CSE 4/521
Introduction to Operating Systems

Lecture 12 – Main Memory I
(Background, Swapping)
Summer 2018
Overview

Objective:
1. To provide a detailed description of various ways of organizing memory hardware.

- Background
- Swapping
Recap

• Deadlock System Model
  • Representation of resources and processes in a system.

• Deadlock Characterization
  • Mutual exclusion, hold-and-wait, no preemption, circular wait.
  • No cycle ⇒ no deadlock, A cycle ⇒ possibility of deadlock.

• Methods for Handling Deadlocks
  • Pretend deadlock never happened, deadlock prevention and avoidance.

• Deadlock Prevention
  • Break all four conditions of deadlocks.
Questions

1. What are the 4 conditions in a system for a deadlock to occur? (Easy)
2. Is there a deadlock here? Why? (Medium)
3. How would you prevent the 4 conditions of deadlocks from happening? (Open-ended)
Overview

• Background
• Swapping
Overview

- Background
- Swapping
Background

• Program must be **brought (from disk) into memory** and placed within a process for it to be run
• **Main memory** and **registers** are only storage CPU can access directly
• **Register access** in **one CPU clock** (or less)
• **Main memory** access can take **many cycles**, causing a **stall**
• **Cache** sits between main memory and CPU registers
• **Protection of memory** required to ensure correct operation
Background: Multistep Processing of User Program
Background: Logical vs. Physical Address Space

• The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
  • Logical address – generated by the CPU; also referred to as virtual address
  • Physical address – address seen by the memory unit

• Logical and physical addresses are the same in compile-time and load-time address-binding schemes; but different in execution-time address-binding scheme
  • Logical address space is the set of all logical addresses generated by a program
  • Physical address space is the set of all physical addresses generated by a program
Background:

Base and Limit Registers

• A pair of **base** and **limit registers** define the logical address space

![Diagram showing address space with base and limit registers](image)
Background: Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user.
Background: Address Binding

• Programs on disk, ready to be brought into memory to execute

• Addresses represented in different ways at different stages of a program’s life
  • Source code addresses usually symbolic
  • Compiled code addresses bind to relocatable addresses
    • i.e. “14 bytes from beginning of this module”
  • Linker or loader will bind relocatable addresses to absolute addresses
    • i.e. 74014

• Each binding maps one address space to another
Background: Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages:
  - **Compile time:** If memory location known a priori, absolute code can be generated.
  - **Load time:** Must generate relocatable code if memory location is not known at compile time.
  - **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another.
    - Need hardware support for address maps (e.g., base and limit registers).
Background:
Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called relocation register
- The user program deals with logical addresses; it never sees the real physical addresses
  - Execution-time binding occurs only when reference is made to location in memory
  - Logical address bound to physical addresses
Background: Dynamic relocation using a relocation register

- Routine is **not loaded until it is called**
- Better **memory-space utilization**; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
Background: Dynamic Linking

• **Static linking** – system libraries and program code combined by the loader into the binary program image
• **Dynamic linking** – linking postponed until execution time
  • Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
  • Stub replaces itself with the address of the routine, and executes the routine
• Operating system checks if routine is in processes’ memory address
  • If not in address space, add to address space
• Dynamic linking is particularly **useful for libraries**
Overview

• Background
• Swapping
Swapping

• A process can be **swapped temporarily out of memory to a backing store**, and then brought back into memory for continued execution

• **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images

• Major part of swap time is **transfer time**; total transfer time is directly proportional to the amount of memory swapped

• System maintains a **ready queue** of ready-to-run processes which have memory images on disk
Swapping:
Schematic View of Swapping

1. **swap out**
2. **swap in**
Swapping

• Does the swapped out process need to swap back in to same physical addresses?
  • Depends on address binding method
    • Plus consider pending I/O to/from process memory space

• Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  • Swapping normally disabled
  • Started if more than threshold amount of memory allocated
  • Disabled again once memory demand reduced below threshold
Swapping: Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
Swapping:
Context Switch Time and Swapping

• Other constraints on swapping
  • Pending I/O – can’t swap out as I/O would occur to wrong process
  • Or always transfer I/O to kernel space, then to I/O device
    • Known as double buffering, adds overhead

• Standard swapping not used in modern operating systems
  • But modified version common
    • Swap only when free memory extremely low
Swapping: Swapping on Mobile Systems

• Not typically supported
  • Flash memory based
    • Small amount of space
    • Limited number of write cycles

• Instead use other methods to free memory if low
  • iOS asks apps to voluntarily relinquish allocated memory
    • Failure to free can result in termination
  • Android terminates apps if low free memory, but first writes application state to flash for fast restart
Credits for slides

Silberschatz, Galvin and Gagne