Memory, part 1

CS 350: Computer Organization & Assembler Language Programming
Lecture 17, Mon 4/1

Memory Management - Intro
- 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
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests

Background
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- Cache sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Address Binding
- When/how do variables in user program get connected to physical memory address?
- Symbolic source code addresses; compile to offsets within a procedure call’s space.
- Linker (= link editor) combines object modules into executable program.
- Loader brings executable program into memory; may also bring in system libraries

Multistep Processing of a User Program
- Binding of Instructions and Data to Memory
- Can a process be moved during execution?
- Combine unused sections of memory by moving used sections.
- Then address binding can’t happen until run time; may also change during runtime
- Need hardware support for address maps (e.g., base and limit registers)
- Address used by code is dynamically transformed.

Logical vs. Physical Address Space
- Logical (a.k.a. virtual) address is generated by the CPU
- Physical address: Address seen by the memory unit
- No distinction between logical & physical addresses back when assemblers, linkers, or loaders generated absolute memory addresses.
• **Memory-Management Unit**
  • MMU: Originally took physical addresses
  • Has been modified over time to take logical address and determine physical address
  • Old, simple scheme: MMU contains relocation register (physical address = logical address + relocation register)
  • Allows for dynamic relocation of program.

• **Why Relocate?**
  • Memory utilization improved if routine is not loaded until it is called.
  • Routines stored in relocatable load format
  • Useful for large infrequently-called routines
  • Implemented through program design
    • OS can help by providing libraries to implement dynamic loading

**Increasing Memory Utilization**

• Various techniques have been created to use memory more efficiently — all based on idea of not using memory to hold code or instructions that aren’t being used.

• **Swapping**: Move entire process out of memory
  • Segmentation & Paging: Break memory up into smaller pieces

• **Dynamic linking**: Don’t bring code into memory if we haven’t called it yet.

• **Static linking**: System libraries & program code combined by loader into binary program image

• **Dynamic linking/shared libraries**: Linking postponed until library call is executed
  • On first call of routine, stub for library call is replaced by call of actual routine (OS adds routine to process’s memory space if necessary)

**Swapping**

• Swap out: Move an entire process out from memory to a backing store.

• Swap in: Move process from backing store into memory.

• Total memory space of processes can exceed physical memory

• Swapping is slow: Can take seconds to read/write an entire process into/out of memory

• Can avoid transferring unused parts of processes if we know what they are.

**Segmentation**

• Split logical address space into segments.
  • Addresses contiguous within a segment.
  • One segment each for code, global data, heap data, stack data, shared libraries.
  • Same addresses appear, but in different in segments
  • Instead of allocating a process’s entire address space as one segment, allocate each segment separately.

**Fragmentation**

• Can get **internal fragmentation**
• Not using all the space for a segment because segments only come in certain sizes.
• Can get **external fragmentation**
  • Hard to allocate small number of large, different-sized pieces of memory.
  • Solving this problem led to paging.
• OS must maintain list of free spaces in memory
  • When a process asks for memory, OS looks for large-enough free space
    • Different techniques for looking for spaces (first-fit, best-fit, worst-fit)
    • In practice, first- and best-fit do roughly equally good jobs; worst-fit is worse.
  • When a process returns memory, OS adds it back to free space list
    • If two free spaces adjoin, we can make them one larger free space
    • Helps to avoid lots of small unused parts of memory.

**Segmented Addresses**
• A logical address is a pair: a segment number and address within segment
• Segment table: Maps segment number to physical address & size of segment
  • Accessing a logical address requires a segment table lookup plus a physical memory lookup
  • MMU maintains segment table to speed up access (putting table in physical memory would slow things down.)

**Paging**
• Paging similar to segmentation: Physical memory for process can be discontiguous even though logical addresses are contiguous
  • In paging, instead of allocating a small number of large, variable-sized pieces of memory, allocate a larger number of smaller, fixed-size pieces ("pages").
• Avoids external fragmentation that segmentation suffers from
• Physical memory is broken up into fixed-size **frames**: a page is stored in a frame.
  • Logical address pairs page number and offset in page.
  • Physical address pairs frame number and same offset
• Frame size = page size, a power of 2, say $2^n$
  • Making it a power of 2 means the last $n$ bits of a logical/physical address is the offset.
  • For a logical address, the left part of the address is the page number
  • For a logical address, left part is physical address of beginning of frame

**Page Table**
• Page table maps page numbers to frame numbers
  • Logical address $(p, d)$ becomes physical address $(table[p], d)$
• Page table is not small. Example: 4 GB physical memory broken up into $2^{20}$ frames each 4 KB long.
  • Each frame number is 20 bits long, so you need $(2^{20}) * 20$ bits for the table, which is 2.5 MB of space.
- That's too large to store in the MMU — the page table needs to be stored in real memory.

- Tension between larger frame/page sizes and smaller sizes
  - The smaller the page size, the less internal fragmentation you get, but you get more page numbers to maintain.
  - The larger the page size, the fewer page numbers there are, but the more internal fragmentation you can have.

- Want MMU to maintain page table, for speed

- Given logical address, MMU figures out physical address and accesses that physical location. It:
  - Breaks up logical address into page# and offset within page
  - Want to look up page# entry in page table
  - Figure out which chunk of page table we want
  - Look up page# within that chunk of page table ← requires a memory access
  - Generate physical address = page # and offset within frame
  - Now access the physical address we wanted ← the memory access we actually wanted.

Activity Questions
1. What is the difference between a logical and a physical address? Which does a user program use?
2. One major goal of improving memory technology is to use memory more efficiently, letting less be wasted while letting users maintain contiguous local addresses. How does paging try to do this?
3. Assume we're not using virtual memory. What does the page table do? How is a logical address mapped to a physical address?

Answers to Activity Questions
1. A logical address is generated by the CPU, a physical address is the corresponding address in actual memory. User code generates logical addresses.
2. Paging allocates memory as a relatively large number of fixed-sized pages. Pages are small enough that little of them contain unused memory (i.e., internal fragmentation is low) A set of pages can have contiguous addresses, but the frames of physical memory they correspond to don't have to be contiguous. The user's logical address space consists of pages that can have contiguous addresses (which makes it easier for users to use), but the corresponding frames of physical memory don't have to be contiguous (which makes it more likely that a large set of memory can be allocated).
3. The page table maps page numbers to physical frame numbers. A logical address is a pair \((p, d)\) where \(p\) is the page number and \(d\) is the offset within the page. If \(P\) is the page table is, the corresponding physical address is \((P[p], d)\).