CS 370: Operating Systems

[Memory Management]

Computer Science
Colorado State University

** Lecture slides created by: Shrdeep Pallickara

Instructor: Louis-Noel Pouchet
Spring 2024
Topics covered in this lecture

- Address binding
- Address spaces
- Swapping
- Contiguous memory allocations
Memory is an important resource that must be managed carefully

- Memory capacities have been increasing
  - But programs are getting bigger faster

- Parkinson's Law
  
  Programs expand to fill the memory available to hold them
What every programmer would like

- Memory that is
  - Private, infinitely large, infinitely fast
  - Non-volatile
  - Inexpensive too

- Unfortunately, no such memory exists as of now
The second choice is to manage a hierarchy of memory

- Registers
- Cache
- Main Memory
- Electronic Disk
- Magnetic Disk
- Optical Disk
- Magnetic Tapes

*Cost/bit increases* → *Access times increase*
MEMORY MANAGEMENT
Memory Management: Why?

- Main objective of system is to execute programs
- Programs and data must be **in memory** (at least partially) during execution
- To improve CPU utilization and response times
  - Several processes need to be memory resident
  - Memory needs to be shared
Memory

- Large array of words or bytes
  - Each word/byte has its own address

- Typical execution cycle:
  1. Fetch instruction from memory where program is stored
  2. Decode
  3. Execute. Operands may be fetched from memory
  4. Result of execution may be stored back to memory
Memory Unit

- Sees only a **stream** of memory addresses

- Oblivious to
  - **How** these addresses are generated
    - Instruction counter, indexing, indirection, etc.
  - **What** they are for
    - Instructions or data
Why processes must be memory resident

- Storage that the CPU can access **directly**
  1. Registers in the processor
  2. Main memory

- Machine instructions take memory addresses as arguments
  - None operate on disk addresses

- Any instructions in execution **plus** needed data
  - Must be in memory
Overheads in direct-access storage devices

- CPUs can decode instructions and perform simple operations on register contents
  - 1 or more per clock cycle
- Registers accessible in 1 clock cycle
- Main memory access is a transaction on the memory bus
  - Takes several cycles to complete
Coping with the speed differential

- Introduce fast memory between CPU and main memory for reused data
  - Cache (data)
  - Cache (instructions!)
  - Cache (translation!)
  - Cache …
Besides coping with the speed differential, **correct** operation needed

- OS must be protected from accesses by user processes
- User processes must be protected from one another
Protection: Making sure each process has separate memory spaces

- Determine **range** of legal addresses for process
- **Ensure** that process can access only those
Providing protection with registers

- **Base**
  - Smallest legal physical address

- **Limit**
  - Size of the range of physical address

- **Eg:** Base = 300040 and limit = 120900
  - Legal: $300040 \leftrightarrow (300040 + 120900 - 1) = 420939$
Base and limit registers loaded only by the OS

- **Privileged** instructions needed to load registers
  - Executed ONLY in kernel mode

- User programs cannot change these registers’ contents

- OS is given unrestricted access to OS and user’s memory
CPU hardware compares every address generated in user mode

CPU

\[ \text{address} \rightarrow \begin{array}{c}
\geq \quad \rightarrow \text{YES} \\
\rightarrow \text{NO}
\end{array} \quad \begin{array}{c}
based \quad \rightarrow \text{YES} \\
base + \text{limit} \rightarrow \text{NO}
\end{array} \quad \rightarrow \text{memory}

\text{TRAP to OS: Addressing ERROR}
Processes and memory

- To execute, a program needs to be placed inside a process.
- Process executes
  - Access instructions and data from memory
- Process terminates
  - Memory reclaimed and declared available
Binding is a mapping from one address space to the next

- Processes can reside in **any part** of the physical memory
  - First address of process need not be x0000
- Addresses in source program are **symbolic**
- Compiler binds symbolic addresses to **relocatable** addresses
- Loader binds relocatable addresses to **absolute** addresses
Binding can be done at ... [1/2]

- Compile time
  - Known that the process will reside at location \( R \)
    - If location changes: recompile
  - MS-DOS .COM programs were bound this way

- Load time
  - Based on compiler generated relocatable code
Binding can be done at ... [2/2]:

Execution-time

- Process can be moved around during execution
  - Binding delayed until run time
  - Special hardware needed
  - Supported by most OS
ADDRESS SPACES
Address spaces

- **Logical**
  - Addresses generated by the program running on CPU

- **Physical**
  - Addresses seen by the memory unit

- Logical address space
  - Set of logical addresses generated by program

- Physical address space
  - Set of physical addresses corresponding to the logical address space
Generation of physical and logical addresses

- Compile-time and load-time
  - *Identical* logical and physical addresses

- Execution time
  - Logical addresses *differ* from physical addresses
  - Logical address referred to as *virtual* address

- Runtime mapping performed in hardware
  - Memory management unit (MMU)
Memory management unit

- Mapping converts logical to physical addresses

- User program *never sees* real physical address
  - Create pointer to location
  - Store in memory, manipulate and compare

- When used as a **memory address** (load/store)
  - Relocated to physical memory
Dynamic relocation using a relocation register

User program *never sees* the real physical addresses
But ...

- Do we need to load the entire program in memory?
In dynamic loading an unused routine is never loaded into memory

- Routine is not loaded until it is called
  - Kept on disk in relocatable load format

- When routine calls another one
  - If routine not present?
    - Load routine and update address tables

- Does not require special support from OS
  - Design programs appropriately
Contrasting Loading and Linking

- **Loading**
  - Load executable into memory prior to execution

- **Linking**
  - Takes some smaller executables and joins them together as a single larger executable.
Static linking

- Language libraries treated as other modules
  - Combined by loader into program image

- Each program includes a copy of library functions called in executable image
  - Wastes disk / memory space, but make the binary self-contained
Dynamic linking is similar to dynamic loading

- **Stub** included for each library reference
  - Locate memory resident routine
  - How to load routine if not in memory

- After routine is loaded, stub replaces itself with address of routine
  - Subsequent accesses to code-segment do not incur dynamic linking costs
Unlike dynamic loading, dynamic linking needs support from the OS

- Only the OS can allow multiple processes to access the same memory region
  - Shared Pages
Swapping
Swapping: Temporarily moving a process out of memory into a backing store.
Swapping and memory space restrictions: Effects of binding

- Process may or may not be swapped back into the same space that it occupied
- Binding at compile or load time?
  - Difficult to relocate
- Execution-time binding
  - Process can be swapped into different memory space
  - Physical addresses computed at run-time
When a CPU scheduler decides to execute a process, it calls the dispatcher

- Check whether the next process is in memory

- If it is not & there is no free memory?
  - Swap out a process that is memory resident
  - Swap in the desired process
Overheads in swapping: Context switch time

- User process size: 100 MB
- Transfer rate: 50 MB/sec
- Transfer time = 2 seconds
- Average latency: 8 milliseconds
- Swap out = transfer time + latency
  - $2000 + 8 = 2008$ milliseconds
- Total swap time = swap in + swap out
  - $4016$ milliseconds
Factors constraining swapping besides swap time

- Process must be completely **idle**
  - No pending I/O

- Device is busy so I/O is **queued**
  - Swap out $P_1$ and swap in $P_2$
  - I/O operation may attempt to use $P_2$’s memory
    - Solution 1: Never swap process with pending I/O
    - Solution 2: Execute I/O operations into OS buffers
Swapping is not a reasonable memory management solution

- Too much swapping time; too little execution time

- Modification of swapping exists in many versions of UNIX
  - Swapping is normally disabled
  - Starts if many processes are running, and a set threshold is breached
  - Halted when system load reduces
Each process is contained in a single continuous section of memory

**CONTIGUOUS MEMORY ALLOCATION**
Partitioning of memory

- Main memory needs to **accommodate** the OS and user processes

- Divided into two partitions
  - Resident OS
    - Usually low memory
  - User processes
Memory Mapping and Protection

- Relocation register
  - Smallest physical address

- Limit register
  - Range of logical addresses
Memory Mapping and Protection

- When CPU scheduler selects a process for execution
  - Relocation and limit registers reloaded as part of context switch

- Every address generated by the CPU
  - Checked against the relocation/limit registers
Memory Mapping and Protection

E.g.: relocation=100040 and limit=74600

TRAP to OS: Addressing ERROR
Memory Allocation: Fixed Partition method

- **Divide** memory into several **fixed-size** partitions
  - Each partition contains exactly one process

- Degree of multiprogramming
  - Bound by the number of partitions
Memory allocation: Variable-partition method [1/2]

- Used in batch environments

- OS maintains table tracking memory utilization
  - What is available?
  - Which ones are occupied?

- Initially all memory is available
  - Considered a large memory hole
  - Eventually many memory holes will exist
Memory allocation: Variable-partition method [2/2]

- OS orders processes according to the scheduling algorithm
- Memory allocated to processes until requirements of the next process cannot be met
  - Wait till a larger block is available
  - Check if smaller requirements of other processes can be met
Variable-partition method: Reclaiming spaces

- When process arrives if space is too large
  - Split into two

- When process terminates
  - If released memory is adjacent to other memory holes
    - **Fuse** to form a larger space
Splitting and Fusing Memory spaces
Dynamic Storage Allocation Problem

- Satisfying a request of size \( n \) from the set of available spaces
  - First fit
  - Best fit
  - Worst fit
First fit

- Scan list of segments until you find a memory-hole that is big enough

- Hole is broken up into two pieces
  - One for the process
  - The other is unused memory
Best Fit

- Scan the entire list from beginning to the end
- Pick the smallest memory-hole that is adequate to host the process
Comparing Best Fit and First Fit

- Best fit is **slower** than first fit

- Surprisingly, it also results in more **wasted memory** than first fit
  - Tends to fill up memory with tiny, useless holes
Worst fit

- How about going the other extreme?
  - Always take the largest available memory-hole
  - Perhaps, the new memory-hole would be useful

- Simulations have shown that worst fit is not a good idea either
The contents of this slide-set are based on the following references
