VIRTUALIZATION: THE CPU

Shivaram Venkataraman
CS 537, Spring 2019
- Project 1a is out! Due Jan 29 at 11.59pm
- Signup for Piazza https://piazza.com/wisc/spring2019/cs537
- Lecture notes at pages.cs.wisc.edu/~shivaram/cs537-sp19/
- Drop? Waitlist? Email enrollment@cs.wisc.edu cc me
AGENDA / OUTCOMES

Abstraction

What is a Process? What is its lifecycle?

Mechanism

How does process interact with the OS?
How does the OS switch between processes?
ABSTRACTION: PROCESS
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[]) {
    char *str = argv[1];

    while (1) {
        printf("%s\n", str);
        Spin(1);
    }
    return 0;
}
WHAT IS A PROCESS?

Stream of executing instructions and their “context”

Instruction Pointer

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Registers</th>
<th>Memory addrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushq %rbp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>movq %rsp, %rbp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subq $32, %rsp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>movl $0, -4(%rbp)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>movl %edi, -8(%rbp)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>movq %rsi, -16(%rbp)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmpl $2, -8(%rbp)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>je LBB0_2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

File descriptors
PROCESS CREATION

CPU

Memory

code
static data
Program
Can run multiple instances of same program

Each program has its own stack, heap etc.
PROCESS VS THREAD DEMO
Threads: “Lightweight process”

- Execution streams that share an address space
- Can directly read / write memory
- Can have multiple threads within a single process
WHY DO WE NEED PROCESSES?
How do we share CPU between processes?
TIME SHARING

code, static data, heap
stack

code, static data, heap
stack

code, static data, heap
stack

CPU
TIME SHARING

code, static data
heap
stack
code, static data
heap
stack
code, static data
heap
stack

CPU
TIME SHARING

code, static data
heap
stack
code, static data
heap
stack
code, static data
heap
stack

CPU
WHAT TO DO WITH PROCESSES THAT ARE NOT RUNNING?

OS Scheduler
- Save context when process is paused
- Restore context on resumption
STATE TRANSITIONS

Running

Descheduled

Ready

Scheduled

Blocked

I/O: initiate

I/O: done
STATE TRANSITIONS

- Running
- Ready
- Scheduled
- Blocked
- Descheduled

I/O: initiate

I/O: done
ASIDE: OSTEP HOMEWORKS!

- Optional homeworks corresponding to each chapter in book
- Little simulators to help you understand
- Can generate problems and solutions!

http://pages.cs.wisc.edu/~remzi/OSTEP/Homework/homework.html
Run ./process_run.py -l 2:100,2:0
≥ ./process-run.py -l 3:50,3:40
Process 0
  io
  io
  io
  cpu

Process 1
  cpu
  io
  io
CPU TIME SHARING

Policy goals

Virtualize CPU resource using processes
Reschedule process for fairness? efficiency?

Mechanism goals

Efficiency: Time sharing should not add overhead
Control: OS should be able to intervene when required
EFFICIENT EXECUTION

Simple answer !?: **Direct Execution**

- Allow user process to run directly
- Create process and transfer control to main()

Challenges

- What if the process wants to do something restricted? Access disk?
- What if the process runs forever? Buggy? Malicious?

Solution: **Limited Direct Execution (LDE)**
**PROBLEM 1: RESTRICTED OPS**

How can we ensure user process can’t harm others?

**Solution:** privilege levels supported by hardware (bit of status)
- User processes run in user mode (restricted mode)
- OS runs in kernel mode (not restricted)

How can process access devices?

- **System calls** (function call implemented by OS)
SYSTEM CALL
Process P wants to call read()
Process P

RAM

P can only see its own memory because of user mode (other areas, including kernel, are hidden)
P wants to call read() but no way to call it directly
movl $6, %eax;  int $64
movl $6, %eax;  int $64
SYSTEM CALL

Process P

movl $6, %eax;  int $64

Syscall table index
SYSTEM CALL

Process P

RAM

Syscall table index

movl $6, %eax;
int $64

Trap table index
Follow entries to correct system call code

```assembly
movl $6, %eax; int $64
```
Kernel can access user memory to fill in user buffer
return-from-trap at end to return to Process P
SYSCALL SUMMARY

Separate user-mode from kernel mode for security

Syscall: call kernel mode functions
  Transfer from user-mode to kernel-mode (trap)
  Return from kernel-mode to user-mode (return-from-trap)
Why not directly specify sys_read address from user-mode?
PROBLEM 2: HOW TO TAKE CPU AWAY

Policy
To decide which process to schedule when
Decision-maker to optimize some workload performance metric

Mechanism
To switch between processes
Low-level code that implements the decision

Separation of policy and mechanism: Recurring theme in OS
OS runs \textit{dispatch loop}

\begin{verbatim}
while (1) {
    run process A for some \textit{time-slice}
    stop process A and save its \textit{context}
    load context of another process B
}
\end{verbatim}

Question 1: How does dispatcher gain control?
Question 2: What must be saved and restored?
HOW DOES DISPATCHER GET CONTROL?

Option 1: *Cooperative Multi-tasking:* Trust process to relinquish CPU through traps

- Examples: System call, page fault (access page not in main memory), or error (illegal instruction or divide by zero)
- Provide special `yield()` system call
PROBLEMS WITH COOPERATIVE?

Disadvantages: Processes can misbehave

By avoiding all traps and performing no I/O, can take over entire machine
Only solution: Reboot!

Not performed in modern operating systems
Option 2: Timer-based Multi-tasking

Guarantee OS can obtain control periodically

Enter OS by enabling periodic alarm clock
   Hardware generates timer interrupt (CPU or separate chip) Example: Every 10ms
   User must not be able to mask timer interrupt
<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>
</tbody>
</table>

**Notes**: The table above outlines the relationship between operating systems, hardware, and programs, with a specific mention of Process A in the program column.
<table>
<thead>
<tr>
<th>Operating System</th>
<th>Hardware</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process A</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
<table>
<thead>
<tr>
<th>Operating System</th>
<th>Hardware</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handle the trap</td>
<td>timer interrupt</td>
<td>Process A</td>
</tr>
<tr>
<td>Call switch() routine</td>
<td>save regs(A) to k-stack(A)</td>
<td></td>
</tr>
<tr>
<td>save regs(A) to proc-struct(A)</td>
<td>move to kernel mode</td>
<td></td>
</tr>
<tr>
<td>restore regs(B) from proc-struct(B)</td>
<td>jump to trap handler</td>
<td></td>
</tr>
<tr>
<td>switch to k-stack(B)</td>
<td>return-from-trap (into B)</td>
<td></td>
</tr>
</tbody>
</table>
Handle the trap
Call switch() routine
save regs(A) to proc-struct(A)
restore regs(B) from proc-struct(B)
switch to k-stack(B)
return-from-trap (into B)

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
<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>Process A</td>
</tr>
<tr>
<td>Call switch() routine</td>
<td>timer interrupt</td>
<td></td>
</tr>
<tr>
<td>save regs(A) to proc-struct(A)</td>
<td>save regs(A) to k-stack(A)</td>
<td></td>
</tr>
<tr>
<td>restore regs(B) from proc-struct(B)</td>
<td>move to kernel mode</td>
<td></td>
</tr>
<tr>
<td>switch to k-stack(B)</td>
<td>jump to trap handler</td>
<td></td>
</tr>
<tr>
<td>return-from-trap (into B)</td>
<td>restore regs(B) from k-stack(B)</td>
<td>Process B</td>
</tr>
<tr>
<td></td>
<td>move to user mode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>jump to B’s IP</td>
<td></td>
</tr>
</tbody>
</table>
SUMMARY

Process: Abstraction to virtualize CPU
Use time-sharing in OS to switch between processes

Key aspects
  Use system calls to run access devices etc. from user mode
  Context-switch using interrupts for multi-tasking
POLICY?
NEXT CLASS!
NEXT STEPS

Project 1a: Due Jan 29th (Tuesday) at 11.59pm
Project 1b: Out on Jan 29th

Discussion section: Thursday 5.30pm-6.30pm

Waitlist? Email enrollment@cs.wisc and cc me (will finalize by Monday)