UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Intro to Operating Systems

A. Arpaci-Dusseau
Spring 2000

Lecture 2: Processes

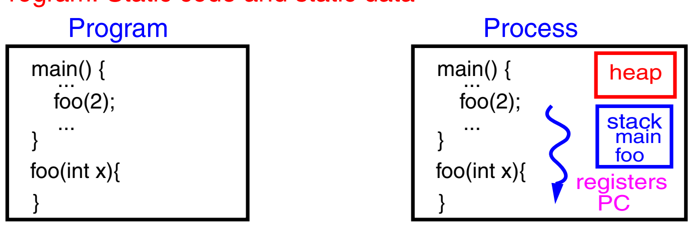
- Questions answered in this lecture
  - What are processes and why are they needed?
  - How does the dispatcher run a process?
  - How does the system start a new process?
- Reading
  - Chapter 4 (Skip 4.6)

What is a Process?

- Process: An execution stream in the context of a process state
- Execution stream
  - Running piece of code
  - Sequential sequence of instruction
  - “thread of control”
- Process state
  - Everything that the running code can affect or be affected by
  - What are all the things that the code can affect or be affected by?

Programs and Processes

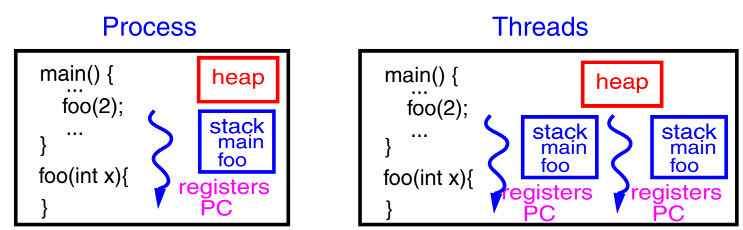
- A process is different than a program
- Program: Static code and static data



  - Can have multiple processes from the same program
    e.g., many users can run “ls” at same time
  - One program can invoke multiple processes
    e.g., “make” runs many processes to accomplish its work
- No one-to-one mapping between programs and processes

Threads and Processes

- A process is different than a thread
- Thread: Separate execution streams that share address space
  - “Lightweight process”



- Can have multiple threads within a process

Why use Processes?

- General principle of divide-and-conquer
  - Decomposition: Divide large, complex problem into simple tasks
  - Easier to think about smaller, well-contained units
- Many operations occur concurrently within system
  - Examples: Multiple users and I/O devices, interrupts
  - OS must coordinate this concurrent activity
- Easier to reason about processes
  - Sequential activity with well-defined interactions

System Classifications

- Uniprogramming: Only one process at a time
  - Examples: Original sytems and older operating systems for PCs
  - Advantages: Easier for OS designer
  - Disadvantages: Not convenient for user and poor performance
- Multiprogramming: Multiple processes at a time
  - Examples: UNIX, OS/2, WindowsNT
  - Note: Multiprogramming is different than multiprocessing

    Multiprocessing: Systems with multiple processors
  - Advantages: Better system performance and user convenience
  - Disadvantages: Complexity in OS

Multiprogramming

- OS requirements for multiprogramming
  - Policy to determine which process to schedule
  - Mechanism to switch between processes
  - Methods to protect processes from one another (memory system)
- Separation of policy and mechanism
  - Reoccuring theme in OS
  - Policy: Decision-maker with some workload performance metric

    Scheduler: Future lecture
  - Mechanism: Low-level code that implements the decision

    Dispatcher: Today's lecture

Dispatch Mechanism

- OS keeps system-wide list of processes
- Each process in one of three modes
  - **Running:** On the CPU (only one in uniprocessor system)
  - **Ready:** Waiting for the CPU
  - **Blocked:** Waiting for I/O or synchronization with another thread
- Dispatch loop:

```
while (1) {
    run a process for a while
    stop process and save its state
    load state of another process      context-switch
}
```

- Question 1: How does dispatcher gain control?
- Question 2: What execution state must be saved and restored?

Q1: How does Dispatcher gain Control?

- Must change from user mode to system mode
  - Problem: Only one CPU, and CPU can only do one thing at a time
  - User process running means that dispatcher isn't!
- Two ways operating system gains control
- 1. Traps: Events internal to user process
  - Examples: System calls, Errors (illegal instructions, divide by zero), Page faults
- 2. Hardware interrupts: Events external to user process
  - Examples: Character typed at terminal, Completion of disk transfer
  - Control given to OS **interrupt service routine** (ISR)
  - Review Chapter 2

Approaches for Dispatcher

- Option 1: Cooperative Multi-tasking
  - Trust process to wake dispatcher
  - System call to relinquish processor
  - Disadvantage: Applications can misbehave

    By avoiding all traps and performing no I/O, can take over entire machine

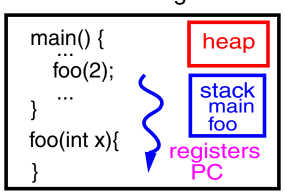
    Solution: Reboot machine!
- Option 2: True Multi-tasking
  - Enter kernel by enabling periodic alarm clock
  - Hardware generates **timer interrupt** (CPU or separate chip)

    Example: Every 10ms in SPARC/Solaris
  - Control returned to dispatcher
  - Dispatcher counts interrupts (2 - 20) between context-switch

    --> Time-slices for processes vary between 20ms and 200ms
  - **Note:** User must not be able to **mask** timer interrupt!

Q2: What State must be Saved?

- Dispatcher must track state of process when not running
  - On every trap or interrupt,
    save process state in Process Control Block (PCB)
- Data structure problem: How to manage all PCBs?



- Information stored in PCB
  - execution state (registers, status word, program counter, stack ptr)
  - i/o status (open files)
  - scheduling information (ready state, priority)
  - accounting information (owner, pid)

Multiprogramming and Memory

- Do we need to save everything in system that could be modified?
  - All of memory? All of disk?
- Option 1: Save all of memory to disk
  - Example: Alto, early personal workstation
  - Disadvantage: Very time-consuming

    How long to save a 10MB process to disk?
- Option 2: Trust next process to not overwrite memory
  - Early multiprogramming in PCs and Macs
  - Same address-space: threads or lightweight-processes
- Option 3: Protect memory (and files) from next process
  - Investigate later in course

Context-Switch Implementation

- Machine-dependent code (written in assembly)
  - Different for MIPS, SPARC, x86, etc.
  - Save all registers to PCB in memory
- Tricky: OS must execute code to save state w/o changing state
- Requires special hardware support
  - Save process PC and PSR on trap and interrupt
  - CISC: Single instruction saves all registers onto stack
  - RISC: Keep some empty registers
    - Alpha and SPARC: Shadow registers
    - MIPS: Two general-purpose registers are scratch; Compiler avoids
- How long does this take?
  - Optimizations so don't save all on interrupts

Process Creation

- Two ways to create a process
  - Build a new one from scratch
  - Clone an existing one
- Option 1: From scratch
  - 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
- Option 2: Cloning (e.g., Unix fork() syscall)
  - Stop current process and save its state
  - Make copy of code, data, stack, and PCB
  - Add new process PCB to ready list
  - Anything else???

Unix Process Creation

- Unix: Combination of fork() and exec()
  - **Fork():** Clone calling process
  - **Exec():** Overlay new image on calling process

Shell example

```
cmd = readcmd();
pid = fork();
if (pid == 0) {
        // Child process -- Setup environment here
        // e.g., standard i/o, signals
        exec(cmd);
        // exec doesn't return
} else {
        // Parent process -- Wait for child to finish
        wait(pid);
}
```

- Advantage: Flexible, clean, simple

Summary

- Processes simplify design of concurrent system
  - Sequential execution stream
  - process state (registers, program counter, address space, etc)
- OS dispatcher: Mechanism
  - Job scheduling: Policy
  - Dispatcher executes a **ready** process
  - Periodic timer interrupts give control to OS
  - Save execution state of process in PBC at context-switch
- Process creation: Fork() and exec()
- Next: Synchronization of processes