Concurrency: Threads, Address Spaces, and Processes

Sarah Diesburg

Operating Systems

CS 3430

Why ***Concurrency***?

- Allows multiple applications to run at the same time
  - Analogy: juggling

Benefits of Concurrency

Benefits of Concurrency

- Ability to run multiple applications at the same time
- Better resource utilization
  - Resources unused by one application can be used by the others
- Better average response time
  - No need to wait for other applications to complete

Benefits of Concurrency

- Better performance
  - One application uses only the processor
  - One application uses only the disk drive
  - Completion time is shorter when running both concurrently than consecutively

Drawbacks of Concurrency

Drawbacks of Concurrency

- Applications need to be protected from one another
- Additional coordination mechanisms among applications
- Overhead to switch among applications
- Potential performance degradation when running too many applications

Thread

- A sequential execution stream
  - The smallest CPU scheduling unit
  - Can be programmed as if it owns the entire CPU
    - Implication: an infinite loop within a thread won't halt the system
  - Illusion of multiple CPUs on a single machine

Thread States

- Program counter
- Register values
- Execution stacks

Thread Benefits

- Simplified programming model per thread
- Example: Microsoft Word
  - One thread for grammar check; one thread for spelling check; one thread for formatting; and so on…
  - Can be programmed independently
  - Simplifies the development of large applications

Address Space

- Contains all states necessary to run a program
  - Code, data, stack
  - Program counter
  - Register values
  - Resources required by the program
  - Status of the running program

Process

- An address space + at least one thread of execution
  - Address space offers protection among processes
  - Threads offer concurrency
- A fundamental unit of computation

Process =? Program

- ***Program***: a collection of statements in C or any programming languages
- Process: a running instance of the program, with additional states and system resources

Process >? Program

- Two processes can run the same program
  - The code segment of two processes are the same program

Program >? Process

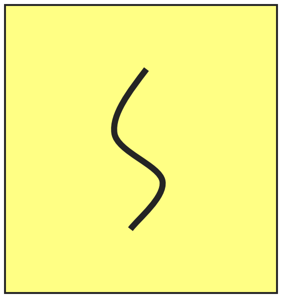
- A program can create multiple processes
  - Example: compilers (like gcc), web browsers

Analogy

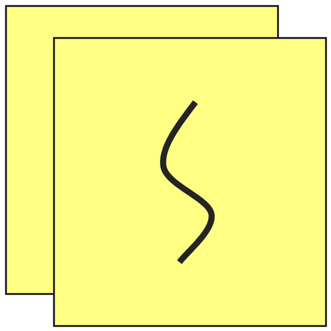
- Program: a recipe
- Process: everything needed to cook
  - e.g., kitchen
- Two chefs can cook the same recipe in different kitchens
- One complex recipe can involve several chefs

Some Definitions

- ***Uniprogramming***: running one process at a time

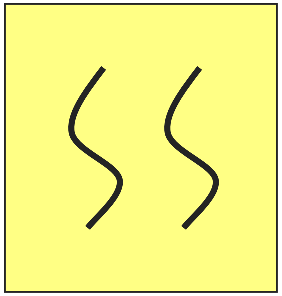


- ***Multiprogramming***: running multiple processes on a machine



Some Definitions

- ***Multithreading***: having multiple threads per address space



- ***Multiprocessing***: running programs on a machine with multiple processors
- ***Multitasking***: a single user can run multiple processes

Classifications of OSes

	Single address space	Multiple address spaces
Single thread	MS DOS, Macintosh	Traditional UNIX
Multiple threads	Embedded systems	Windows 8, Linux, OSX

Threads & Dispatching Loop

- A thread owns a ***thread control block***
  - Execution states of the thread
  - The status of the thread
    - Running or sleeping
  - Scheduling information of the thread
    - e.g., priority

Dispatching Loop

- Threads are run from a ***dispatching loop***
  - LOOP
    - Run thread ← Jump to the first instruction
    - Save states
    - Choose a new thread to run ← ***Scheduling***
    - Load states from a different thread

Context switch

Simple? Not quite…

- How does the dispatcher regain control after a thread starts running?
- What states should a thread save?
- How does the dispatcher choose the next thread?

How does the dispatcher regain control?

- Two ways:
  1. Internal events (“Sleeping Beauty”)
     - A thread is waiting for I/O
     - A thread is waiting for some other thread
     - Yield—a thread gives up CPU voluntarily
  2. External events
     - Interrupts—a complete disk request
     - Timer—it’s like an alarm clock

What states should a thread save?

- Anything that the next thread may trash before a context switch
  - Program counter
  - Registers
  - Changes in execution stack

How does the dispatcher choose the next thread?

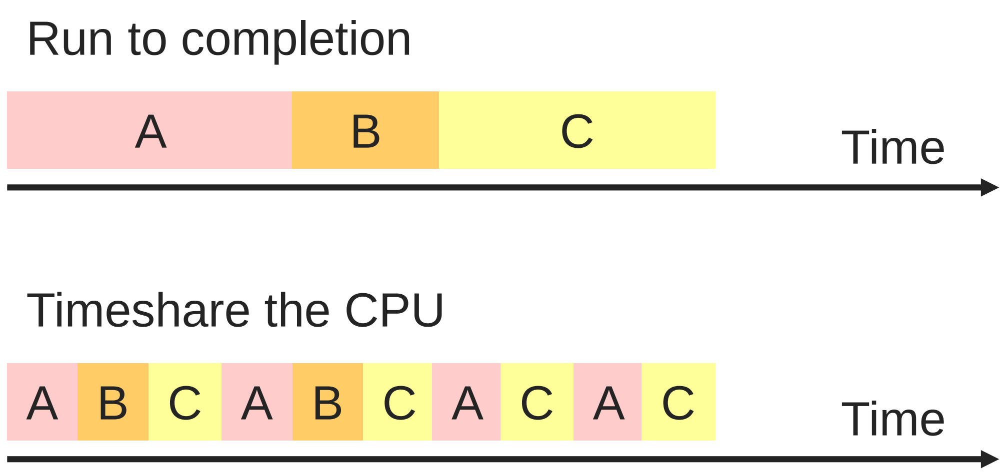
- The dispatcher keeps a list of threads that are ready to run
- If no threads are ready
  - Dispatcher just loops
- If one thread is ready
  - Easy

How does the dispatcher choose the next thread?

- If more than one thread are ready
  - We choose the next thread based on the scheduling policies
  - Examples
    - FIFO (first in, first out)
    - LIFO (last in, first out)
    - Priority-based policies

How does the dispatcher choose the next thread?

- Additional control by the dispatcher on how to share the CPU
  - Examples:



Per-thread States

- Each thread can be in one of the three states
  1. ***Running***: has the CPU
  2. ***Blocked***: waiting for I/O or another thread
  3. ***Ready to run***: on the ready list, waiting for the CPU

Per-thread State Diagram