[537] Processes

Tyler Harter
Thinking about systems

- Abstractions
- Mechanisms
- Policies
Thinking about systems

- **Abstractions** [TODAY: processes and virtual CPUs]
- Mechanisms
- Policies
Thinking about systems

- **Abstractions** [TODAY: processes and virtual CPUs]
- **Mechanisms** [TODAY: exceptions and LDE]
- **Policies**
Thinking about systems

• **Abstractions** [TODAY: processes and virtual CPUs]

• **Mechanisms** [TODAY: exceptions and LDE]

• **Policies** [NEXT WEEK: CPU Scheduling]
What is a process?
What is a Process?

Programs are code.
Processes are running programs.

Java analogy:

- class => “program”
- object => “process”
What is in a Process?

what things change as a program runs?
What is in a Process?

what things change as a program runs?

process

memory
What is in a Process?

what things change as a program runs?
What is in a Process?

what things change as a program runs?

- registers
- memory
- CODE
- HEAP
- STACK
What is in a Process?

What things change as a program runs?
What is in a Process?

what things change as a program runs?
Peeking Inside

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/*`
Process Creation
Process Creation

CPU

Memory

code
static data
Program
Process Creation

CPU

Memory

- code
- static data
- heap
- stack
- Process

Program

code
static data
Program
Process Creation

CPU

Memory

- code
- static data
- heap
- stack
- Process

Program

- code
- static data
Process Creation

CPU

PC

Memory

code
static data
heap

stack
Process

code
static data
Program
Process Creation

CPU

PC

Memory

code

static data

heap

stack

Process

Before, PC pointed at kernel code
Process Creation

Now, CPU begins directly executing process code
Challenge: how to prevent process from doing "kernel stuff"
Limited Direct Execution
Goals

Make processes fast

Don’t let them cause harm

Provide way to do sensitive things safely
Limited Direct Execution

Usually let processes run with no OS involvement

Limit what processes can do

Offer privileged operations through well-defined channels with help of OS.
What to limit?

General memory access

Disk I/O

Certain x86 instructions
How to limit?

Need H/W support

Add additional execution mode to CPU

“user mode” — restricted, limited capabilities

“kernel mode” — not restricted

**Processes** start in **user mode**, **Oses** start in **kernel mode**.
LDE: Remaining challenges

(1) What if P wants to do something privileged?

(2) What if we want to run a different process?

Problem: how can OS switch processes (or do anything) if it’s not running?
LDE: Remaining challenges

1) What if P wants to do something privileged?

2) What if we want to run a different process?

Problem: how can OS switch processes (or do anything) if it's not running?
Taking turns

- Process
- OS
- Hardware
Taking turns

Process running

OS

Hardware
Taking turns

- Process
- OS: running
- Hardware
Taking turns

- Hardware
- OS
- Process running
Taking turns

when/how do we switch to OS?
Exceptions
Interrupt

- Process
- OS
- Hardware
Interrupt

Process

OS

Hardware key
Interrupt

Process

OS

Hardware

hardware "interrupt"
Interrupt

Process

OS

Hardware
System Call

- Process
- OS
- Hardware
System Call

- Process
  - open
- OS
- Hardware
System Call

Process  open
OS  handler
Hardware

system call “trap”
System Call

- Process
- OS
- Hardware
Exception Handling
Implementation

**Goal**: processes and H/W should be able to call functions in the OS
Implementation

**Goal**: processes and H/W should be able to call functions in the OS

Functions should be:

- at *well-known* locations
- *safe* from processes
Implementation

**Goal**: processes and H/W should be able to call functions in the OS

Functions should be:
- at **well-known** locations
- **safe** from processes
hard code handler addresses into CPU?
what if layout differs?
use array of function pointers (H/W knows where this is)
use array of function pointers (H/W knows where this is)
use array of function pointers
(H/W knows where this is)
How does H/W know where trap table is?
How does H/W know where trap table is? *lidt* instruction
how to handle variable number of system calls?
trap table

syscall

syscall table

disk
network
timer
keyboard
system call
open
read
getpid
Implementation

**Goal**: processes and H/W should be able to call functions in the OS

Functions should be:
- at *well-known* locations
- *safe* from processes
Implementation

**Goal**: processes and H/W should be able to call functions in the OS

Functions should be:

- at well-known locations
- **safe** from processes
Safe Transfers

Only certain kernel functions should be callable.

Privileges should escalate at the moment of the call.
  • examples?
Safe Transfers

Only certain kernel functions should be callable

Privileges should escalate at the moment of the call

- read/write disk
- kill processes
- access all memory
- etc
Safe Transfers

Only certain kernel functions should be callable

Privileges should escalate at the moment of the call

- read/write disk
- kill processes
- access all memory
- etc

H/W support: use x86 int, instead of jmp or call

- escalate priority and call simultaneously
System Call Example
Process P
P can only see its own memory because of user mode
(other areas, including kernel, are hidden)
Process P

P wants to call read()
movl $6, %eax;  int $64
movl $6, %eax;  int $64

trap-table index
static int (*syscalls[]) (void) (syscall.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

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
movl $6, %eax;  int $64

syscall-table index  trap-table index
```c
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \ 
  name: \ 
  movl $SYS_ ## name, %eax; \ 
  int $T_SYSCALL; \ 
  ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
```
```c
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
    .globl name; \
    name: \
    movl 6, %eax; \
    int 64; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
```
Malicious Example
lidt example

Process P

RAM

trap-table index

syscall-table index
lidt example

Process P

RAM

trap-table index

syscall-table index
Process P tries to call \texttt{lidt}!
CPU warns OS, OS kills P
LDE: Remaining challenges

(1) What if P wants to do something privileged?

(2) What if we want to run a different process?

Problem: how can OS switch processes (or do anything) if it’s not running?
LDE: Remaining challenges

(1) What if P wants to do something privileged?

(2) What if we want to run a different process?

Problem: how can OS switch processes (or do anything) if it’s not running?
LDE: Remaining challenges

(1) What if P wants to do something privileged?

(2) What if we want to run a different process? Why?

**Problem**: how can OS switch processes (or do anything) if it’s not running?
Sharing the CPU
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)

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 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**
  - Descheduled to **Ready**
  - Scheduled to **Blocked**
  - I/O: initiate

- **Ready**
  - Scheduled to **Running**
  - I/O: done

- **Blocked**
  - Descheduled from **Running**
State Transitions

- **Running**
- **Ready**
- **Blocked**

Transition arrows:
- Running to Ready: **Scheduled**
- Ready to Running: **Scheduled**
- Running to Blocked: **I/O: initiate**
- Blocked to Running: **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**

- **Descheduled**
- **Scheduled**
Mechanism: Context Switch
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
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

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

Switch contexts for syscall interrupt.

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

Switch contexts for syscall interrupt.

Provide special `yield()` system call.
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.

![Diagram](image.png)
Cooperative Approach

Switch contexts for syscall interrupt.

Provide special yield() system call.
Cooperative Approach

Switch contexts for syscall interrupt.

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

\texttt{yield()} return
Cooperative Approach

Switch contexts for syscall interrupt.

Provide special `yield()` system call.

P2

`yield()` return
Cooperative Approach

Switch contexts for syscall interrupt.

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

Switch contexts for **syscall interrupt**.

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

```
P2
yield() call
```
Cooperative Approach

Switch contexts for **syscall interrupt**.

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

Switch contexts for syscall interrupt.

Provide special \texttt{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()** return
Cooperative Approach

Switch contexts for **syscall interrupt**.

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

Switch contexts for **syscall interrupt**.

Provide special `yield()` system call.

Critiques?
Non-Cooperative Approach

Switch contexts on timer interrupt

Set up before running any processes

H/W does not let processes prevent this
<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></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>
<tr>
<td></td>
<td>timer interrupt</td>
<td>Process A</td>
</tr>
<tr>
<td></td>
<td>save regs(A) to k-stack(A)</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>move to kernel mode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>jump to trap handler</td>
<td></td>
</tr>
<tr>
<td>Operating System</td>
<td>Hardware</td>
<td>Program</td>
</tr>
<tr>
<td>------------------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td>Handle the trap</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Call **switch()** routine | save regs(A) to k-stack(A) | Process A  
<p>| save regs(A) to proc-struct(A) | move to kernel mode | ... |
| restore regs(B) from proc-struct(B) | jump to trap handler | |
| switch to k-stack | return-from-trap (into B) | |</p>
<table>
<thead>
<tr>
<th>Operating System</th>
<th>Hardware</th>
<th>Program</th>
</tr>
</thead>
</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 | restore regs(B) from k-stack(B)  
move to user mode  
jump to B’s IP | |
| save regs(A) to proc-struct(A)  
restore regs(B) from proc-struct(B)  
switch to k-stack  
return-from-trap (into B) | | |
<table>
<thead>
<tr>
<th>Operating System</th>
<th>Program</th>
</tr>
</thead>
</table>
|                  | Process A ...
|                  | Process B ...

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

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

P1

tick
Preemptive Approach
Preemptive Approach
Preemptive Approach
Preemptive Approach
Preemptive Approach
Preemptive Approach
Preemptive Approach

OS

tick
Preemptive Approach
Preemptive Approach
Preemptive Approach
Preemptive Approach
Preemptive Approach

Is preemptive always better than cooperative?
Summary

- Smooth **context switching** makes each process think it has its own CPU (virtualization!)
- **Limited direct execution** makes processes fast
- Hardware provides a lot of OS support
  - limited direct execution
  - timer interrupts
  - automatic register saving