Computer Science 162
Discussion Section
Week 2
Agenda

• Recap “What is an OS?” and Why?
• Process vs. Thread
• “THE” System
Note: Many slides are modifications of slides from Matei Zaharia
Who referenced slides from Steve Gribble, Ed Lazowska, Hank Levy, and John Zahorian
Why do we want an OS?

• **Isolation**
  – **Fault**: “if my program crashes yours shouldn’t”
  – **Performance**: “if my program starts to do some massive computation, it shouldn’t starve yours from running”

• **Mediation (multiplexing/sharing + protection)**
  – Manage the sharing of hardware resources (CPU, NIC, RAM, disk, keyboard, sound card, etc),

• **Abstractions and Primitives**
  – Set of constructs and well-defined interfaces to simplify application development: “all the code you didn’t write” in order to implement your application
    • Because hardware changes faster than applications!
    • Because some concepts are useful across applications
Why bother with an OS?

• **User benefits**
  – **Efficiency** (cost and performance)
    • share one computer across many users
    • concurrent execution of multiple programs
  – **Safety**
    • OS protects programs from each other
    • OS fairly multiplexes resources across programs

• **Application benefits**
  – **Simplicity**
    • sockets instead of ethernet cards
  – **Portability**
    • device independence: 3COM card or Intel card?
Concurrency and Parallelism

• Concurrency means *multiple threads of computation can make progress*, but possibly by sharing the same processor
  — Like doing homework while chatting on IM

• Parallelism means *leveraging multiple processors* to compute a result faster
  — Like dividing a pile of work among people
Why Concurrency?

- Consider a web server: while it’s waiting for a response from one client, it could read a request for another client.
- Consider a browser: while it’s waiting for a response from a web server, it wants to react to mouse or keyboard input.

Concurrency increases/enables responsiveness.
Why Parallelism?

- Because we actually have multiple CPUs!
- Because matrix multiply goes so much faster!

NOTE: Parallelism requires multiple processors, while concurrency also helps on a uniprocessor.
Lifecycle of a Thread (or Process)

- As a thread executes, it changes state:
  - **new**: The thread is being created
  - **ready**: The thread is waiting to run
  - **running**: Instructions are being executed
  - **waiting**: Thread waiting for some event to occur
  - **terminated**: The thread has finished execution

- “Active” threads are represented by their TCBs
  - TCBs organized into queues based on their state
How does OS do it?

• Kernel: The highly privileged code that carries out lowest level OS functions

• Use multiple processes, OS schedules them (i.e. multiplexes resources between them)
  – Each process has its own address space
  – Each process maintains a list of open files, open network connections ...

• Use multiple threads within a process, either OS or user schedules them
  – Threads share the process’s address space

Threads are cheaper than processes and can more easily share state! But have no isolation.
Recall, an OS needs to mediate access to resources: how do we share the CPU?

- **Strategy 1:** force everyone to cooperate
  - a thread willingly gives up the CPU by calling `yield()` which calls into the scheduler, which context-switches to another thread
  - what if a thread never calls `yield()`?

- **Strategy 2:** use preemption
  - at timer interrupt, scheduler gains control and context switches as appropriate
Review: Two Thread Yield Example

• Consider the following code blocks:

```plaintext
proc A() {
    B();
}
proc B() {
    while(TRUE) {
        yield();
    }
}
```

• Suppose we have 2 threads:
  – Threads S and T
“THE” System

• Dijkstra
  – Algorithm (shortest path)
  – OS (“THE”)  
  – Software Engineering (“GOTO Considered Harmful”)
  – Programming Language and Formal Verification
“THE” Multiprogramming System

• Why Multiprogramming?
  – A Reduction in turnaround time for short programs
  – Economic use of peripheral devices
  – Automatic control of backing store and efficient use of CPU
  – Need general processor but not all the power
Storage

• Core ->RAM, Drum ->Disk
• Separation of Virtual and Physical location
• Page Swap, the content of the page can be written to a different location on the Drum.
• No need for consecutive physical locations
Processor

• A Collection of Sequential Processes Working Together
• Process State
• Mutual Synchronization
Hierarchy

- Level 0 – Present a virtual processor
- Level 1 – Present virtual segments
- Level 2 – Present a virtual console
- Level 3 – Present a buffered IO interface to devices
- Level 4 – User Programs
Benefit of Layering

• Limited Interface
• Fewer Bugs
• Easier to Test
• Easier to Communicate
Semaphore

• Found in Appendix, but so Important!
• Shared between sequential processes for synchronization
• P -> decrease -> Possibly lock (if value <0)
• V -> increase -> unlock
• P, V are indivisible (atomic)