CS 326: Operating Systems

Segmentation

Lecture 13

Today's Schedule

- Dissecting Memory
- Augmenting the MMU
- Dealing with Fragmentation

Today's Schedule

- **Dissecting Memory**
- Augmenting the MMU
- Dealing with Fragmentation

Dissecting Memory

- We can print the memory locations of the stack, heap, etc. in C
- From a Raspberry Pi:
  - `Address of uninitialized data = 0x43405c`
  - `Address of initialized data   = 0x4238c4`
  - `Address of code               = 0x4236f0`
  - `Address of a (stack)          = 0x7ed9b2e4`
  - `Address of b (stack)          = 0x7ed9b2e8`
  - `Address of c (heap)           = 0x1f44150`
  - `Address of d (heap)           = 0x1f44160`
- Given this, what do we know about memory organization?

Organization

- The first thing we notice is that these locations may or may not match our logical model of memory
  - (See: stack, heap, code, etc. figure)
- They also vary depending on the machine we run our code on
  - macOS: stack seems to be growing ”up” (each subsequent address is smaller than the last). Heap grows “down”
  - Pi: both stack and heap seem to grow down?

Digging Deeper

- As you may suspect, the **virtual** memory allocated to processes is totally removed from **physical** memory
- ...Why?
- The first obvious reason: having a massive gap between the stack and heap wastes memory
  - How would we even figure out how much space to allocate to each process?
    - Too little: processes run out quickly
    - Too much: small processes waste resources

Tweaking the Address Space

- Before, we had a single **base** + **bound** pair per process
- Why not have a base + bound for each process **segment**?
  - Segment: one of those memory regions, like the stack or the heap
- This means the MMU must support **segmentation**

Segmented Addresses

- With segmentation, each segment has its own base + bound pair
- This allows for a **sparse** memory space
  - No need to allocate big, contiguous blocks of memory!
  - We can even move segments around if they need to be resized
- So now we finally know what a **segfault** really is: an illegal access outside of the segment

Handling a Segfault

- When an illegal access takes place, the MMU will generate a hardware **trap**
- This executes some privileged code in kernel space
- Next, a SIGSEGV signal is sent to the offending process
- (Usually) the process is terminated!

Today's Schedule

- Dissecting Memory
- **Augmenting the MMU**
- Dealing with Fragmentation

Augmenting the MMU

- Since we're going to allow for multiple segments, we need to update our concept of a hardware MMU
- Before, we tracked base + bound
- What do we need to track now?

Things to Keep Track Of

- The type of segment
- The base address
- Segment size
- ***Plus***:
  - Whether or not the segment grows up or down
  - Protection bits

Protection Bits

- Protection bits allow us to flag particular segments based on what they are intended to be used for
- For example, the **code** segment should be read-only
  - If it's not, then a program could modify itself at runtime… Essentially rewriting its code in memory
    - Sounds crazy, but is quite common for rogue software: if you can disable protection, you can insert your own executable code in memory!
- Stack/heap: **read+write**

Execute Bit

- Segments also support an **execute bit** – whether or not the location in memory can be executed by the CPU
- Code segment: **execute**
- Stack/heap: **not executable!**
  - Otherwise, you could load CPU instructions into memory, move the program counter there, and begin executing
    - So? Well, imagine if a user visiting your website figures out how to specially craft requests to modify memory and add instructions…

NX Bit

- Modern processors also support the **NX Bit**, which is separate from segmentation
- This allows entire regions of memory to be marked as non-executable
  - Segmentation's restrictions are very fine-grained in comparison

Context Switching

- Now that our MMU is more complex, how does this impact context switching?
- Well, mostly as we'd expect:
  - We can't just store a single base+bound, we also need to store the different segment locations, sizes, permissions, etc.
  - Context switching takes more effort in general

Today's Schedule

- Dissecting Memory
- Augmenting the MMU
- **Dealing with Fragmentation**

Fragmentation

- We briefly discussed fragmentation last class
- With segmentation, the likelihood that things are fragmented increases
  - Why?
- Lots more ”chunks” – segments of memory to deal with!

Handling Fragmentation

- We can **move** segments around, and as long as we update the physical addresses the process won't know the difference
- Unfortunately, moving memory requires copies to be made, and copies are bad
  - Slow
  - Requires enough free space to do the transfer

Another Approach

- Let's keep track of where segments are located in memory
- When we allocate a new segment, we'll find a suitable region of free space and put it there
- Need to know where all the segments are located

Free Memory Management Algorithms

- First fit – find the first free space available and put the segment there
- Best fit – find free space that closely matches the size of the segment
  - Optimal: exactly the same size
- Worst fit – find the largest empty region of memory and use that

Worst Fit?!

- Wait, why would you use a **worst** fit algorithm?
- Worst fit actually has an advantage: after allocating space, the remainder is probably **also** big enough to store a memory segment
- Whereas placing a 15 KB segment in a 16 KB free region will leave 1 KB… not as useful