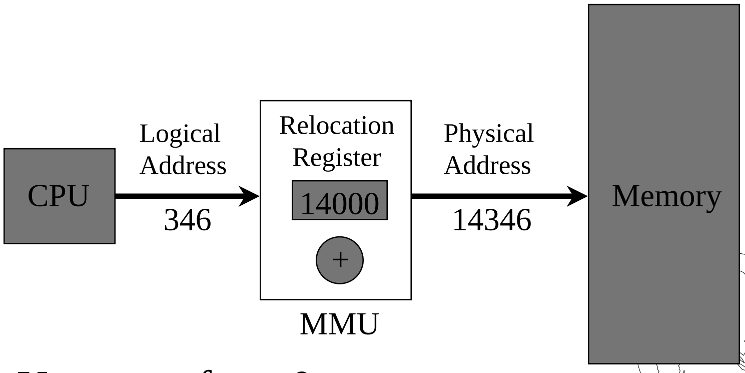
Linking Method	Disk Space	Load Time	Run Time (4 used)	Run Time (2 used)	Run Time (0 used)
Static	**3Mb**	**3.1s**	**0**	**0**	**0**
Load Time	**1Mb**	**3.1s**	**0**	**0**	**0**
Run Time	**1Mb**	**1.1s**	**2.4s**	**1.2s**	**0**

Design Technique: Static vs. Dynamic

- Static solutions
  - compute ahead of time
  - for predictable situations
- Dynamic solutions
  - compute when needed
  - for unpredictable situations
- Some situations use dynamic because static too restrictive (`malloc`)
- ex: memory allocation, type checking

Logical vs. Physical Addresses

- Compile-Time + Load Time addresses same
- Run time addresses different



CPU

Logical Address

346

Relocation Register

14000

+

MMU

Physical Address

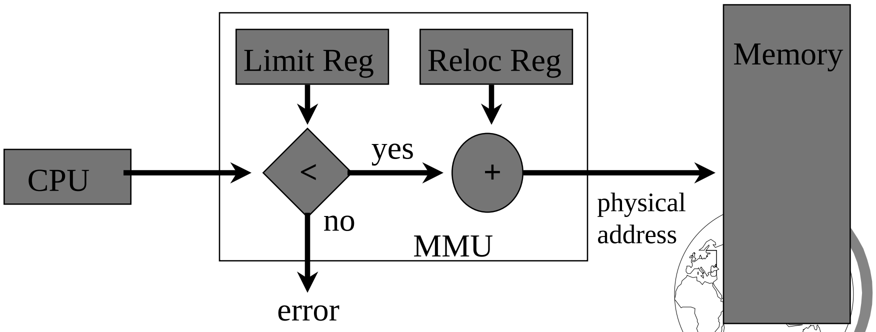
14346

Memory

- User goes from 0 to *max*
- Physical goes from `R`+0 to `R`+*max*

Relocatable Code Basics

- Allow *logical* addresses
- Protect other processes

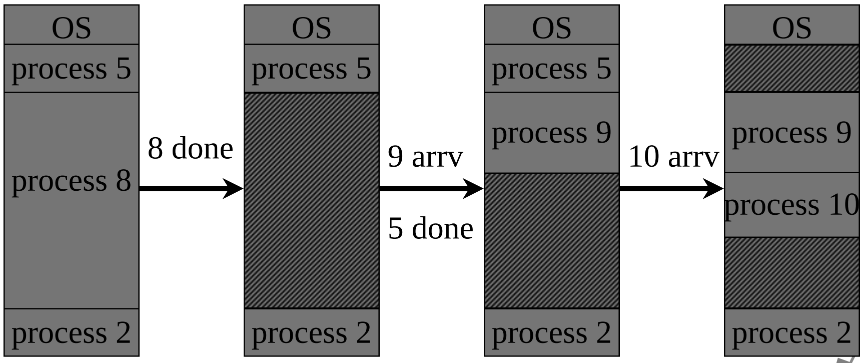


- Addresses must be contiguous!

Variable-Sized Partitions

- Idea: want to remove “wasted” memory that is not needed in each partition
- Definition:
  - *Hole* - a block of available memory
  - scattered throughout physical memory
- New process allocated memory from hole large enough to fit it

Variable-Sized Partitions



OS

process 5

process 8

process 2

8 done

OS

process 5

process 2

9 arrv

5 done

OS

process 5

process 9

process 2

10 arrv

OS

process 9

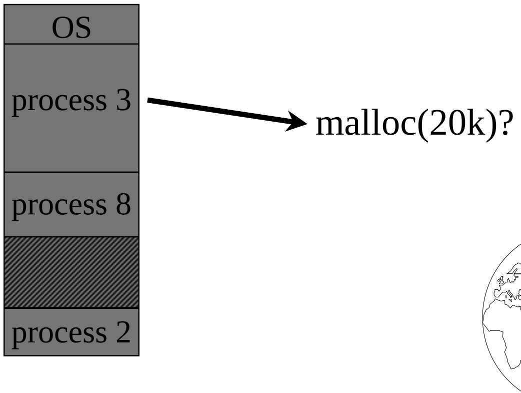
process 10

process 2

- OS keeps track of:
  - allocated partitions
  - free partitions (holes)
  - queues!

Memory Request?

- What if a request for additional memory?



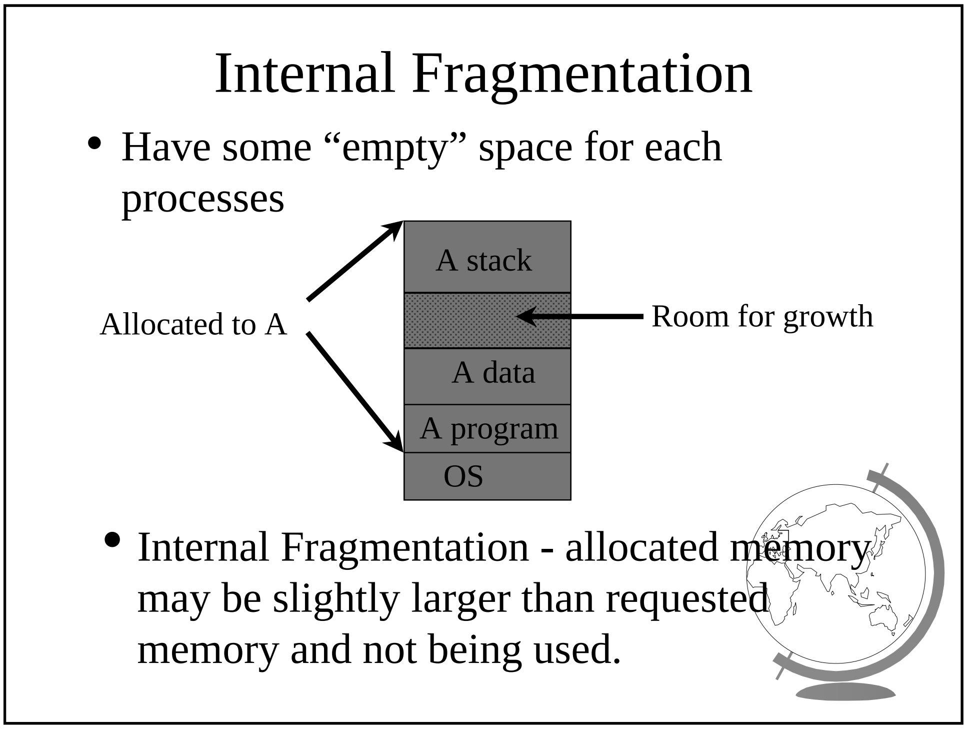
OS

process 3

process 8

process 2

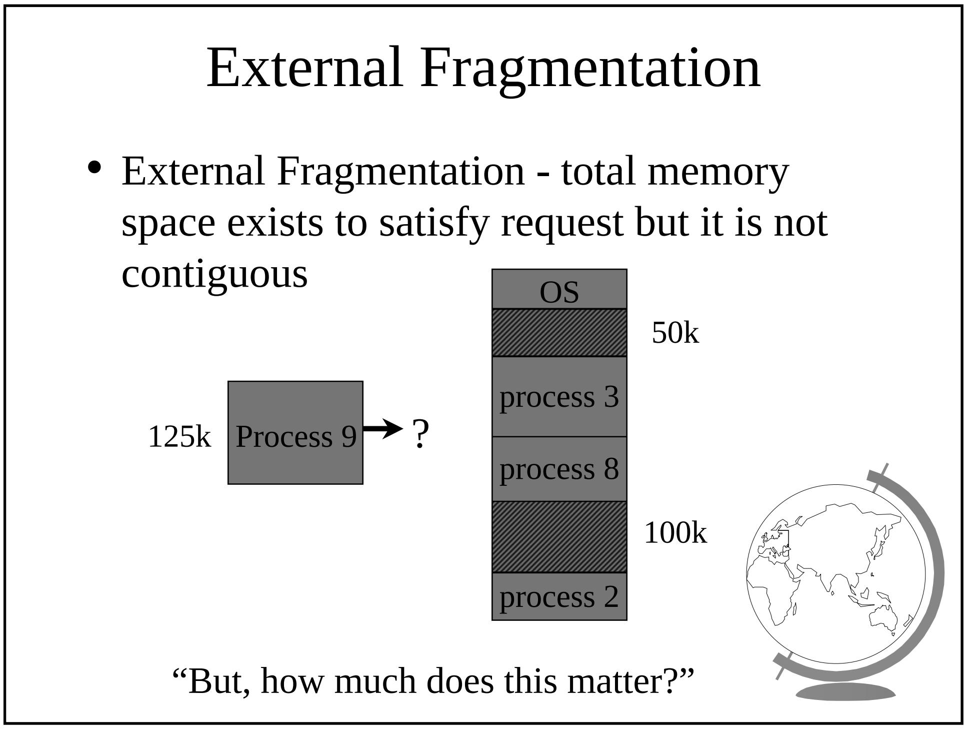
malloc(20k)?



Internal Fragmentation

- Have some “empty” space for each processes

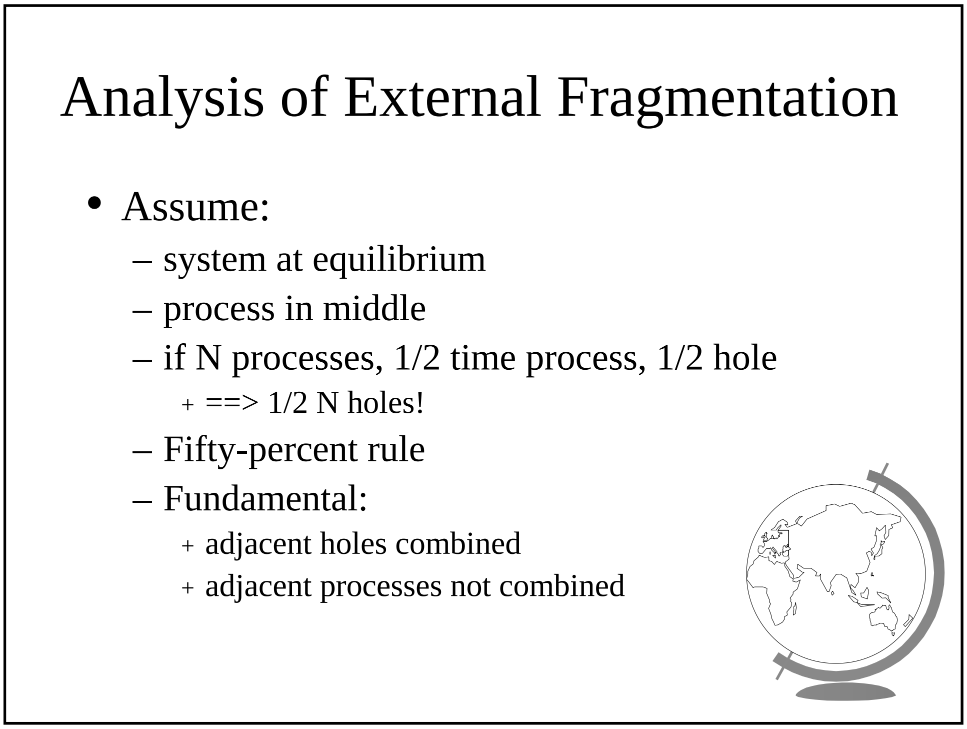
- Internal Fragmentation - allocated memory may be slightly larger than requested memory and not being used.



External Fragmentation

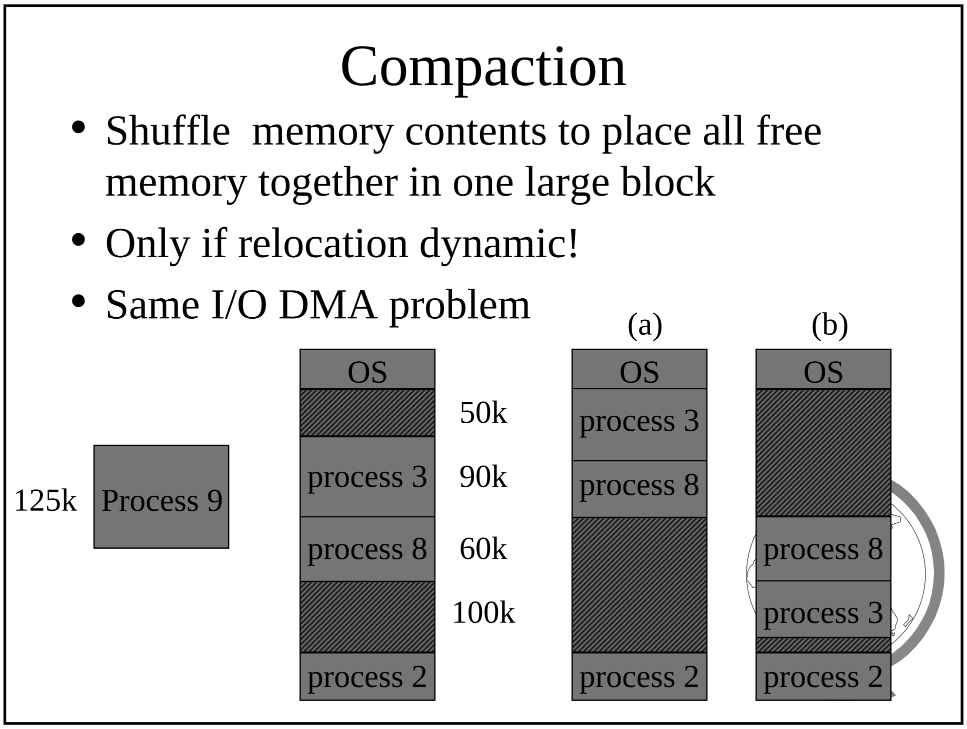
- External Fragmentation - total memory space exists to satisfy request but it is not contiguous

“But, how much does this matter?”



Analysis of External Fragmentation

- Assume:
  - system at equilibrium
  - process in middle
  - if N processes, 1/2 time process, 1/2 hole
    - ==> 1/2 N holes!
  - Fifty-percent rule
  - Fundamental:
    - adjacent holes combined
    - adjacent processes not combined

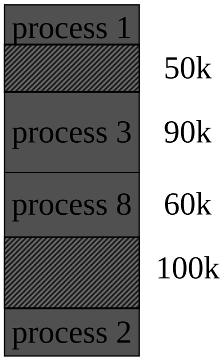


Compaction

- Shuffle memory contents to place all free memory together in one large block
- Only if relocation dynamic!
- Same I/O DMA problem

125k
Process 9
OS
50k
process 3
90k
process 8
60k
100k
process 2
(a)
OS
process 3
process 8
process 2
(b)
OS
process 8
process 3
process 2

Cost of Compaction



- 2 GB RAM, 10 nsec/access (cycle time)

  ➔ 5 seconds to compact!
- Disk much slower!

Solution?

- Want to minimize external fragmentation
  - Large Blocks
  - But internal fragmentation!
- Tradeoff
  - Sacrifice some internal fragmentation for reduced external fragmentation
  - *Paging*

Where Are We?

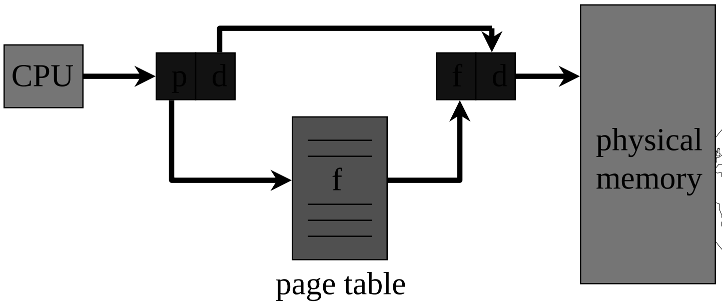
- Memory Management
  - – fixed partitions (done)
  - – linking and loading (done)
  - – variable partitions (done)
- Paging ←
- Misc

Paging

- Logical address space noncontiguous; process gets memory wherever available
  - – Divide physical memory into fixed-size blocks
    - + size is a power of 2, between 512 and 8192 bytes
    - + called *Frames*
  - – Divide logical memory into bocks of same size
    - + called *Pages*

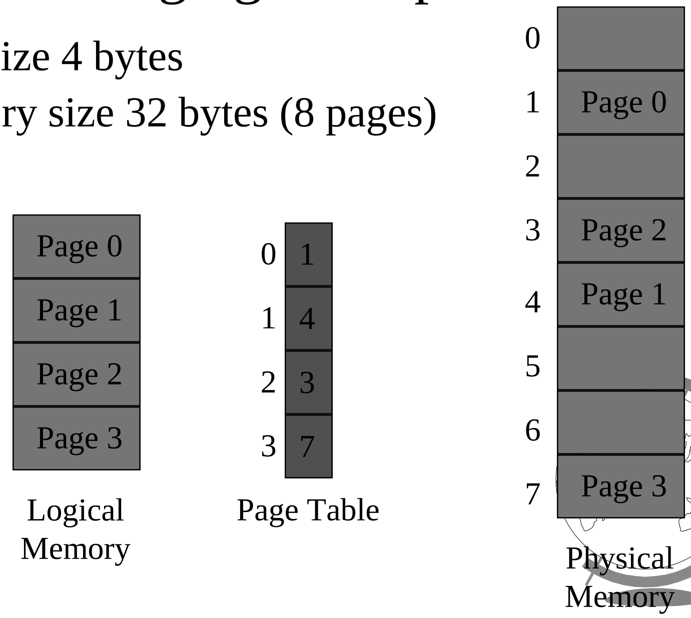
Paging

- Address generated by CPU divided into:
  - *Page number (p)* - index to page table
    - *page table* contains base address of each page in physical memory (frame)
  - *Page offset (d)* - offset into page/frame

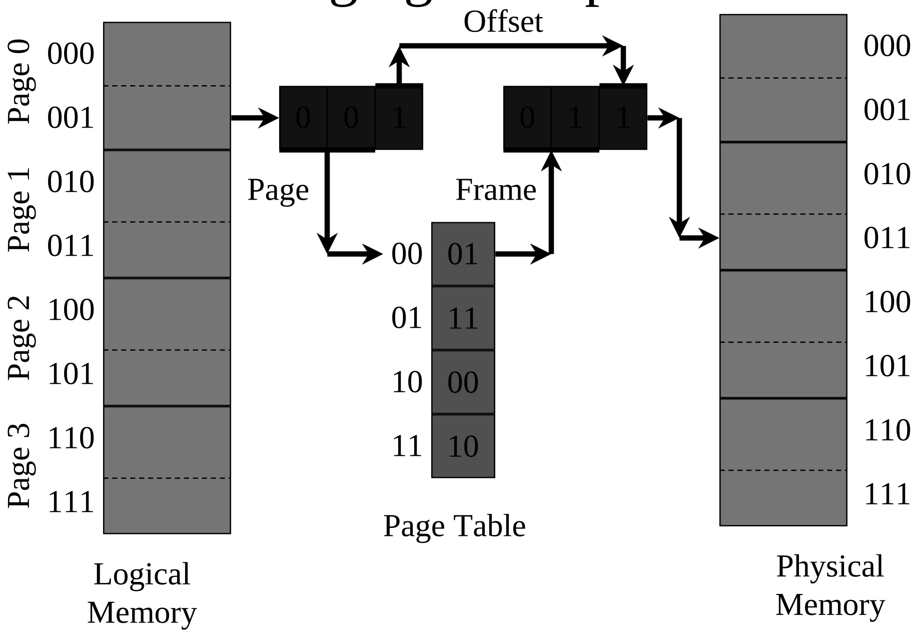


Paging Example

- Page size 4 bytes
- Memory size 32 bytes (8 pages)

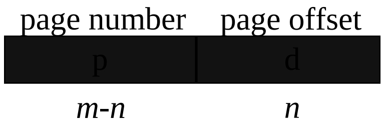


Paging Example

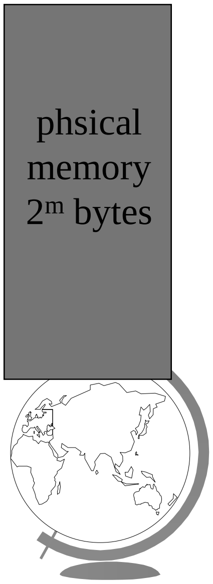


Paging Hardware

- address space 2^m
- page offset 2^n
- page number 2^{m-n}



- note: not losing any bytes!



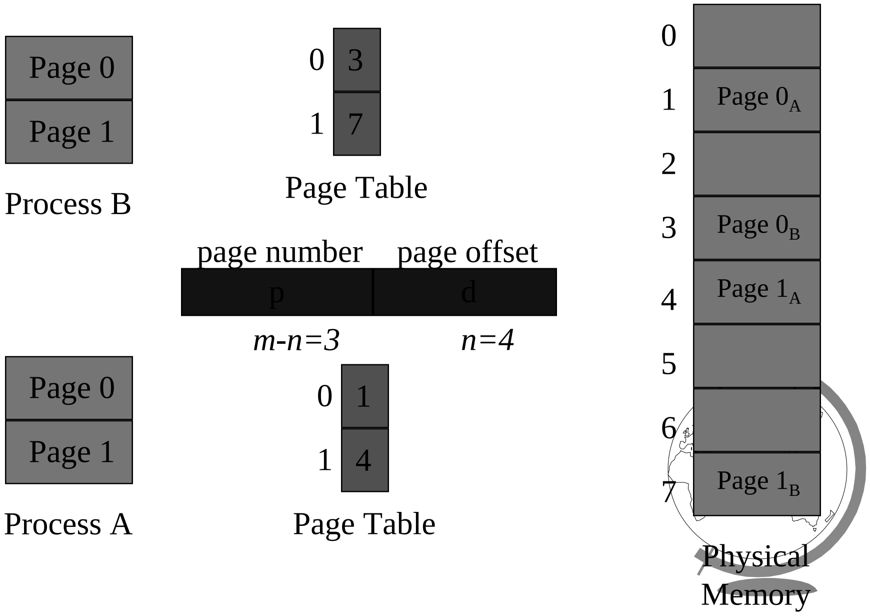
Paging Example

- Consider:
  - Physical memory = 128 bytes
  - Physical address space = 8 frames
- How many bits in an address?
- How many bits for page number?
- How many bits for page offset?
- Can a logical address space have only 2 pages? How big would the page table be?

Another Paging Example

- Consider:
  - 8 bits in an address
  - 3 bits for the frame/page number
- How many bytes (words) of physical memory?
- How many frames are there?
- How many bytes is a page?
- How many bits for page offset?
- If a process’ page table is 12 bits, how many logical pages does it have?

Page Table Example

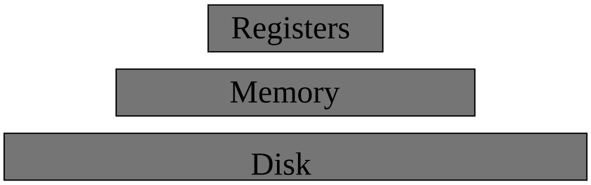


Paging Tradeoffs

- Advantages
  - no external fragmentation (no compaction)
  - relocation (now pages, before were processes)
- Disadvantages
  - internal fragmentation
    - consider: 2048 byte pages, 72,766 byte proc
      - 35 pages + 1086 bytes = 962 bytes
    - avg: 1/2 page per process
    - small pages!
  - overhead
    - page table / process (context switch + space)
    - lookup (especially if page to disk)

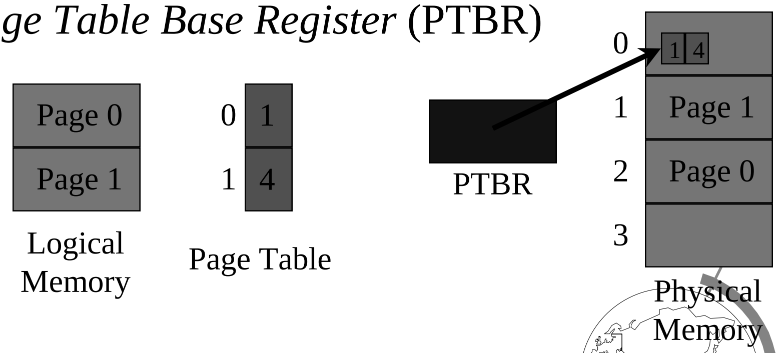
Implementation of Page Table

- Page table kept in registers
- Fast!
- Only good when number of frames is small
- Expensive!

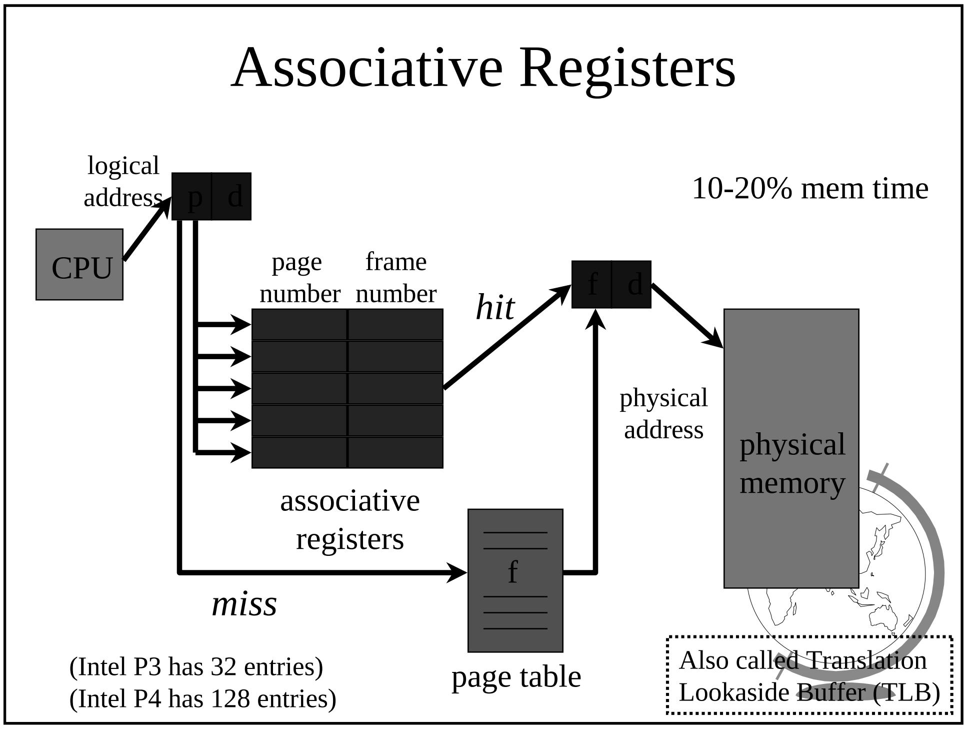


Implementation of Page Table

- Page table kept in main memory
- *Page Table Base Register* (PTBR)



- Page Table Length
- Two memory accesses per data/inst access.
  - Solution? *Associative Registers*

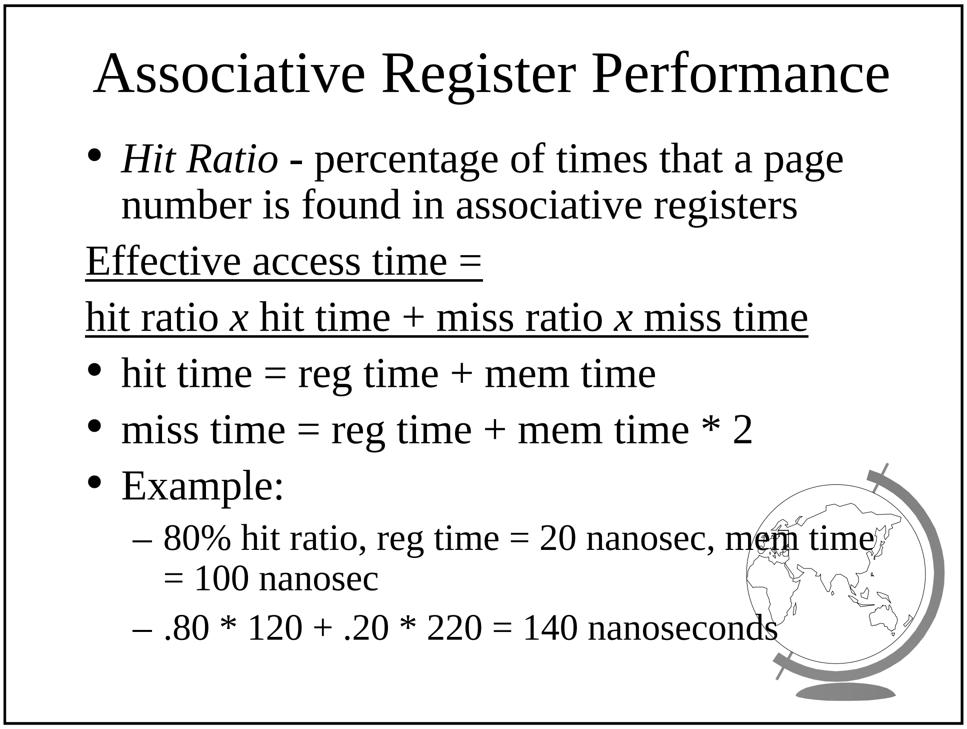


Associative Registers

10-20% mem time

(Intel P3 has 32 entries)
(Intel P4 has 128 entries)

Also called Translation Lookaside Buffer (TLB)



Associative Register Performance

- *Hit Ratio* - percentage of times that a page number is found in associative registers

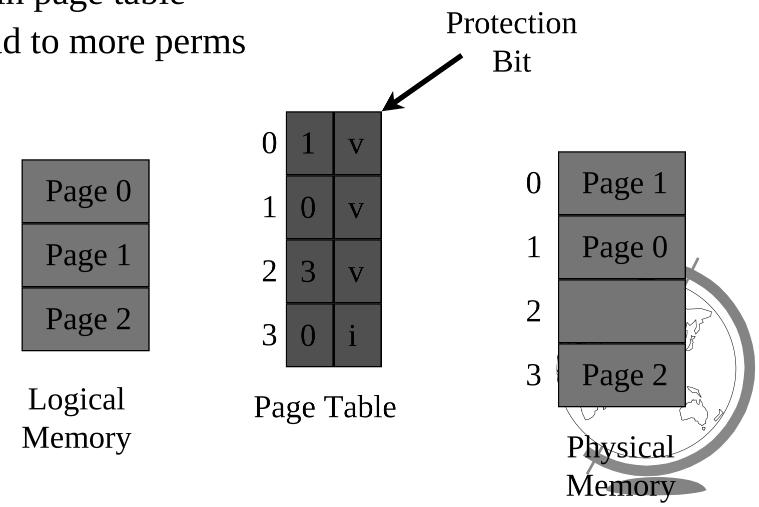
Effective access time =

hit ratio *x* hit time + miss ratio *x* miss time

- hit time = reg time + mem time
- miss time = reg time + mem time * 2
- Example:
  - 80% hit ratio, reg time = 20 nanosec, mem time = 100 nanosec
  - .80 * 120 + .20 * 220 = 140 nanoseconds

Protection

- Protection bits with each frame
- Store in page table
- Expand to more perms



Protection Bit

Logical Memory

Page 0	
Page 1	
Page 2	

Page Table

0	1	v
1	0	v
2	3	v
3	0	i

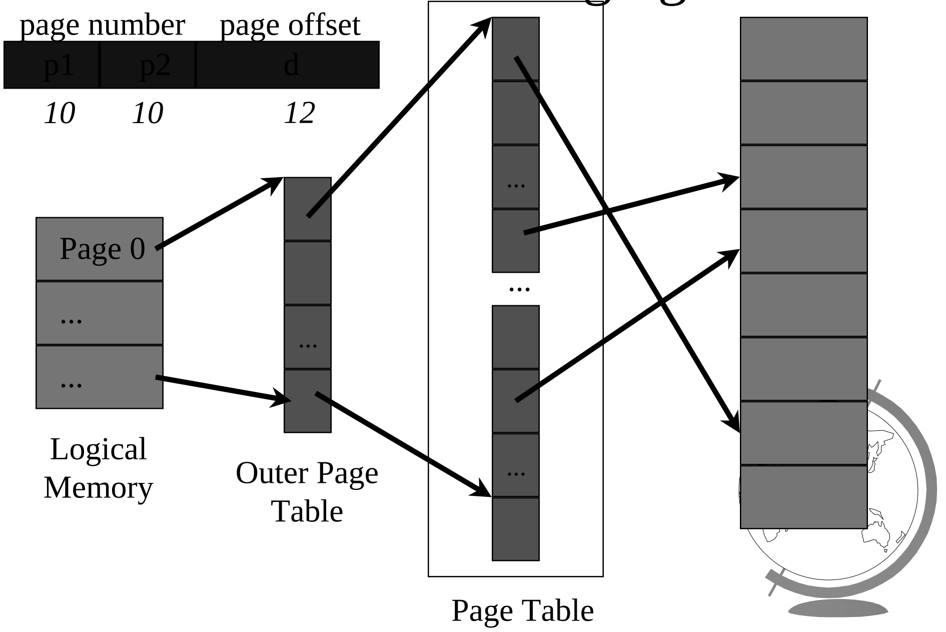
Physical Memory

0	Page 1
1	Page 0
2	
3	Page 2

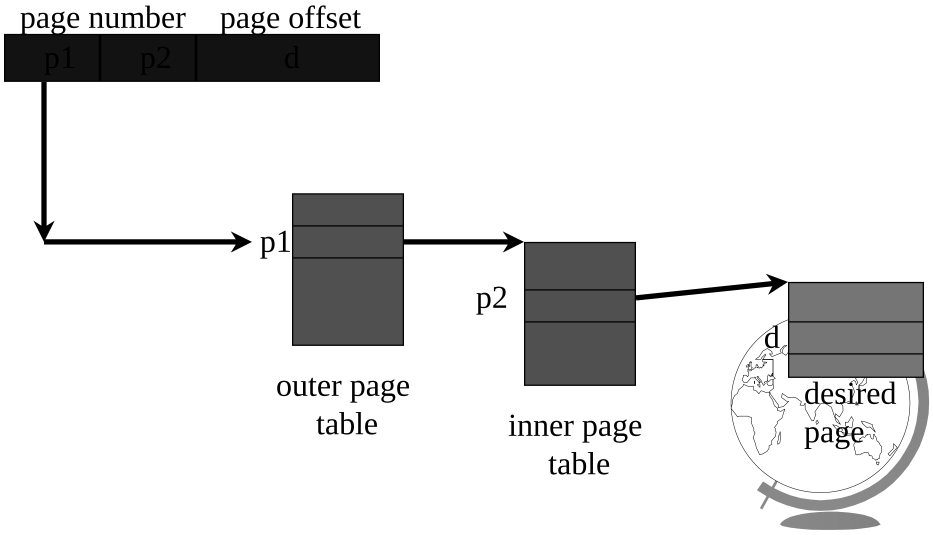
Large Address Spaces

- Typical logical address spaces:
  - 4 Gbytes => 2^{32} address bits (4-byte address)
- Typical page size:
  - 4 Kbytes = 2^{12} bits
- Page table may have:
  - $2^{32} / 2^{12} = 2^{20}$ = 1million entries
- Each entry 3 bytes => 3MB per process!
- Do not want that all in RAM
- Solution? Page the page table
  - Multilevel paging

Multilevel Paging

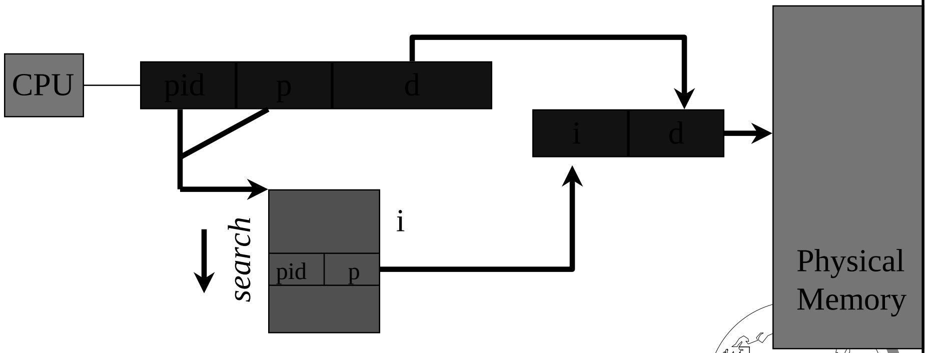


Multilevel Paging Translation



Inverted Page Table

- Page table maps to physical addresses

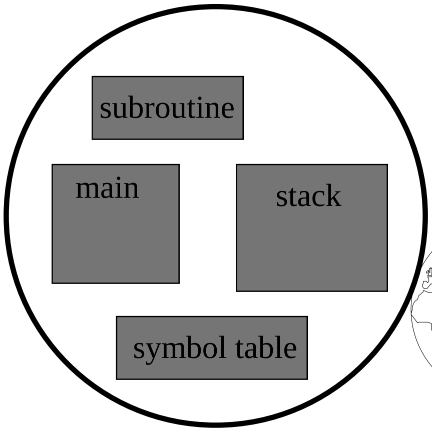


- Still need page per process --> backing store
- Memory accesses longer! (search + swap)

Memory View

- Paging lost users' view of memory
- Need "logical" memory units that grow and contract

ex: stack,
shared library

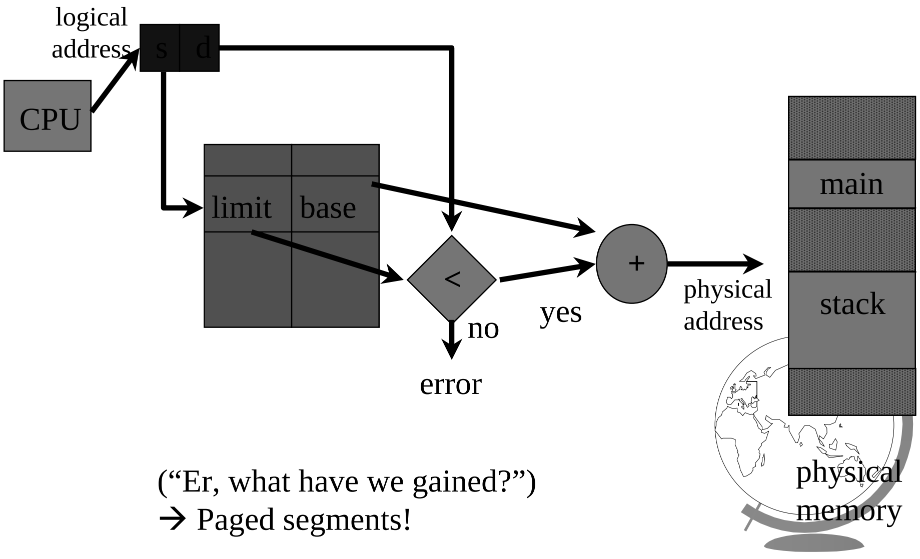


- Solution?
  - Segmentation!

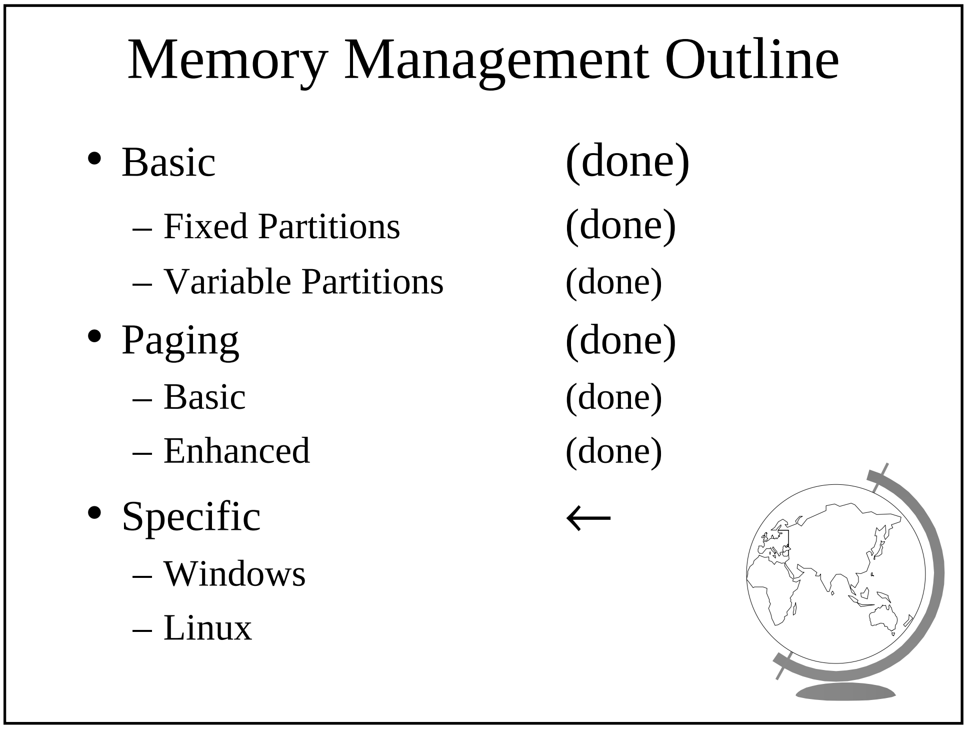
Segmentation

- Logical address: <segment, offset>
- Segment table - maps two-dimensional user defined address into one-dimensional physical address
  - base - starting physical location
  - limit - length of segment
- Hardware support
  - Segment Table Base Register
  - Segment Table Length Register

Segmentation

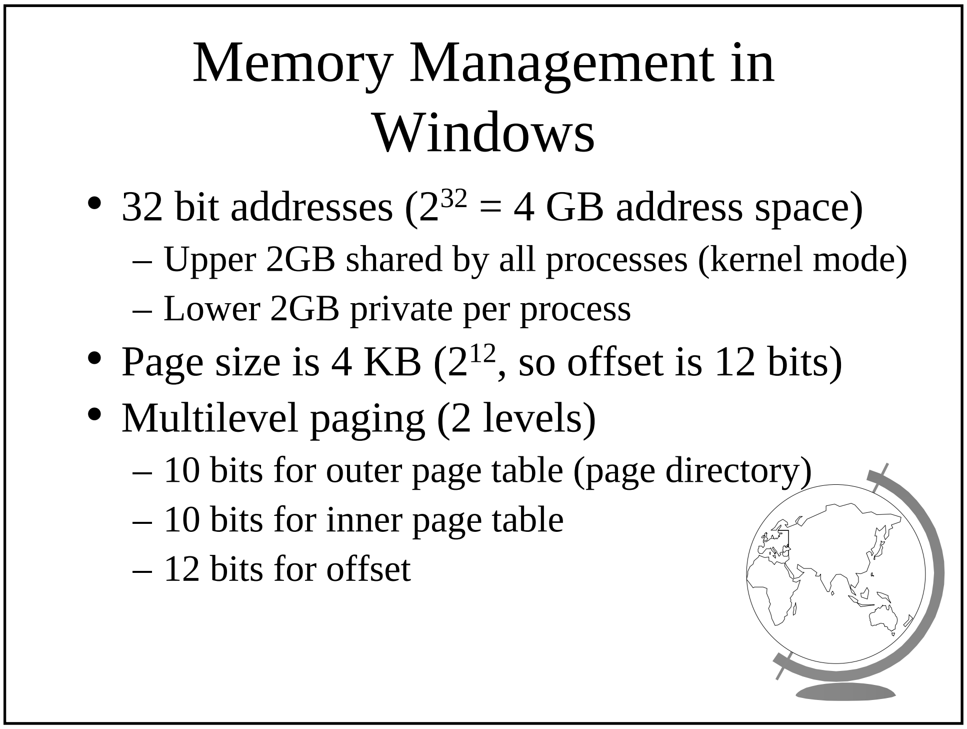


("Er, what have we gained?")
→ Paged segments!



Memory Management Outline

- Basic (done)
  - Fixed Partitions (done)
  - Variable Partitions (done)
- Paging (done)
  - Basic (done)
  - Enhanced (done)
- Specific ←
  - Windows
  - Linux



Memory Management in Windows

- 32 bit addresses (2^{32} = 4 GB address space)
  - Upper 2GB shared by all processes (kernel mode)
  - Lower 2GB private per process
- Page size is 4 KB (2^{12}, so offset is 12 bits)
- Multilevel paging (2 levels)
  - 10 bits for outer page table (page directory)
  - 10 bits for inner page table
  - 12 bits for offset

Memory Management in Windows

- Each page-table entry has 32 bits
  - only 20 needed for address translation
  - 12 bits “left-over”
- Characteristics
  - Access: read only, read-write
  - States: valid, zeroed, free …
- Inverted page table
  - points to page table entries
  - list of free frames

Memory Management in Linux

- Page size:
  - Alpha AXP has 8 Kbyte page
  - Intel x86 has 4 Kbyte page
- Segments
  - Kernel code, kernel data, user code, user data …
- Multilevel paging (3 levels)
  - Makes code more portable
  - Even though no hardware support on x86!
    - “middle-layer” defined to be 0