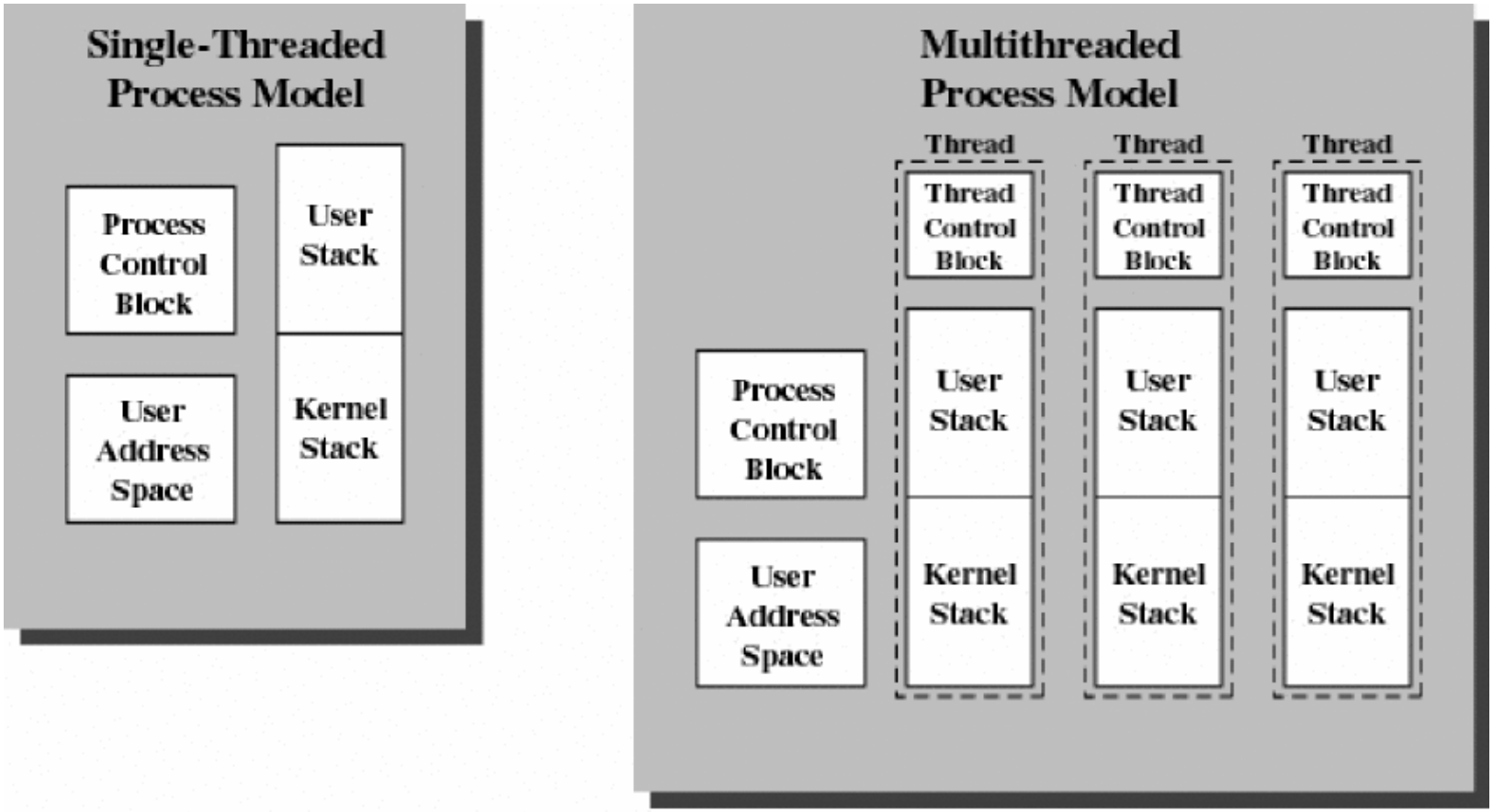
# Programming Threads in C

## CS 360 Internet Programming

Daniel Zappala

Brigham Young University
Computer Science Department

# Process and Thread Models




*from Operating Systems: Internals and Design Principles, fifth edition, by William Stallings, Prentice Hall, 2005*

# Process vs. Thread

- process: unit of resource ownership
  - process is allocated a virtual address space: process image
  - assigned main memory, I/O devices, files
  - protection from other processes
- thread: unit of dispatch/execution
  - instruction trace through a program
  - thread execution state
  - context: stack, storage for local variables
  - access to memory and resources of the process – shared with all other threads for the same process

# Benefits of Threads

- faster to create a new thread than a process
- faster to terminate a thread than a process
- faster to switch between two threads within the same process
- more efficient communication between threads
  - process communication requires protection and communication provided by kernel
  - threads can avoid the kernel
- parallel processing

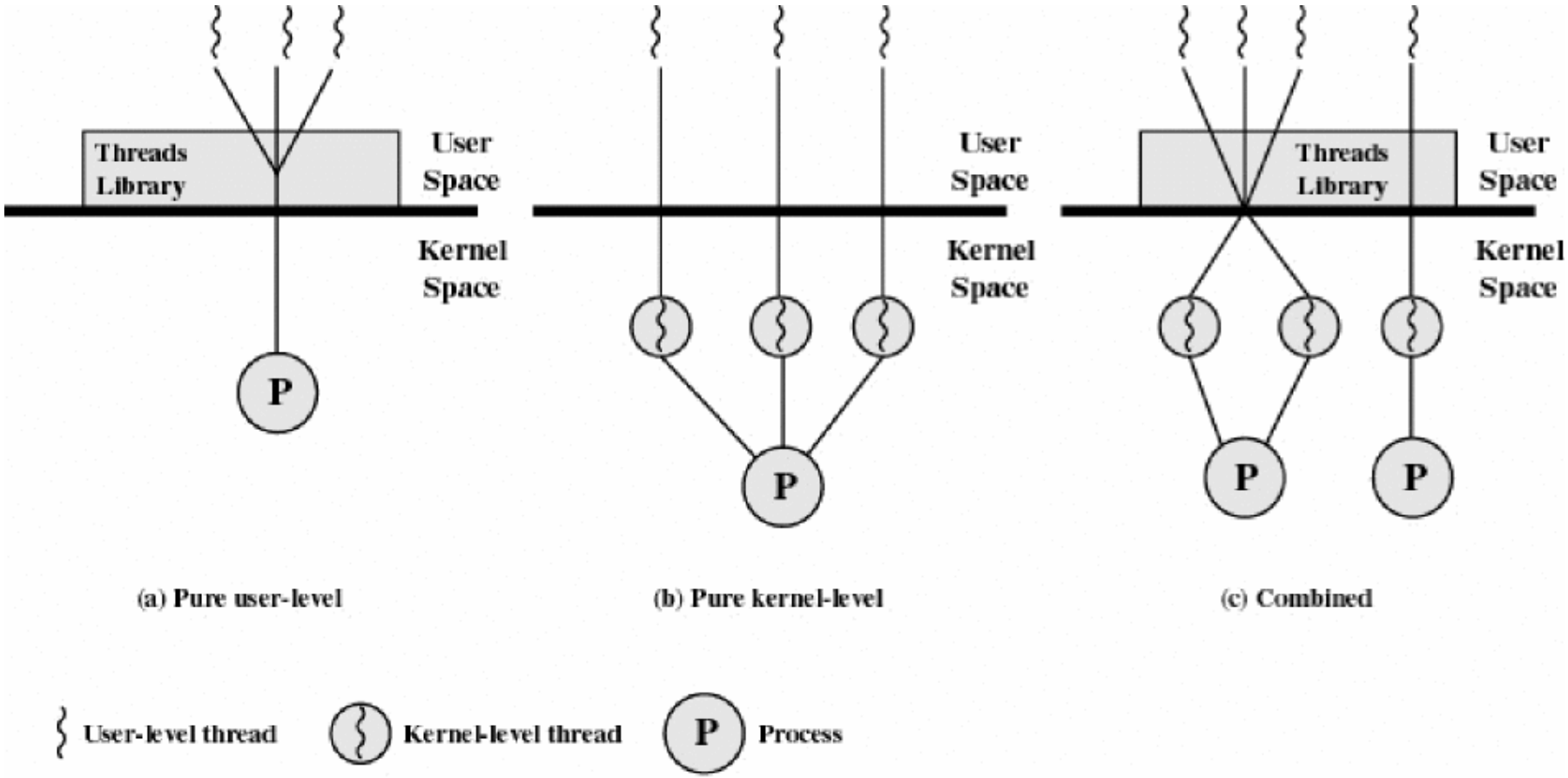
# Using Threads

- foreground and background work
  - a thread for the GUI and another to execute tasks
  - faster response time for the user
- networking communication
  - separate threads to handle blocking system calls so multiple connections can be handled simultaneously
- modular program structure
  - give logically separate functionality to different threads
  - easily program with blocking system calls

# Thread Support in Operating Systems

- MS-DOS: one process, one thread
- old Windows, UNIX: multiple user processes, but only one thread per process
- JVM: one process, multiple threads
- modern operating systems (Linux, Windows 2000+, Solaris, Mach): multiple threads per process

# User-Level and Kernel-Level Threads




*from Operating Systems: Internals and Design Principles, fifth edition, by William Stallings, Prentice Hall, 2005*

# User-Level Threads

- all thread management done by the application
  - creating and destroying threads
  - thread communication
  - thread synchronization
  - thread scheduling
- runs in a single process, no kernel involvement
- advantages
  - efficient: no kernel mode switch to handle a different thread
  - application-specific scheduling
  - O/S independent
- disadvantages
  - thread system call blocks entire process
  - no multiprocessing: threads of the same process cannot run on different processors

# Kernel-Level Threads

- thread management handled by kernel
- kernel schedules threads, not processes
- advantages
  - multiprocessing support
  - blocked thread doesn't block entire process
  - kernel can be multithreaded
- disadvantages
  - thread switching more expensive: requires mode switch

# Introduction

- *Pthreads*: POSIX threads library
  - *POSIX is the collective name of a family of related standards specified by the IEEE to define the application programming interface (API) for software compatible with variants of the Unix OS.* – Wikipedia
  - threads standardized in 1995
- Linux
  - 1:1 mapping to kernel level threads
  - compile application with *gcc/g++ -pthread*
  - Native POSIX Thread Library (NPTL): *In tests, NPTL succeeded in running 100,000 threads simultaneously on a IA-32 which were started in two seconds. In comparison, this test under a kernel without NPTL would have taken around 15 minutes.* – Wikipedia
- http://www.llnl.gov/computing/tutorials/pthreads/

# Creating a Thread

- when a program starts, it runs in a single thread called the main thread
- create threads with `pthread_create()`

```
1 #include <pthread.h>
2
3 int pthread_create(pthread_t * thread, pthread_attr_t * attr,
4 void * (*start_routine)(void *), void * arg);
```

- the thread identifier is returned through the thread pointer
- the new thread runs the given start routine with the given arguments, terminates by finishing this routine
- attributes include priority, stack size, etc. - leave as default by passing a null pointer
- return value is normally zero, return positive error value otherwise

# Joining a Thread

- wait for threads to terminate with `pthread_join()`

```
1 #include <pthread.h>
2
3 int pthread_join(pthread_t th, void **thread_return);
```

- specify thread identifier of thread to wait for
- return value of thread in given in returned pointer if non-null
- must call join to reclaim thread memory and thus avoid memory leaks

# Getting a Thread ID

- get your own thread ID with `pthread_self()`

```
1 #include <pthread.h>
2
3 pthread_t pthread_self(void);
```

- returns the thread's thread identifier

# Exiting a Thread

- exit a thread with `pthread_exit()`

```
1 #include <pthread.h>
2
3 void pthread_exit(void *retval);
```

- returned value can be any object that is not local to the thread

# Print Test

```
1  // initialize data to pass to thread 1
2  data1.number = 1;
3  strcpy(data1.message, "Hello!");
4
5  // initialize data to pass to thread 2
6  data2.number = 2;
7  strcpy(data2.message, "Goodbye!");
8
9  // create threads
10 pthread_create (&thread1, NULL, &print, (void *) &data1);
11 pthread_create (&thread2, NULL, &print, (void *) &data2);
```

- *see threading example code on web site*

# Threaded Server

```
1  while ((c = accept(s,(struct sockaddr *)&client,&clientlen)) > 0) {
2    // create a thread to handle the client
3    pthread_t tid;
4    int *arg = new int;
5    *arg = c;
6    pthread_create(&tid, NULL, &worker, arg);
7  }
```

- be sure to allocate memory for arguments and free it in thread
- creating one thread per client is not scalable
- *see threaded server example code on web site*