UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

Processes

Questions answered in this lecture:

- What is a process?
- How does the dispatcher context-switch between processes?
- How does the OS create a new process?

What is a Process?

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

Execution stream

1) Stream of executing instructions
2) Running piece of code
3) Sequential sequence of instructions
4) "thread of control"

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
  - Everything process can address through memory
  - Represented by array of bytes
  - Heap, stack, and code

Processes vs. Programs

A process is different than a program

- Program: Static code and static data
- Process: Dynamic instance of code and data

No one-to-one mapping between programs and processes

- Can have multiple processes of the same program
  Example: many users can run "ls" at the same time
- One program can invoke multiple processes
  Example: "make" runs many processes to accomplish its work

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 memory address 0xffe84264 see different values (I.e., different contents)
- Two threads examining memory address 0xffe84264 see same value (I.e., same contents)

System Classifications

All systems support processes, but the number varies

Uniprogramming: Only one process resident at a time

- Examples: First systems and DOS for PCs
- Advantages: Runs on all hardware
- Disadvantages: Not convenient for user and poor performance

Multiprogramming: Multiple processes resident at a time (or, multitasking)

- Note: Different than multiprocessing
  - Multiprocessing: Systems with multiple processors
- Examples: Unix variants, WindowsNT
- Advantages: Better user convenience and system performance
- Disadvantages: Complexity in OS

Multiprogramming

OS requirements for multiprogramming

- Mechanism
  - To switch between processes
  - To protect processes from one another
- Policy
  - To decide which process to schedule

Separation of policy and mechanism

- Reoccuring 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
}
```

Context-switch

Question 1: How does dispatcher gain control?

Question 2: What execution context must be saved and restored?

Q1: Hardware for Multiprogramming

Must differentiate application process and OS

Hardware support

- Bit in status word designates whether currently running in user or system mode
- System mode (or privileged or supervisor) allows functionality
  - Execution of special instructions (e.g., access hardware devices)
  - Access to all of memory
  - Change stack pointer
- Usage
  - Applications run in user mode
  - OS runs in system mode

Q1: Entering system mode

How does OS get control?

1) Synchronous interrupts, or traps
   - Event internal to a process that gives control to OS
   - Examples: System calls, page faults (access page not in main memory), or errors (illegal instruction or divide by zero)
2) Asynchronous interrupts
   - Events external to a process, generated by hardware
   - Examples: Characters typed, or completion of a disk transfer

How are interrupts handled?

- Each type of interrupt has corresponding routine (handler or interrupt service routine (ISR)
- Hardware saves current process and passes control to ISR

Q1: How does Dispatcher run?

Option 1: Cooperative Multi-tasking

- Trust process to relinquish CPU through traps
  - Trap: Event internal to process that gives control to OS
  - Examples: System call, page fault (access page not in main memory), or error (illegal instruction or divide by zero)
- 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 run?

Option 2: True Multi-tasking

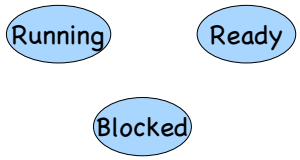
- Guarantee OS can obtain control periodically
- Enter OS by enabling periodic alarm clock
  - Hardware generates timer interrupt (CPU or separate chip)
  - Example: Every 10ms
- User must not be able to mask timer interrupt
- Dispatcher counts interrupts between context switches
  - Example: Waiting 20 timer ticks gives 200 ms time slice
  - Common time slices range from 10 ms to 200 ms

Q2: Process States

Each process is in one of three modes:

- Running: On the CPU (only one on a uniprocessor)
- Ready: Waiting for the CPU
- Blocked (or asleep): Waiting for I/O or synchronization to complete

Transitions?



Q2: Tracking Processes

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

Q2: What Context must be Saved?

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)

Q2: Context-Switch Implementation

Basic idea: Save all registers to PCB in memory

- Tricky: Must execute code without using registers
- Machine-dependent code (written in assembly)
  - Different for x86, SPARC, MIPS, etc.

Requires special hardware support

- Hardware saves process PC and PSR on interrupts
- CISC: Single instruction saves all registers onto stack
- RISC: Agreement to keep some registers empty
  - Alpha and SPARC: Shadow registers
  - MIPS: Two general-purpose registers are scratch, avoided by compiler

How long does a context-switch take?

- 10s of microseconds

Multiprogramming and Memory

Does OS need to save all of memory between processes?

Option 1: Save all of memory to disk

- Example: Alto, early personal workstation
- Disadvantage: Very time consuming
- How long to save a 10 MB process to disk?

Option 2: Trust next process to not overwrite memory

- Example: Early multiprogramming in PCs and Macs (single user)
- Disadvantage: Very hard to track down bugs

Option 3: Protect memory (and files) from next process

- Requires hardware support
- Investigate later in course

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

Process Creation

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 PCB?
- 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

Unix Process Creation

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);
   }
}
```