Operating Systems Programming

Discussion Section

January 27, 2010
Introductions + Administrivia

• I’m Benjamin Hindman.
• You can email me at: 
  
  cs162-tc@inst.eecs.berkeley.edu

• Office Hours: 4:00 – 5:00 pm in 711 Soda (right after section).
Plan for Today

• Recap “what is an operating system” and why

• Processes and threads

• Getting started with Nachos

• Project group formation
Note: Many slides are modifications of 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 details 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
Why bother with an OS?

• **User benefits**
  – **Efficiency** (cost and speed)
    • 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?
What about the JVM? .NET? A Web Browser?
Concurrency vs Parallelism

• Concurrency means multiple computations executing simultaneously, 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 its waiting for a response from one client it could read a request for another client
• Consider a browser, while its waiting for a response from a web server it could react to any mouse or keyboard input

Doing things concurrently lets us be more efficient and increases responsiveness!
Why Parallelism?

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

True parallelism requires multiple processors, while concurrency is still helpful on a uniprocessor.
But how?

- Use multiple **processes**, OS *schedules* them (i.e. multiplexes resources between them)

- Use multiple **threads** within a process, either OS or user *schedules* them

  Threads are **cheaper** than processes and can more easily *share state*!
Process

• Process is not a program, its an instantiation of a program (i.e. lots of people might be running ‘java’ at the same time)
(old) Process address space

0xFFFFFFFF

address space

0x00000000

stack
(dynamic allocated mem)

heap
(dynamic allocated mem)

static data
(data segment)

code
(text segment)

PC

SP
(new) Address space with threads

0xFFFFFFFF

address space

0x00000000

thread 1 stack  

thread 2 stack  

thread 3 stack  

heap  
(dynamic allocated mem)

static data  
(data segment)

code  
(text segment)

SP (T1)

SP (T2)

SP (T3)

PC (T1)

PC (T2)

PC (T3)
Kernel threads

address space

thread

Mach, NT,
Chorus,
Linux, ...

os kernel

CPU

(thread create, destroy, signal, wait, etc.)
Are kernel level threads too expensive?

• Historically yes (thread operations require system calls), but aren’t too bad in practice today, if you use them correctly.

Alternatives?
User-level threads

address space

thread

os kernel

CPU

user-level thread library

(thread create, destroy, signal, wait, etc.)
User-level threads: what the kernel sees
User-level threads: the full story

- Mach, NT, Chorus, Linux, ...

- OS kernel

- CPU

- (kernel thread create, destroy, signal, wait, etc.)

- User-level thread library

- (thread create, destroy, signal, wait, etc.)

- Address space

- Thread
Are user-level threads the answer?

- No, Google “scheduler activations” for a great discussion of why user-level threads aren’t enough!
The design space

<table>
<thead>
<tr>
<th>Key</th>
<th>MS/DOS</th>
<th>older UNIXes</th>
</tr>
</thead>
<tbody>
<tr>
<td>address space</td>
<td>one thread per process</td>
<td>many threads per process</td>
</tr>
<tr>
<td>thread</td>
<td>one process</td>
<td>many processes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Java</th>
<th>many threads per process</th>
<th>Mach, NT, Chorus, Linux, ...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>one process</td>
<td></td>
</tr>
<tr>
<td></td>
<td>many processes</td>
<td></td>
</tr>
</tbody>
</table>
Recall, an OS is meant 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 ready thread
  – what if a thread never calls `yield()`?

• Strategy 2: use preemption
  – at timer interrupt, scheduler gains control and 
    context switches as appropriate
We’ll talk more about threads and processes next week!
Course Project

• 4 Phases
  – Threads, Multiprogramming, Caching and Virtual Memory, Networks and Distributed Systems

• Nachos
Nachos Demonstration

• Compiling Nachos
• Running Nachos
• Changing Nachos Configuration
• Changing/Adding code