CPU Virtualization
Virtualization: The CPU

Questions answered in this lecture:

What is a process? (Chapter 4-5)
Why is limited direct execution a good approach for virtualizing the CPU? (Chapter 6)
What execution state must be saved for a process? (Chapter 6)
What 3 modes could a process in? (Chapter 6)
What is a Process?

Process: An execution stream in the context of a process state

What is an execution stream?
- Stream of executing instructions
- Running piece of code
- “thread of control”

What is process state?
- Everything that the running code can affect or be affected by
- Registers
  - General purpose, floating point, status, program counter, stack pointer
- Address space
- Heap, stack, and code
- Open files
A process is different than a program
  • Program: Static code and static data
  • Process: Dynamic instance of code and data

Can have multiple process instances of same program
  • Example: many users can run “ls” at the same time
Recall: Process Memory Segments

• The OS allocates memory for each process - ie. a running program – for data and code

• This memory consists of different segments

• Stack - for local variables – incl. command line arguments and environment variables

• Heap - for dynamic memory

• Data segment for – global uninitialised variables (.bss) – global initialised variables (.data)

• Code segment typically read-only
Processes vs. Threads

• A process is different than a thread

• Thread: “Lightweight process” (LWP)
  • An execution stream that shares an address space
  • Multiple threads within a single process

• Example:
  • Two processes examining same memory address 0xffe84264
    see different values (I.e., different contents)
  • Two threads examining memory address 0xffe84264
    see same value (I.e., same contents)
**Goal:** Give each process the impression that it alone is actively using the CPU

Resources can be shared in **time** and **space**

Assume single uniprocessor

  **Time-sharing** (today’s multi-processors: more nuanced)

But while sharing, processes

  should not perform restricted operations

  should not run forever or make the entire system slow

One possibility: let the OS inspect each process instruction before running

  The problem? **Performance**
How to Provide Good CPU Performance?

Direct execution
- Allow user process to run directly on hardware
- OS creates process and transfers control to starting point (i.e., main())

Problems with direct execution?
1. Process could do something restricted
   - Could read/write other process data (disk or memory)
2. Process could run forever (slow, buggy, or malicious)
   - OS needs to be able to switch between processes
3. Process could do something slow (like I/O)
   - OS wants to use resources efficiently and switch CPU to other process

Solution: Limited direct execution:

OS and the hardware maintain some control
Problem 1: Restricted Ops

How can we ensure user process can’t unilaterally perform restricted operations?

Solution: privilege levels/separation provided by hardware (status bit on a register)
- OS runs in kernel mode (not restricted)
  - Instructions for interacting with devices enabled
  - Could have many privilege levels (advanced topic)
- User processes run in user mode (restricted mode)
  - Interacting with devices directly will trap (software interrupt)
  - Pre-set routines that run when privileged/restricted instructions run

How can a process legitimately access a device?
- System calls (function call implemented by OS)
- Change privilege level through system call (trap)
Legitimate use: System Call

syscall(SYS_call, arg1, arg2, ...);

```c
#include <syscall.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>

int main(void) {

    long ID1, ID2;
    /* -------------------------------*/
    /* direct system call */
    /* SYS getpid (func no. is 20) */
    /* -------------------------------*/
    ID1 = syscall(SYS_getpid);
    printf("syscall(SYS_getpid)=%ld\n", ID1);

    /* -------------------------------*/
    /* "libc" wrapped system call */
    /* SYS getpid (Func No. is 20) */
    /* -------------------------------*/
    ID2 = getpid();
    printf("getpid()=%ld\n", ID2);

    return(0);
}
```
System Call

P wants to call read()
P can only see its own memory because of **user mode** (other areas, including kernel, are hidden)
Process P wants to call read() but no way to call it directly.

List of Linux System Calls:
http://www.cheat-sheets.org/saved-copy/Linux_Syscall_quickref.pdf
System Call

read():

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

Assembly convention: `movl %eax, …`
- CPU uses contents of EAX register as source operand
movl $6, %eax;  int $64

syscall-table index

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

movl $6, %eax; int $64

syscall-table index  trap-table index

Follow entries to correct system call code
Kernel can access user memory to fill in user buffer

return-from-trap at end to return to Process P
System Call

Syscall() {
    sysnum = %eax
    sys_handle = get_fn_table(sysnum)
    sys_handle();
}

movl $6, %eax; int $64

System Call

H/W-level Trap Table

<table>
<thead>
<tr>
<th>Num</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>illegal access</td>
</tr>
<tr>
<td>64</td>
<td>system call</td>
</tr>
<tr>
<td>65</td>
<td>Device Interrupt</td>
</tr>
</tbody>
</table>

Syscall table

<table>
<thead>
<tr>
<th>Num</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>sys_read</td>
</tr>
<tr>
<td>7</td>
<td>sys_write</td>
</tr>
</tbody>
</table>
User processes are not allowed to directly perform:
• Arbitrary memory access
• Disk I/O
• Special x86 instructions like \texttt{lidt}

What if a process tries to do something privileged/restricted on its own?

Typical response: trap (hardware); OS kills process
Problem 2: How to take the CPU away?

OS requirements for **multiprogramming** (or multitasking)
- **Mechanism**: To switch between processes
- **Policy**: To decide which process to run at what time

Separation of policy and mechanism
- Recurring theme in OS design
- **Policy**: Decision-maker to optimize some workload performance metric
  - Which process to run when?
  - Process **Scheduler**: next lecture
- **Mechanism**: Low-level code that implements the decision
  - "How"?
  - Process **Dispatcher**: Today’s lecture
Dispatch Mechanism

OS runs `dispatch loop`

```c
while (1) {
    run process A for some time-slice
    stop process A and save its context
    load context of another process B
}
```

Question 1: How does dispatcher regain control after the time slice?
Question 2: What execution context must be saved and restored?
Q1: How does Dispatcher regain control?

Option 1: **Cooperative Multi-tasking**
- Trust process to relinquish CPU to OS 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
Cooperative Approach

yield() call

OS
yield() return
yield() return
P2

yield() call
Q1: How Does Dispatcher regain control?

- Problem with cooperative approach?
- 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
Q1: How does Dispatcher regain control?

Option 2: **Regain control without cooperation**

- Guarantee OS can obtain control periodically. How?
- 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 (privileged operation)
- Dispatcher counts interrupts between context switches
  - Example: Waiting 20 timer ticks gives 200 ms time slice
  - Common time slices range from 10 ms to 200 ms
  - Research systems today: ~5 microseconds

Use hardware mechanisms (timer, traps) to regain control
Q2: What Context must be Saved?

“Now where was I...”

Context save and restore.
Q2: What Context must be Saved?

Dispatcher must save the context of the process when it’s not running

- Save it in **process control block (PCB)** (or process descriptor)
- PCB is a structure maintained for each process in the OS

What information is stored in PCB?

- PID
- Process state (i.e., running, ready, or blocked)
- **Execution state (all registers, PC, stack pointer) -- Context**
- Scheduling priority
- Accounting information (parent and child processes)
- Credentials (which resources can be accessed, owner)
- Pointers to other allocated resources (e.g., open files)

Requires special hardware support. Why?

- Hardware saves process PC and PSR on interrupts
Q3: What’s inside a PCB?

// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem;          // Start of process memory
    uint sz;            // Size of process memory
    char *kstack;       // Bottom of kernel stack
                       // for this process
    enum proc_state state;  // Process state
    int pid;            // Process ID
    struct proc *parent;  // Parent process
    int killed;         // If non-zero, have been killed
    struct file *ofile[NOFILE];  // Open files
    struct inode *cwd;    // Current directory
    struct context context;  // Switch here to run process
    struct trapframe *tf;  // Trap frame for the
                            // current interrupt
};

Conceptually:
Separate kernel thread of execution per process
<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>
<tr>
<td>timer interrupt</td>
<td>save regs(A) to k-stack(A)</td>
<td>Process A</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>
</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)

Must have been saved the last time OS switched B out
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)

Hardware

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

Program

Process A

…

restore regs(B) from k-stack(B)
move to user mode
jump to B's IP
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
Q4: What Context must be Saved?

// the registers will save and restore
// to stop and subsequently restart a process
struct context {
    int eip; // Index pointer register
    int esp; // Stack pointer register
    int ebx; // Called the base register
    int ecx; // Called the counter register
    int edx; // Called the data register
    int esi; // Source index register
    int edi; // Destination index register
    int ebp; // Stack base pointer register
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };
Problem 3: Slow Ops such as I/O?

When running process performs op that does not use CPU, OS switches to process that needs CPU (policy issues)

OS must track state of each process:

- **Running**:
  - On the CPU (only one on a uniprocessor)
- **Ready**:
  - Waiting for the CPU
- **Blocked**
  - Asleep: Waiting for I/O or synchronization to complete
OS must track every process in system
  • Each process identified by unique Process ID (PID)

OS maintains queues of all processes
  • Ready queue: Contains all ready processes
  • Event queue: One logical queue per event
    • e.g., disk I/O and locks
    • Contains all processes waiting for that event to complete

Next Lecture: Policy for determining which ready process to run
Virtualization: Context switching gives each process impression it has its own CPU

Direct execution makes processes fast

Limited execution at key points ensures OS retains control

Hardware is crucial for limited direct execution

- Privilege separation: user vs kernel mode
- Timer interrupts
- Automatic register saves and restores