CS 140: Midterm Review

6th February 2015

Administrivia

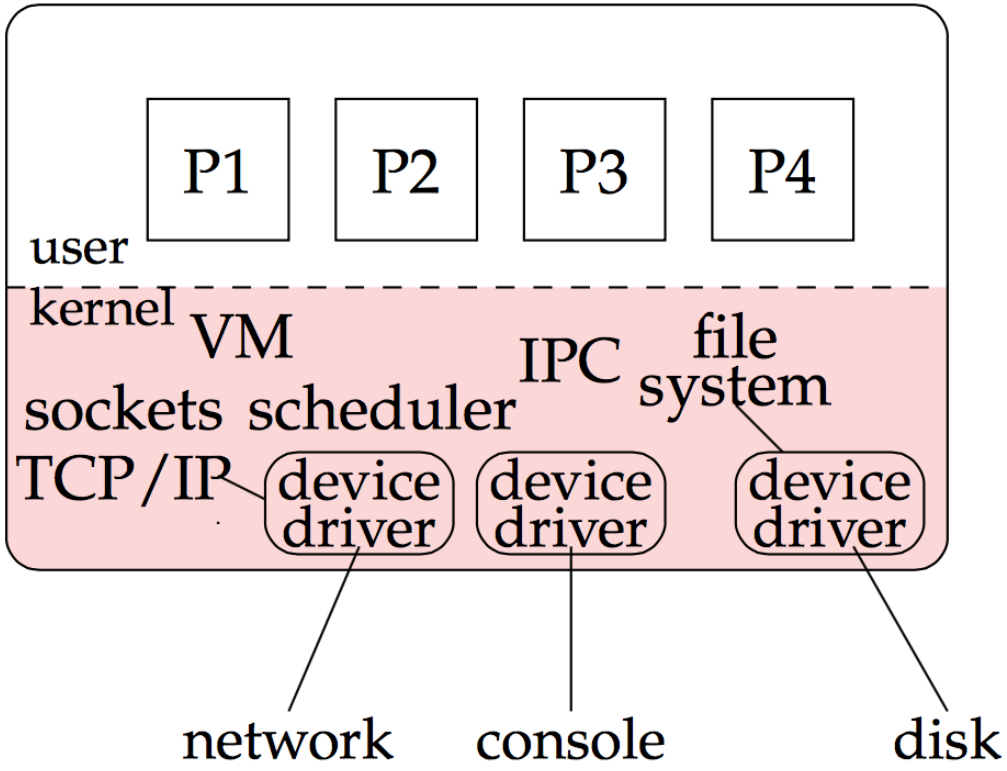
- **Project 2 was due at noon**
- **Project 3 will be due on Feb 27th**
  - Review Session on the 13th
- **Midterm on Monday**
  - Open notes + any material that you print
  - No electronic devices
  - No textbook

Midterm Coverage

- Processes and Threads
- Concurrency
- Scheduling
- Virtual Memory
- Synchronization
- Memory Allocation

Processes and Threads

- OSes need to provide some mechanisms for multitasking: *preemption* and *memory protection*



Processes and Threads

- Provide two examples of an application that benefits from multithreading, and two examples where multithreading does not provide better performance than a single-threaded solution.

Processes and Threads

- Yay Multithreading:
  - Application with User interface. UI thread separate from background processing thread gives better responsiveness.
  - Any program with heavy CPU usage along with I/O.
- Boo Multithreading:
  - Programs with heavy I/O, hardly any CPU usage. Bottleneck will always be at I/O, threading will only add more overhead.
  - Mainly sequential programs that modify small chunks of data. Synchronization overhead will outweigh threading benefits.

It’s all about context

- User -> Kernel?
  - Syscall, Page Fault
- User -> Interrupt Handler?
  - Hardware/Software Interrupt
- User/Process -> User?
  - Return

Kernel and User Threads

- Kernel threads: Faster than a process (still pretty heavy though), kernel does everything for you, every operation has to go through the kernel, one-size fits all.
- User threads: User library keeps a list of runnable threads, all threads belong to the process that started them, can be custom-built for particular applications

Concurrency

- What is Sequential Consistency?
  - We often assume that our machines support it
  - But they mostly don't.
  - SC prevents a lot a compiler optimizations and makes performance hardware a lot harder to design.
  - Other models? (refer to slides)
- How do we prevent data races?
  - Critical Sections! (These will have to guarantee **mutual exclusion**, **progress** and **bounded waiting times**)
  - You've seen different synchronization primitives in your Pintos projects

Concurrency

Initialization

```
int a = 4; int b = 0; int c = 0;
```

Thread 1

```
if (a < 0) {
    c = b – a;
} else {
    c = b + a;
}
```

Thread 2

```
b = 10;
a = -3;
```

What are the possible values for c after both threads complete? Assume that reads and writes of variables are atomic, and that the order of statements within each thread is preserved in the code generated by the C compiler (i.e., this code is sequentially consistent)

Concurrency

Initialization

```
int a = 4; int b = 0; int c = 0;
```

Thread 1

```
if (a < 0) {
    c = b – a;
} else {
    c = b + a;
}
```

Thread 2

```
b = 10;
a = -3;
```

Ans: 4, 7, 13, 14, -3

Concurrency

Consider the following three threads, where resource1, resource2, resource3 are generic resources that only one thread can hold at a time.

Can we have deadlock? Remember that we have 4 conditions for deadlock:

- Limited Access (Mutual Exclusion)
- No preemption
- Multiple Independent Requests
- Circularity in request graph

```
a () {
        acquire(resource1);
        acquire(resource2);
        doWorkA();
        release(resource2);
        release(resource1);
}
b (){
        acquire(resource2);
        acquire(resource3);
        doWorkB();
        release(resource3);
        release(resource2);
}
c () {
        acquire(resource3);
        acquire(resource1);
        doWorkC();
        release(resource1);
        release(resource3);
}
```

```
a () {
    acquire(resource1);
    acquire(resource2);
    doWorkA();
    release(resource2);
    release(resource1);
}
b (){
    acquire(resource2);
    acquire(resource3);
    doWorkB();
    release(resource3);
    release(resource2);
}
c () {
    acquire(resource3);
    acquire(resource1);
    doWorkC();
    release(resource1);
    release(resource3);
}
```

Yep! This could deadlock.

Mutual Exclusion. Each of the resources are mutually exclusive and only one thread can access at a time.

No Preemption. Resources cannot be forcibly taken back.

Multiple Independent Requests. Each of the resources is requested individually

Circularity. A can wait on B, who can wait on C, who in turn can wait on A

Scheduling

- What do we care about?

  Throughput, Turnaround Time, Response Time, CPU Utilization, Waiting Times

- Scheduling Algorithms:
  - FCFS (FIFO)
  - Shortest Job First (can also be SRTF)
  - Round-Robin
  - Priority Scheduling
  - Real-time Scheduling

Scheduling

(a) Describe a CPU scheduling algorithm that guarantees that no process will starve given a finite number of processes.

(b) Describe a CPU scheduling algorithm that can result in starvation. Provide an example workload that will result in starvation.

Scheduling

(a) Describe a CPU scheduling algorithm that guarantees that no process will starve given a finite number of processes.

Round-Robin Schedulers with time-slices and preemption

(b) Describe a CPU scheduling algorithm that can result in starvation. Provide an example workload that will result in starvation.

SJC with a lot of short jobs will starve long running software

Virtual Memory HW

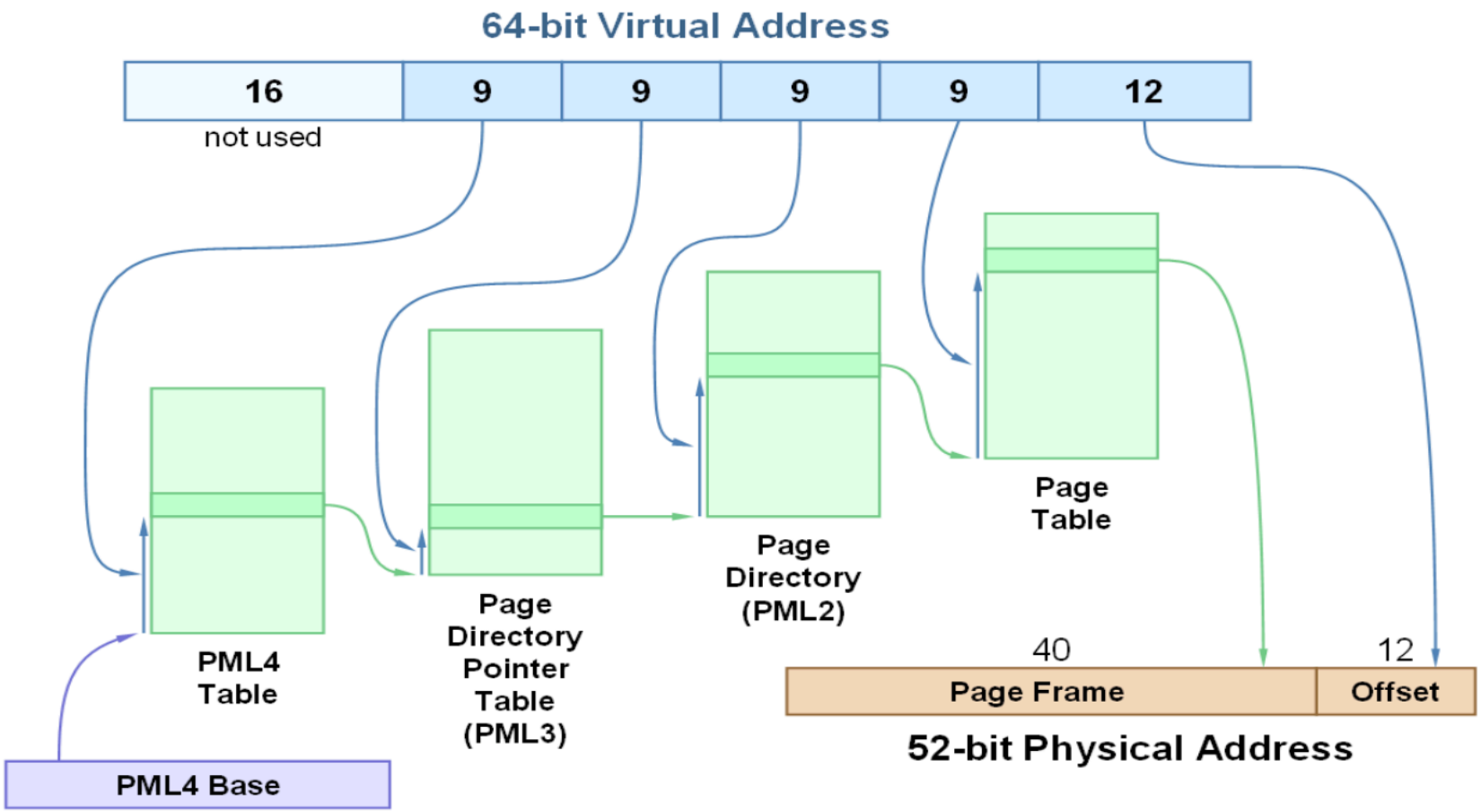
- Issues of *protection*, *transparency*, *resources*
  - Give each program its own *virtual* address space
- Segmentation: base + bounds registers
  - Simple, not transparent, *external* fragmentation
- Paging: map virtual pages to physical pages
  - Simplifies allocating and freeing, transparent, *internal* fragmentation
- x86 page translation
  - An interesting case. Has segmentation AND paging.

Virtual Memory OS

- We typically have more virtual memory than actual physical memory.
- But now, we need to decide what to fetch and what to eject
  - FIFO
  - LRU
  - Clock Algorithm
  - A few more
- Other issues: Thrashing, Working Sets, Memory Mapped Files

Virtual Memory

The following figure shows the address translation mechanism used for 64-bit mode in Intel x86 processors. Starting with a 48-bit virtual addresses, it uses 4 levels of page table to translate the address to a 52-bit physical address. Each page table entry is 8 bytes long.



If the page size is increased, the number of levels of page table can be reduced. How large must pages be in order to translate 48-bit virtual addresses with only 2 levels of page table? Draw the address translation mechanism corresponding to this page size.

Let the number of offset bits be x and number of bits to index into the page tables by y:

Total number of bits = 48, so..

$2y + x = 48$

Each page table needs to fit into a page, so..

$(2^y)*8 = 2^x$

$\Rightarrow y = 15;$

$\Rightarrow x = 18;$

$\Rightarrow$Page = (2^{18}) bytes = 256 kB per page

Synchronization

- Implementing shared locks
  - The multiple readers, single writer problem
- Dealing with ordering requirements can be hard
  - The test&set spinlock could be useless if we don't compensate for no sequential consistency
- Memory barriers in locks are useful for relaxed consistency models
- Performance needs caches, and that in turn needs coherence.

Synchronization

- cc-NUMA and spinlocks issues
  - Spinlocks are unfair and cause a lot of traffic
  - Either avoid spinlocks completely, or spin on a lock in local memory
- Locks and scheduling issues
  - Expensive to switch to the kernel
  - *futex*: ask kernel to sleep only if memory has changed

Memory Allocation

- Dynamic allocation's main problem is fragmentation
  - different lifetimes, different sizes
- Decisions: where in free memory to put a block?
  - There will always be pathological cases
  - Best fit: allocate to leave the smallest fragment
  - First fit: pick the best block that fits
- Other Stuff:
  - memory usage patterns (peaks, ramp, plateaus)
  - fault + resumption = power
  - distributed shared memory
  - garbage collection
  - reference counting

Best Of Luck!