LECTURE 2.1
MEMORY CONCEPTS

COP4600

Dr. Matthew Gerber

2/29/2016

MEMORY

Memory and the Fetch-Execute Cycle (8.1)

- Main memory is the only storage that the CPU can directly address
- Example: The fetch/execute cycle
- When we say *addressed*, we mean each byte of memory has a location expressible with a number or set of numbers
- There are several levels of addressing
- In this course, we care about *the memory addresses that processes want to access*
- We *do not,* in general, care *why*; what a program does with its allocated memory is entirely up to the program as written

Memory Access (8.1.1)

- Important concepts:
  - *Registers* are single words of memory that the CPU can do arithmetic and logic in – and more importantly for the moment, use to control memory
    - The *base register* is used to hold the lowest memory address a process can access
    - The *limit register* is used to hold the size of the range a process can access
  - Accessing registers can be done in a cycle; accessing main memory requires a *stall*
  - The *cache* is a smaller, faster space of memory used to hold memory “closer” to the processor
  - Accessing it either does not require a stall, or requires only a very short one

Memory Protection (8.1.1)

- In general we need to confine each process to its own memory space
- We can do this with the base and limit registers
- Limit setting the base and limit registers to the operating system – make them *privileged instructions*
- If a process tries to access memory below the base or past the limit, it is terminated
- We will see versions of this concept for each form of memory management
- This termination has several names
  - Segmentation Fault
  - General Protection Fault
  - Access Violation

BINDING BASICS

Address Binding (8.1.2)

- A program on disk is typically a binary executable file
- To run, it's loaded into memory and made a process
- If swapping is involved, the process can be moved between disk and memory multiple times during execution
- The *input queue* is the queue of processes on disk waiting to get into memory
- The process usually can run anywhere in memory – but for that to work, its addresses need to be *bound*
- Eventually, those addresses need to become *physical addresses* – the ones used by the actual memory unit
  - (Even those aren't necessarily physical!)

Address Binding (8.1.2)

- Recall from programming basics…
  - *Symbolic addresses* are those that un-compiled (or un-assembled) programs use – variable names, and the like. The compiler binds them into…
  - *Relocatable addresses*, such as "*n* bytes from the beginning of this module", which the linker in turn binds into…
  - *Absolute addresses*, like "byte #*n*"
- These "absolute" addresses then need to be bound *again* when the operating system deals with the program, and subsequently the process

Binding Times (8.1.2)

- *Compile Time*
  - Bind all addresses into physical addresses at link time, generating *absolute code*
  - .COM-format programs in MS-DOS were compile-time bound
  - Only works if we know exactly where programs need to be loaded into memory
- *Load Time*
  - Leave addresses zero-relative at link time, generating *relocatable code* that the OS binds when it loads the program
  - Only works if we know programs will never be moved in memory
- *Execution Time*
  - Bind addresses every time they're used
  - All modern general-purpose operating systems use this
  - Requires hardware support

Logical and Physical Addresses (8.1.3)

- Execution time binding creates a distinction between the addresses that the CPU uses in normal operation and the addresses that are actually accessed in memory
  - The former are called *logical* or *virtual addresses*
  - The latter remain *physical addresses*
- The mapping is done by a *memory management unit*
  - This absolutely has to be in hardware
- We already have most of a simple version
  - Think of the base register as, instead, the *relocation register*
  - Every single time a process accesses memory, the base register is added to the address
  - The operating system changes the base and limit registers on every context switch
  - Every program now sees a zero-based main memory space

DYNAMIC LOADING BASICS

Dynamic Loading and Linking (8.1.4-8.1.5)

- Similar concepts that operate on different levels
- With dynamic *loading*, when a program is loaded, only its main routine is loaded and started; additional routines are loaded as they're called
- With dynamic *linking*, or *shared libraries*, rather than including all referenced routines from a library in the executable file, the linker includes *stubs* for those routines
- The routines are then dynamically loaded from system-managed libraries when called
- Allows for all processes that use the same library to call the same version
  - Saves memory
  - Allows for updates

Swapping (8.2)

- A process must be in memory to be executed
- But it doesn't have to *stay* in memory
- Standard *swapping* is moving processes between memory and a *backing store*
- The context-switch time for a swapped-out process is very high
- Processes must be *completely* idle to be swapped
- We basically don't do this any more
  - Why will be clear soon enough…

MEMORY MANAGEMENT BASICS

Allocating Contiguous Memory (8.3)

- The main memory has to have the operating system and the user processes in it
- Divide the memory into two partitions: *system* and *user*
  - The system partition is at either the top or bottom of memory
- We want to fill the rest of memory with processes
- Not too long ago, we allocated processes *contiguous* chunks of memory
  - Older versions of MacOS did this!
- The relocation register allows us to easily decide where in memory a process is; the limit register allows us to protect other processes from it

Contiguous Allocation (8.3.2)

- In *fixed-partition* allocation we divide memory into multiple partitions (that can even be of different size), allocating exactly one per process
  - Easy to implement and manage
  - Wastes a lot of memory
- In *variable-partition* allocation the system has to keep a table indicating which parts of memory are occupied
  - Initially, there is one large *hole* for memory
  - As processes enter and leave, the one hole will become a set of holes
  - This process is called *fragmentation*
  - If holes are adjacent to each other, they become one hole
- How best to allocate processes to available holes?

Contiguous Allocation Strategies (8.3.2)

- First Fit
  - Allocate a new process to the first hole in the set large enough to hold it
- Best Fit
  - Allocate a new process to the smallest hole large enough to hold it
  - Idea is to waste the least amount of memory
- Worst Fit
  - Allocate a new process to the largest hole
  - Idea is to leave the largest available holes
- In simulations, First Fit and Best Fit are both faster, and both produce better results, than Worst Fit
- First Fit is faster than Best Fit; neither is clearly better

THE PROBLEM

Fragmentation (8.3.3)

- The First Fit and Best Fit strategies suffer from *external fragmentation*
  - Over time, the holes get smaller and smaller
  - As much as 1/3 of memory space can be lost to this!
- One solution is to *compact* memory whenever fragmentation becomes sufficiently severe
- Relocate all processes to place all free memory together in one large block
- Only possible with execution-time binding
  - …and with execution-time binding, we've got better ways to handle it…

Preliminaries: Segmentation (8.4)

- Programs don't use memory perfectly as a linear array of bytes
- They use memory in chunks – a collection of variably-sized *segments*
- Each segment has a number, and memory bytes are referred to as *offsets* in that segment
- A logical address thus consists of a *segment* and an *offset*
- Typical segments might be code, globals, heap and stack
- Segments are implemented via a *segment table*; each segment is given its own base and limit register
- What would you call it if a program tried to read or write past the end of one of its segments?

A Question

- Segmentation allows code to be relocated much, more easily
- Doesn’t eliminate external fragmentation but does break up programs into more manageable pieces
- What if we could do a radical version of this?
- Break up *all of memory* into an arbitrary number of fully relocatable chunks
- Could we take this concept to the point of eliminating fragmentation entirely?

NEXT TIME: PAGING