CS 326: Operating Systems

Memory

Lecture 12

Welcome Back

- Hope you had a great break!

P2

- A tip: don't hardcode paths, names, etc.
  - I saw some repos with "/home" in cd or prompt creation – why is this dangerous?
  - Be sure to use system calls / functions to get host names, user names, …
- Proposed timeline:
  - This week: History (and related command execution functions)
  - Next week: Pipes and redirection by Oct 26
  - Final week: Job control, bug fixes

Today's Schedule

- Mapped Memory
- Memory Management
- Virtual Memory
- Memory Addresses

Today's Schedule

- **Mapped Memory**
- Memory Management
- Virtual Memory
- Memory Addresses

Mapped Memory

- The last type of IPC we will cover is **mapped memory**
- Think back to using pthreads: we were able to access a shared memory pool
  - "Heap memory"
- We can also share regions of memory between processes!
- Unlike with pthreads, we can restrict sharing to a particular region of memory

mmap

- Mapping a region of memory is accomplished by the **mmap** function
- mmap asks the OS kernel to reserve a portion of memory and allow other process(es) to access it
- It's also possible to memory map a file
  - In fact, mmap() takes in a file descriptor
  - More efficient than reading/writing buffered data to/from disk manually

Working with Mapped Memory

- After the call to mmap completes, participating processes are given a void pointer
  - Just like with malloc
- This pointer provides the first offset of the shared memory region
- Processes can read/write directly as if they are working with their own memory space

Use Cases

- Mapped memory is extremely efficient
- Once set up, processes do **not** have to go through the kernel to communicate
  - Better performance
- It's also more error-prone than other IPC mechanisms
  - If a cooperating program writes to memory incorrectly, it's game over

Demo: mmap

Today's Schedule

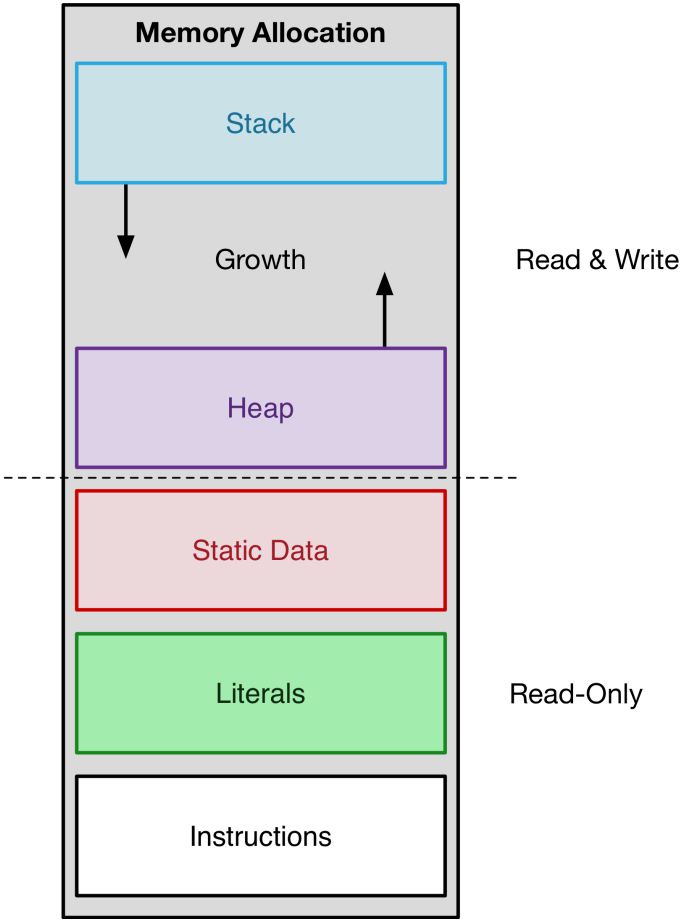
- Mapped Memory
- **Memory Management**
- Virtual Memory
- Memory Addresses

Virtualizing Memory

- Thus far, we've focused on how the OS runs and schedules processes
  - Each process can pretend it owns the machine
- However, this doesn't paint a complete picture…
- What about memory?
- How can we allow several processes to share system memory without them knowing it?

Memory Layout

- Look! There's that figure again!
- We've discussed how programs' memory is organized
- What happens when we have several programs running at once?



Isolating Memory

- How would you add support for memory isolation to an operating system?
  - …
- One approach: every time we swap out a process, let's store its current memory state to disk
- Great! Now we just add a bit more logic to our scheduler and we're off to the races!

Store-to-Disk Downsides

- What are some issues with this approach?
  - …
- Programs can't share memory with each other because everything is getting moved to disk
- The disk is **way too slow** to handle this
  - We may context switch every 100ms… say our program has 2 GB of memory in use and our disk can write 300 MB/s
    - Yikes!

Virtual Memory

- Since it's too inefficient to swap memory in and out from disk constantly, we need another approach
- We'll keep everything in memory!
  - Ok, great, except now processes can read each others' data!
- We need to provide processes with their own **address spaces**
  - Virtual addresses that correspond to actual hardware memory locations

Today's Schedule

- Mapped Memory
- Memory Management
- **Virtual Memory**
- Memory Addresses

Implementing Virtual Memory

- Making this work isn't too difficult: we will require programs to manage memory through the kernel
- When a program starts up, we'll give it its very own address space and **map** it to some location in memory
  - Process: "Whoa cool, I got memory starting at address 0! I must be the only process on this machine!"
  - Kernel: "Poor naïve process, you were actually allocated memory locations 3240 – 3800."

The Benefit of Virtual Memory

- Besides isolating processes, virtual memory also eases the burden on our software
- When you write a C program, you can just assume your memory addresses start at some particular location
  - No need to worry about how things work at a hardware level!
  - Imagine if the C runtime had to worry about other processes' memory
    - It'd be **way** more complex!

Translating Addresses

- When a process asks to read from memory location, for example, **0x20**, the OS determines its hardware address and performs the read operation
- We now have control over the portions of memory a process is allowed to read and write
- However… what are the downsides of this approach?

Address Translation Performance

- Doing all this work starts to sound somewhat like the "full virtualization" execution strategy
  - Slow!
- To boost performance, many CPUs include a **memory management unit** (MMU)
- Set up by the OS, this hardware feature enables fast translation from **logical addresses** to **physical addresses**

Address Types

- Logical address:
  - Memory location seen by a process
  - A **virtual** address
- Physical address:
  - **Actual** location in system memory (RAM) where the data resides

Translation

- How does the mapping from logical to virtual work?
- The simplest approach is to use **base** and **bound** addresses
  - Or in plain English: the starting memory location and the ending memory location
- If a program asks for 0x20, give it
  base + 0x20

Protection

- When accessing memory, we also need to make sure the request (**req**) is valid
- If req < base:

  // out of bounds!

  If req > bound:

  // out of bounds!
- What type of error would you expect to receive when you go out of bounds?

Memory Migration

- Using the base+bound pair, we can now **move** programs around in memory dynamically
- Simply copy the data to a new location, update the base+bound
- Want to increase the amount of virtual memory allocated to a process?
  - Increase its bound!

Today's Schedule

- Mapped Memory
- Memory Management
- Virtual Memory
- **Memory Addresses**

Inspecting Memory Addresses

- We can print out the memory addresses of pointers using the %p printf format specifier
- `printf("Address of a = %p\n", &a);`
- This allows us to determine the addresses of:
  - Stack
  - Heap
  - Literals
  - Uninitialized data
  - Code (functions)

Demo: **mem.c**

Gaps

- There is a giant gap between the stack and the heap… Is this wasted space?
- Interestingly, our nice diagram with memory stacked in order starting from 0 does not seem to hold up
  - (Has Prof Malensek been lying to you this whole time? Tune in next class to find out!)
- These gaps vary from system to system and have a variety of usage scenarios
  - Modern Macs have an interesting memory layout…

Wrapping Up: Memory

- We now can isolate processes' memory and even move memory around dynamically!
  - Neato!
- Unfortunately, memory is a bit more complex
- Consider the scenario where you:
  1. Start 3 processes
  2. Terminate process 2
- What happens to memory in this case?

Fragmentation

- When there are gaps between processes, we have **memory fragmentation**
- We could move processes around to eliminate fragmentation, but this is costly
  - Copies must be made, OS is has more work to do
- Another thing: what happens when process 1 or 2 grow too large and collide with a neighboring process?

Segmentation

- Instead of mapping large, contiguous blocks of space to processes, we will allocate **segments** of memory as necessary
  - **Memory Segmentation**
- Simple approach: put the stack in one place, heap in another, data somewhere else…
- Hmm, wonder what those "seg fault" things are?

Segmentation & Fragmentation