Operating Systems
CS220

Lecture 8

Threads

3rd May 2021

By: Dr. Rana Asif Rehman

What's in today's lecture

- Thread Concept
- Multithreading Models
- User & Kernel Threads
- Pthreads
- Threads in Solaris, Linux, Windows

Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Introduction

- Each process has
  1. Own Address Space
  2. Single thread of control
- A process model has two concepts:
  1. Resource grouping
  2. Execution
- Sometimes it is useful to separate them

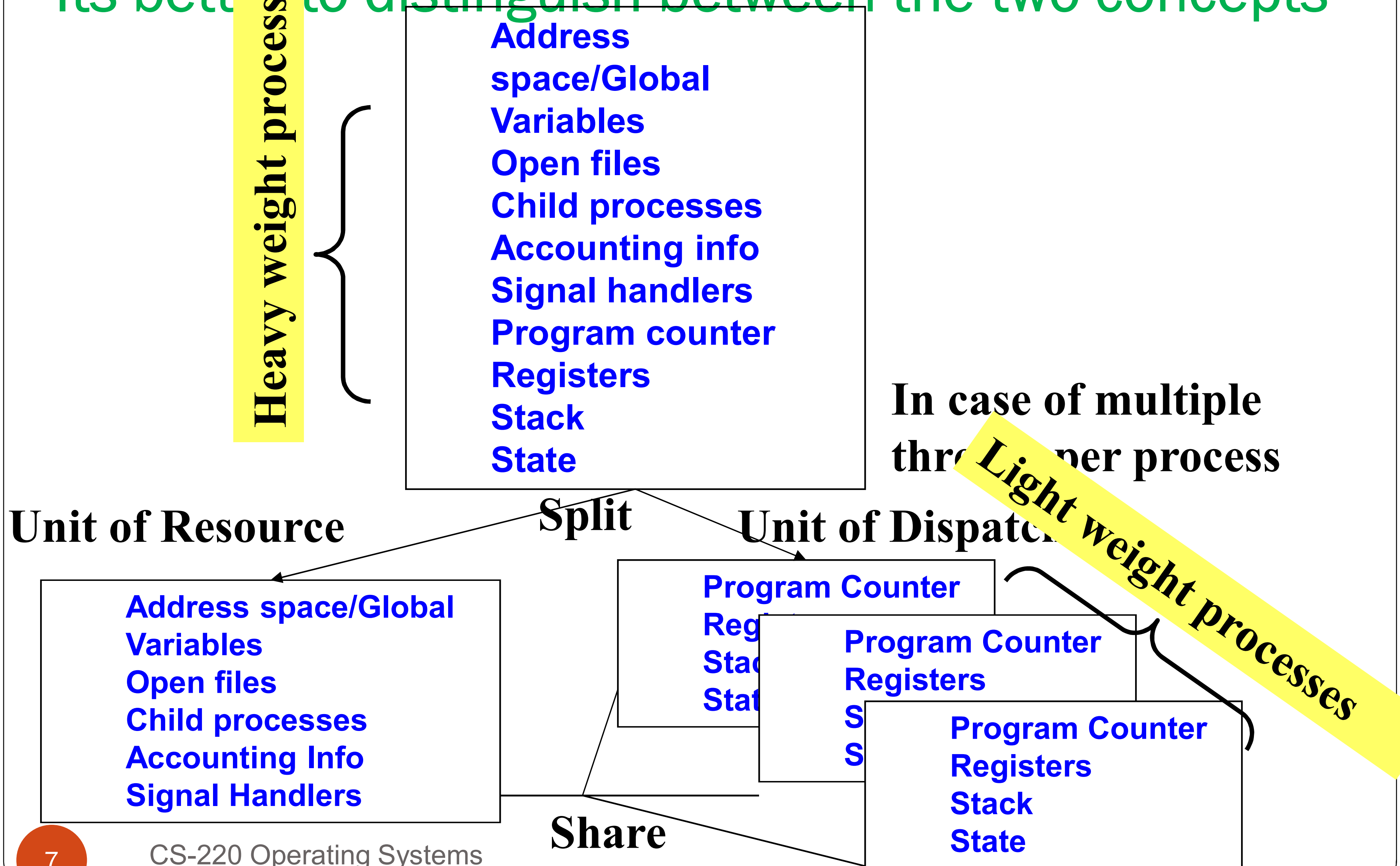
Unit of Resource Ownership

- A process has an
  - Address space
  - Open files
  - Child processes
  - Accounting information
  - Signal handlers
- If these are put together in a form of a process, can be managed more easily

Unit of Dispatching

- Path of execution
  - Program counter: which instruction is running
  - Registers:
    - holds current working variables
  - Stack:
    - Contains the execution history, with one entry for each procedure called but not yet returned
  - State
- Processes are used to group resources together
- Threads are the entities scheduled for execution on the CPU
- Threads are also called ***lightweight*** process (LWP)

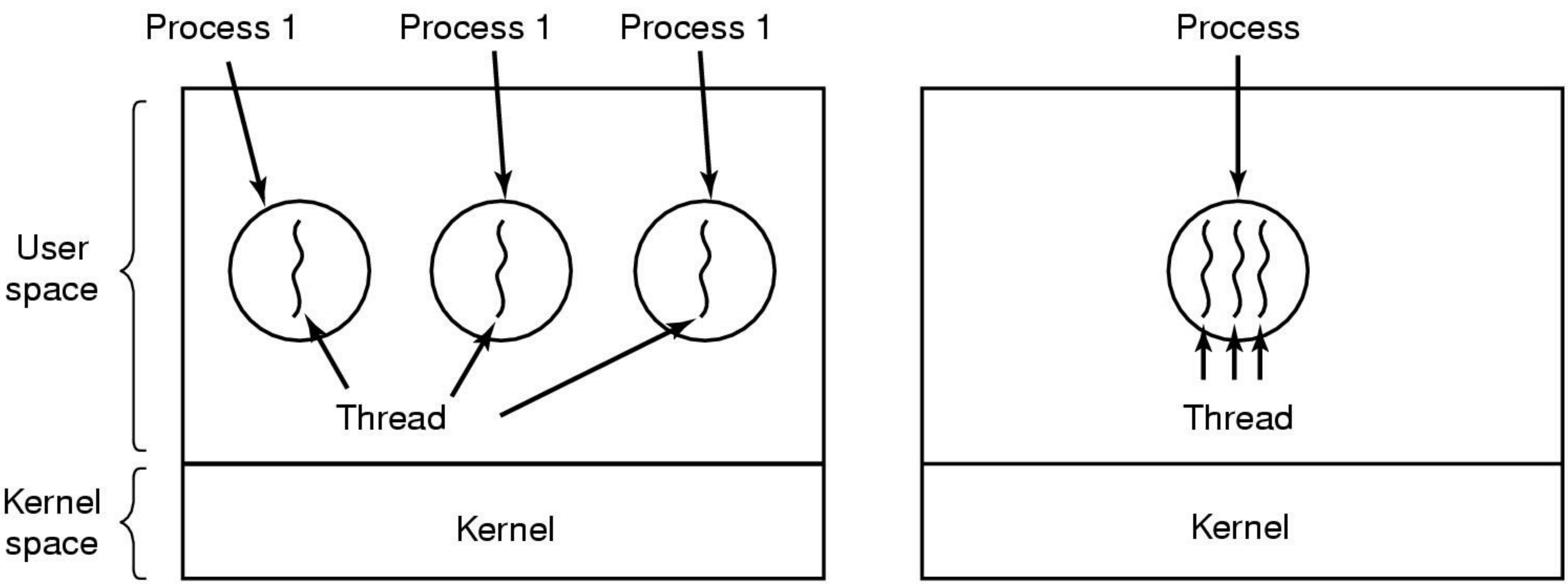
Its bett to distinguish between the two concepts



Process Vs. Thread

- Process
  - All resources allocated: IPC channels, files etc..
  - A virtual address space that holds the process image
  - Protected access to processors, other processes, I/O resources, and files
- Threads
  - a dispatchable unit of work
  - an execution state: running, ready, etc..
  - saved thread context (when not running)
  - an execution context: PC, SP, other registers
  - a per-thread stack
  - Access to the memory and resources of the process it belongs to
    - all threads of the same process share this

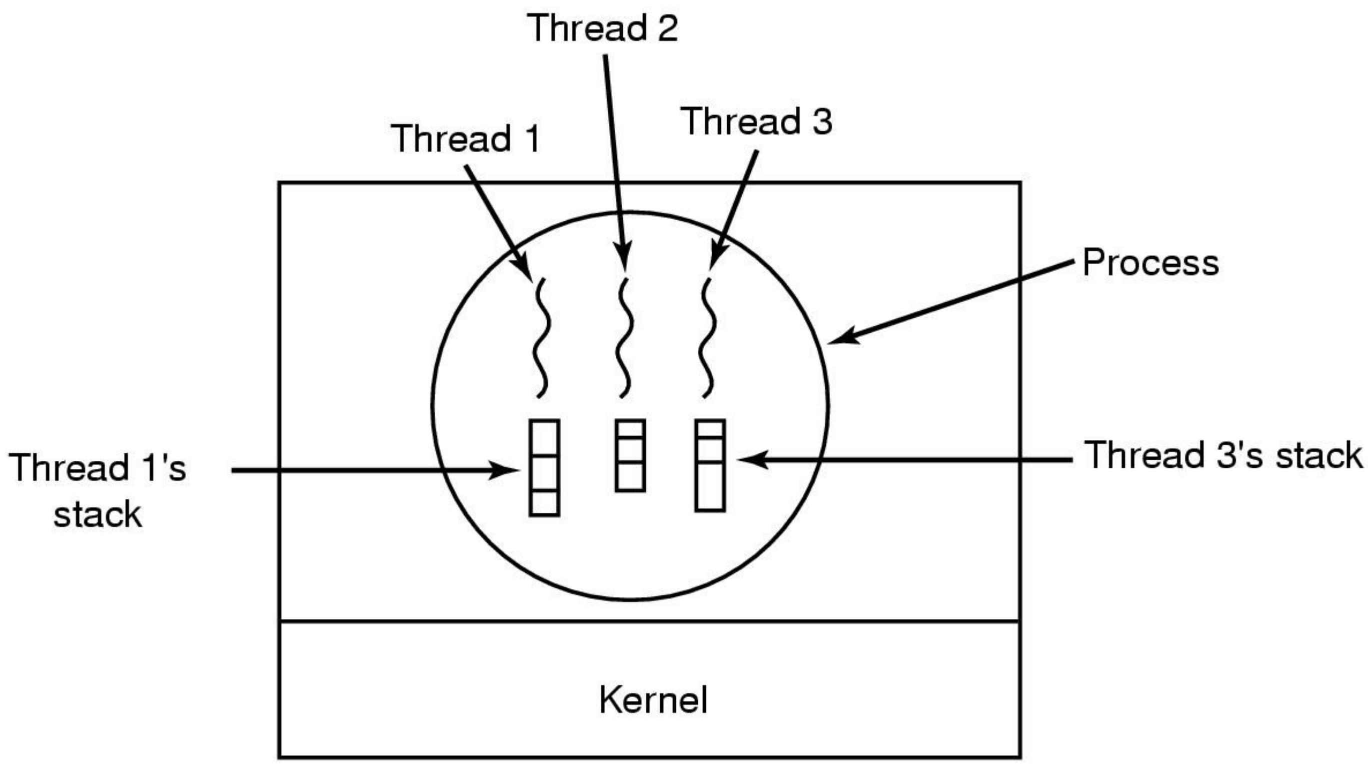
Process Vs. Threads



(a) Three threads, each running in a separate address space

(b) Three threads, sharing the same address space

The Thread Model



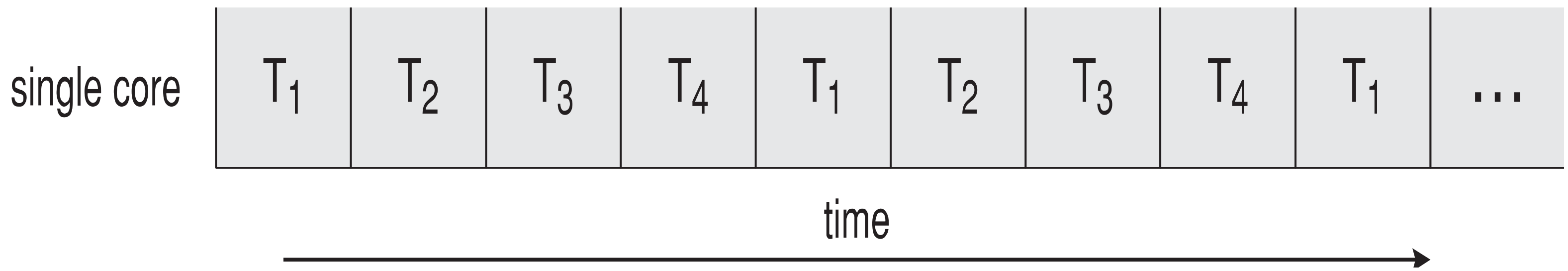
Each thread has its own stack

Multicore Programming

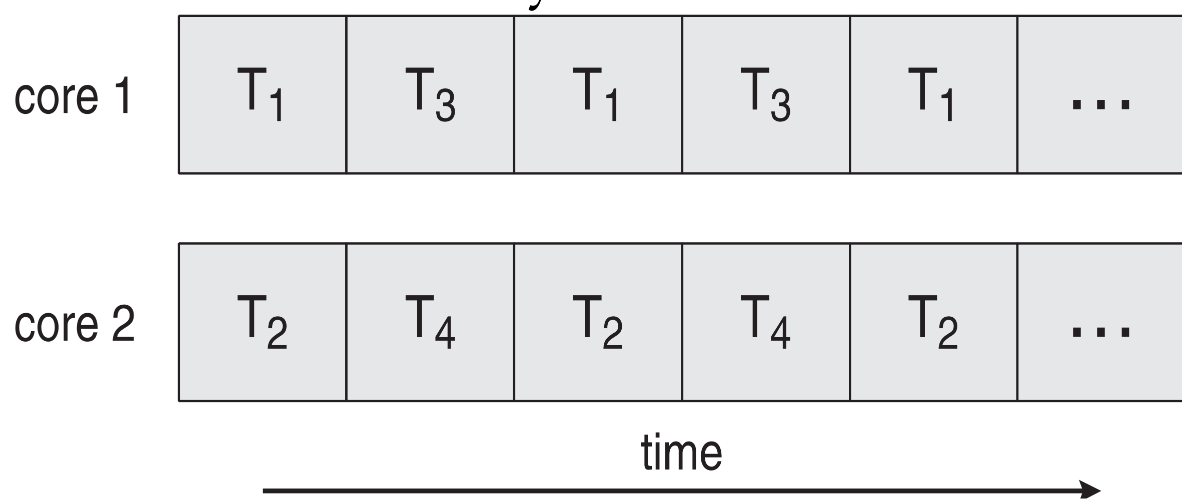
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

Concurrency vs. Parallelism

- Concurrent execution on single-core system:



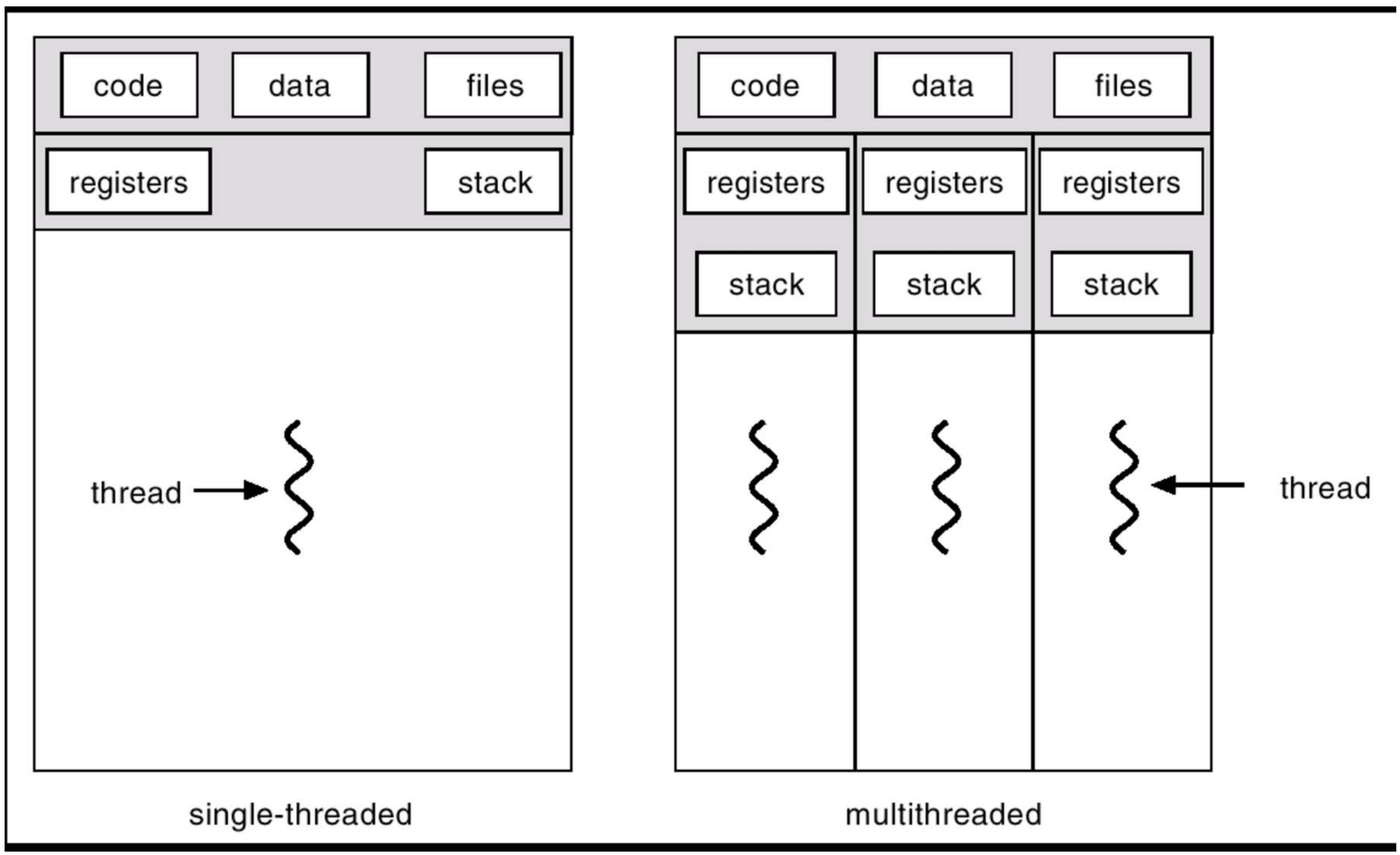
- Parallelism on a multi-core system:



Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
  - CPUs have cores as well as ***hardware threads***
  - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

Single and Multithreaded Processes



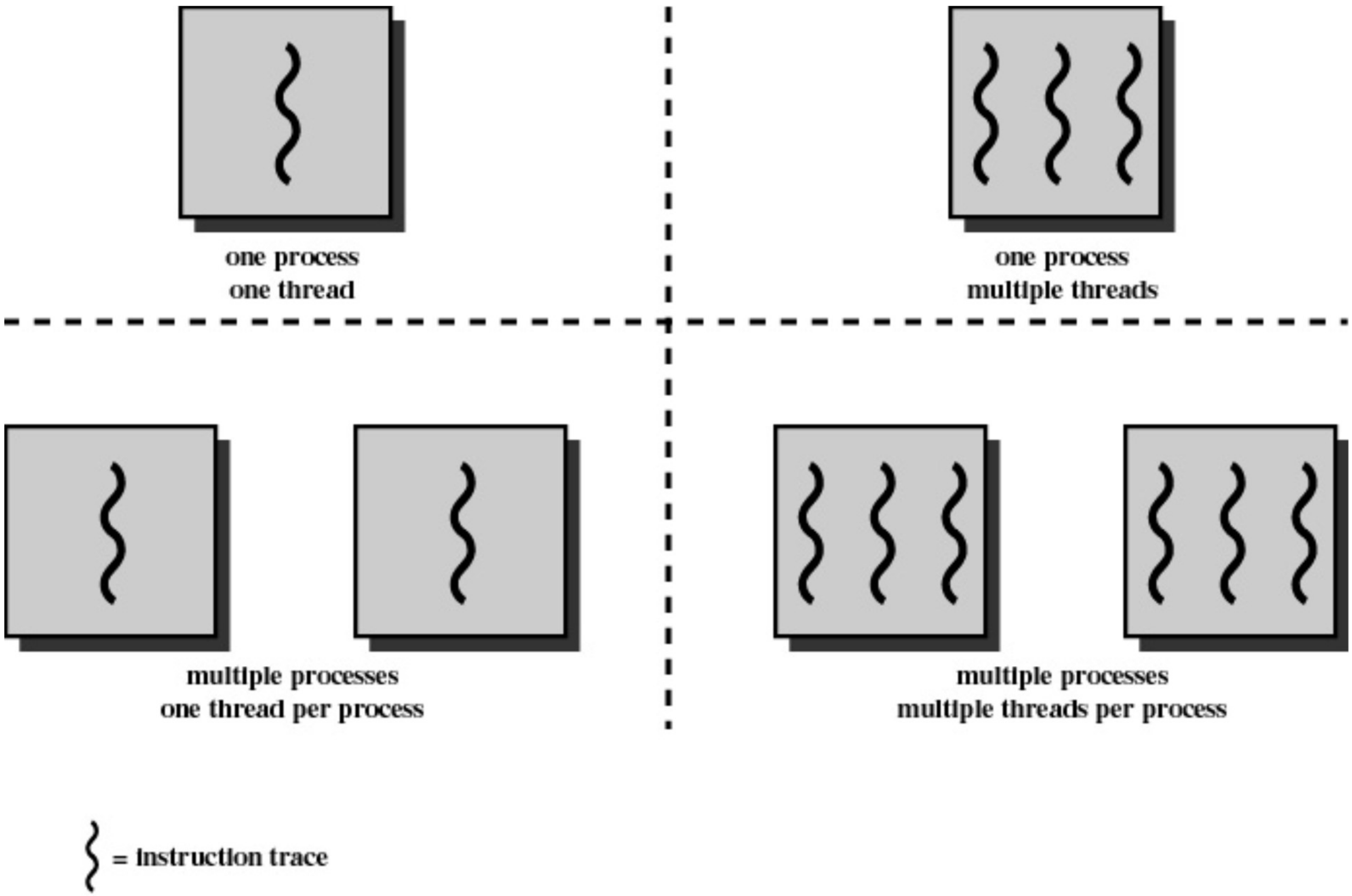


Figure 4.1 Threads and Processes [ANDE97]

Threads

- Allow multiple execution paths in the same process environment (Within an address space, we can have more units of execution: **threads)**
- All the threads of a process share the same address space and the same resources
- But have own set of Program counter, Stack etc
- The first thread starts execution with
  - `int main(int argc, char *argv[])`
- The threads appear to the Scheduling part of an OS just like any other process

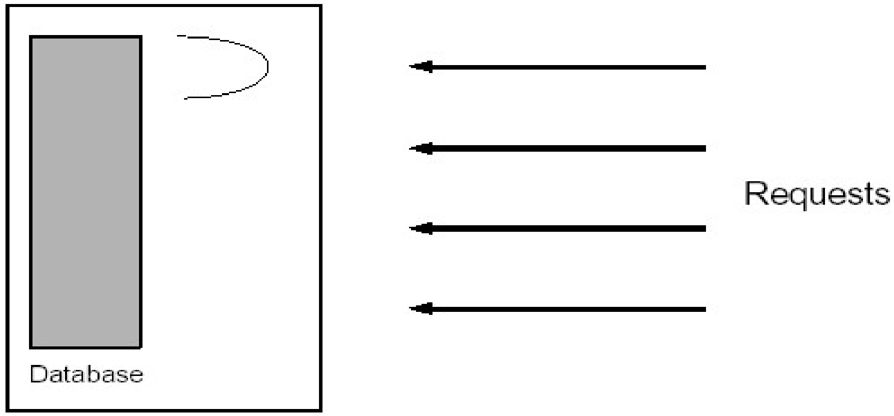
Threads

- Advantages
  - operations on threads (creation, termination, scheduling, etc..) are **cheaper** than the corresponding operations on processes
  - inter-thread communication is supported through **shared memory** without kernel intervention
  - Responsiveness, Resource Sharing, Utilization of MP Architectures
- Disadvantages
  - easy to introduce race conditions
  - synchronization is necessary

Thread Usage

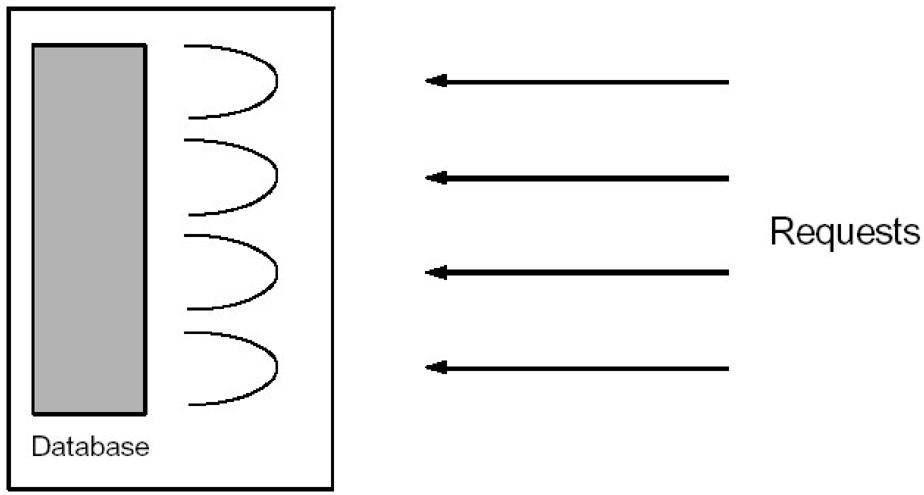
- Less time to create a new thread than a process
  - the newly created thread uses the current process address space
  - no resources attached to them
- Less time to terminate a thread than a process.
- Less time to switch between two threads within the same process, because the newly created thread uses the current process address space.
- Less communication overheads
  - threads share everything: address space, in particular. So, data produced by one thread is immediately available to all the other threads
- Performance gain
  - Substantial Computing and Substantial Input/Output
- Useful on systems with multiple processors

1. Single threaded database server



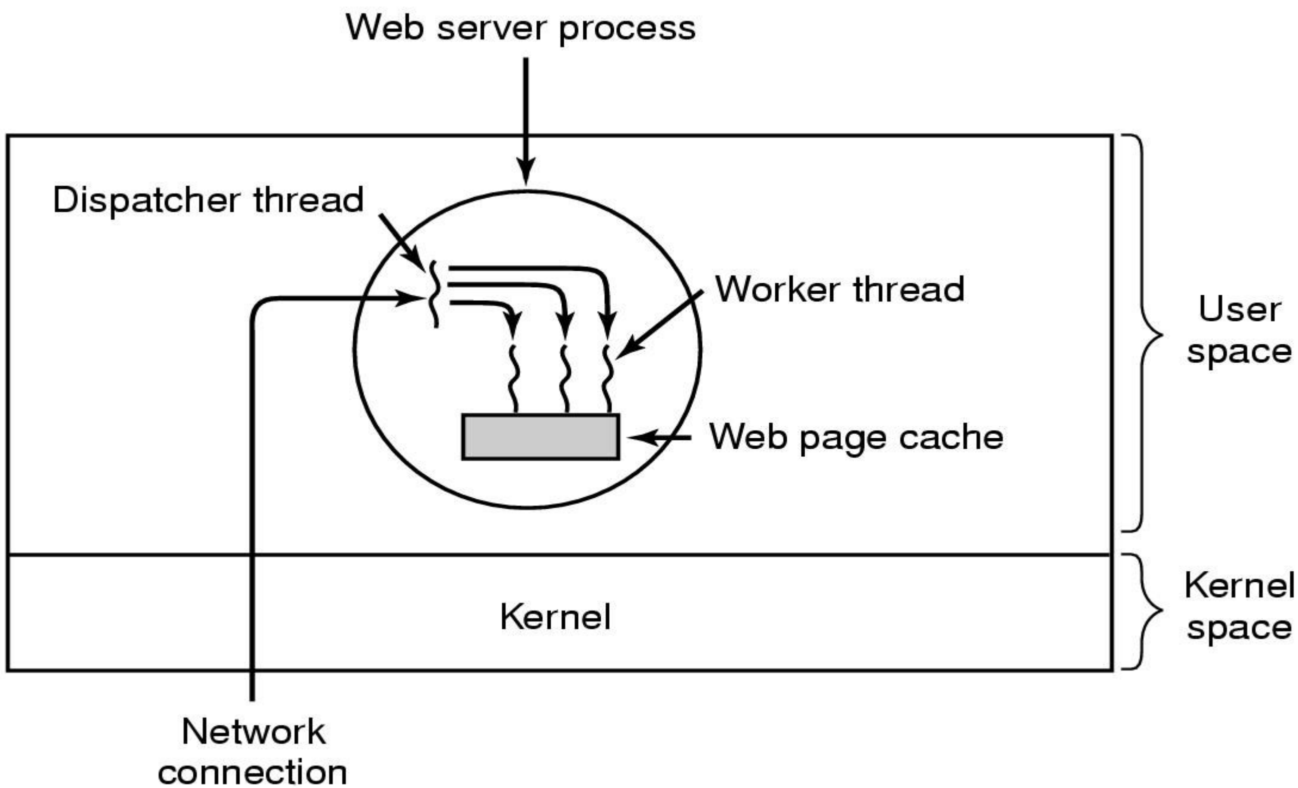
- Handles multiple clients
  - Either handle the requests sequentially
  - Or multiplex explicitly by creating multiple processes
- Problems
  - Unfair for quick requests, occurring behind lengthy request
  - Complex and error prone
  - Heavy IPC required

1. Multithreaded database server



- Assign a separate thread to each request
- As fair as in the multiplexed approach.
- The code is as simple as in the sequential approach, since the address space is shared - all variables are available
- Some synchronization of access to the database is required, this is not terribly complicated.

e.g. A multithreaded web server

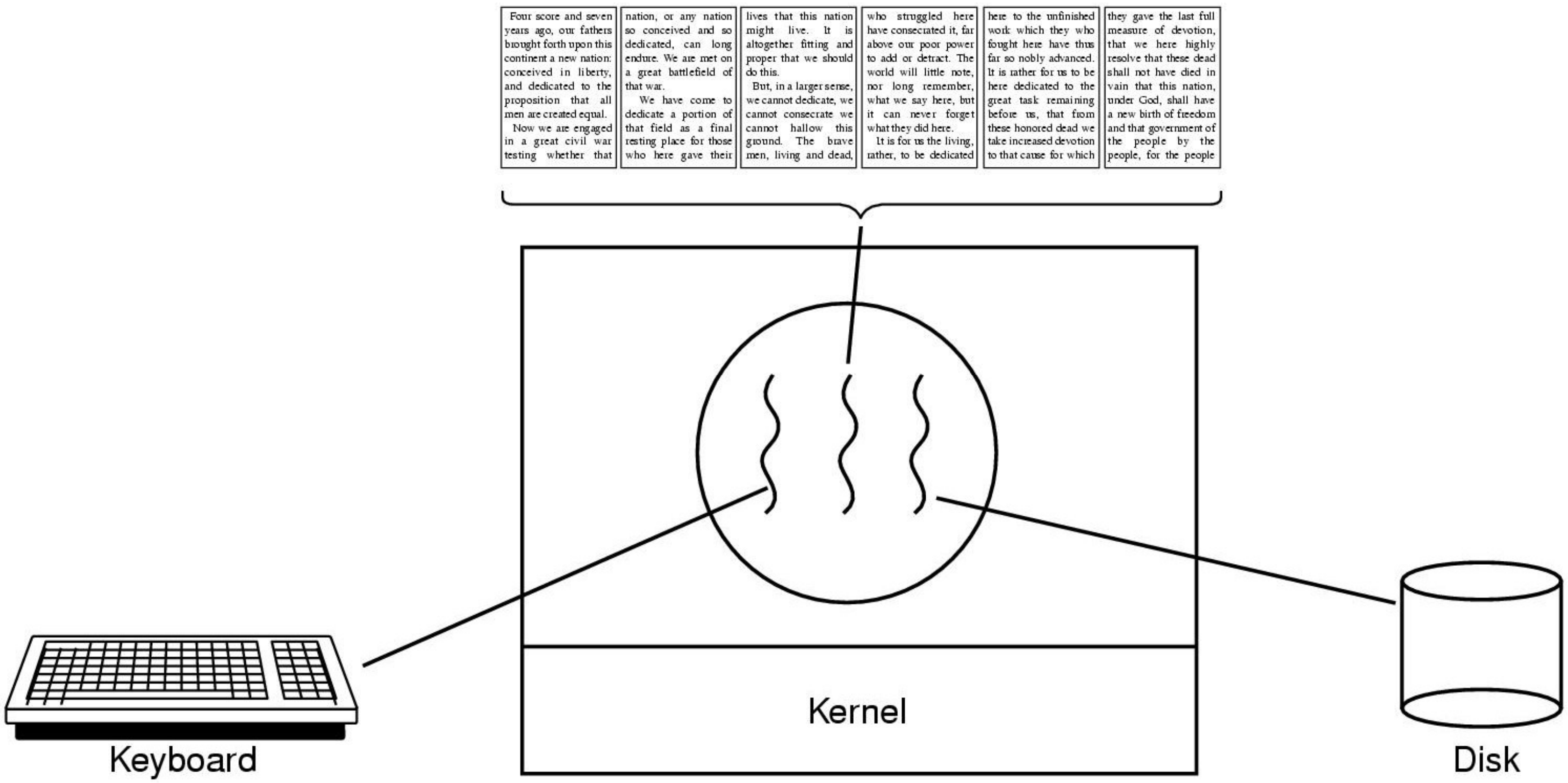


2. Background Processing

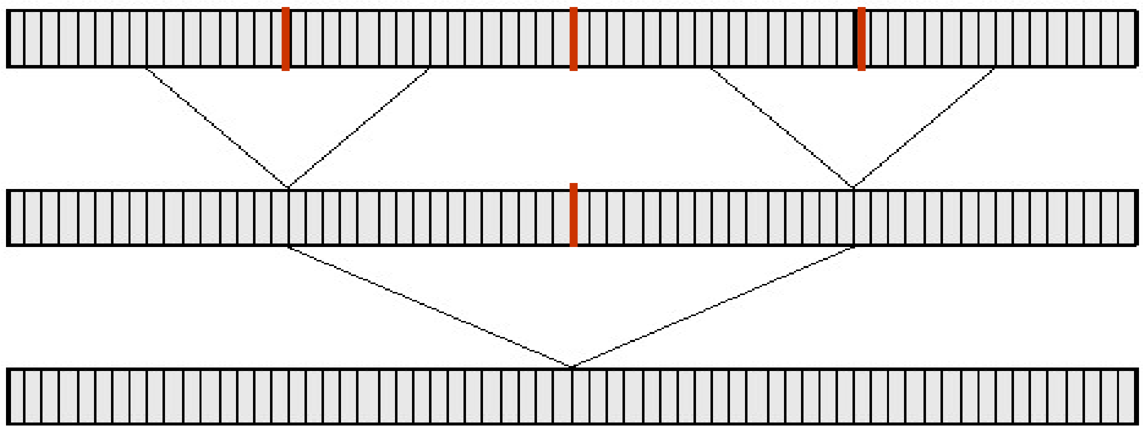
- Consider writing a GUI-based application that uses:
  - Mouse
  - Keyboard input
  - Handles various clock-based events

- In a single threaded application, if the application is busy with one activity, it cannot respond (quickly enough) to other events, such as mouse or keyboard input.
- Handling such concurrency is difficult and complex
- But simple in a multithreaded process

e.g. A word processor with 3 threads



3. Parallel Algorithms e.g. Merge Sort



- Sort some data items on a shared-memory parallel computer.
- Our task is merely to implement a multithreaded sorting algorithm.
  - Divide the data into four pieces
  - Have each processor sort a different piece.
  - Two processors each merge two pieces
  - One processor merges the final two combined pieces.

Threads

- User Threads
- Kernel Threads

User Threads

- All thread management is done by the application. A user thread maintains all its state in user space.
- Thread switching does not require kernel mode privileges (no mode switch)
- Scheduling is application specific
- **The kernel is not aware of the existence of user threads**
- Examples
  - POSIX *Pthreads*
  - Mach *C-threads*
  - Solaris *threads*

User thread (cont..)

- Threads library contains code for:
  - creating and destroying threads
  - passing messages and data between threads
  - scheduling thread execution
  - saving and restoring thread contexts

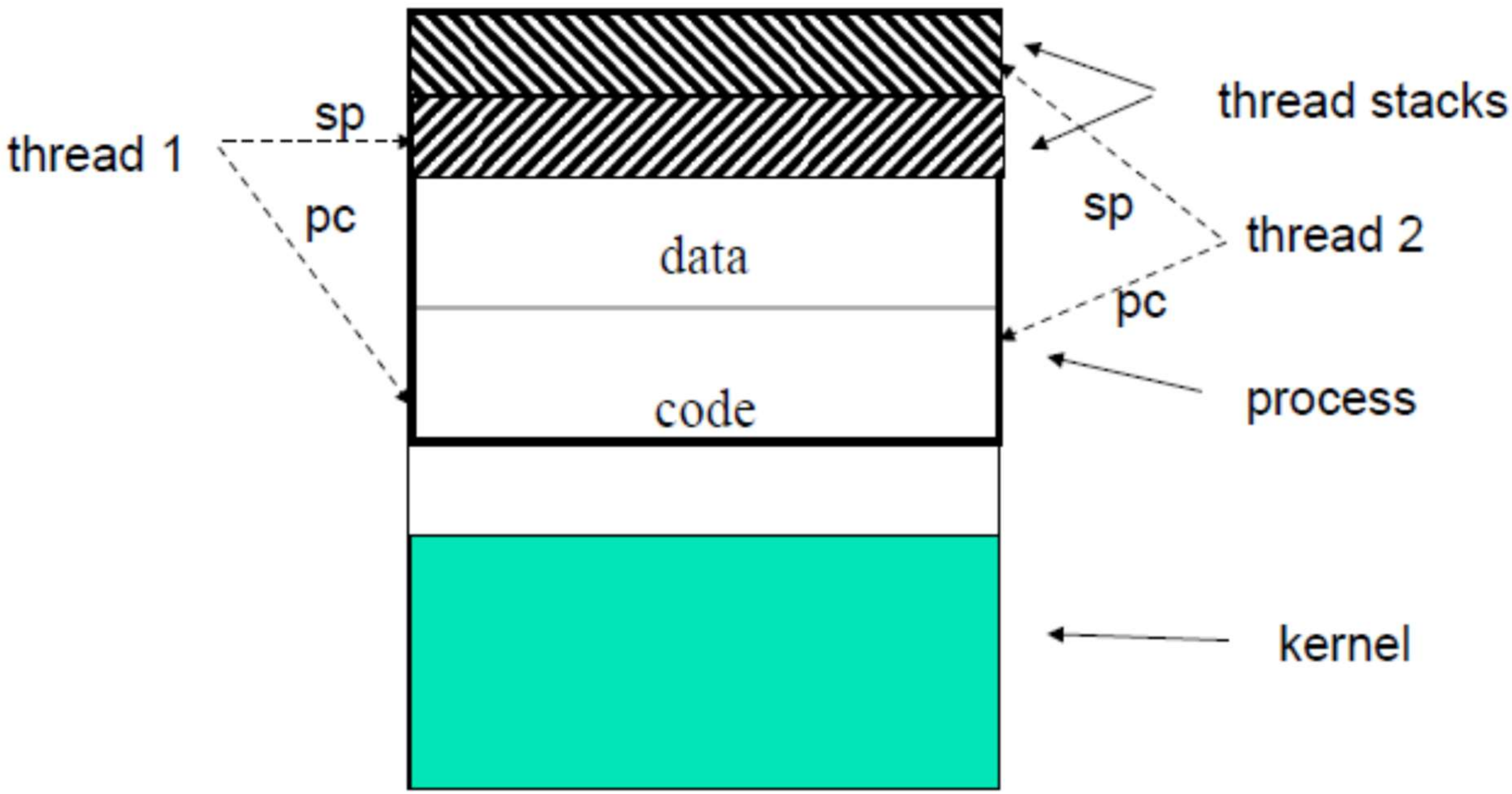
Kernel Threads

- Kernel threads are supported directly by the operating system
- The kernel performs creation, scheduling, and management
- Kernel threads are generally slower to create and manage than user threads
- However since the kernel manages the threads, if a thread performs a system call, the kernel can schedule another thread
- OS Kernel maintains context information for the process and the threads (LWP)
- Scheduling is done on a thread basis
- Examples
  - Windows NT/XP/Vista/7, Solaris, Tru64 UNIX, Linux

Thread Implementation

- User-level threads
  - Implemented as a thread library, which contains the code for thread creation, termination, scheduling, and switching
  - Kernel sees one process and it is unaware of its thread activity
  - can be preemptive or not (co-routines)
- Kernel-level threads
  - Thread management done by the kernel

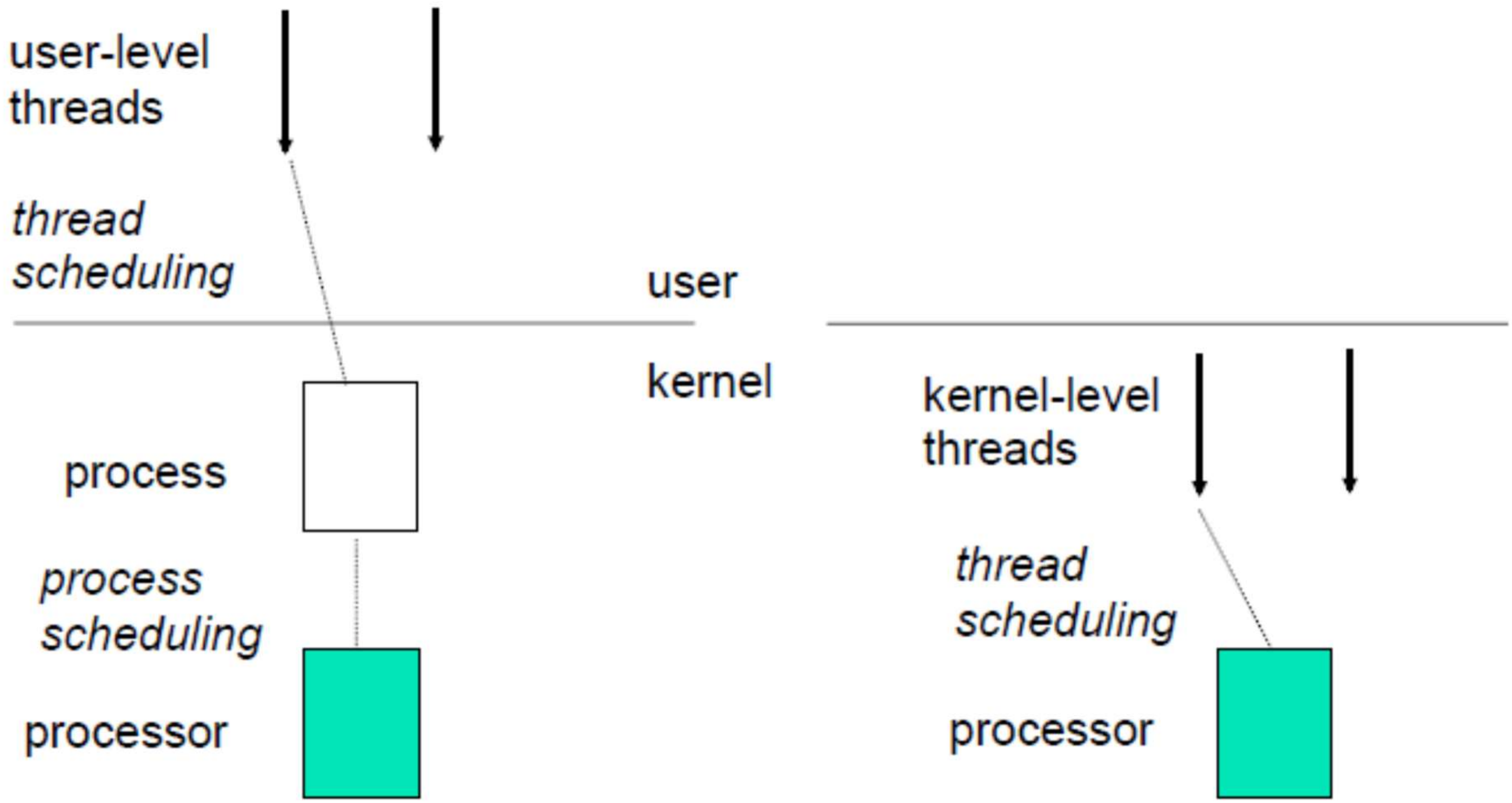
User-Level Thread Implementation



User-Level vs. Kernel-Level Threads

- Advantages of the user-level threads
  - Performance: low-cost thread operations (do not require crossing protection domains), fast context switching
  - Flexibility: scheduling can be application specific
  - Portability: user-level thread library easy to port
- Disadvantages of the user-level threads
  - If a user-level thread is blocked in the kernel, the entire process (all threads of that process) are blocked
  - cannot take advantage of multiprocessing (the kernel assigns one process to only one processor)

User-Level vs. Kernel-Level Threads

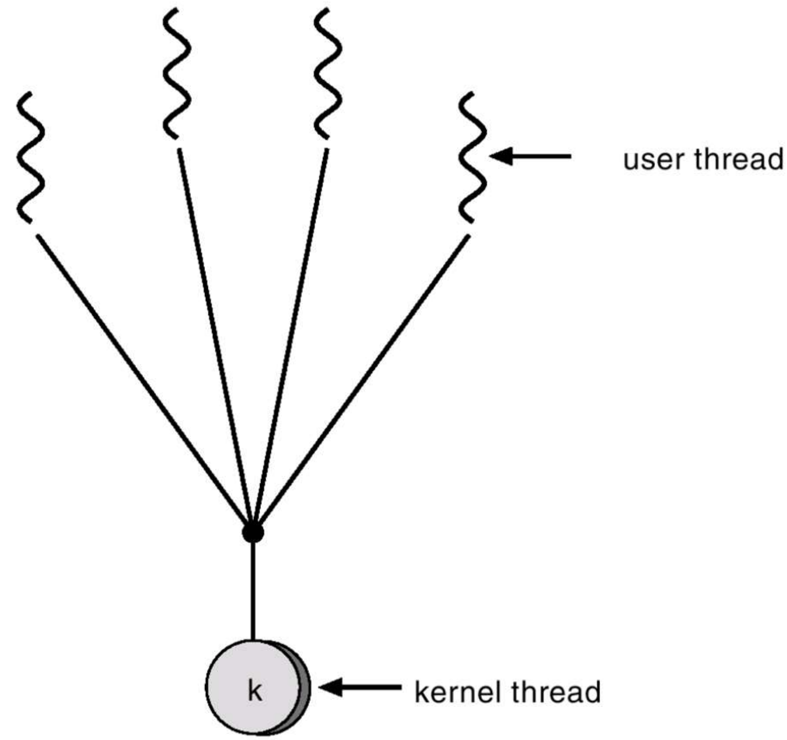


Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

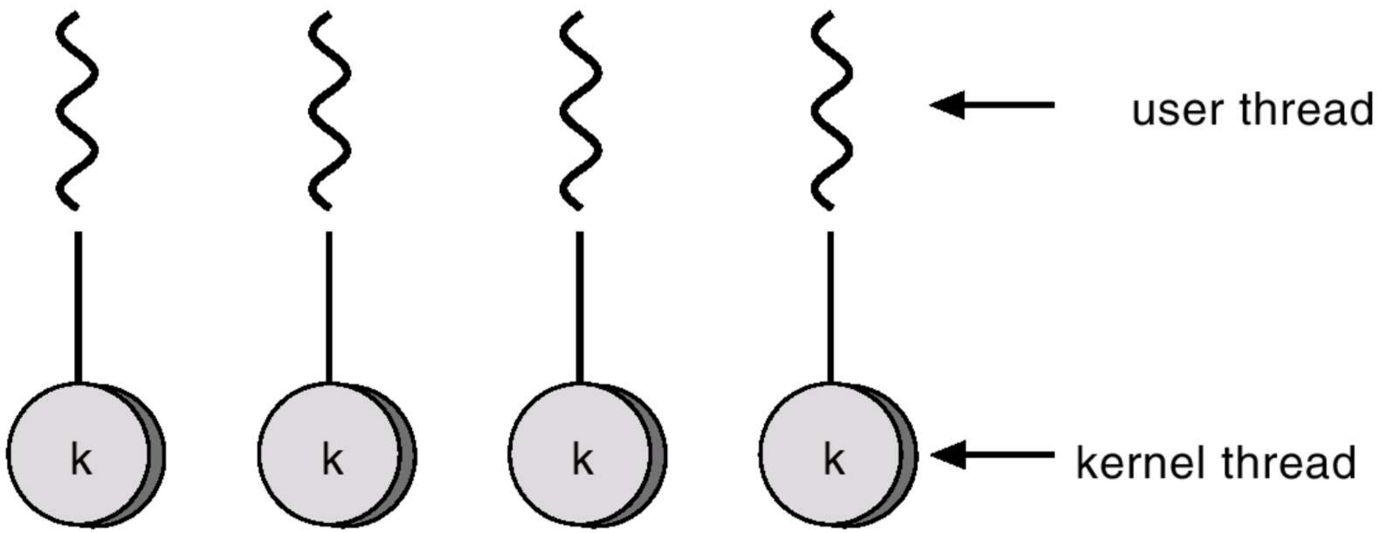
Many-to-One Model

- Many user-level threads mapped to single kernel thread
- Used on systems that do not support multiple kernel threads
- E.g., Solaris green threads; GNU Portable Threads



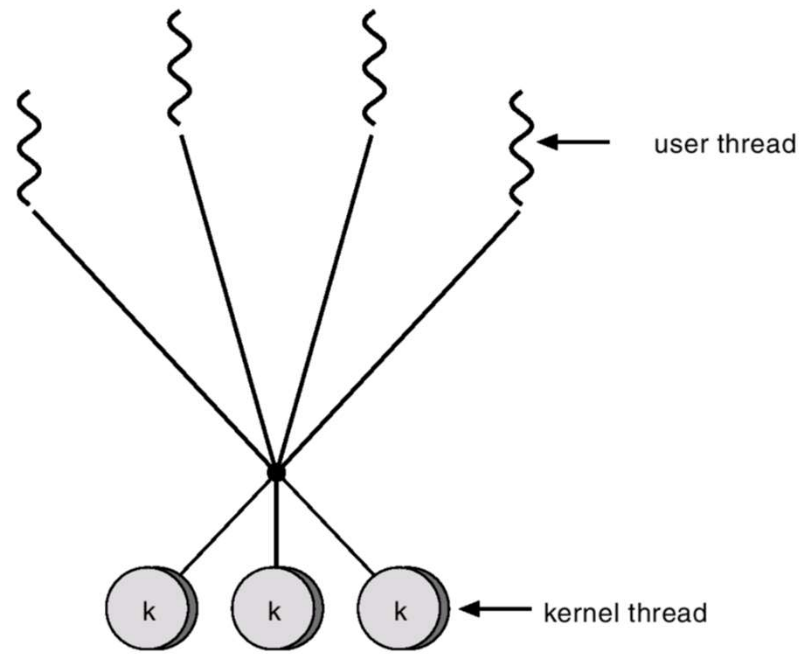
One-to-One Model

- Each user-level thread maps to kernel thread.
- Examples
  - Windows NT/XP/Vista/7
  - Linux, Solaris 9



Many-to-Many Model

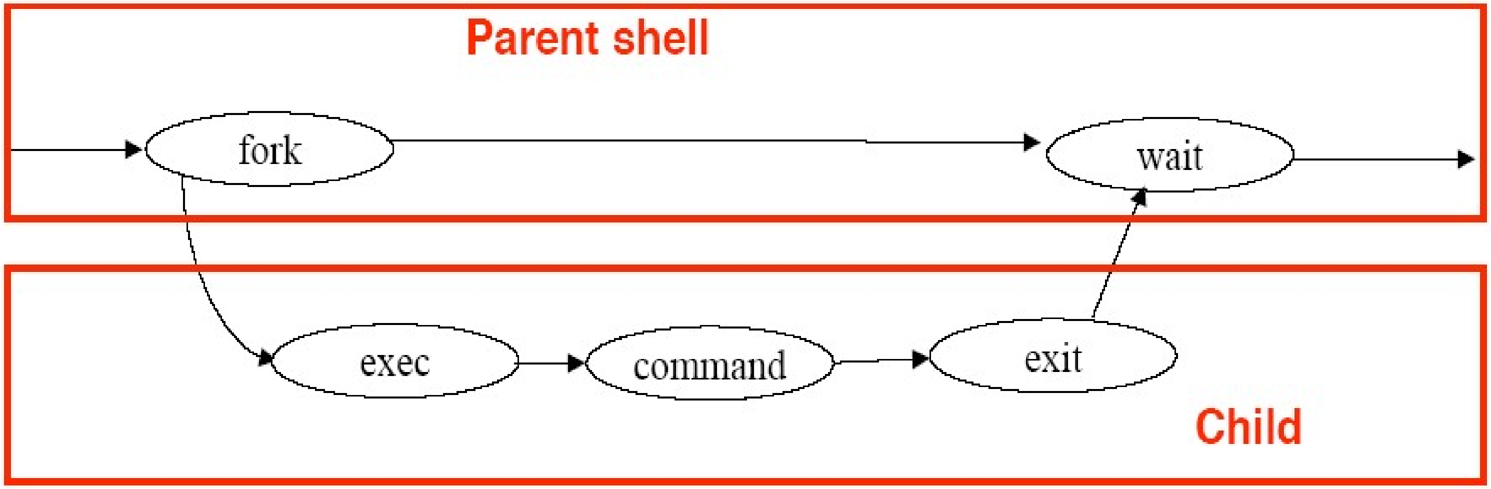
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris 2
- HP-UX, Tru64 UNIX



Multithreaded Programming Issues – fork and exec system calls

- When a thread (associated with process A) calls fork, two things can happen:
  - The new process duplicates all threads associated with process A
  - The new process will be single-threaded
- Some Unix operating systems support these two versions of fork
- Typically, the exec system call is used after a fork system call
- The exec system call:
  - Loads a binary file into memory
  - Destroys the memory image containing the exec system call
  - Starts its execution

How the unix shell runs commands



- when you type a command, the shell forks a clone of itself
- the child process makes an exec call, which causes it to stop executing the shell and start executing your command
- the parent process, still running the shell, waits for the child to terminate

Multithreaded Programming Issues – Cancellation

- Thread cancellation is the task of terminating a thread before it is completed
  - For example: assume that multiple threads are searching a database. As soon as one thread returns the search result, we can terminate the remaining threads
- A thread to be cancelled is referred to as a target thread
- Thread cancellation can happen in two ways:
  - **Asynchronous cancellation:** One thread immediately terminates the target thread
  - **Deferred cancellation:** the target thread can periodically checks if it should terminate. This allows the cancellation to happen in an orderly manner

Multithreaded Programming Issues – Cancellation

- Thread cancellation is not as easy as it appears
  - What about resources allocated to a thread
    - A thread might be cancelled while in the middle of updating a shared variable
    - This becomes especially troublesome with asynchronous cancellation
  - An OS usually reclaim all system resources from a cancelled thread. But often does not reclaim all resources. Why?
- Deferred cancellation happens when a thread can be safely cancelled. This is referred to as **cancellation points**
- The Pthreads API provides **cancellation points**

Multithreaded Programming Issues – Signal Handling

- A signal is used to notify a process that a particular event has occurred
  - Examples of signals are illegal memory access, division by zero, etc
- A signal might occur synchronously and asynchronously
- All signals follow the same pattern
  - A signal is generated by the occurrence of a particular event
  - A generated signal is delivered to a process
  - Once delivered, the signal must be dealt with
- **Synchronous signals** are generated by events internal to a running process
- **Asynchronous signals** are generated by events external to a running process

Multithreaded Programming Issues – Signal Handling

- Synchronous signals:
  - Generated by events internal to the running process
  - Examples include illegal memory access, division by zero, etc
  - Synchronous signals are delivered to the same process that performed the operation causing the signal
- Asynchronous signals:
  - Generated by events external to the running process
  - Examples include terminating a process by specific keystrokes (e.g., control c)
  - Asynchronous signals are more complicated

Multithreaded Programming Issues – Signal Handling

- Every signal, whether synchronous or asynchronous, is handled in two ways
  - A default signal handler
  - A user-defined signal handler
- By default, every signal has a **default signal handler** that is run by the kernel
- This default signal handler can be overwritten by the user-defined signal handler
- **Single-threaded programs:** straightforward, signals are always delivered to the process
- **Multithreaded programs:** more complicated

Multithreaded Programming Issues – Signal Handling

- When a signal is delivered to a multithreaded program, the following can happen:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process
- Examples:
  - A terminating signal should be sent to all thread in the process
  - Solaris 2 implements the fourth option (i.e., creates a special thread within each process solely for signal handling)

Multithreaded Programming Issues

- Thread Pools
  - Creating threads can be time consuming
  - Too many threads can bog down the system
  - Thread pools help with this problem
  - Threads are pre-allocated
  - The number of threads available at a given time is fixed
  - Some systems may adjust the thread pool size depending on usage

Multithreaded Programming Issues

- Thread Specific Data/ Thread-local Storage (TLS)
  - Threads belonging to the same process share the process data. This provided the benefit of multithreaded programming
  - However, in some instances, each thread might need its own specific data.
    - For example, a transaction processing multithreaded application might service each transaction in a separate thread
  - Most thread libraries such as Win32 and Pthreads provides support for thread specific data

Thread Implementation (POSIX)

- The most important of these APIs, in the Unix world, is the one developed by the group known as POSIX.
- POSIX is a standard API supported
- Portable across most UNIX platforms.
- PTHREAD library contains implementation of POSIX standard
- To link this library to your program use *–lpthread*
  - **`gcc MyThreads.c -o MyThreadExecutable - lpthread`**

POSIX Thread (Pthread)

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.
- API *specifies* behavior of the thread library, *implementation* is up to developer of the library.
- Common in UNIX operating systems.

Pthread API

- **thread creation and termination**
  - *pthread_create(&tid,NULL,start_fn,arg);*
  - *pthread_exit(status);*
- **thread join**
  - *pthread_join(tid, &status);*
- **mutual exclusion**
  - *pthread_mutex_lock(&lock);*
  - *pthread_mutex_unlock(&lock);*
- **condition variable**
  - *pthread_cond_wait(&c, &lock);*
  - *pthread_cond_signal(&c);*

Condition Variables (Example)

- thread 1

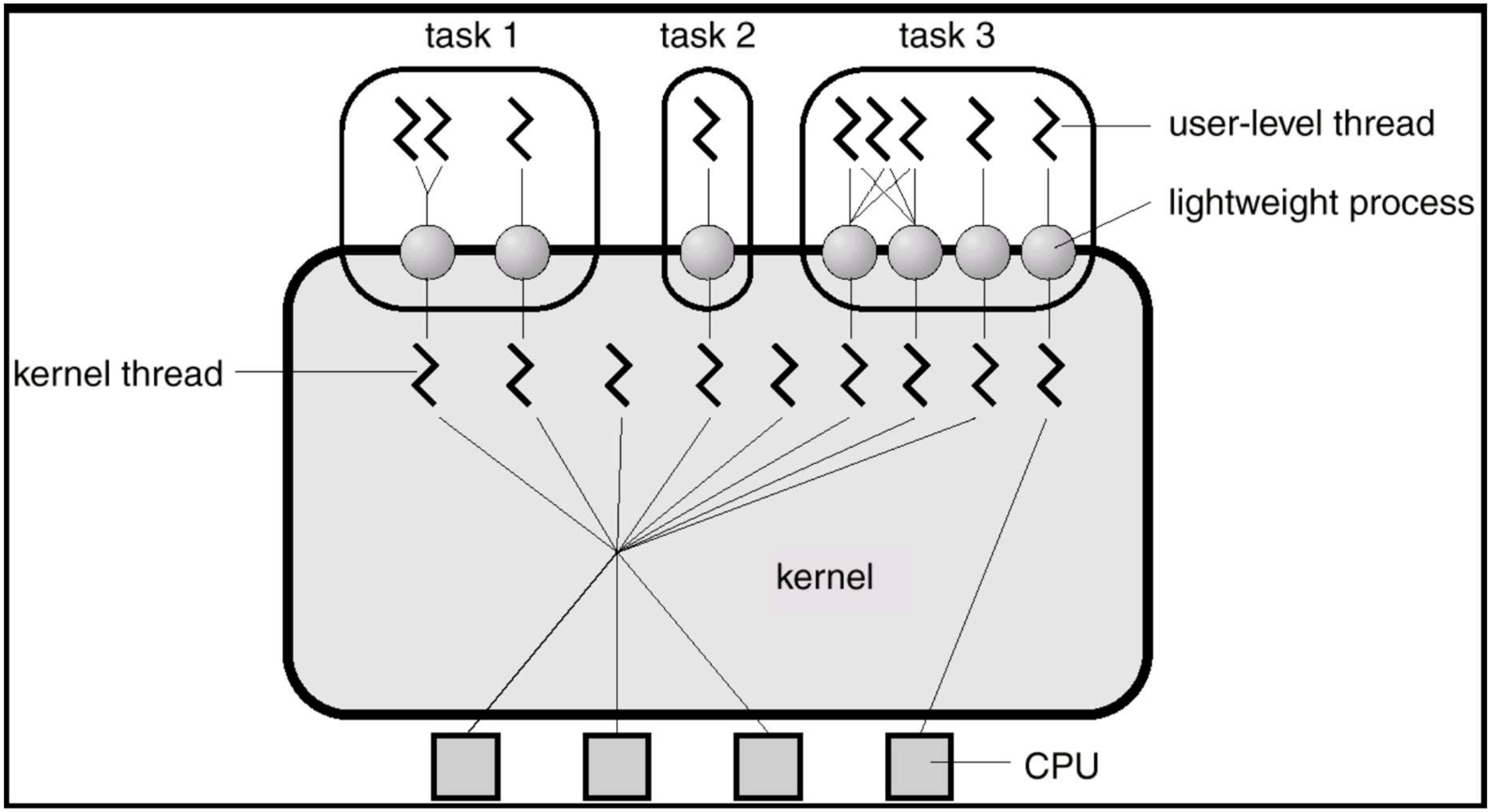
```
pthread_mutex_lock(&lock);
while (!my-condition)
        pthread_cond_wait(&c, &lock);
do_critical_section();
pthread_mutex_unlock(&lock);
```

- thread 2

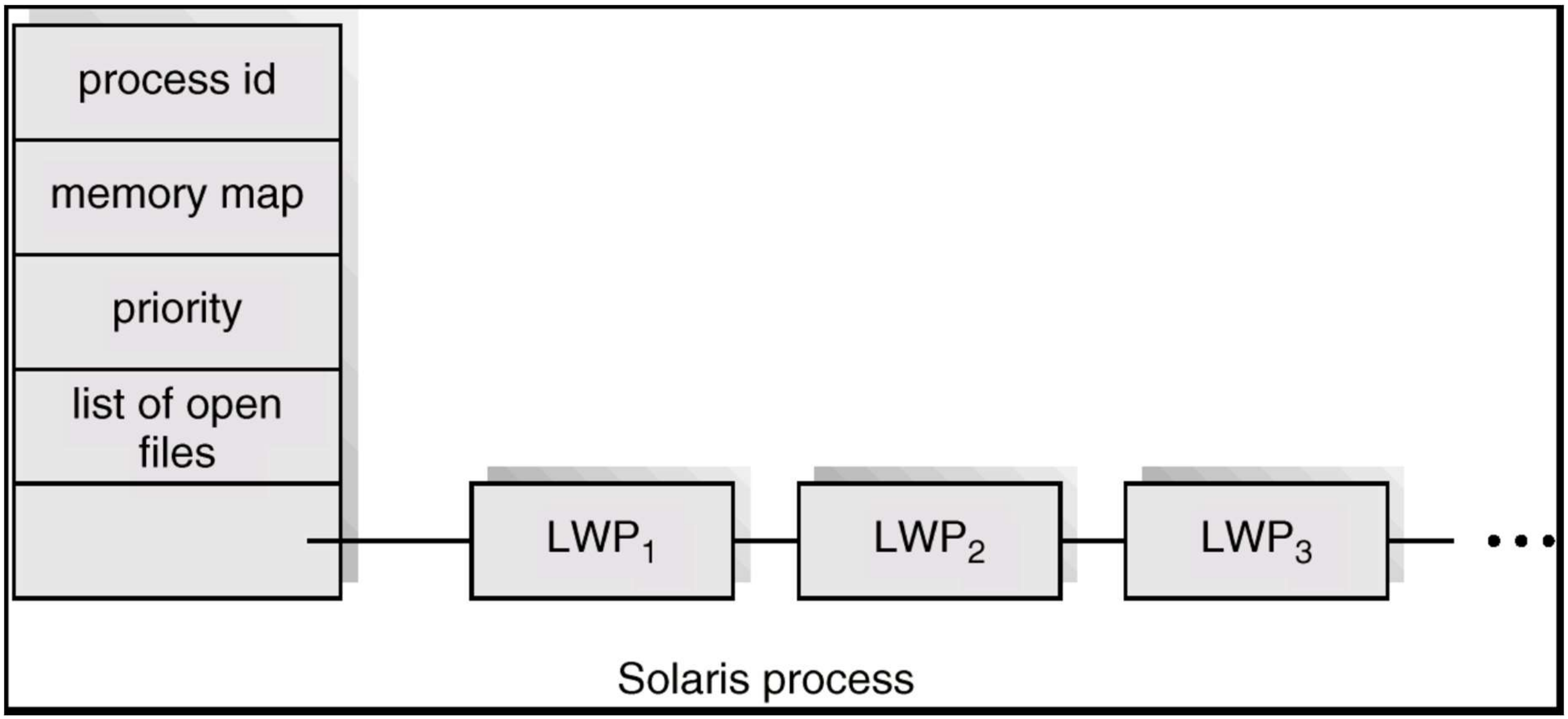
```
pthread_mutex_lock(&lock);
my-condition = true;
pthread_mutex_unlock(&lock);
pthread_cond_signal(&c);
```

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
void * PrintHello(void *threadid)
{
        printf("\n%d: Hello World!\n", threadid);
        pthread_exit(NULL);
}
int main()
{
        pthread_t threads[NUM_THREADS]; /* the thread identifier */
        int rc, t;
        for (t = 0; t < NUM_THREADS; t++)
        {
                printf("Creating thread %d\n", t);
                rc = pthread_create(&threads[t], NULL, PrintHello, (void *)&t);
                if (rc)
                {
                        printf("ERROR; return code from pthread_create() is %d\n", rc);
                        exit(-1);
                }
        }
        pthread_exit(NULL);
}
```

Solaris 2 Threads



Solaris Process



Linux Threads

- Linux refers to them as tasks rather than threads
- Thread creation is done through clone() system call
- Clone() allows a child task to share the address space of the parent task (process)

Windows NT Threads

- Implements the one-to-one mapping.
- Each thread contains
  - a thread ID
  - register set
  - separate user and kernel stacks
  - private data storage area

References

- Chapter 2, Modern Operating Systems, Tanenbaum
- Chapter 4, Silberschartz Operating System Concept 9th Edition.
- http://www.thegeekstuff.com/2012/04/create-threads-in-linux/
- http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
- http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_create.html