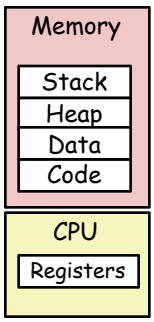
Processes & Threads

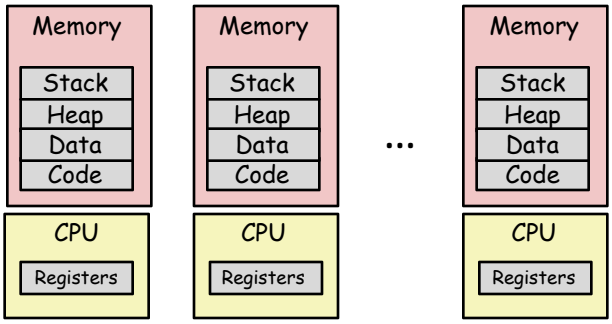
CS 475

Processes

- Definition: A *process* is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as "program" or "processor"
- Process provides each program with two key abstractions:
  - ***Logical control flow***
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called *context switching*
  - ***Private address space***
    - Each program seems to have exclusive use of main memory.
    - Provided by kernel mechanism called *virtual memory*

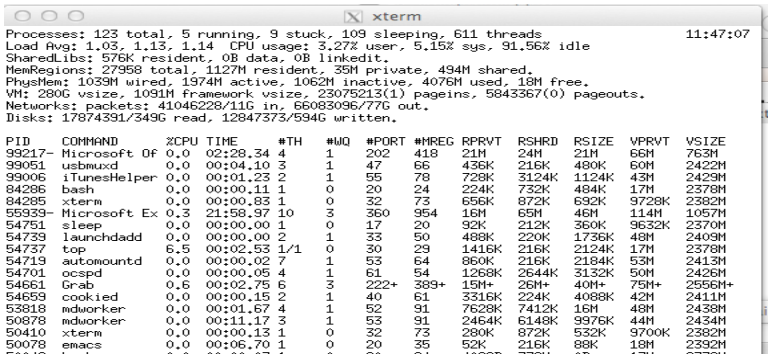


Multiprocessing: The Illusion



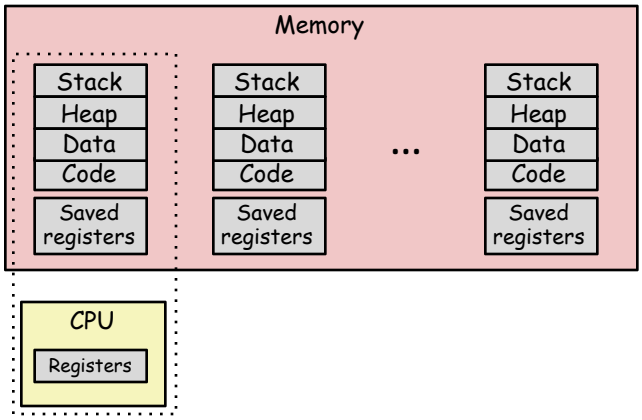
- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Multiprocessing Example



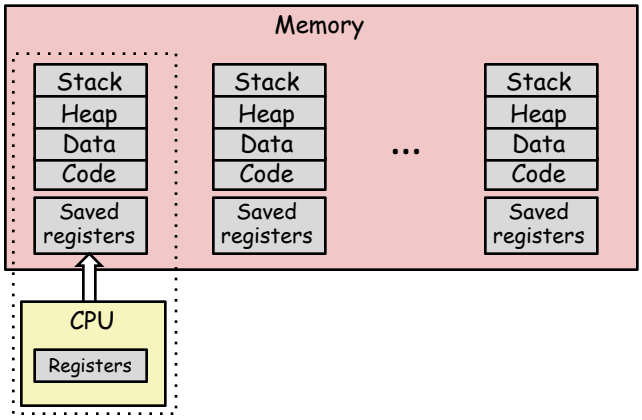
- Running program "top" on Mac or a Linux system
  - System has 123 processes, 5 of which are active
  - Identified by Process ID (PID)

Multiprocessing: The (Traditional) Reality



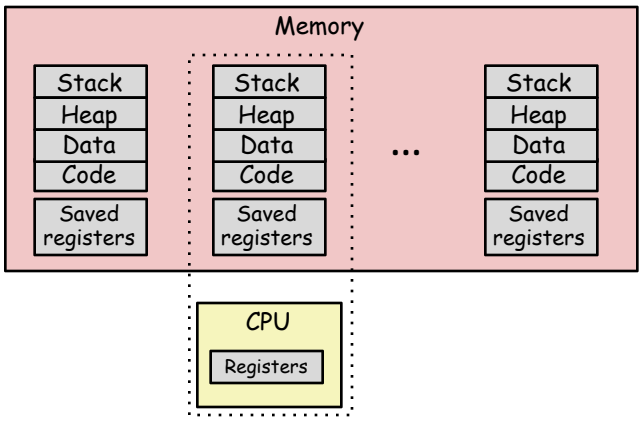
- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (CS 471)
  - Register values for non-executing processes saved in memory

Multiprocessing: The (Traditional) Reality



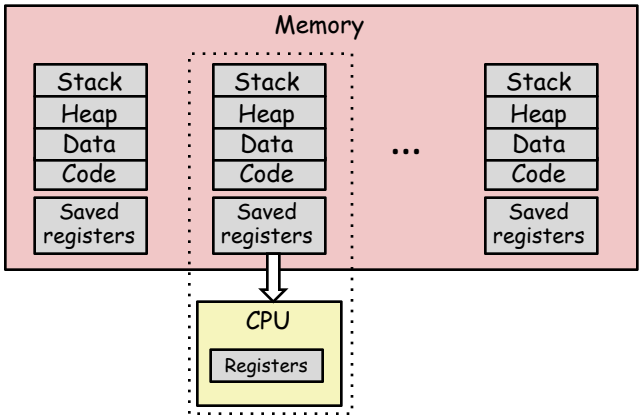
- Save current registers in memory

Multiprocessing: The (Traditional) Reality



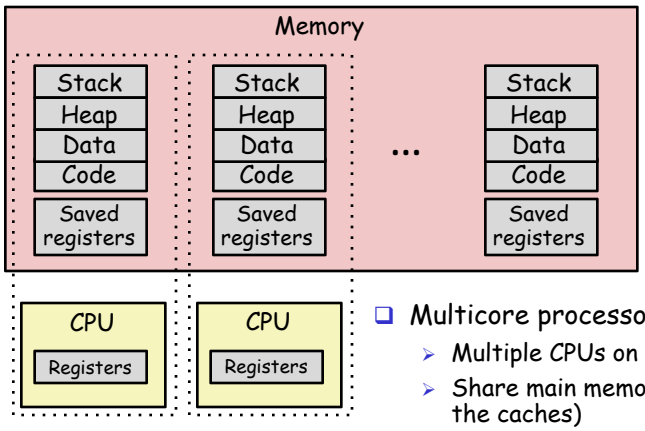
- Schedule next process for execution

Multiprocessing: The (Traditional) Reality



- Load saved registers and switch address space (context switch)

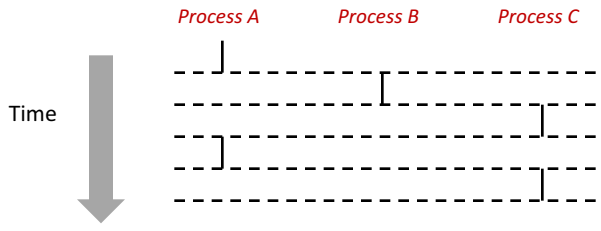
Multiprocessing: The (Modern) Reality



- Multicore processors
  - Multiple CPUs on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Scheduling of processes onto cores done by kernel

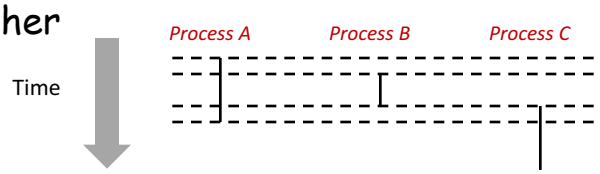
Concurrent Processes

- Each process is a logical control flow.
- Two processes *run concurrently* (*are concurrent*) if their flows overlap in time
- Otherwise, they are *sequential*
- Examples (running on single core):
  - Concurrent: A & B, A & C
  - Sequential: B & C



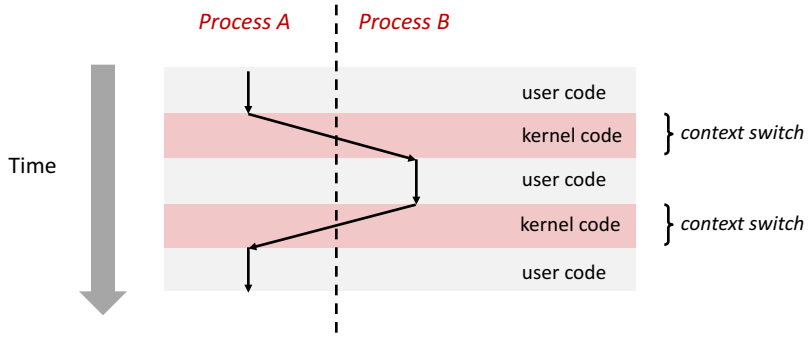
User View of Concurrent Processes

- Control flows for concurrent processes are physically disjoint in time
- However, we can think of concurrent processes as running in parallel with each other



Context Switching

- Processes are managed by a shared chunk of memory-resident OS code called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of some existing process.
- Control flow passes from one process to another via a *context switch*



Cooperating Concurrent Processes

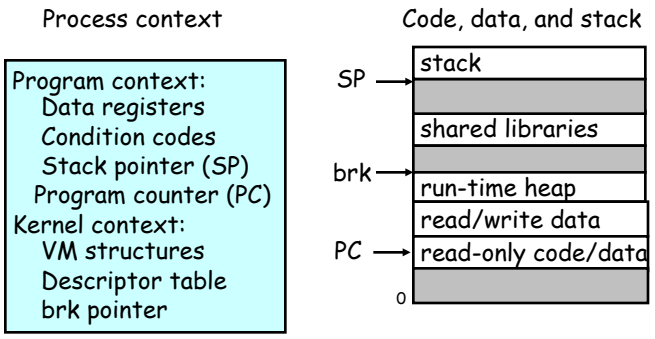
- Concurrent processes part of the same application
- Processes "cooperate" on task
- Motivation
  - Support inherent concurrency in application
    - Window systems, web servers
  - Improved performance - can make use of multiple processors

Concurrent Programs

- Process = Address space + one thread of control
- Concurrent program = **multiple threads of control**
  - Multiple single-threaded processes
  - Multi-threaded process

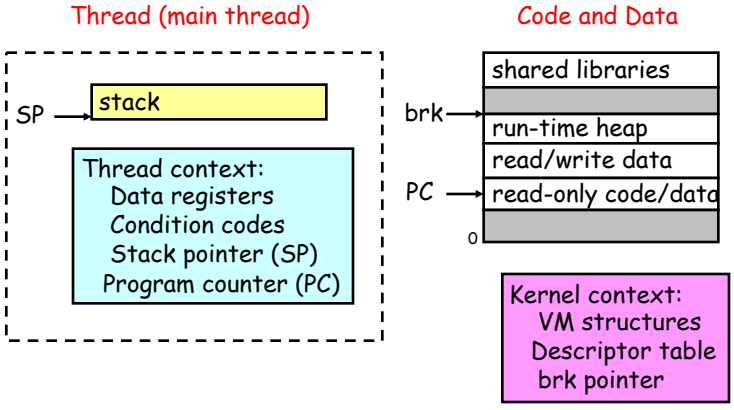
Traditional View of a Process

- Process = process context + code, data, and stack



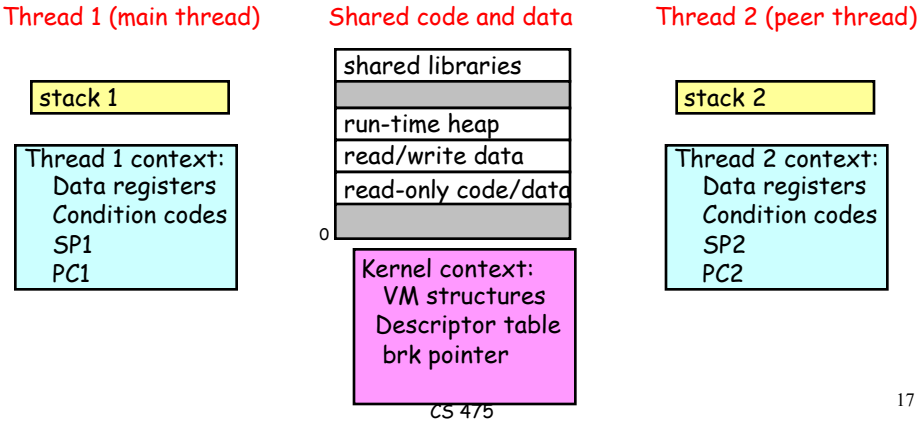
Alternate View of a Process

- Process = thread + code, data, and kernel context



A Process With Multiple Threads

- Multiple threads can be associated with a process
  - Each thread has its own logical control flow (sequence of PC values)
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own thread id (TID)



Threads: Motivation

- Traditional processes created and managed by the OS kernel
- Process creation expensive - fork system call in UNIX
- Context switching expensive
- Cooperating processes - no need for memory protection (separate address spaces)

Threads

- Execute in same address space
  - separate execution stack, share access to code and (global) data
- Smaller creation and context-switch time
- Can exploit fine-grain concurrency

Creating processes

- UNIX
  - fork system call
  - Used in conjunction with exec system call

`fork`: Creating new processes

- `int fork(void)`
  - creates a new process (child process) that is identical to the calling process (parent process)
  - returns 0 to the child process
  - returns child's `pid` to the parent process

```
if (fork() == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Fork is interesting (and often confusing) because it is called *once* but returns *twice*

Creating and Using threads

- Pthreads Multi-threading Library
  - API for thread creation and management
  - `pthread_create, pthread_join, pthread_self, pthread_exit, pthread_detach`
- Java
  - provides a `Runnable` interface and a `Thread` class as part of standard Java libraries
    - users program threads by implementing the `Runnable` interface or extending the `Thread` class

Concurrent Systems

- Essential aspects of any concurrent system
  - Execution context - state of a concurrent entity
    - Processes: process context
    - Threads: thread context
  - Scheduling - deciding which context will run next
    - Processes: Operating System scheduler
    - Threads: Library thread scheduler (Pthreads), Java runtime
  - Synchronization - mechanisms that enable execution contexts to coordinate their use of shared resources
    - Semaphores, locks, monitors, condition variables
    - Provided at both operating system and library/language level

Road Map

- Next two lectures: Processes & Signals in UNIX
  - Some of this will be a repetition of material discussed in CS 367
  - Assignment 1 (Shell Lab)
- Thread creation and management in Java and Pthreads (one lecture)
- Process & Thread synchronization mechanisms (two – three lectures)