VIRTUALIZATION: 
The CPU

Questions answered in this lecture:
What is a process? Why is limited direct execution a good approach for virtualizing the CPU? What execution state must be saved for a process? What 3 modes could a process be in?

Announcements:
Waiting list:
Seats available in Section 303 (Wed 3:30-4:20)
Sign-up sheet at front of lecture to show continued interest in enrolling
Read chapters 1 - 6
Begin Project 1 (part 1 - sorting)
Watch P1 video
Attend discussion section tomorrow for sorting help

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

PROCESSES VS. PROGRAMS

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
• Can have multiple processes of the same program
  Example: many users can run “ls” at the same time

PROCESS CREATION

CPU

Memory
**PROCESS CREATION**

- CPU
- Memory
- Code
- Static data
- Heap
- Stack

**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)

**Virtualizing the CPU**

- Goal: Give each process impression it alone is actively using CPU
- Resources can be shared in time and space
- Assume single uniprocessor
  - Time-sharing (multi-processors: advanced issue)
- Memory?
  - Space-sharing (later)
- Disk?
  - Space-sharing (later)

**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 hardware maintain some control
**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)
- Instructions for interacting with devices
- Could have many privilege levels (advanced topic)

How can process access device?
- System calls (function call implemented by OS)
- Change privilege level through system call (trap)

**System Call**

Process P

RAM

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
Kernel mode: we can do anything!

Follow entries to correct system call code
**SYSTEM CALL**

User processes are not allowed to perform:
- General memory access
- Disk I/O
- Special x86 instructions like `lidt`

What if process tries to do something restricted?

**WHAT TO LIMIT?**

**PROBLEM 2: HOW TO TAKE CPU AWAY?**

OS requirements for *multiprocessing* (or multitasking)
- Mechanism
  - To switch between processes
- Policy
  - To decide which process to schedule when

Separation of policy and mechanism
- Reoccurring theme in OS
- Policy: Decision-maker to optimize some workload performance metric
  - Which process when?
- Process Scheduler: Future lecture
- Mechanism: Low-level code that implements the decision
  - How?
- Process Dispatcher: Today's lecture

**DISPATCH MECHANISM**

OS runs *dispatch loop*

```
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 gain control?
Question 2: What execution context must be saved and restored?
Q1: HOW DOES DISPATCHER GET 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

Q1: How Does Dispatcher Run?
- 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

Option 2: True 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
- 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
Dispatcher must track context of process when not running
- Save context in process control block (PCB) (or, process descriptor)

What information is stored in PCB?
- PID
- Process state (i.e., running, ready, or blocked)
- Execution state (all registers, PC, stack ptr)
- 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
- Hardware saves process PC and PSR on interrupts
**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 mode 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 Topic: 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 to ensure OS retains control
Hardware provides a lot of OS support
  • user vs kernel mode
  • timer interrupts
  • automatic register saving

Virtualization: The CPU
Questions answered in this lecture:
  What is a process?
  Why is limited direct execution a good approach for virtualizing the CPU?
  What execution state must be saved for a process?
  What 3 modes could a process be in?

Announcements:
  Waiting list:
  Seats available in Section 303 (Wed 3:30-4:20)
  Sign-up sheet at front of lecture to show continued interest in enrolling
  Read chapters 1-6
  Begin Project 1 (post a sorting)
  Watch P1 video
  Attend discussion section tomorrow for sorting help

Process Creation
Two ways to create a process
  • Build a new empty process from scratch
  • Copy an existing process and change it appropriately

Option 1: New process from scratch
  • Steps
    • Load specified code and data into memory;
    • Create empty call stack
    • Create and initialize PCB (make look like context-switch)
    • Put process on ready list
  • Advantages: No wasted work
  • Disadvantages: Difficult to setup process correctly and to express all possible options
    • Process permissions, where to write I/O, environment variables
    • Example: WindowsNT has call with 10 arguments

Option 2: Clone existing process and change
  • Example: Unix fork() and exec()
    • Fork(): Clones calling process
    • Exec(char *file): Overlays file image on calling process
    • Fork()
      • Stop current process and save its state
      • Make copy of code, data, stack, and PCB
      • Add new PCB to ready list
      • Any changes needed to child process?
    • Exec(char *file)
      • Replace current data and code segments with those in specified file
      • Advantages: Flexible, clean, simple
      • Disadvantages: Wasteful to perform copy and then overwrite of memory
How are Unix shells implemented?

While (1) {
    Char *cmd = getcmd();
    Int retval = fork();
    If (retval == 0) {
        // This is the child process
        // Setup the child’s process environment here
        // E.g., where is standard I/O, how to handle signals?
        exec(cmd);
        // exec does not return if it succeeds
        printf("ERROR: Could not execute %s\n", cmd);
        exit(1);
    } else {
        // This is the parent process; Wait for child to finish
        int pid = retval;
        wait(pid);
    }
}