



Operating Systems

Memory Management
(Ch 9)

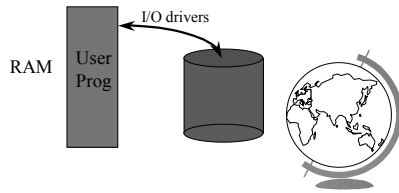
Overview

- Provide Services (done)
 - processes (done)
 - files (briefly here, more in cs4513)
- Manage Devices (done)
 - processor (done)
 - memory (next!)
 - disk (done after files)



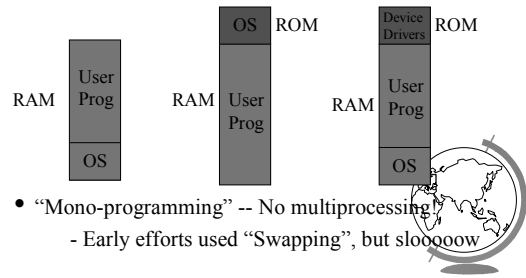
Simple Memory Management

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



Simple Memory Management

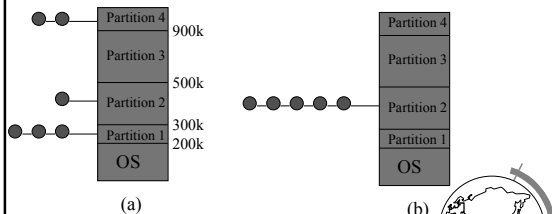
- Small, protected OS, drivers
 - DOS



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

Multiprocessing w/Fixed Partitions

Simple!

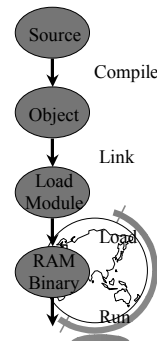


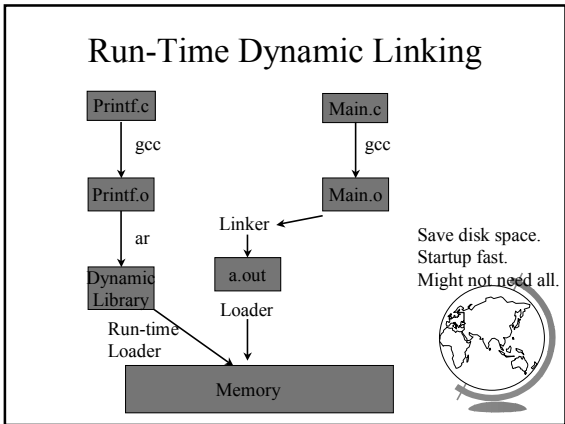
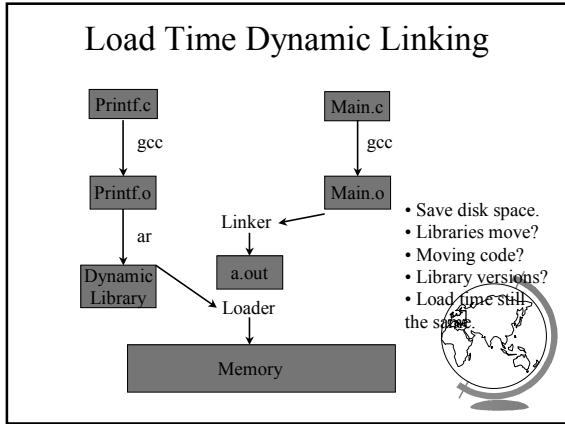
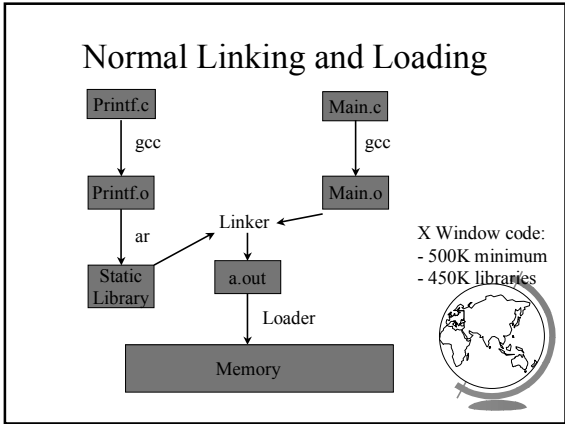
- Unequal queues
- Waste large partition
- Skip small jobs

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

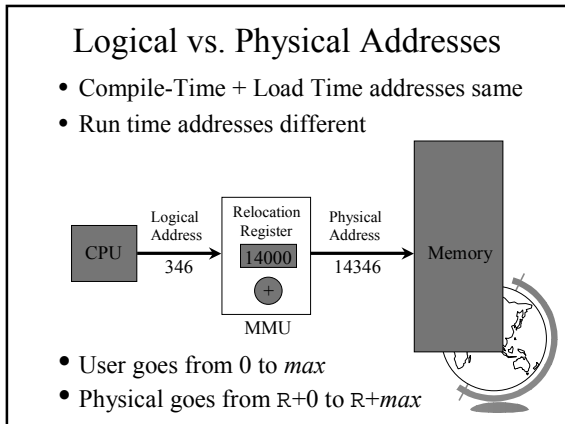




Memory Linking Performance Comparisons

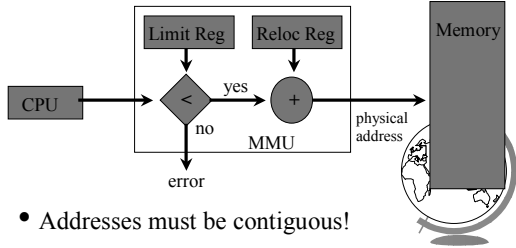
Linking Method	Disk Space	Load Time	Run Time (4 used)	Run Time (2 used)	Run Time (0 used)
Static	3Mb	3.1s	0	0	0
Load Time	1Mb	3.1s	0	0	0
Run Time	1Mb	1.1s	2.4s	1.2s	0

- ### 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



Relocatable Code Basics

- Allow *logical* addresses
- Protect other processes



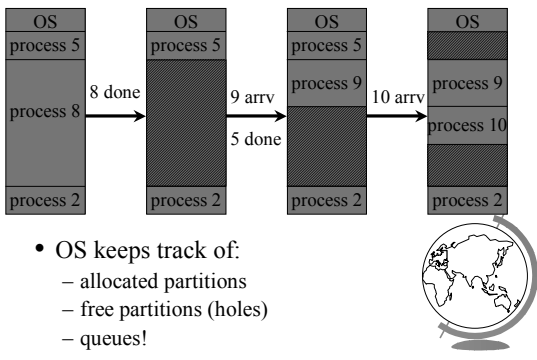
- Addresses must be contiguous!

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

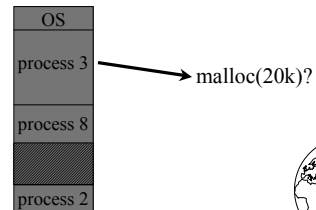


- OS keeps track of:
 - allocated partitions
 - free partitions (holes)
 - queues!



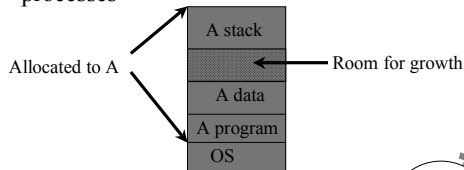
Memory Request?

- What if a request for additional memory?



Internal Fragmentation

- Have some “empty” space for each processes

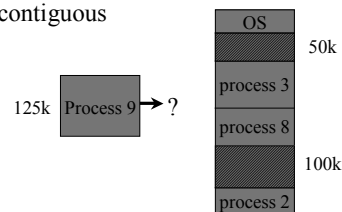


- Internal Fragmentation - allocated memory may be slightly larger than requested and not being used.



External Fragmentation

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



“But, how much does this matter?”



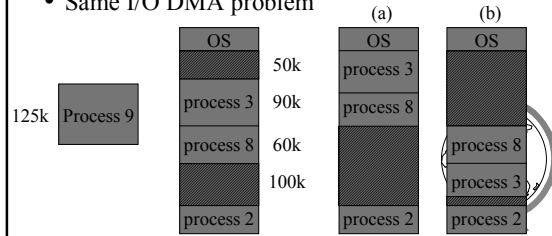
Analysis of External Fragmentation

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

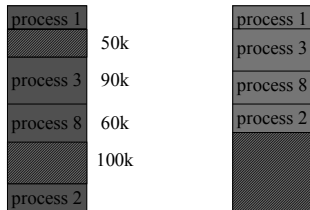


Compaction

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



Cost of Compaction



- 128 MB RAM, 100 nsec/access
 - \rightarrow 1.5 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 blocks 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)

Page 0	0	1
Page 1	1	4
Page 2	2	3
Page 3	3	7

Paging Example

Page	Frame
00	01
01	11
10	00
11	10

Paging Hardware

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

page number
page offset

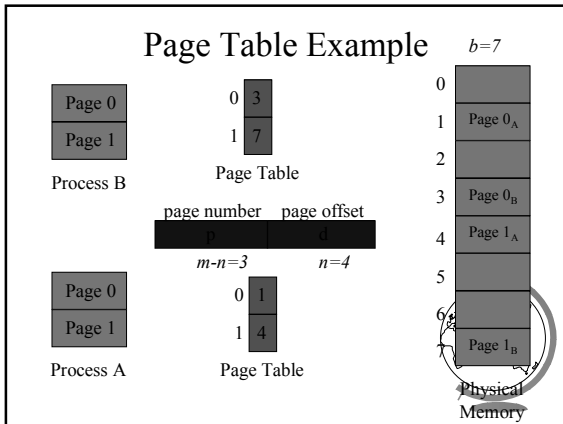
$m-n$ n

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 4 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?



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)*

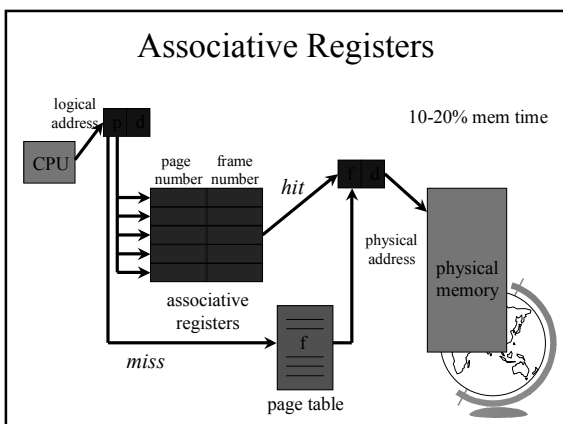
Logical Memory Page Table

Page 0	0	1
Page 1	1	4

PTBR

Physical Memory

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



Associative Register Performance

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

Effective access time =
 $\text{hit ratio} \times \text{hit time} + \text{miss ratio} \times \text{miss time}$

- 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
 - $.80 \times 120 + .20 \times 220 = 140$ nanoseconds

Protection

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

The diagram illustrates the mapping from logical memory to physical memory. On the left, 'Logical Memory' contains three pages: Page 0, Page 1, and Page 2. In the center, the 'Page Table' maps these logical pages to physical frames: Page 0 maps to frame 0, Page 1 to frame 1, and Page 2 to frame 3. Each frame in the page table has a 'Protection Bit' (v for valid, i for invalid). On the right, 'Physical Memory' shows the actual layout: frame 0 contains Page 1, frame 1 contains Page 0, and frame 3 contains Page 2.

Large Address Spaces

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

Multilevel Paging

The diagram shows 'Logical Memory' with pages 0 through 12. An 'Outer Page Table' maps these logical pages to entries in a 'Page Table'. The 'Page Table' is a large structure where each entry points to a specific page in physical memory.

Multilevel Paging Translation

The diagram illustrates the translation of a logical address. The address is split into a 'page number' (10) and a 'page offset' (12). The page number is used to look up an entry 'p1' in the 'outer page table'. This entry 'p1' points to an 'inner page table'. The page offset is then used to look up an entry 'p2' in the inner page table, which finally points to the 'desired page' in physical memory.

Inverted Page Table

- Page table maps to physical addresses

The diagram shows a 'CPU' with a register containing 'pid' and 'p'. This register points to an 'Inverted Page Table' which is a list of entries. Each entry contains 'pid' and 'p'. The 'CPU' also has a 'search' pointer. The 'Physical Memory' is shown as a large block. The diagram indicates that the CPU searches for the 'pid' in the inverted page table to find the corresponding 'p' (physical address).

- Memory accesses longer! (search + swap)
(Used by PowerPC and Ultra Sparc)

Memory View

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

ex: stack, shared library

The diagram shows a circular 'Memory View' containing several logical memory units: 'subroutine', 'main', 'stack', and 'symbol table'. A globe icon is also present.

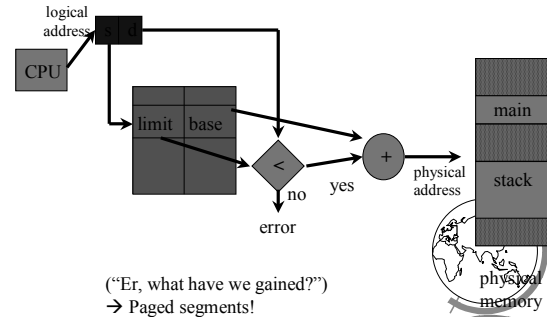
- Solution?
 - Segmentation!

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 (done)
 - Fixed Partitions (done)
 - Variable Partitions (done)
- Paging (done)
 - Basic (done)
 - Enhanced (done)
- Specific ←
 - WinNT/2000
 - Linux



Memory Management in WinNT/2000

- 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 WinNT/2000

- 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
- Multilevel paging (3 levels)
 - Makes code more portable
 - Even though no hardware support on x86!
+ "middle-layer" defined to be 1

