CS 370: Operating Systems
[Memory Management]

Shrideep Pallickara
Computer Science
Colorado State University

There is nothing wrong with your television set. Do not attempt to adjust the picture. We are controlling transmission. If we wish to make it louder, we will bring up the volume. If we wish to make it softer, we will tune it to a whisper. We will control the horizontal. We will control the vertical. We can roll the image, make it flutter. We can change the focus to a soft blur or sharpen it to crystal clarity. For the next hour, sit quietly and we will control all that you see and hear. We repeat: there is nothing wrong with your television set.

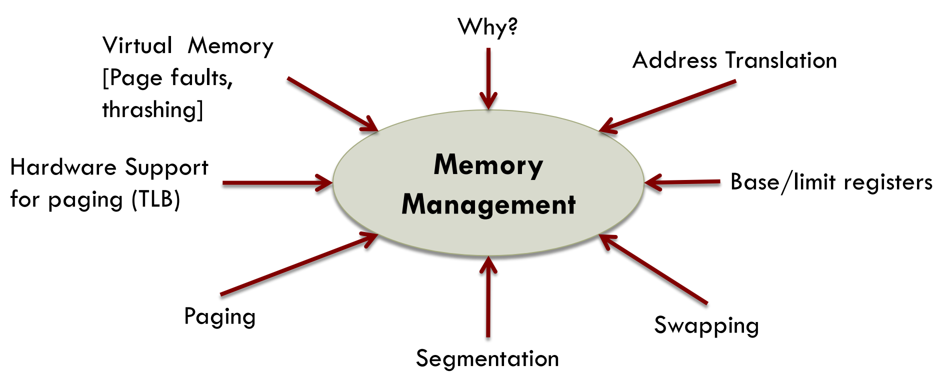
— Opening narration, The Outer Limits
Broadcast on ABC from 1963 to 1965 at 7:30 PM ET on Mondays

Memory Management

Frequently asked questions from the previous class survey

- Can a system transition from a safe state to an unsafe state even if it currently has several safe sequences?
  - In a safe state is it possible to have an unsafe sequence?
- Deadlock avoidance: possible for resources to be added or removed from the system?
- Recovery
  - Does a deadlocked process have to be terminated to retrieve resources?
- Communications deadlock: Lost messages?

Memory Management



Topics covered in this lecture

- Address Translation
- 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

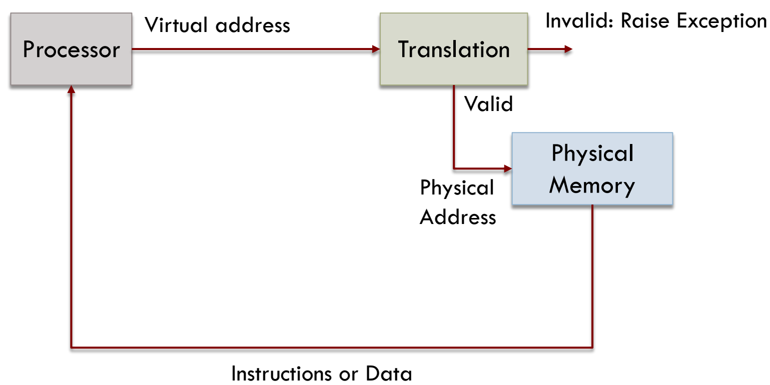
You were taught in an early programming class that a memory address is just an address

- A pointer in a linked list contains the actual memory address of what it is pointing to
- Jump instructions contain memory address of the next instruction to be executed
- This is **useful fiction**!
  - Programmer is better off not thinking about how each memory reference is converted into the data/instruction being referenced

Several features enabled by putting OS in control of **address translation**

- Address translation
  - **Conversion** of memory addresses the program *thinks* it is referencing to the physical location of memory cell
- From the programmer's perspective
  - Address translation occurs *transparently*
  - **Program behaves correctly** despite the fact that memory is stored somewhere <u>completely different</u> from where it **thinks** it's stored

Address translation in the abstract



9

Address translation concept

- Translator takes **each** instruction and data memory reference generated by a process
  - Checks whether the address is legal
  - Converts it to a physical memory address that can be used to store or fetch instructions or data
  - The data itself – whatever is stored in memory – is returned as is; it is not transformed in any way
- Translation usually **implemented in hardware**
  - The kernel configures the hardware to accomplish this

10

Address translation is a simple concept but turns out to be incredibly powerful

- What does it allow the OS to do?
  - Process isolation
  - IPC via shared memory
  - Share code segments
  - Program initialization
  - Dynamic memory allocations
  - Cache management
  - Program debugging
  - Virtual memory
  - Efficient I/O …

11

A pen – to register; a key –
That winds through secret wards
Are well assigned to Memory
By allegoric Bards.
…

Memory, William Wordsworth

MEMORY MANAGEMENT: WHY?

12

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
  2. Decode
     - Operands may be fetched from memory
  3. Results of execution may be stored back

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
  - <u>None operate on disk addresses</u>
- 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
  - 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

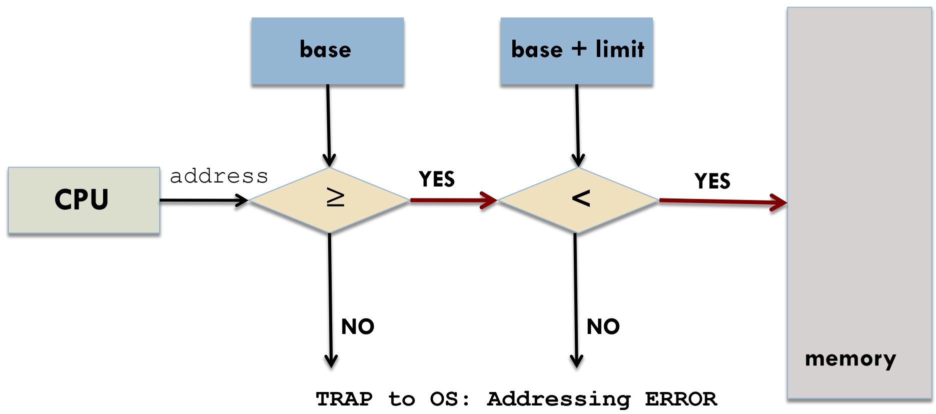
Providing protection with registers

- Base
  - **Smallest** legal physical address
- Limit
  - Size of the **range** of physical address
- E.g.: Base = 300040 and limit = 120900
  - Legal: 300040 ←→ (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



Trap?

- Interrupt generated by the CPU when there is
  - An attempt to execute a privileged instruction
  - Divide by zero
  - Illegal memory access
- Causes the OS to switch over to kernel mode

Processes and memory

- To execute, a program needs to be **placed** inside a process
- When process **executes**
  - Access instructions and data from memory
- When 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

COMPUTER SCIENCE DEPARTMENT
COLORADO STATE UNIVERSITY

29

Address spaces

- **Logical**
  - Addresses *generated* by the 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

COLORADO STATE UNIVERSITY | Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT | MEMORY MANAGEMENT | L19.30

30

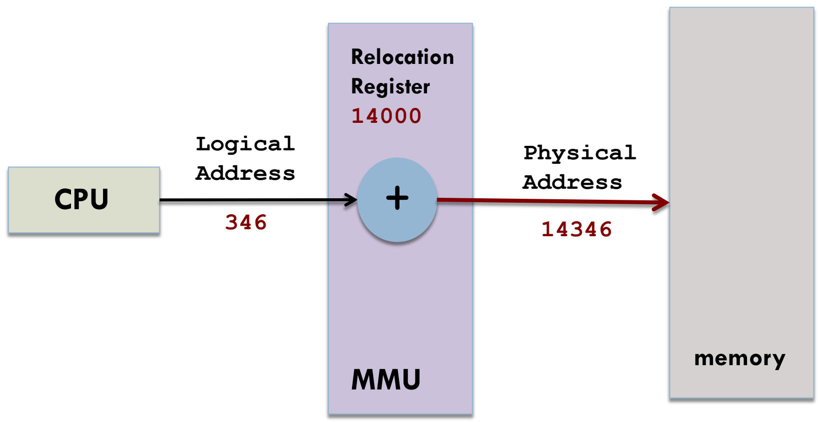
Generation of physical and logical addresses

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

33

But …

- Do we need to load the entire program in memory?

34

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 language library in its executable image
  - Wastes disk and memory space

Dynamic linking is similar to dynamic loading

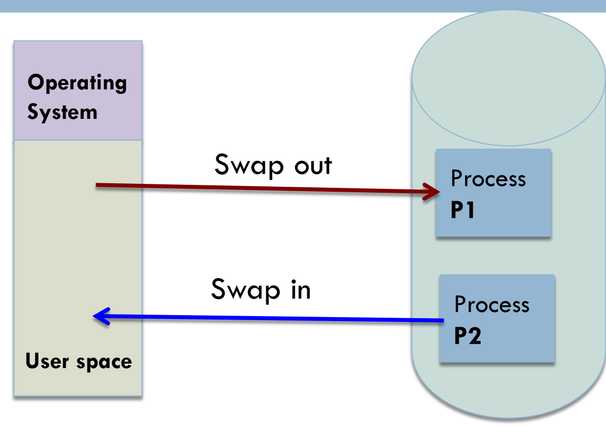
- **Stub** included for each library reference; includes information about
  - How to locate memory resident routine
  - How to load routine if not in memory
- After routine is loaded, stub replaces itself with address of routine
  - Subsequent accesses to code-segment do not incur dynamic linking costs

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 [disk seeks]: 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

Summarizing the pure Swapping based approach

- Bring in each process, in its *entirety*, into memory
- Run process for a while before eviction due to:
  - Space being needed for another process
  - Process becomes idle
    - Idle processes should not take up space in memory

The contents of this slide-set are based on the following references

- *Avi Silberschatz, Peter Galvin, Greg Gagne. Operating Systems Concepts, 9th edition. John Wiley & Sons, Inc. ISBN-13: 978-1118063330.* [Chapter 8]
- *Andrew S Tanenbaum and Herbert Bos. Modern Operating Systems. 4th Edition, 2014. Prentice Hall. ISBN: 013359162X/ 978-0133591620. [Chapter 3]*
- *Thomas Anderson and Michael Dahlin. Operating Systems Principles and Practice. 2nd Edition. ISBN: 978-0985673529.* [Chapter 8]