

Components of a Process

- **Program vs. process**
- **Process:**
 - object code of program ("program text" in UNIX)
 - data on which the program will execute (from file or user interaction)
 - resources required by the program (e.g., files)
 - status of the process execution (e.g., PC and registers)
- OS keeps **process descriptor** for each (generally) non-terminated process
- **Traditional process:** one address space and one flow of control
- **"Modern" process:** one address space and one or more flows of execution (*threads*) – **MORE LATER**

CS414: Operating Systems

Process Creation

- Parent process creates **children** processes, which, in turn create other processes, forming a tree of processes
- **Resource sharing options**
 - Parent and child share all resources
 - Children share subset of parent's resource
 - Parent and child share no resources
- **Execution options**
 - Parent and child execute concurrently
 - Parent waits until children terminate
- **Address Space**
 - Child duplicate of parent
 - Child has a program loaded into it

CS414: Operating Systems

UNIX fork

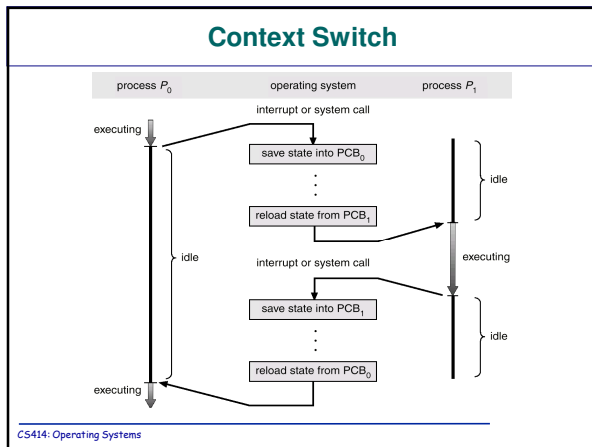
- **fork()** creates **new child process**; one process executes **fork()** and two become ready
- Child is identical to parent
 - **except** for return code from fork()
- **No part** of the address space is **shared**
 - parent and child communicate via **pipes, explicitly shared memory, and shared files**
- Parent can **wait** or **continue**
- Child can **exec**
- **Example:** your **shell** program; typing *date* forks a new process and then execs date; "&" after the command means it runs in parallel with your shell, otherwise it waits {**DEMO**}

CS414: Operating Systems

Context Switch

- To run a process, the OS **loads** the values of the **hardware registers** (PC, SP, other registers) from the values stored in that process' **PCB**
- As the program executes, the CPU registers **changes values** (PC, SP)
- When CPU **switches** to another process, the system must **save** the state of the old process and **load** the **saved state** for the new process
- Context-switching is **overhead**; the system does no **useful work** while switching
- The **duration** of a context switch is dependent on hardware
- Typically, time sharing OS performs 100 to 1000 **context switches per second**
- **Picture on next slide....**

CS414: Operating Systems



Threads

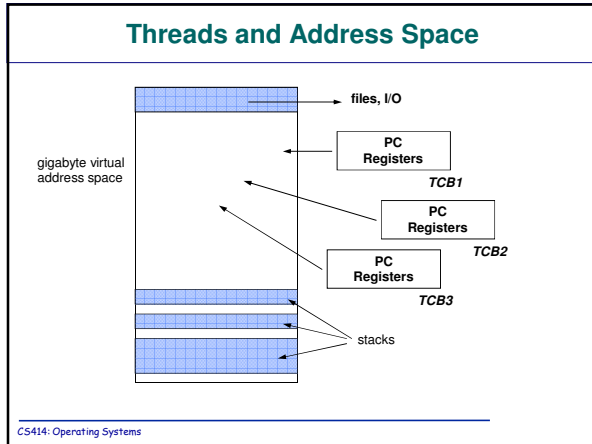
- **"Original" Process:** address space and (single) flow of execution
- **Thread**
 - we must separate address space (*process or task*) and flow of execution (*thread or lightweight process, LWP*)
- **Motivation:**
 - Context switch between cooperating processes is **HUGE** (reestablishing address space); context switch between cooperating threads is cheap
 - **fork(...)** of cooperating process is expensive; spawn of thread is cheap
 - programming is easier(?)

traditional UNIX

embedded systems

Windows, Solaris (POSIX)

CS414: Operating Systems



Sample Pthreads Code (1/2)

```

// Pthreads example code
// compile with: gcc -Wall pthreads_example.c -o pthreads_example -lpthread
// execute with: ./pthreads_example (or "time ./pthreads_example")

#include <pthread.h>
#include <stdio.h>

void thread_func (int num)
{
    long long i;

    printf("Thread %d executing.\n", num);
    j = 0;
    for (i=0; i < 700000000U; ++i){
        ++i;
    }
    printf("Thread %d done (%lld).\n", num, j);
}
    
```

CS414: Operating Systems

Sample Pthreads Code (2/2)

```

int main ()
{
    pthread_t thread1, thread2;
    pthread_t thread3, thread4;

    pthread_create(&thread1, NULL, (void *) &thread_func, (void *) 1);
    pthread_create(&thread2, NULL, (void *) &thread_func, (void *) 2);
    pthread_create(&thread3, NULL, (void *) &thread_func, (void *) 3);
    pthread_create(&thread4, NULL, (void *) &thread_func, (void *) 4);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);
    pthread_join(thread4, NULL);

    return 0;
}
    
```

CS414: Operating Systems

DEMO

- Run the previous program on Linux
 - Hmmm.. Why is "top" saying that only one of the CPUs is pegged at 100%? Why aren't both pegged?

CS414: Operating Systems

- ### Kernel Threads
- An improvement over only processes
 - Creation/Switching still requires trapping to the kernel (system call)
 - Thread data structure resides within the kernel:
 - Thread Control Block (TCB) (execution state and scheduling info), so more complexity for the kernel
 - Only one scheduling policy per system
 - OS (still) does not trust the user, so there must be a lot of checking on kernel calls
- CS414: Operating Systems

- ### User-level Threads
- Faster than kernel-level threads
 - In what sense?
 - Managed by run-time system in user-space (no kernel calls)
 - Creation, switching, and synchronizing between thread calls can be done without kernel involvement
 - Process-specific scheduling policies are possible
 - Problem: whole process blocks when one thread blocks (there are ways around this, but they're complex)
 - OS can make poor scheduling choices, because OS has no notion of "amount of work" each process must do
- CS414: Operating Systems

Comparing

	PROS	CONS
User-level threads		
Kernel-level threads		

CS414: Operating Systems

“Hybrid”: Solaris 2

- Solaris 2 is a version of UNIX with support for threads at the kernel and user levels, symmetric multiprocessing, and real-time scheduling.
- LWP – intermediate level between user-level threads and kernel-level threads.
- Resource needs of thread types:
 - Kernel thread: small data structure and a stack; thread switching does not require changing memory access information – relatively fast.
 - LWP: PCB with register data, accounting and memory information,; switching between LWPs is relatively slow.
 - User-level thread: only need stack and program counter; no kernel involvement means fast switching. Kernel only sees the LWPs that support user-level threads.

CS414: Operating Systems

