Review: System Calls
P can only see its own memory because of \textit{user mode}
(other areas, including kernel, are hidden)
Process P

P wants to call read()
movl $6, %eax;  int $64
static int (*syscalls[])(void) (syscall.c)

Process P

movl $6, %eax; int $64

syscall-table index
struct gatedesc idt[256] (trap.c)

Process P

movl $6, %eax;  int $64

syscall-table index  trap-table index
movl $6, %eax;   int $64

syscall-table index

trap-table index
Kernel mode: we can do anything!

Process P

RAM

```
movl $6, %eax;  int $64
```

syscall-table index

trap-table index
Process P

```
movl $6, %eax;  int $64
```

syscall-table index

trap-table index

RAM
Process P

RAM

movl $6, %eax;  int $64

syscall-table index

trap-table index
`movl $6, %eax; int $64`

**syscall-table index**

**trap-table index**
movl $6, %eax; int $64

syscall-table index

trap-table index
Processes
What’s a Process?

Java analogy:

class => “program”
object => “process”

Programs are just code.
Processes are running programs.

A process is an instance of a program.
There may be 0 or more processes per program.
Process Creation

- CPU
- Memory

Program

- code
- static data
Process Creation

CPU

Memory

- code
- static data
- heap
- stack
- Process

Program

- code
- static data
What’s in a Process?

Processes share code, but each has its own “context”

CPU
   Instruction Pointer (aka Program Counter)
   Stack Pointer

Memory
   set of memory addresses (“address space”)
   `cat /proc/<PID>/maps`

Disk
   set of file descriptors
   `cat /proc/9506/fdinfo/*`
Do we enough CPUs?

Linux commands:

```
ps ax | wc

top

cat /proc/cpuinfo | grep 'model name'
```
How do we share?

CPU?

Memory?

Disk?
How do we share?

CPU? (a: time sharing)

Memory? (a: space sharing)

Disk? (a: space sharing)
How do we share?

CPU? (a: time sharing)  TODAY

Memory? (a: space sharing)

Disk? (a: space sharing)
How do we share?

CPU? (a: time sharing)  
TODAY

Memory? (a: space sharing)

Disk? (a: space sharing)

Goal: processes should NOT even know they are sharing (each process will get its own virtual CPU)
What to Do with Processes That Are Not Running?

A: store context in OS struct

Look in kernel/proc.h

context (CPU registers)
ofile (file descriptors)
state (sleeping, running, etc)
What to Do with Processes That Are Not Running?

A: store context in OS struct

Look in kernel/proc.h

- **context** (CPU registers)
- **ofile** (file descriptors)
- **state** (sleeping, running, etc)
State Transitions

- Running
- Ready
- Blocked

Transition arrows:
- Running to Ready: Descheduled
- Ready to Running: Scheduled
- I/O: initiate to Blocked
- Blocked to I/O: done
State Transitions

Running → Descheduled → Scheduled → Ready

I/O: initiate → Blocked → I/O: done

View process state with “ps xa”
How to transition? ("mechanism")
When to transition? ("policy")

- Running
- Ready
- Blocked

- I/O: initiate
- I/O: done
Administrative Stuff

- P1 due on 9/16 (eight days left!)
- Office hours: today after class (in lab), Wed 2-3pm
- Exam prep: understand book and exams
- Reading: chapters 1-2 (last time) and 3-6 (today)
- Learning names
- Wait list: good news!
CPU Time Sharing

Goal 1: efficiency
OS should have minimal overhead

Goal 2: control
Processes shouldn’t do anything bad
OS should decide when processes run

Solution: limited direct execution
Limited Direct Execution
What to limit?

- General memory access
- Disk I/O
- Special x86 instructions like `lidt`

How? Get HW help, put processes in “user mode”
What to limit?

General memory access

Disk I/O

Special x86 instructions like `lidt`

How? Get HW help, put processes in “user mode”
lidt example

Process P

RAM

trap-table index

syscall-table index
$\text{P tries to call $\text{lidt}$!}$
The diagram illustrates the process of the `lidt` example. The CPU warns the OS, and the OS eventually kills process P.

Key components include:
- **RAM**: Represents memory where trap-table and syscall-table indices are located.
- **trap-table index**: Indicates the starting point for trap-table entries.
- **syscall-table index**: Indicates the starting point for syscall-table entries.

The diagram visually aligns these components to show the flow and interaction within the system.
Context Switch

Problem: when to switch process contexts?

Direct execution => OS can’t run while process runs

How can the OS do anything while it’s not running?
Context Switch

Problem: when to switch process contexts?

Direct execution => OS can’t run while process runs

How can the OS do anything while it’s not running? A: it can’t
Problem: when to switch process contexts?

Direct execution => OS can’t run while process runs

How can the OS do anything while it’s not running? A: it can’t

Solution: **switch on interrupts**. But which interrupt?
Cooperative Approach

Switch contexts for syscall interrupt.

Provide special `yield()` system call.
Cooperative Approach

Switch contexts for *syscall interrupt*.

Provide special `yield()` system call.
Cooperative Approach

Switch contexts for **syscall interrupt**.

Provide special **yield()** system call.

```
P1
```

```
yield() call
```
Cooperative Approach

Switch contexts for **syscall interrupt**.

Provide special **yield()** system call.
Cooperative Approach

Switch contexts for syscall interrupt.

Provide special yield() system call.
Cooperative Approach

Switch contexts for **syscall interrupt**.

Provide special `yield()` system call.

↑ `yield()` return

OS
Cooperative Approach

Switch contexts for syscall interrupt.

Provide special \texttt{yield()} system call.

\begin{align*}
\text{P2} & \quad \text{yield()} \text{ return}
\end{align*}
Cooperative Approach

Switch contexts for **syscall interrupt**.

Provide special **yield()** system call.

P2
Cooperative Approach

Switch contexts for syscall interrupt.

Provide special yield() system call.
Cooperative Approach

Switch contexts for **syscall interrupt**.

Provide special **yield()** system call.
Cooperative Approach

Switch contexts for syscall interrupt.

Provide special yield() system call.
Cooperative Approach

Switch contexts for **syscall interrupt**.

Provide special `yield()` system call.

```
yield() return
```
Cooperative Approach

Switch contexts for \texttt{syscall interrupt}.

Provide special \texttt{yield()} system call.
Cooperative Approach

Switch contexts for *syscall interrupt*.

Provide special `yield()` system call.
Non-Cooperative Approach

Switch contexts on timer interrupt.

Set up before running any processes.

HW does not let processes prevent this.

Is it better to be cooperative or non-cooperative?
<table>
<thead>
<tr>
<th>Operating System</th>
<th>Hardware</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Process A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Operating System</td>
<td>Hardware</td>
<td>Program</td>
</tr>
<tr>
<td>------------------</td>
<td>----------</td>
<td>---------</td>
</tr>
</tbody>
</table>
|                  | timer interrupt \  
save regs(A) to k-stack(A) \  
move to kernel mode \  
jump to trap handler | Process A \  
... |
Handle the trap
Call **switch()** routine
  save regs(A) to proc-struct(A)
  restore regs(B) from proc-struct(B)
  switch to k-stack
  return-from-trap (into B)

**Hardware**
- timer interrupt
  - save regs(A) to k-stack(A)
  - move to kernel mode
  - jump to trap handler

**Program**
- Process A
  ...

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Hardware</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Process A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Operating System</td>
<td>Hardware</td>
<td>Program</td>
</tr>
<tr>
<td>------------------</td>
<td>----------</td>
<td>---------</td>
</tr>
</tbody>
</table>
| Handle the trap  | timer interrupt  
save regs(A) to k-stack(A)  
move to kernel mode  
jump to trap handler | Process A |
| Call **switch()** routine  
save regs(A) to proc-struct(A)  
restore regs(B) from proc-struct(B)  
switch to k-stack  
return-from-trap (into B) | restore regs(B) from k-stack(B)  
move to user mode  
jump to B’s IP | ... |
<table>
<thead>
<tr>
<th>Operating System</th>
<th>Hardware</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handle the trap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call <strong>switch()</strong> routine</td>
<td></td>
<td></td>
</tr>
<tr>
<td>save regs(A) to proc-struct(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>restore regs(B) from proc-struct(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>switch to k-stack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return-from-trap (into B)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|            |          |         |
| timer interrupt |          |         |
| save regs(A) to k-stack(A) |          |         |
| move to kernel mode |          |         |
| jump to trap handler |          |         |

|            |          |         |
| restore regs(B) from k-stack(B) |          |         |
| move to user mode |          |         |
| jump to B’s IP |          |         |

**Process A**

... 

**Process B**

...
Summary

• Smooth context switching makes each process think it has its own CPU (virtualization!)

• Direct execution makes processes fast

• Hardware provides a lot of OS support
  - limited direct execution
  - timer interrupts
  - automatic register saving
Things to Look Forward to

- CPU-sharing policy (Wed lecture)
- Process APIs (Thu discussion)
  Also: syscall timing and more C review
- Memory virtualization (next Mon lecture)