Threads
Readings

- Silberschatz et al: Chapter 4
Motivation

- Sometimes a program needs to do multiple tasks concurrently
- **Example:** Word processor
  - Tasks include: Display graphics, respond to keystrokes from the user, and perform spelling and grammar checking
Motivation

- Example: Web server
  - It is desirable to service requests concurrently

```
client ➔ server ➔ thread
```

(1) request
(2) create new thread to service the request
(3) resume listening for additional client requests
Introduction

- Earlier we discussed the use of forking to create a process

- For example we could
  - Word processor example: fork a process for each task
  - Web server example: fork a process for each request

- It turns out there is a more efficient alternative
Threads

- A thread is a basic unit of CPU utilization.
- Threads of a process share memory but can execute independently.
- A traditional process can be viewed as a memory address space with a single thread.
A word processor program with three threads.
Thread Usage - Web Server

A multithreaded Web server.
Why Not Fork?

- You certainly can fork a new process
- In fact, the first implementation of Apache web servers (Apache 1.0) forked N processes when the web server was started
  - “N” was defined in a configuration file
  - Each child process handled one connection at a time
- Problem: Process creation is time consuming and resource intensive
- Creating threads is not as expensive
Why Not Fork?

Let's look at web servers

- This allowed a child process to handle multiple connections at a time
- Web servers have caches for read pages
- Forking means that these caches cannot be shared
- Using threads allows for these caches to be shared
Thread State

- Threads share
  - Process address space
    - Text
    - Data (global variables)
    - Heap (dynamically allocated data)
  - OS state
    - Open files, sockets, locks
- Threads have their own CPU context
  - Program counter (PC), Stack pointer (SP), register state, stack
Single and Multithreaded Processes

single-threaded process

multithreaded process
Benefits of Threads

- Responsiveness
  - Overlap computation and blocking due to I/O on a single CPU

- Resource Sharing
  - Example: Word processor
    - The document is shared in memory.
    - Forking would require replication

- Allocating memory and resources for process creation is costly

- Context-switching is faster
Questions

Can one thread access local variables of another thread?

If one thread allocates a data structure from the heap, can other threads access it?
Thread Libraries

- A thread library provides the programmer with an API for creating and managing threads
- Three main libraries:
  - POSIX Pthreads
  - Win32
  - Java
Pthreads: POSIX Threads

- **Pthreads** is a standard set of C library functions for multithreaded programming
  - IEEE Portable Operating System Interface, POSIX, section 1003.1 standard, 1995

- Pthread Library (60+ functions)

- Programs must include the file `pthread.h`

- Programs must be linked with the pthread library (`-lpthread`)
  - Done by default by some gcc's (e.g., on Mac OS X)
pthread_create()

- Creates a new thread

```c
int pthread_create (  
    pthread_t *thread,  
    pthread_attr_t *attr,  
    void * (*start_routine) (void *),  
    void *arg);
```

- Returns 0 to indicate success, otherwise returns error code
- **thread**: output argument for the id of the new thread
- **attr**: input argument that specifies the attributes of the thread to be created (NULL = default attributes)
- **start_routine**: function to use as the start of the new thread
  - must have prototype: void *foo(void*)
- **arg**: argument to pass to the new thread routine
  - If the thread routine requires multiple arguments, they must be passed bundled up in an array or a structure
Let us say that you want to create a thread to compute the sum of the elements of an array.

```c
void *do_work(void *arg);
```

Needs three arguments:
- the array, its size, where to store the sum
- we need to bundle these arguments in a structure

```c
typedef struct arguments {
    double *array;
    int size;
    double *sum;
} arguments;
```
int main(int argc, char *argv) {
    double array[100];
    double sum;
    pthread_t worker_thread;
    arguments *arg;

    arg = (struct arguments *)malloc(sizeof(struct arguments));

    arg->array = array;
    arg->size=100;
    arg->sum = &sum;

    if (pthread_create(&worker_thread, NULL,
                        do_work, (void *)arg)) {
        fprintf(stderr,"Error while creating thread\n");
        exit(1);
    }
    ...
}
void *do_work(void *arg) {
    arguments *argument;
    int i, size;
    double *array;
    double *sum;

    argument = (struct arguments*)arg;

    size = argument->size;
    array = argument->array;
    sum = argument->sum;

    *sum = 0;
    for (i=0; i<size; i++)
        *sum += array[i];

    return NULL;
}
Thread Creation

- Thread identifiers
  - Each thread has a unique identifier (ID), a thread can find out its ID by calling `pthread_self()`.
  - Thread IDs are of type `pthread_t` which is usually an unsigned int. When debugging it's often useful to do something like this:

```c
printf("Thread %u: blah\n", pthread_self());
```
Shared Global Variables

All threads within a process can access global variables.

```c
int counter=0;
void *blah(void *arg) {
    counter++;
    printf("Thread %u is number %d\n",
           pthread_self(),counter);
}
main() {
    int i; pthread_t threadid;
    for (i=0;i<10;i++)
        pthread_create(&threadid,NULL,blah,NULL);
}
```
Problem

- Sharing global variables is dangerous - two threads may attempt to modify the same variable at the same time.

- **pthreads** includes support for mutual exclusion primitives that can be used to protect against this problem.

- The general idea is to **lock** something before accessing global variables and to **unlock** as soon as you are done.

- More on this topic later in the course
Processes vs Threads

- A beta version of Google's browser, Chrome, was released on 2 September 2008.
- Before Chrome browsers allocated a thread for each tab
- Chrome allocates a process for each tab
Question: In a multithreaded program, when a `fork()` system call is executed by a thread, does the new process duplicate all the threads or is the new process single threaded?

Answer: Depends on the implementation
- Some UNIX systems have two versions of `fork()`
  - One that duplicates all threads
  - One that duplicates only the thread that invoked for the `fork()` system call
Threading Issues - fork() and exec()

- The `exec()` system call works the same way we saw earlier - i.e., it replaces the entire process including all threads.
Thread Libraries

- Three main libraries in use
  - POSIX PThreads
    - User or kernel level
  - Win32
    - Kernel level library
    - Used in Windows OS
  - Java threads
    - JVM is running on top of a host OS
    - The Java thread API is implemented using a thread library available on the system
User Space Threads

- All code and data structures for the library exist in user space
- Invoking a function in the library results in a local function call in user space and not a system call
- Kernel knows nothing about the threads package
- Kernel uses the Process Control Block (PCB) data structure to represent threads
- How do user and kernel threads map into each other?
One to One Model

- Each user-level thread maps to kernel thread.
- Examples
  - Windows, Linux, Solaris versions starting with Solaris 9
Many to One

- Many user-level threads mapped to single kernel thread

- Examples:
  - Older versions of Solaris
  - GNU Portable Threads
One to One vs Many to One

- One to one has higher overhead
- One to one provides more concurrency
- Many to many limits concurrency
Thread Scheduling

- In Linux-based system threads compete with processes (treated the same)
- In other systems competition for the CPU takes place among threads belong to the same process
Question

Which of the following components are shared among threads?

- Register values
- Heap memory
- Global variables
- Stack Memory
Question

Can you give an example in which multithreading does not provide better performance than a single-threaded solution?
Multithreaded programming provides a mechanism for more efficient use of multiple computing cores.

On a single core we have concurrency where the execution of the processes/threads are interleaved over time.

Multicore allows for parallelism where more than one task can be done simultaneously.
Summary

- Introduction to the concept of threads
- There will be more discussion