

Today

- Memory Management
 - Virtual Memory Motivation and Requirements
- Project 5

Review

- What is RAID?
 - What is its motivation?
 - What are its goals?
 - What are some RAID levels?
 - What techniques do they use?
 - Benefits? Tradeoffs?
- What is the abstraction that virtual memory provides?

Review: RAID

- Disks fail
 - Want them to be more reliable
- Add redundant data to allow recovery in case of failure
- Improve performance with parallel reads/writes
- Costs/Tradeoffs:
 - Capacity overhead
 - Bandwidth overhead
- Approaches used:
 - Striping, mirroring, parity disk

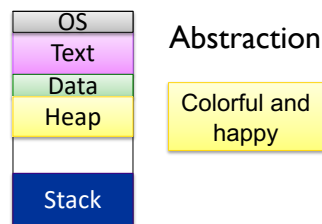
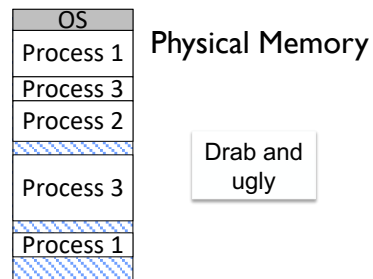
Nov 28, 2018

Sprenkle - CSCI330

3

Review: Memory

- Reality
 - there's only so much memory to go around
 - no two processes should use the same (physical) memory addresses.
- Abstraction goal: make every process think it has the same memory layout



Nov 28, 2018

Sprenkle - CSCI330

4

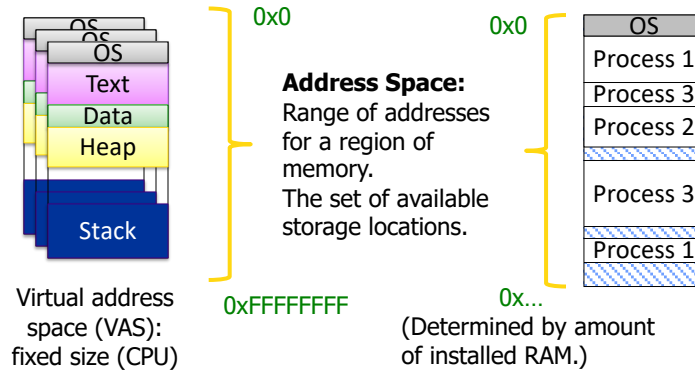
Review: Memory Terminology

Virtual (logical) Memory:

The abstract view of memory given to processes. Each process gets an independent view of the memory

Physical Memory:

The contents of the hardware (RAM) memory. Managed by OS. Only **one** of these for the entire machine!



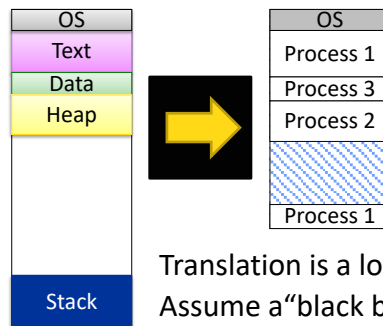
Nov 28, 2018

Sprenkle - CSCI330

5

Review: Address Translation

- Virtual addresses must be translated to physical addresses



Translation is a lot of work.
Assume a "black box" mechanism does it.

- Is logical addressing worth it/necessary?
- What do we want it to provide for us?

Nov 28, 2018

Sprenkle - CSCI330

6

User Perspective

- Average user doesn't care about "address spaces" or memory sizes
- User might say:
 - I want all of my programs to be able to run at the same time
 - I don't want to worry about running out of memory
- If OS has no virtual memory:
 - Best we can do is give them all of the physical memory
 - Is that enough?
 - VAS size can be larger than PAS...

Nov 28, 2018

Sprenkle - CSCI330

7

Multiprogramming, Revisited

- Multiple programs available to the machine, even if you only have one CPU core that can execute them.
- How to give the illusion: context switch quickly between processes on the CPU

Nov 28, 2018

Sprenkle - CSCI330

8

Multiprogramming, Revisited

- Can we do something analogous to a context switch for process memory?

A. Yes (how? Where will process memory be stored?)

B. No (why not?)

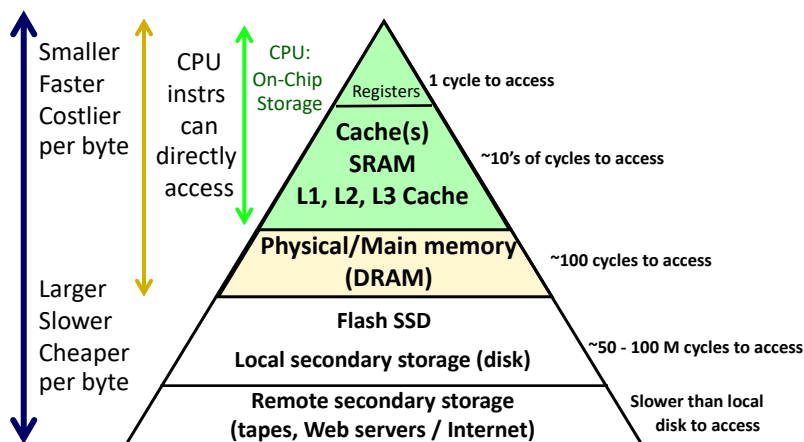
C. It depends (on what?)

Nov 28, 2018

Sprenkle - CSCI330

9

The Memory Hierarchy



Nov 28, 2018

Sprenkle - CSCI330

10

Memory Management

- Processor can only *directly* use data from registers
 - Need to move data closer (memory)
- Ideally, programmers want memory that is large, fast, and non-volatile
- Memory hierarchy
 - Small amount of fast, expensive memory – cache
 - Some medium-speed, medium-price – main memory
 - Gigabytes of slow, cheap disk storage – swap/virtual memory
- Multiprogramming makes memory management trickier

Nov 28, 2018

Sprenkle - CSCI330

11

Multiprogramming, Revisited

- Can we do something analogous to a context switch for process memory?
 - Suppose disk transfer rate is 100 MB/s
 - “switching” a 1 MB process would take 10 ms (+ disk seek time)
 - CPU context switch: approx. 10 – 50 μ s
 - Moving that 1 MB would make context switch take 200 – 1000 times longer!

Conclusion: We can't swap entirety of process memory on a context switch. It needs to be in memory already.

Nov 28, 2018

Sprenkle - CSCI330

12

Multiprogramming Requirements

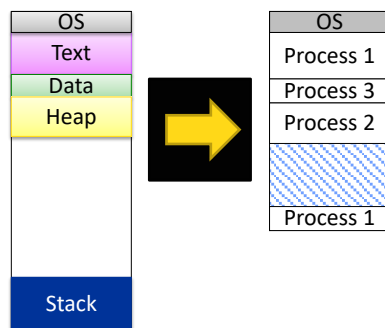
- Multiple processes **will** be in memory at the same time
 - Too costly to switch otherwise
- Processes should *not* be able to read/write each other's memory
 - unless we approve them to, with shared memory

Nov 28, 2018

Sprenkle - CSCI330

13

Address Translation: Wish List



- Map virtual addresses to physical addresses
- Allow multiple processes to be in memory at once, but isolate them from each other

Nov 28, 2018

Sprenkle - CSCI330

14

Using Disk

- We still have a large amount of [cheap]disk space though!
- If the total size of desired memory is larger than the Physical Address Space (PAS), overflow to disk
 - Disk: can store a lot, but relatively slow to access
 - Memory: much faster than disk, but can only store a subset

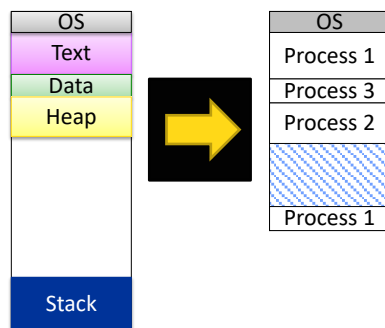
Caching!
(Swap Space)

Nov 28, 2018

Sprenkle - CSCI330

15

Address Translation: Wish List



- Map virtual addresses to physical addresses
- Allow multiple processes to be in memory at once, but isolate them from each other
- Determine which subset of data to keep in memory/move to disk

Nov 28, 2018

Sprenkle - CSCI330

16

Programmer Perspective

- Mix of user and compiler needs
 - High-level language: probably cares more about memory availability
 - Low-level language: probably cares a lot about memory addresses
- One major concern: library code
 - I want to #include lots of functionality for free!

Nov 28, 2018

Sprenkle - CSCI330

17

If multiple processes want to use the same library, how should we support that?

- A. Add a copy of the library code to the executable file at compile time.
- B. Load a copy of the library code into memory when the process begins executing.
- C. Map a shared copy of the library code in each process's virtual address space.

Nov 28, 2018

Sprenkle - CSCI330

18

Linking Tradeoffs

(A) Static Linking

- Bundle up one giant executable, with copies of all library code
 - Advantage: fully self-contained, not dependent on system libraries (*portable*)
 - Disadvantage: makes executables take up lots of space (on disk and in memory)

(B/C) Dynamic Linking

- Executable refers to external library code, which must be installed on system (or runtime error)
 - Advantage: memory efficiency, only one copy of library code needed
 - Disadvantage: must have library installed on system to use it

Nov 28, 2018

Sprenkle - CSCI330

19

Dynamic Libraries

- On Linux: `.so` (shared object) file
- On Windows: `.dll` (dynamically linked library) file
- Example: C standard library (`libc`)
 - Every process can use the same `libc` code (`printf`, `malloc`, `strlen`, etc.)

Displays shared objects required

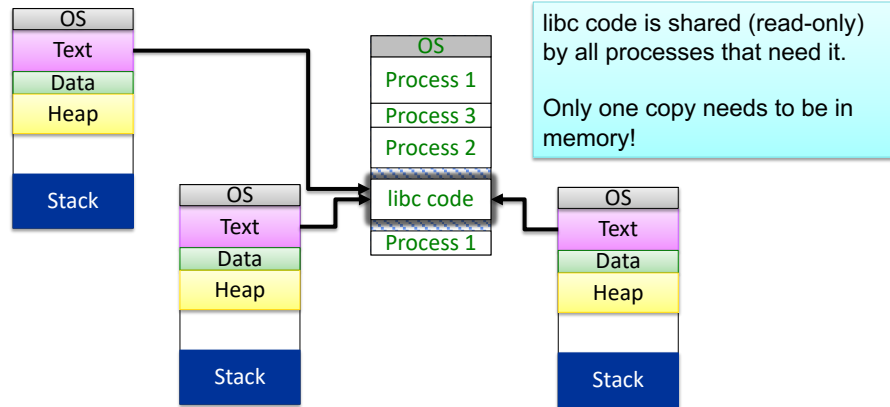
```
$ ldd strcmp_example
linux-vdso.so.1 (0x00007ffd41ffd000)
libc.so.6 => /lib64/libc.so.6 (0x00007fc954d72000)
/lib64/ld-linux-x86-64.so.2 (0x000055af5e366000)
```

Nov 28, 2018

Sprenkle - CSCI330

20

Dynamic Library in Memory

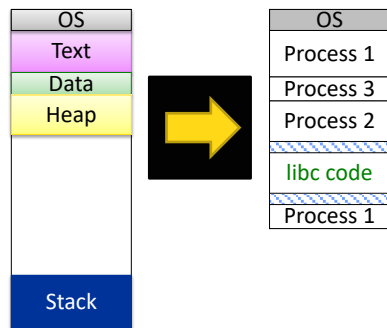


Nov 28, 2018

Sprenkle - CSCI330

21

Address Translation: Wish List



- Map virtual addresses to physical addresses
- Allow multiple processes to be in memory at once, but isolate them from each other
- Determine which subset of data to keep in memory/move to disk
- Allow the same physical memory to be mapped in multiple processes' VASes

Nov 28, 2018

Sprenkle - CSCI330

22

Compiler Perspective

- Compiler's goal: generate assembly code that will run... *later*.
- It generates the instructions for code and puts them somewhere in the resulting executable

Nov 28, 2018

Sprenkle - CSCI330

23

Changing the Program Counter

- Recall: PC register contains address of next instruction
- The compiler must change the PC when program control flow needs it
 - if / else: skip over some section of code
 - jump over instructions
 - loops: keep repeating the same code
 - jump back to same instructions
 - function call: execute code at some other location, come back later
- All of these cases: compiler must be setting the PC to *some* value

Nov 28, 2018

Sprenkle - CSCI330

24

Placing and Finding Code

Suppose we're generating code for two functions: f1() and f2(), and f1 calls f2

Option A: Choose addresses

```
f1: 0x1000 add %eax, %ecx
    ...
    0x100C call f2 (jump to 0x104C)
    ...
f2: 0x104C movl (%edx), %eax
    ...
    ret
```

Option B: Use relative addresses

```
f1: BASE add %eax, %ecx
    ...
    BASE + 0x0C call f2 (jump forward 0x40)
    ...
f2: BASE + 0x4C movl (%edx), %eax
    ...
    ret
```

Nov 28, 2018

Sprenkle - CSCI330

25

Placing and Finding Code

Now suppose we're generating a function that makes a *library call*

Option A: Choose addresses

```
f1: 0x1000 add %eax, %ecx
    ...
    0x100C call lib_f (jump to 0x0xF460)
    ...
    Elsewhere in memory...
lib_f: 0xF460 movl (%edx), %eax
    ...
    ret
```

Option B: Use relative addresses

```
f1: BASE add %eax, %ecx
    ...
    BASE + 0x0C movl (load LIB_BASE)
    BASE + 0x10 call f2 (jump to loaded LIB_BASE)
    ...
lib_f: LIB_BASE movl (%edx), %eax
    ...
    ret
```

Nov 28, 2018

Sprenkle - CSCI330

26

Which would you use? Why? How does it relate to OS / virtual memory?

Now suppose we're generating a function that makes a library call.

Option A: Choose addresses

```
f1: 0x1000 add %eax, %ecx
    ...
    0x100C call lib_f (jump to 0x0xF460)
    ...
```

Elsewhere in memory...

```
lib_f: 0xF460 movl (%edx), %eax
    ...
    ret
```

Option B: Use relative addresses

```
f1: BASE      add %eax, %ecx
    ...
    BASE + 0x0C movl (load LIB_BASE)
    ...
    BASE + 0x10 call f2 (jump to loaded
    LIB_BASE)
```

Elsewhere in memory...

```
lib_f: LIB_BASE movl (%edx), %eax
    ...
    ret
```

Nov 28, 2018

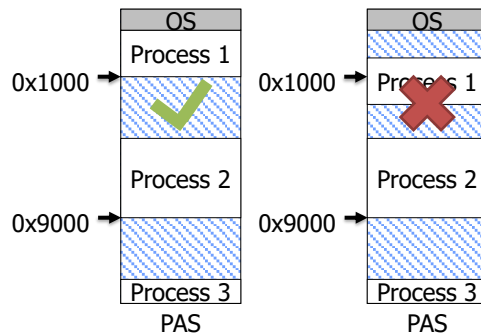
Sprenkle - CSCI330

27

Without Help (Virtual Memory or Hardware)

- Without help from the OS/hardware, can't do B.
- Option A works...*sometimes*.

```
f1: 0x1000 add %eax, %ecx
    ...
    0x100C call f2 (jump to 0x1050)
    ...
f2: 0x104C movl (%edx), %eax
    ...
    ret
```



Nov 28, 2018

Sprenkle - CSCI330

28

Challenge: Dynamic Environment

- Compiler can't realistically know:
 - When will the code run?
 - Which machine(s) will the code run on?
 - How much memory will be available at the time?
 - Where in the address space will that memory be available?

Conclusion: the compiler's job is much easier if it can rely on the OS/Hardware to help with placement.

With Virtual Memory (OS and Hardware)

Both options A and B work easily

- Compiler has abstract view of memory to use however it wants

For your local code generation
(VM provides the relative address)

Option A: Choose addresses

```
f1: 0x1000 add %eax, %ecx
    ...
    0x100C call lib_f (jump to 0x0xF460)
    ...
lib_f: 0xF460 movl (%edx), %eax
    ...
    ret
```

Elsewhere in memory...

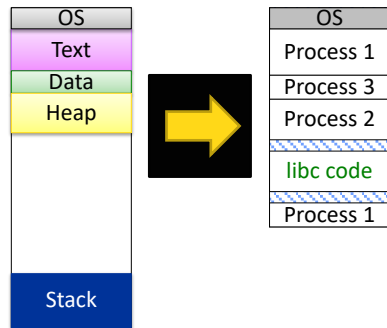
For shared libraries

Option B: Use relative addresses

```
f1: BASE add %eax, %ecx
    ...
    BASE + 0x0C movl (load LIB_BASE)
    BASE + 0x10 call f2 (jump to loaded LIB_BASE)
lib_f: LIB_BASE movl (%edx), %eax
    ...
    ret
```

Elsewhere in memory...

Address Translation: Wish List



- **Map virtual addresses to physical addresses**
- Allow multiple processes to be in memory at once, but isolate them from each other
- Determine which subset of data to keep in memory/move to disk
- Allow the same physical memory to be mapped in multiple process VASes

Nov 28, 2018

Sprenkle - CSCI330

31

OS Perspective

- Primary challenge: Which physical memory do we give to processes?
- Other important considerations:
 - Protection: OS is resource gatekeeper, must isolate itself (and processes)
 - Performance: OS should map memory for best performance, as long as it doesn't violate protection

Nov 28, 2018

Sprenkle - CSCI330

32

Without Virtual Memory Abstraction...

- Physical memory starts as one big empty space
- When starting new processes, allocate memory
 - At first, placement is easy: lots of large chunks free

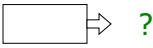


Nov 28, 2018

Sprenkle - CSCI330

33

Without Virtual Memory Abstraction...

- Physical memory starts as one big empty space
- When starting new processes, allocate memory
 - At first, placement is easy: lots of large chunks free
- Over time, processes will terminate, leaving gaps 
- Now we have to decide, for new processes, where should they go?



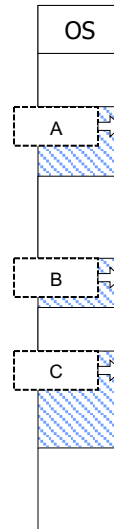
Nov 28, 2018

Sprenkle - CSCI330

Where should process P be placed?

- Why place it there?
 - Give an argument for each option

Process P →



Nov 28, 2018

Sprenkle - CSCI330

Where should process P be placed?

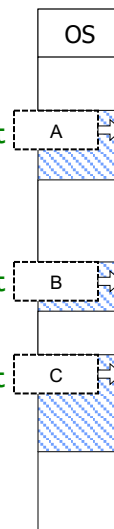
- First fit
 - Don't spend time searching!
- Best fit
 - It fits tightly!
 - Maybe no other process will fit in that spot
- Worst fit
 - Leaves lots of space for another process

Process P →

First Fit

Best Fit

Worst Fit

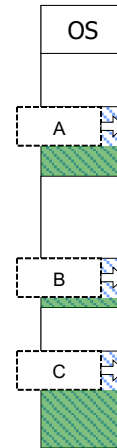


Nov 28, 2018

Sprenkle - CSCI330

(External) Fragmentation

- No matter where it ends up, the remaining gaps get smaller
- Large gaps are probably still usable, small ones likely aren't
- Fragmentation: over time, we end up with small gaps that become more difficult to use (eventually, wasted)
- “External” because the gaps are *between allocated pieces*



Nov 28, 2018

Sprenkle - CSCI330

37

Looking Ahead

- Project 5 due next Friday
 - [Background slides](#)

Nov 28, 2018

Sprenkle - CSCI330

38