Lecture 38:

Parallel Programming Systems and Models & Processes and Threads

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
June 2nd, 2010

Common Parallel Programming Models

- Message-Passing
- Shared Address Space

Common Parallel Programming Models

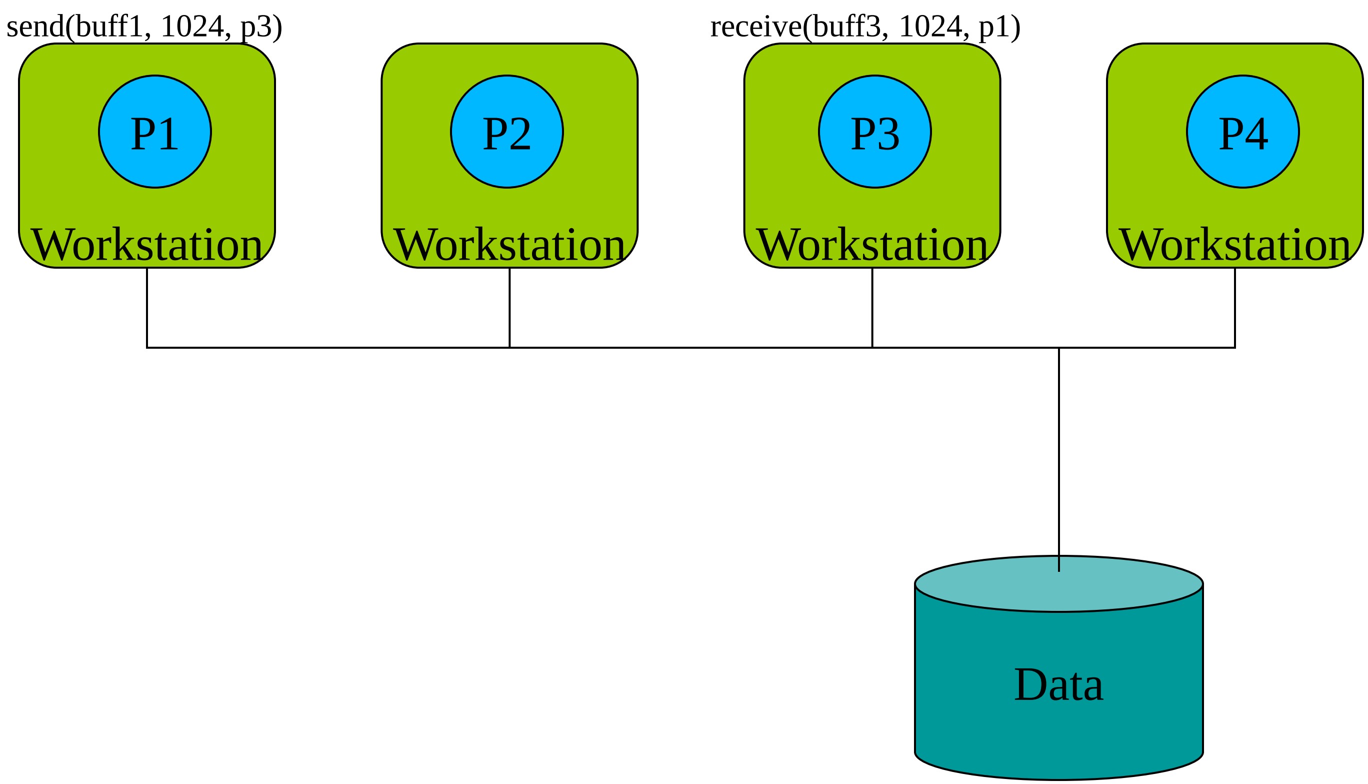
- Message-Passing
  - Most widely used for programming parallel computers (clusters of workstations)
  - Key attributes:
    - Partitioned address space
    - Explicit parallelization
  - Process interactions
    - Send and receive data

Common Parallel Programming Models

- Message-Passing
  - Communications
    - Sending and receiving messages
    - Primitives
      - send(buff, size, destination)
      - receive(buff, size, source)
      - Blocking vs non-blocking
      - Buffered vs non-buffered
    - Message Passing Interface (MPI)
      - Popular message passing library
      - ~125 functions

Common Parallel Programming Models

- Message-Passing



Common Parallel Programming Models

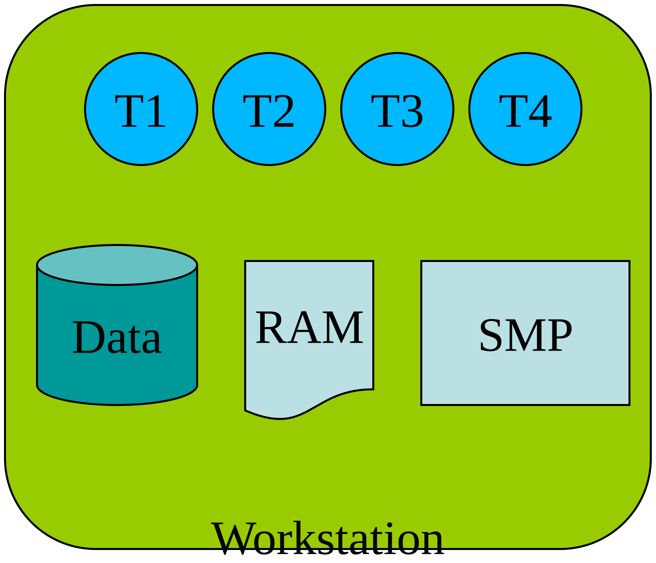
- Shared Address Space
  - Mostly used for programming SMP machines (multicore chips)
  - Key attributes
    - Shared address space
      - Threads
      - Shmget/shmat UNIX operations
    - Implicit parallelization
  - Process/Thread communication
    - Memory reads/stores

Common Parallel Programming Models

- Shared Address Space
  - Communication
    - Read/write memory
      - EX: x++;
  - Posix Thread API
    - Popular thread API
    - Operations
      - Creation/deletion of threads
      - Synchronization (mutexes, semaphores)
      - Thread management

Common Parallel Programming Models

- Shared Address Space



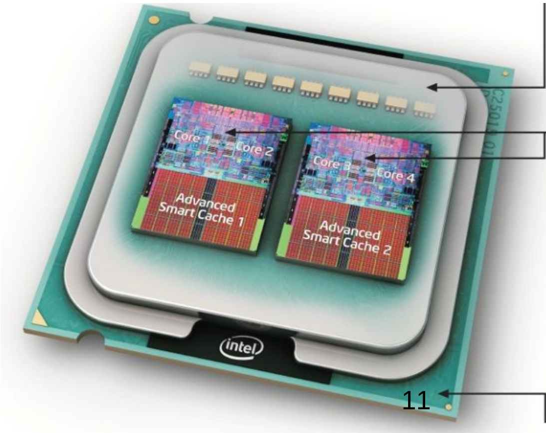
Parallel Programming Pitfalls

- Synchronization
  - Deadlock
  - Livelock
  - Fairness
- Efficiency
  - Maximize parallelism
- Reliability
  - Correctness
  - Debugging

Processes and Threads

Review: Multicore everywhere!

- Multicore processors are taking over, *many*core is coming
- The processor is the "new transistor"
- This is a "sea change" for HW designers and especially for programmers

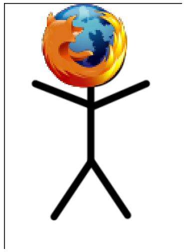


Outline for Today

- Motivation and definitions
- Processes
- Threads
- Synchronization constructs
- Speedup issues
  - Overhead
  - Caches
  - Amdahl's Law

How can we harness (many | multi)cores?

- Is it good enough to just have multiple programs running simultaneously?
- We want per-program performance gains!



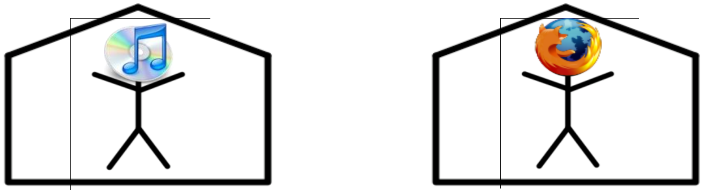
Crysis, Crytek 2007

Multiprogramming/Timesharing Systems

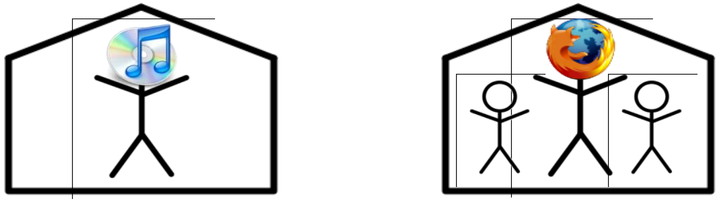
- Goal: to provide interleaved execution of several processes to give an illusion of many simultaneously executing processes.
- Computer can be a single-processor or multi-processor machine.
- The OS must keep track of the state for each active process and make sure that the correct information is properly installed when a process is given control of the CPU.
- Many resource allocation issues to consider:
  - How to give each process a chance to run?
  - How is main memory allocated to processes?
  - How are I/O devices scheduled among processes?

Definitions: threads v.s. processes

- A *process* is a “program” with its own address space.
  - A process has at least one thread!



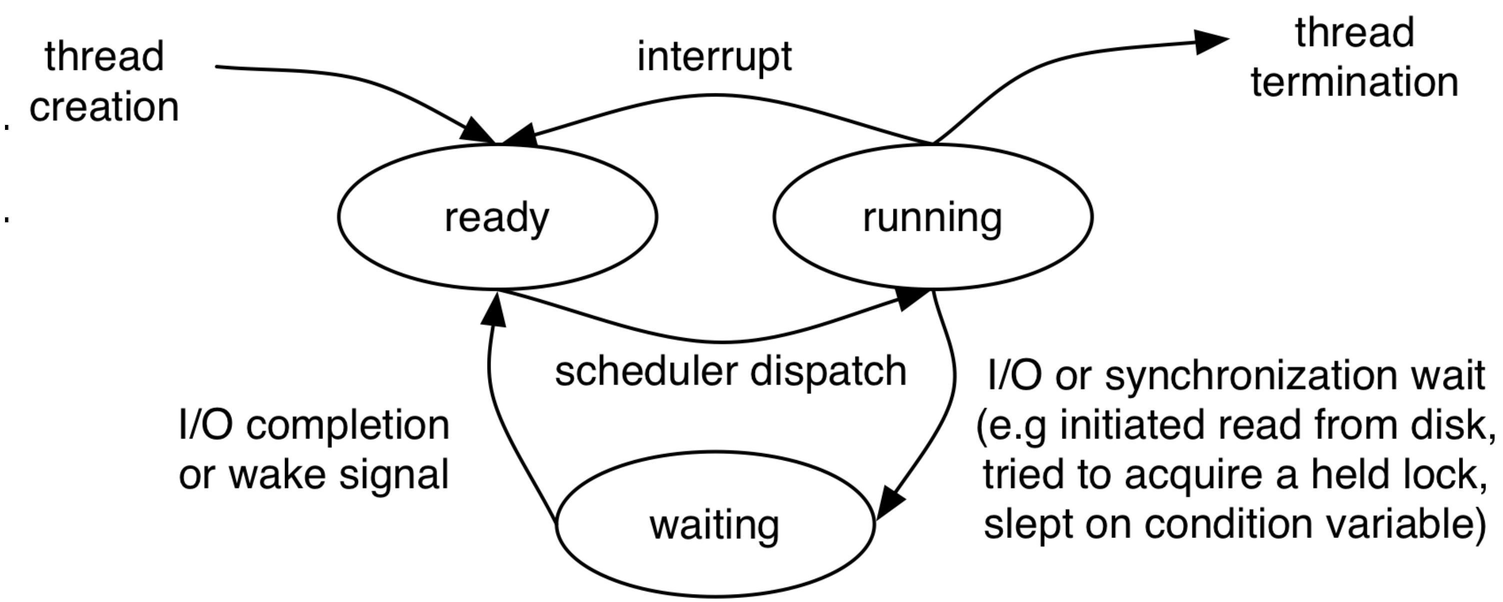
- A *thread of execution* is an independent sequential computational task with its own control flow, stack, registers, etc.
  - There can be many threads in the same process sharing the same address space



  - There are several APIs for threads in several languages. We will cover the PThread API in C.

How are threads *scheduled?*

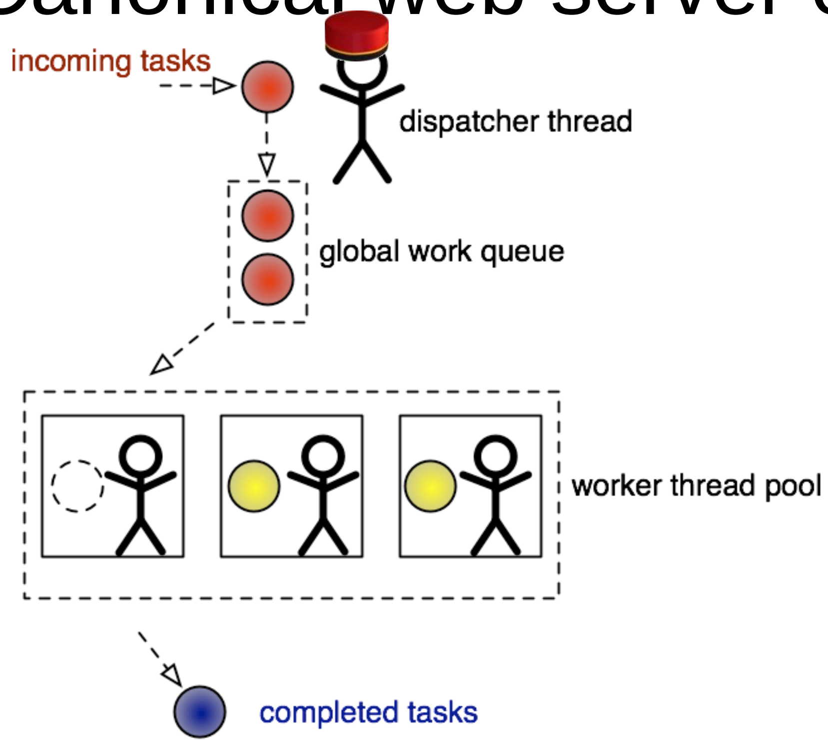
- Threads/processes are run sequentially on one core or simultaneously on multiple cores
  - The operating system schedules threads and



Based on diagram from Silberschatz, Galvin, and Gagne

Side: threading without multicore?

- Is threading useful without multicore?
  – Yes, because of I/O blocking!
- Canonical web server example:




```
global workQueue;

dispatcher() {
  createThreadPool();
  while(true) {
    task = receiveTask();
    if (task != NULL) {
      workQueue.add(task);
      workQueue.wake();
    }
  }
}

worker() {
  while(true) {
    task = workQueue.get();
    doWorkWithIO(task);
  }
}
```

Outline for Today

- Motivation and definitions
- Processes
- Threads
- Synchronization constructs
- Speedup issues
  - Overhead
  - Caches
  - Amdahl’s Law

Creating processes in UNIX

- To see how processes can be used in application and how they are implemented, we study how processes are created and manipulated in UNIX.
- Important source of information on UNIX is "man."
- UNIX supports multiprogramming, so there will be many processes in existence at any given time.
  - Processes are created in UNIX with the fork() system call.
  - When a process P creates a process Q, Q is called the child of P and P is called the parent of Q.

Process Hierarchies

- Parent creates a child process, child processes can create its own process
- Forms a hierarchy
  - UNIX calls this a *process group*
- Signals can be sent all processes of a group
- Windows has no concept of process hierarchy
  - all processes are created equal

Initialization

At the root of the family tree of processes in a UNIX system is the special process init:

- created as part of the bootstrapping procedure
- process-id = 1
- among other things, init spawns a child to listen to each terminal, so that a user may log on.
- do "man init” to learn more about it

UNIX Process Control

UNIX provides a number of system calls for process control including:

- fork - used to create a new process
- exec - to change the program a process is executing
- exit - used by a process to terminate itself normally
- abort - used by a process to terminate itself abnormally
- kill - used by one process to kill or signal another
- wait - to wait for termination of a child process
- sleep - suspend execution for a specified time interval
- getpid - get process id
- getppid - get parent process id

Questions

?