Operating Systems I

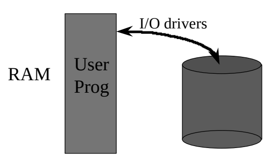
Memory Management

Overview

- ✦ Provide Services
  - – processes ✓
  - – files ✗
- ✦ Manage Devices
  - – processor ✓
  - – memory ✓
  - – disk ✗

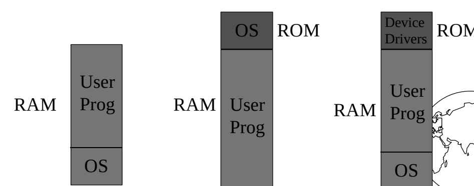
Simple Memory Management

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



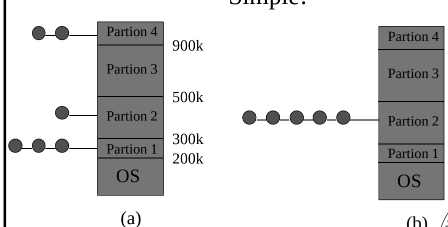
Simple Memory Management

- ✦ Small, protected OS, drivers
  - – DOS
  - – “Mono-programming” -- No multiprocessing



Multiprocessing w/Fixed Partitions

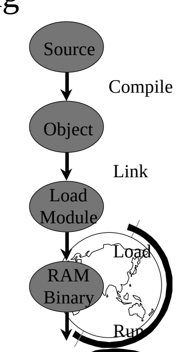
Simple!



- Unequal queues
- Waste large partition
- Skip small jobs

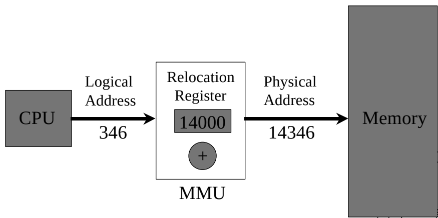
Address Binding

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



Logical vs Physical Addresses

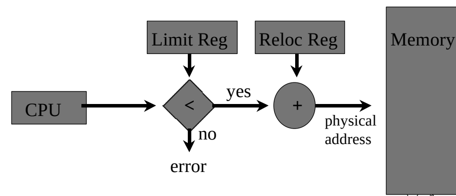
- Compile-Time + Load Time addresses same
- Run time addresses different



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

Relocatable Code

- Allow *logical* addresses
- Protect other processes



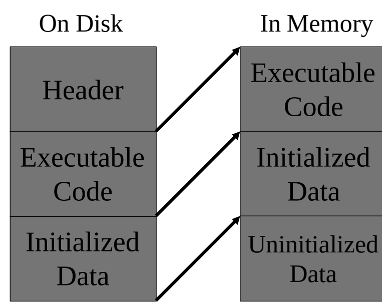
- Addresses must be contiguous!

Object Module

- Information required to “load” into memory
- Header Information
- Machine Code
- Initialized Data
- Symbol Table
- Relocation Information
- (see sample)

Loading an Object

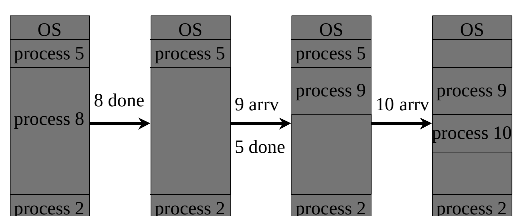
- Resolve references of object module



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

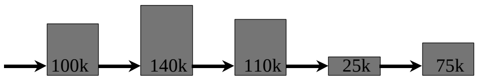
Variable-Sized Partitions



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

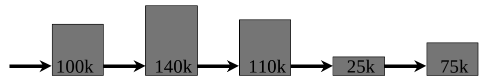
Variable-Sized Partitions

- Given a list of free holes:



- How do you satisfy a request of sizes?
  - 20k, 130k, 70k

Variable-Sized Partitions



- Requests: 20k, 130k, 70k
  - First-fit: allocate *first* hole that is big enough
  - Best-fit: allocate *smallest* hole that is big enough
  - Worst-fit: allocate *largest* hole (say, 120k)

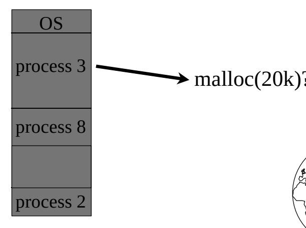
Variable-Sized Partitions

- First-fit: might not search the entire list
- Best-fit: must search the entire list
- Worst-fit: must search the entire list

- First-fit and Best-ft better than Worst-fit in terms of speed and storage utilization

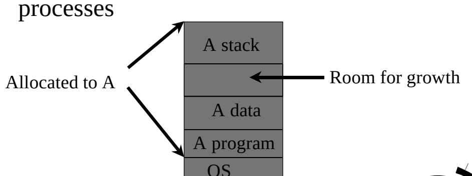
Memory Request?

- What if a request for additional memory?



Internal Fragmentation

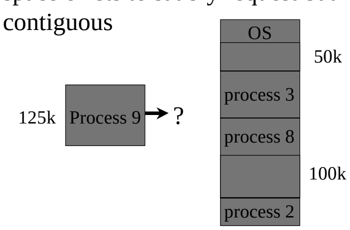
- Have some "empty" space for each processes



- 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

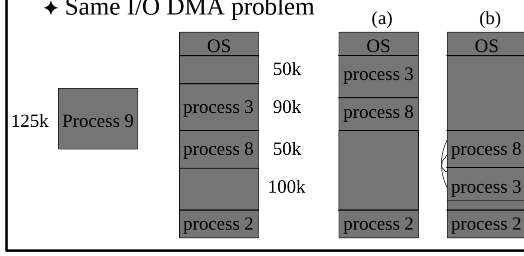


Analysis of External Fragmentation

- 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

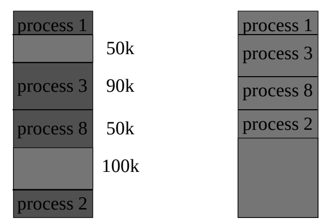


Review

- What is a relocation register?
- What is external fragmentation? How to fix?
- Given fixed partitions. List three ways to handle a job that requests too much memory.

Cost of Compaction

- Compaction of Memory vs. Swap (Disk)



- Disk much slower!

Solution?

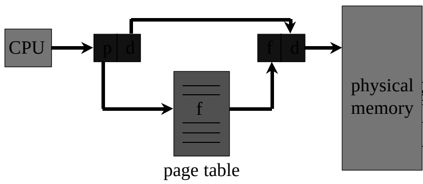
- Want to minimize external fragmentation
  - Large Blocks
  - But internal fagmentation!
- Tradeoff
  - Sacrifice some internal fragmentation for reduced external fragmentation
  - *Paging*

Paging

- Logical address space of a process can be noncontiguous; process is allocated memory wherever latter is 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*

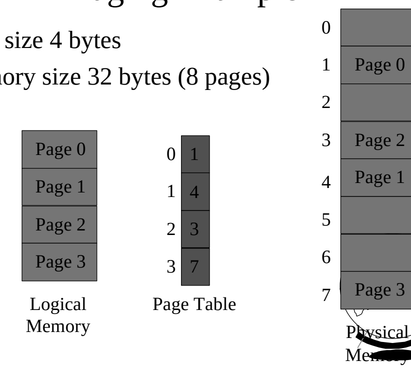
Paging

- 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

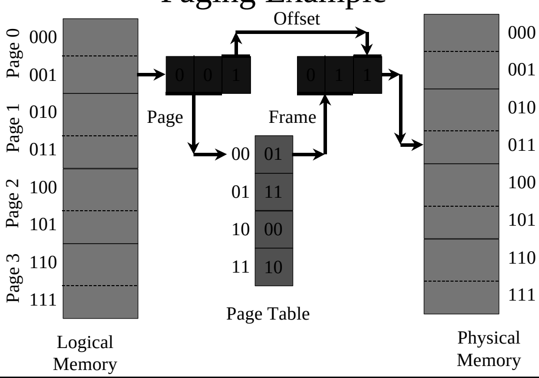


Paging Example

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

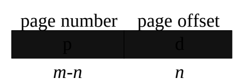


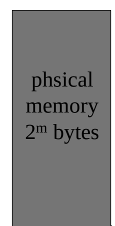
Paging Example



Paging Hardware

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





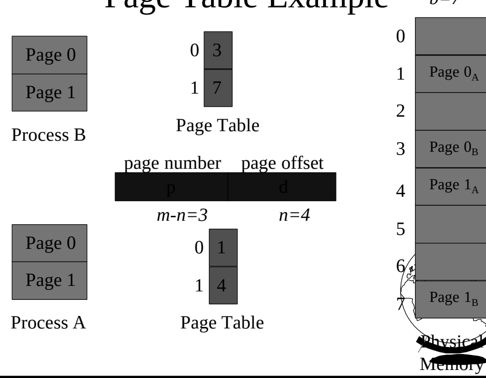
- note: not losing any bytes!

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?

Page Table Example

b=7

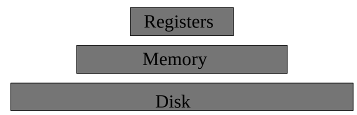


Paging Tradeoffs

- 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)

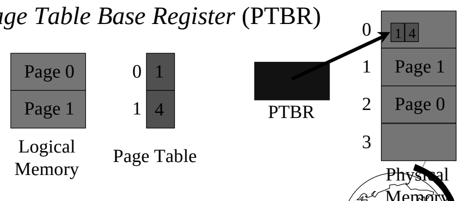
Implementation of Page Table

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



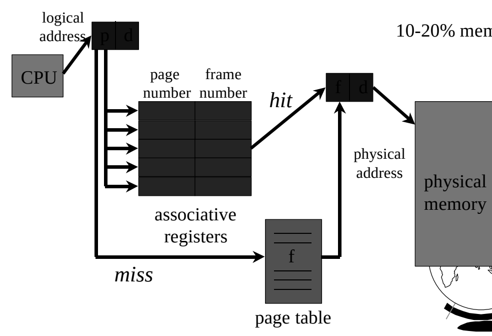
Implementation of Page Table

- 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



Associative Register Performance

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

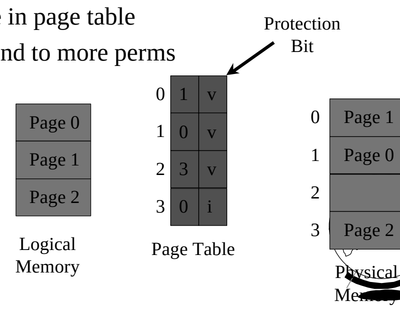
<u>Effective access time =</u>

<u>hit ratio *x* hit time + miss ratio *x* miss time</u>

- 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

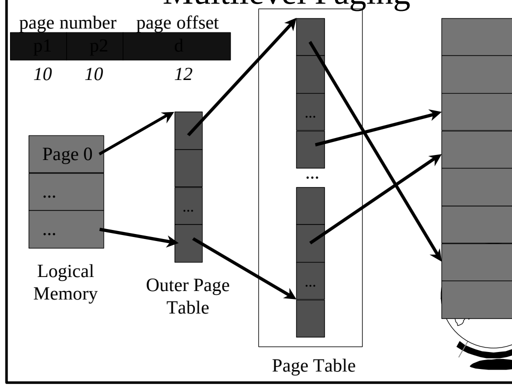
- Protection bits with each frame
- Store in page table
- Expand to more perms



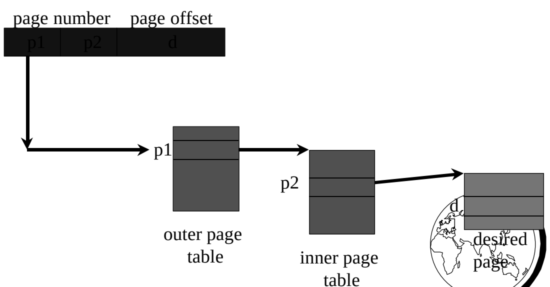
Large Address Spaces

- 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

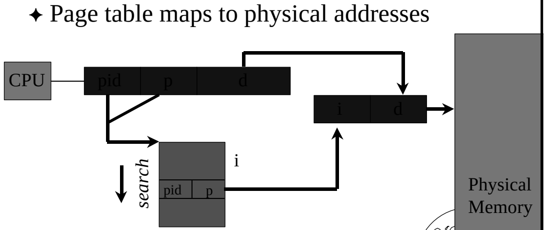


Multilevel Paging Performance

- 2 memory access if miss
- Effective access time?
  - 90% hit rate
  - .80 * 120 + .20 * 320 = 160 nanoseconds

Inverted Page Table

- Page table maps to physical addresses

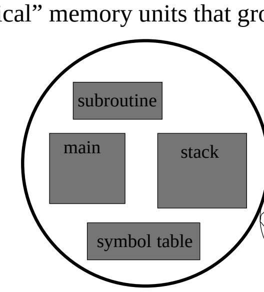


- Still need page per process --> backing store
- Memory accesses longer! (search + swap)

Memory View

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

ex: stack,
shared library

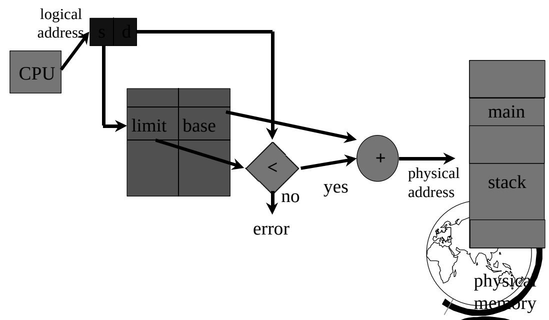


- Solution?
  - Segmentation!

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



Memory Management Outline

- Basic ✓
  - Fixed Partitions ✓
  - Variable Partitions ✓
- Paging ✓
  - Basic ✓
  - Enhanced ✓
- Specific ←
  - WinNT
  - Linux
- Virtual Memory

Memory Management in WinNT

- 32 bit addressess (2^{32} = 4 GB address space)
  - Upper 2GB shared by all processes (kernel mode)
  - Lower 2GB privater 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 WinNT

- 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 ha 8 Kbyte page
  - Intel x86 has 4 Kbyte page
- Multilevel paging (3 levels)
  - Even though hardware support on x86!

Memory Management in Linux

- Buddy-heap
- Buddy-blocks are combined to larger block
- Linked list of free blocks at each size
- If not small enough, broken down