What does an OS do?

- Creates abstractions to make hardware easier to use
- Manages shared hardware resources

OS Abstractions

- Applications
- File system
- Virtual memory
- Operating System
- CPU
- Disk
- RAM

Upcoming Schedule

- This lecture starts a class segment that covers processes, threads, and synchronization
  - Perhaps the most important in this class
  - Basis for Projects 1 and 2

Managing Concurrency

- Recall: Source of OS complexity
  - Multiple users, programs, I/O devices, etc.
  - Originally for efficient use of H/W, but useful even now

  How to manage this complexity?
  - Divide and conquer
  - Modularity and abstraction

The Process

- The process is the OS abstraction for execution
  - Also sometimes called a job or a task
- A process is a program in execution
  - Programs are static entities with potential for execution

  Recall: For each area of OS, ask
  - What interface does the hardware provide?
  - What interface does the OS provide?

Process Components

- A process, named using its process ID (PID), contains all the state for a program in execution
  - Set of threads (active)
  - An address space (passive)
    - The code and input data for the executing program
    - The memory allocated by the executing program
    - An execution stack encapsulating the state of procedure calls
    - The program counter (PC) indicating the next instruction
    - A set of general-purpose registers with current values
    - Open files, network connections, etc.
Process Address Space

Multiple Threads
- Can have several threads in a single address space
  - Sometimes they interact
  - Sometimes they work independently
- What state is private to a thread?
  - Stack (and SP)
  - PC
  - Code
- What state is shared between threads?
  - Data segment

Review of Stack Frames
A(int tmp) {
    B(tmp);
}
B(int val) {
    C(val, val + 2);
    A(val - 1);
}
C(int foo, int bar) {
    int v = bar - foo;
}

Upcoming Topics
- Threads: unit of concurrency
  - How multiple threads can cooperate to accomplish a single task?
  - How multiple threads can share limited number of CPUs?
- Address spaces: unit of state partitioning
  - How multiple address spaces can share a single physical memory efficiently, flexibly, and safely?

Why do we need threads?
- Example: Web server
  - Receives multiple simultaneous requests
  - Reads web pages from disk to satisfy each request

Option 1: Handle one request at a time
- Example execution schedule:
  - Request 1 arrives
  - Server receives request 1
  - Server starts disk I/O 1a
  - Request 2 arrives
  - Server waits for I/O 1a to finish
  - Easy to program, but slow
    - Can’t overlap disk requests with computation, or with network receives
**Option 2: Event-driven web server (asynchronous I/O)**

- Issue I/Os, but don’t wait for them to complete
  - Request 1 arrives
  - Server receives request 1
  - Server starts disk I/O 1a to satisfy request 1
  - Request 2 arrives
  - Server receives request 2
  - Server starts disk I/O 2a to satisfy request 2

Web server must remember:
- What requests are being served, and what stage they’re in
- What disk I/Os are outstanding (and which requests they belong to)

Lots of extra state!

**Multi-threaded web server**

- One thread per request
  - Thread issues disk (or n/w) I/O, then waits for it to finish
  - Though thread is blocked on I/O, other threads can run
  - Where is the state of each request stored?

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request 1 arrives</td>
<td>Request 1 arrives</td>
<td>Request 3 arrives</td>
</tr>
<tr>
<td>Receive request 1</td>
<td>Receive request 2</td>
<td>Receive request 3</td>
</tr>
<tr>
<td>Start disk I/O 1a</td>
<td>Start disk I/O 2a</td>
<td></td>
</tr>
<tr>
<td>Disk I/O 1a finishes</td>
<td>Continue handling request 1</td>
<td></td>
</tr>
</tbody>
</table>

**Benefits of Threads**

- Thread manager takes care of CPU sharing
  - Other threads can progress when one thread issues blocking I/Os
  - Private state for each thread

- Applications get a simpler programming model
  - The illusion of a dedicated CPU per thread

**When are threads useful?**

- Multiple things happening at once
  - Usually some slow resource

- Examples:
  - Network server
  - Controlling a physical system (e.g., airplane controller)
  - Window system
  - Parallel programming

**Ideal Scenario**

- Split computation into threads
  - Threads run independent of each other
    - Divide and conquer works best if divided parts are independent

Is independence of threads practical?

**Dependence between threads**

- Example 1: Microsoft Word
  - One thread formats document
  - Another thread spell checks document

- Example 2: Desktop computer
  - One thread plays World of Warcraft
  - Another thread compiles EECS 482 project

- Two types of sharing: app resource or H/W
- Example of non-interacting threads?
Cooperating threads

- How can multiple threads cooperate on a single task?
  - Example: Ticketmaster’s webserver
  - Assume each thread has a dedicated processor
  - Later, we’ll talk about how to provide the illusion of infinite CPUs

- Problem:
  - Ordering of events across threads is non-deterministic
  - Speed of each processor is unpredictable

  Thread A
  Thread B
  Thread C

  Many possible global ordering of events
  Some may produce incorrect results

Non-deterministic ordering → Non-deterministic results

- Printing example
  Thread 1
  Print ABC
  Thread 2
  Print 123

  Possible outputs?
  » ABC321

  Ordering within thread is sequential, but many ways to merge per-thread order into a global order
  What's being shared between these threads?

- Arithmetic example (y is initially 10)
  - What’s being shared between these threads?
  - Possible results?
    » If A runs first: x = 11 and y = 20
    » If B runs first: x = 21 and y = 20

- Another example (x is initially 0)
  - Possible results?
    » x = 1 or -1
  - Impossible results?
    » x = 0

Atomic operations

- Before we can reason at all about cooperating threads, we must know that some operation is atomic
  - Indivisible, i.e., happens in its entirety or not at all
  - No events from other threads can occur in between the start and end of an atomic operation

  Print example:
  » What if each print statement were atomic?
  » What if printing a single character were not atomic?

  Most computers
  - Memory load and store are atomic
  - Many other instructions are not atomic
  » Example: double-precision floating point
  » Need an atomic operation to build a bigger atomic operation