Chapter 4: Threads

Chapter 4: Threads

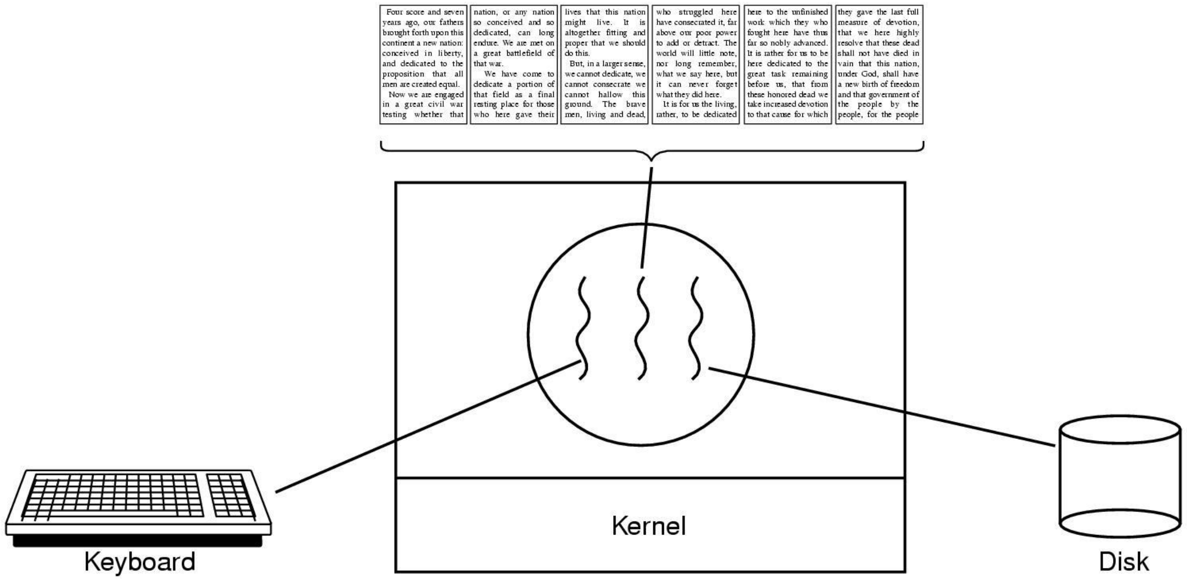
- **Overview**
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Threading Issues

Objectives

- To introduce the notion of a ***thread***—a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the ***Pthreads***, Windows, and Java thread libraries
- To examine issues related to multithreaded programming

Motivation

- Most modern applications and computers are multithreaded
- Threads run within application
- Multiple tasks with the application (e.g. word processor) can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request



Four score and seven years ago, our fathers brought forth upon this continent a new nation: conceived in liberty, and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war.

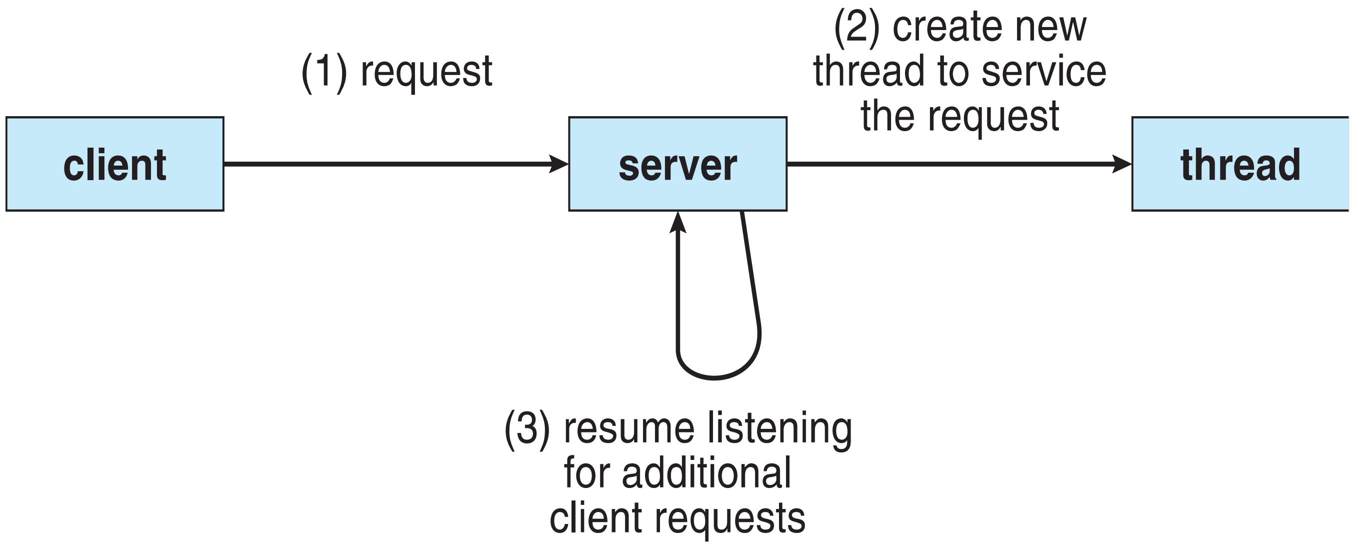
We have come to dedicate a portion of that field as a final resting place for those who here gave their lives that this nation might live. It is altogether fitting and proper that we should do this.

But, in a larger sense, we cannot dedicate, we cannot consecrate we cannot hallow this ground. The brave men, living and dead, who struggled here have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember, what we say here, but it can never forget what they did here.

It is for us the living, rather, to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us, that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion, that we here highly resolve that these dead shall not have died in vain that this nation, under God, shall have a new birth of freedom and that government of the people by the people, for the people

Motivation

- Process creation is ***heavy-weight*** while thread creation is ***light-weight***
  - For applications performing multiple similar tasks
    - It is costly to create a process for each task
  - Ex: web server serving multiple clients requesting the same service
    - Create a separate thread for each service
- Multithreaded programs: Can simplify code, increase efficiency

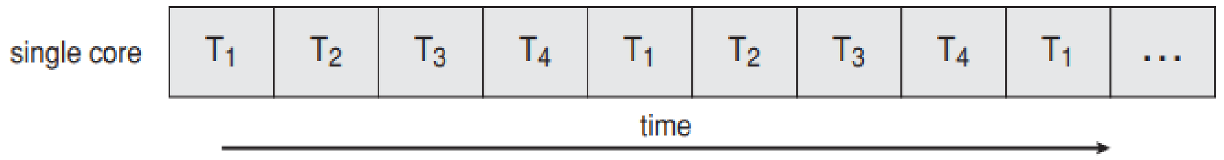

client
(1) request
server
(2) create new thread to service the request
thread
(3) resume listening for additional client requests

What is a Thread?

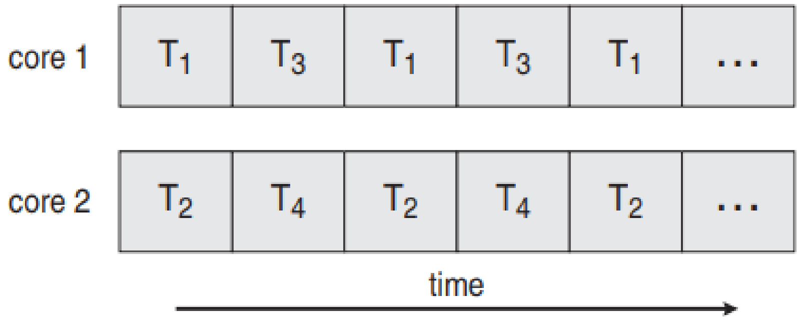
- A *thread*, also known as *lightweight process* (LWP), is a basic unit of CPU execution, and is created by a process.
- A thread has a ***thread ID***, a ***program counter***, a ***register set***, and a ***stack***. Thus, it is similar to a process.
- However, a thread *shares* with other threads in the *same* process its **code section**, **data section**, and other **OS resources** (e.g., files and signals).
- A process, or *heavyweight* process, has a *single* thread of control.

Thread: Concurrency vs. Parallelism

- Threads are about concurrency and parallelism
- ***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
- Concurrent execution on single-core system:



- Parallelism on a multi-core system:



Benefits of threads

- **Responsiveness –** Other part (i.e., threads) of a program may still be running even if one part (e.g., a thread) is blocked-- especially important for user interfaces. Users need not wait
- **Resource Sharing –** Threads of a process, by default, share many system resources (e.g., files and memory). Several threads run in the same address space.
- **Economy –** Creating and terminating processes, allocating memory and resources, and context switching processes are very time consuming. Threads share the resources of the process they belong to, therefore, it is more economical to create and context switch threads.
- **Scalability (Utilization of multiprocessor architecture)–** process can take advantage of multiprocessor architectures. Multiple CPUs may run multiple threads of the same process. No program change is necessary.

Chapter 4: Threads

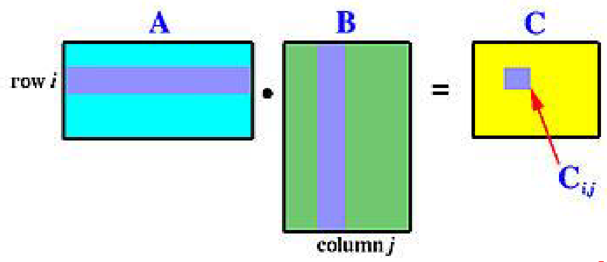
- Overview-Motivation
- Benefits of threads
- **Multithreading Programming Challenges**
- Multithreading Models
- Thread Libraries

Programming Challenges (1/4)

- With a single-core CPU, threads are scheduled by a scheduler and can only run one at a time.
- With a multicore CPU, multiple threads may run at the same time, one on each core.
- Therefore, system design becomes more complex than one may expect.
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**

Programming Challenges (2/4)

- ***Dividing Activities***:
  - Since each thread can run on a core, one must study the problem in hand so that program activities can be divided and run concurrently.
- Matrix multiplication is a good example.



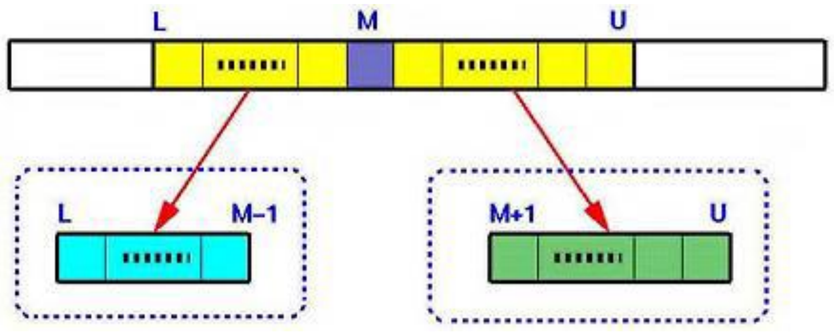
$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \times B_{k,j}$$

We may create a thread for each $C_{i,j}$

- ***Balance:***
  - Make sure that each thread has equal contribution, if possible, to the whole computation.
  - If an insignificant thread runs frequently, occupying a core, other more useful threads would have less chance to run.

Programming Challenges (3/4)

- ***Data Splitting*:**
  - Data may also be split into different sections so that each of which can be processed separately.
- Quicksort is an example.
  - After partitioning, the two sections can be sorted separately.



After partitioning a[L..U] into a[L..M-1] and a[M+1..U], we may create two threads, one for each section.
Then, each thread sorts its own section.

Programming Challenges (4/4)

- ***Data Dependency***:
  - Watch for data items that are used by different threads. For example, two threads may update a common variable at the same time.
- Should this happen, unexpected results may occur. As a result, the execution of threads has to be ***synchronized*** so that only one thread can update a shared variable at any time.

Programming Challenges (5/5)

- ***Testing and Debugging***:
  - The behavior of a threaded program is *dynamic*. A bug that appears in this test run may not occur in the next. Some bugs may never surface throughout the life-span of a threaded program, or may appear at an unexpected time.
  - Some debugging issues (e.g., race condition – updating a shared resource at the same time, and system deadlock) do not have efficient solutions.
- Thus, testing and debugging is an art, and requires a careful design and planning.

Types of Parallelism

- Types of parallelism (and concurrencies)
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
    - Ex: summing elements of a length-*N* array
    - Divide array into multiple subarrays, each thread perform summation on the assigned subarray.
  - **Task parallelism** – distributing tasks (threads) across cores, each thread performing unique operation. Different threads may use same data or distinct data
    - Ex: one thread performs summation and another thread perform multiplication
- **Hybrid Data-and-Task parallelism** in practice –
  - Ex: sorting and summing a length-N array

What’s needed?

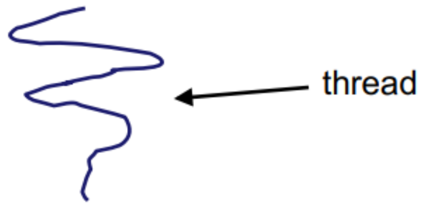
- **In many cases**
  - Everybody wants to run the same code
  - Everybody wants to access the same data
  - Everybody has the same privileges
  - Everybody uses the same resources (open files, network connections, etc.)
- **But you’d like to have multiple hardware execution states:**
  - an execution stack and stack pointer (SP)
    - traces state of procedure calls made
  - the program counter (PC), indicating the next instruction
  - a set of general-purpose processor registers and their values

How could we achieve this?

- **Given the process abstraction as we know it:**
  - fork several processes
  - cause each to map to the same physical memory to share data
    - see the **shmget()** system call for one way to do this
- **This is really inefficient**
  - **space:** PCB, page tables, etc.
  - **time:** creating OS structures, fork/copy address space, etc.

Can we do better?

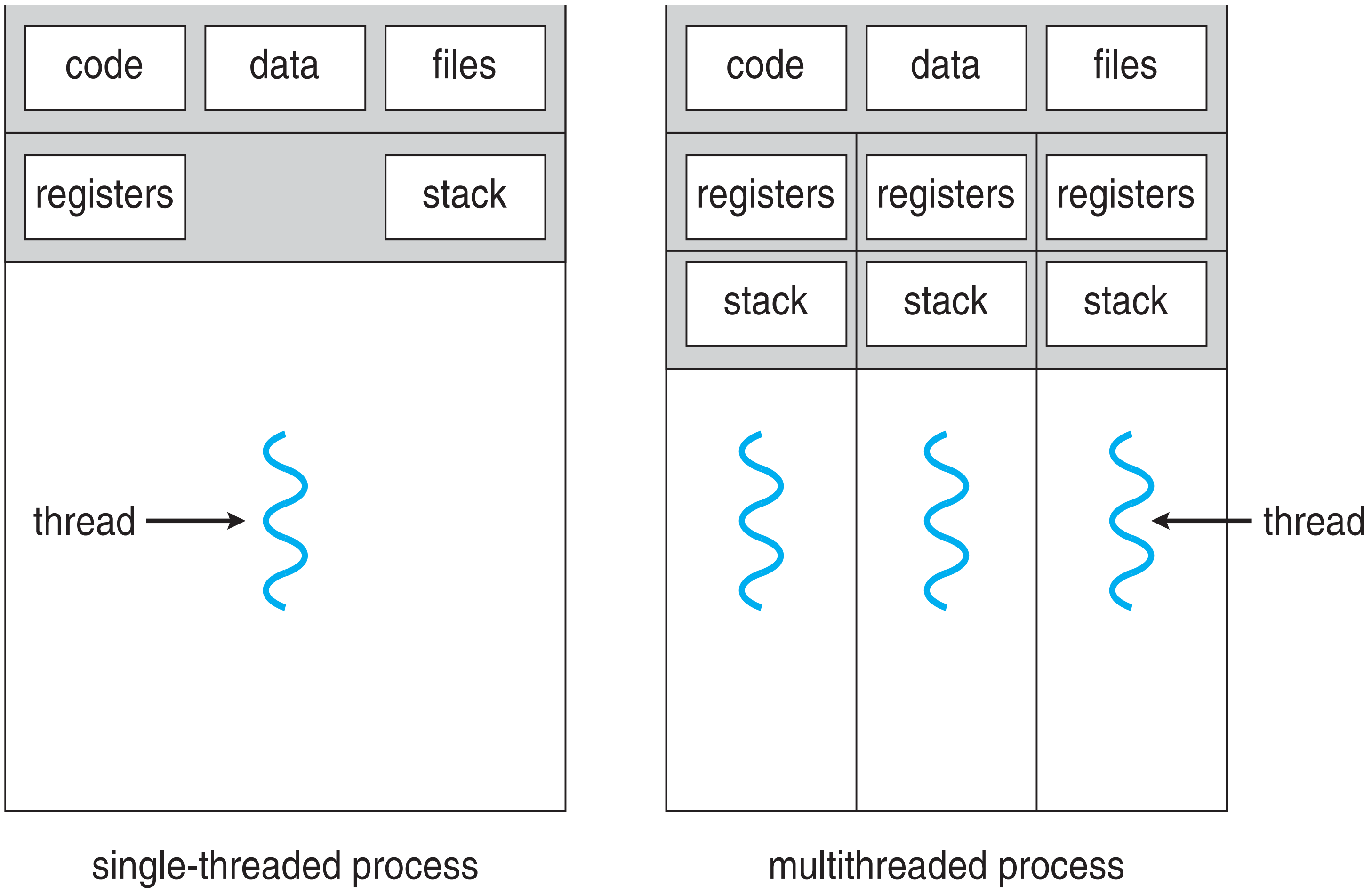
- Key idea:
  - separate the concept of a process (address space, OS resources)
  - … from that of a minimal "***thread of control***" (**execution state:** stack, stack pointer, program counter, registers)
- This execution state is usually called a ***thread***, or sometimes, a lightweight process.



Threads and processes

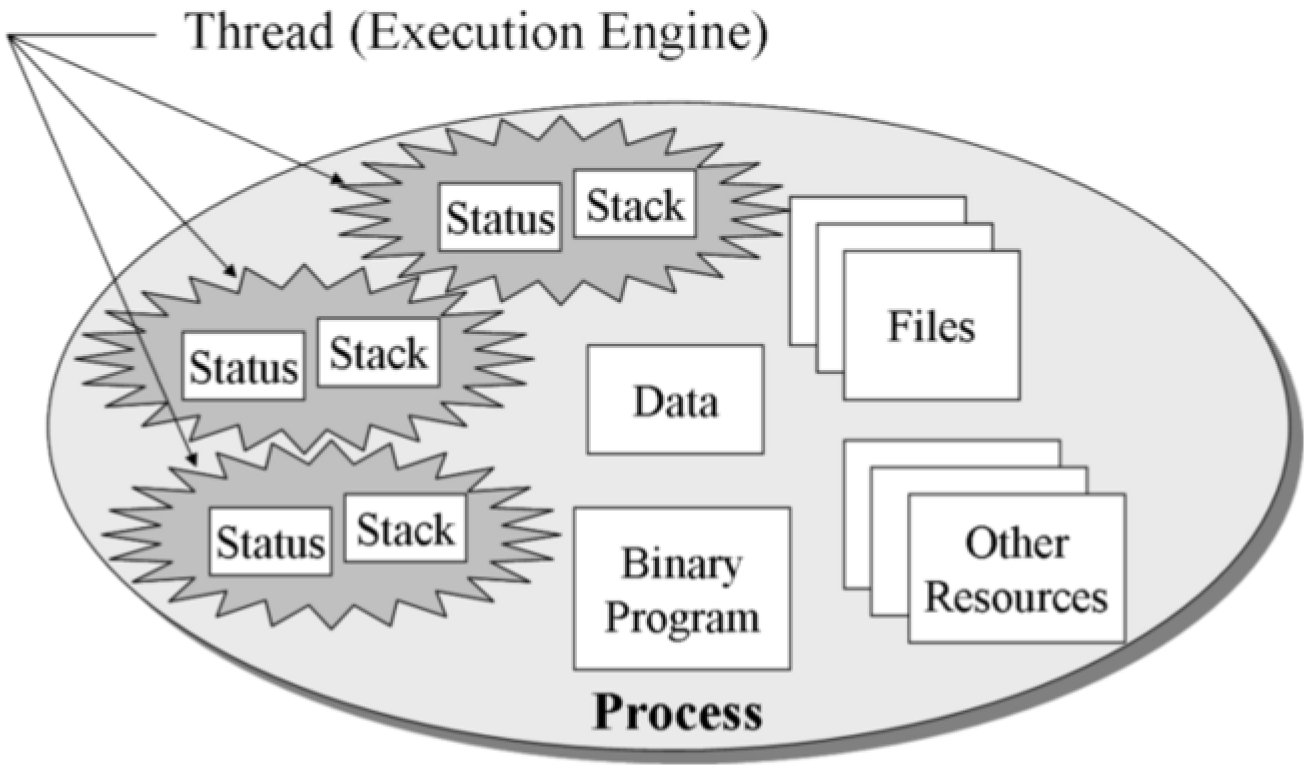
- Most modern OS's (Mac OS, Windows, UNIX) support two entities:
  - **the process**, which defines the address space and general process attributes (such as open files, etc.)
  - **the thread**, which defines a sequential execution stream within a process
- A thread is bound to a single process / address space
  - address spaces, however, can have multiple threads executing within them
  - sharing data between threads is cheap: all see the same address space
  - creating threads is cheap too!
- Threads become the unit of scheduling
  - processes / address spaces are just containers in which threads execute

Single and Multithreaded Processes



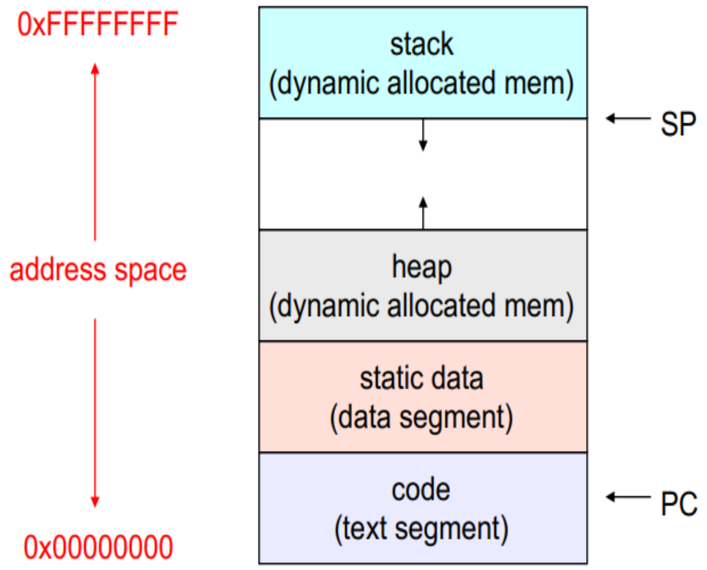
Process with multiple threads

- Per process items: Address space, global variables, open files, open files, child processes, pending alarms, signals and signal handlers, accounting information
- Per thread items: Program counter, Registers, Stack, state

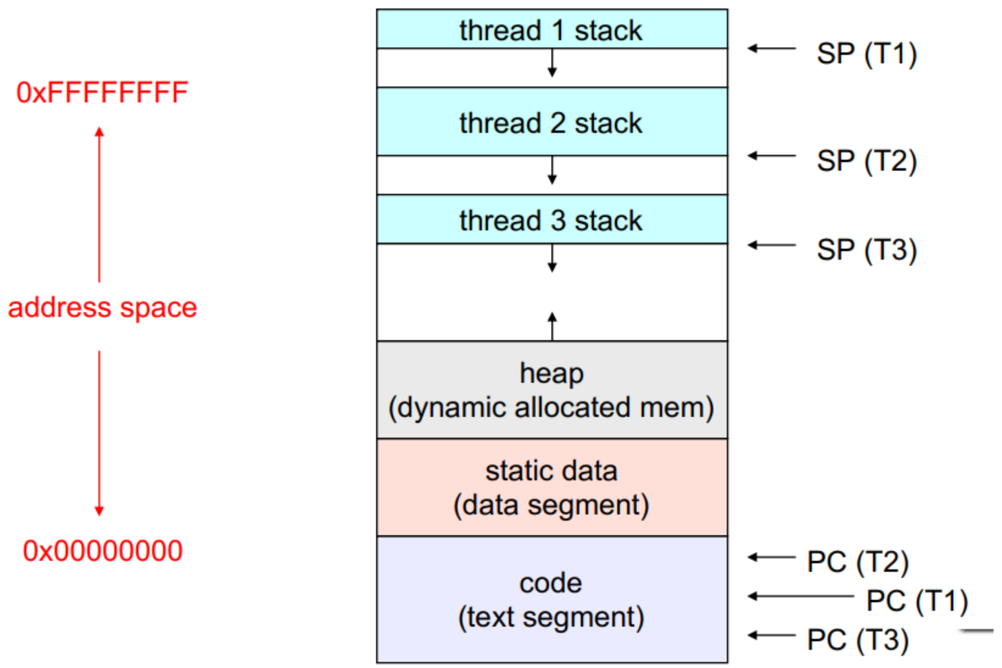


Address Spaces

Traditional process address space

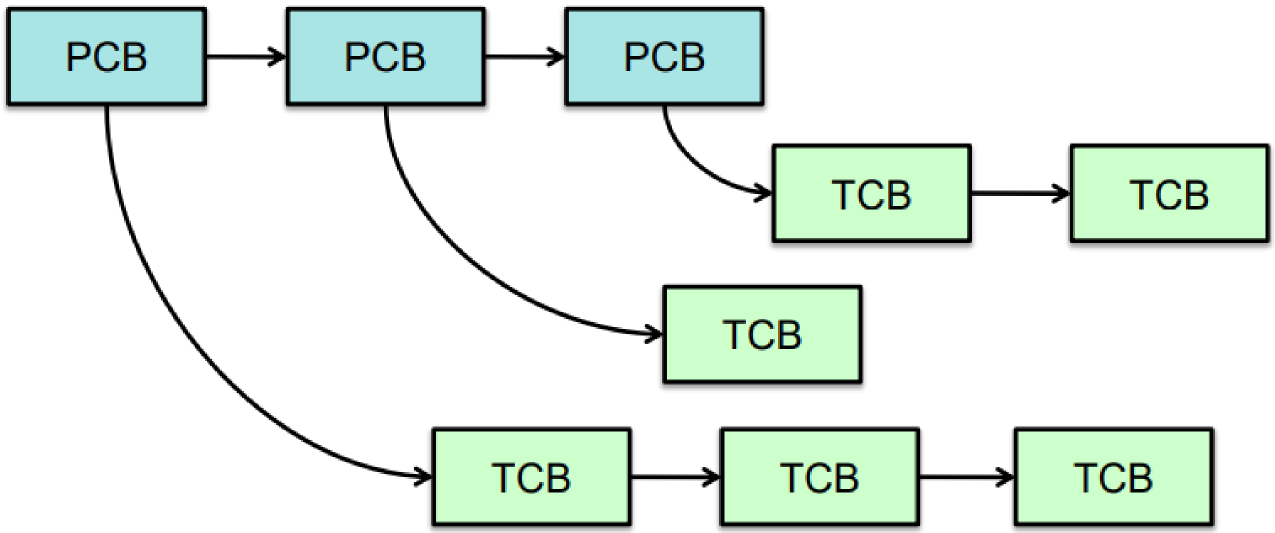


Address Space of Process with two threads



Implementation

- Process info (***Process Control Block***) contains one or more ***Thread Control Blocks*** (TCB):
  - Thread ID
  - Saved registers
  - Other per-thread info (e.g., scheduling parameters)



Amdahl's Law

- Identifies *performance gains* from adding additional cores to an application that has both serial and parallel components
- *S* is serial portion
- *N* processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

Chapter 4: Threads

- Overview
- Multicore Programming
- **User-level and Kernel-level Threads**
- Multithreading Models
- Thread Libraries
- Threading Issues

User Threads and Kernel Threads

- Support for threads either at user level or kernel level
- User threads - management done by user-level threads library
  - Supports thread programming: creating and managing program threads
- Three primary thread libraries (threads are managed without kernel support):
  - POSIX Pthreads
  - Windows threads
  - Java threads
- Kernel threads - Supported by the Kernel
  - Managed directly by the OS
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

User-level vs Kernel Level threads

User-level threads

All **code** and **data structures** for the library exist in user space.

Invoking a function in the API results in a **local function call** in user **space** and not a system call.

user mode

Kernel-level threads

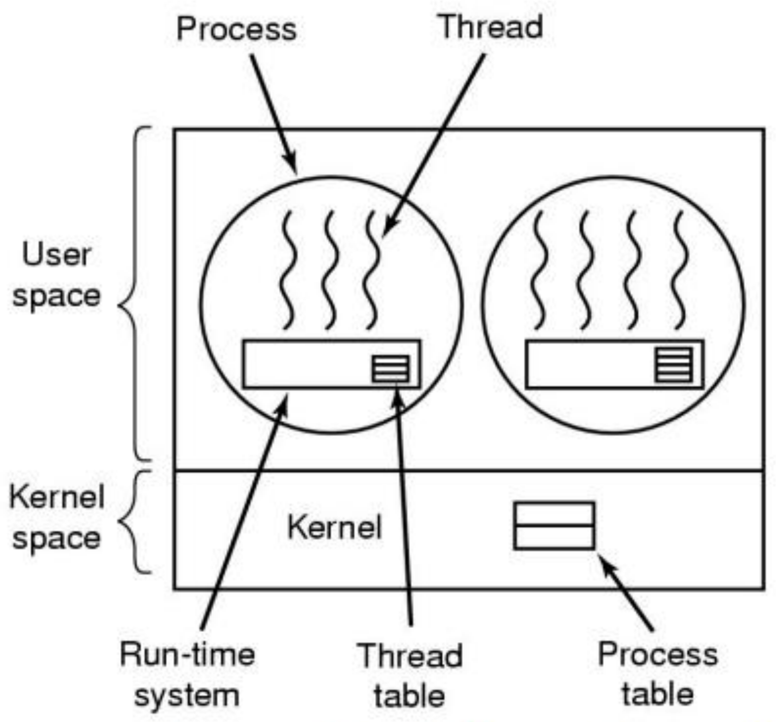
All **code** and **data structures** for the library exists in **kernel space**.

Invoking a function in the API typically results in a **system call** to the kernel.

kernel mode

Implementing threads in user space

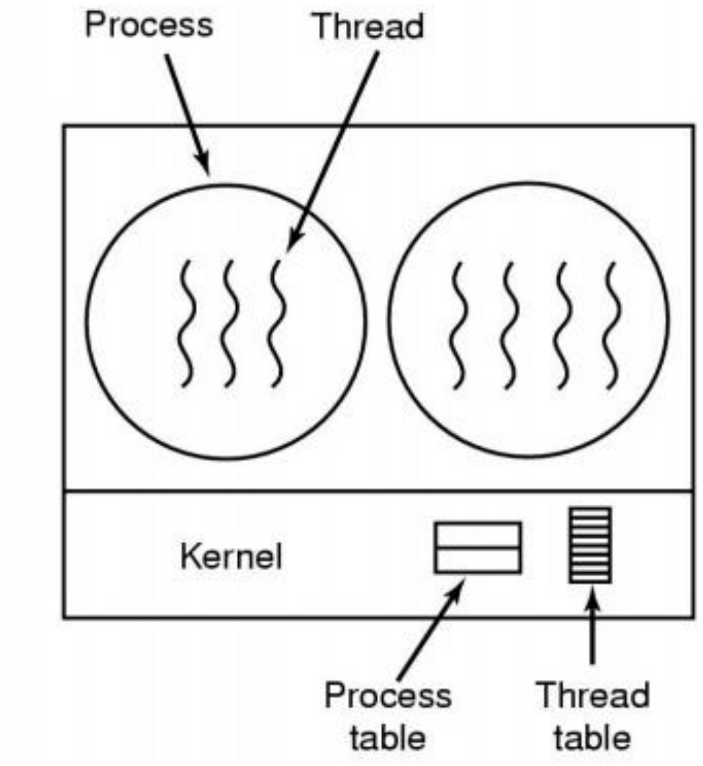
- Kernel knows nothing about threads.
  - Only thread library manages the threads.
- It knows only about the two processes in the system and holds its information in the Process table.



A user-level threads package

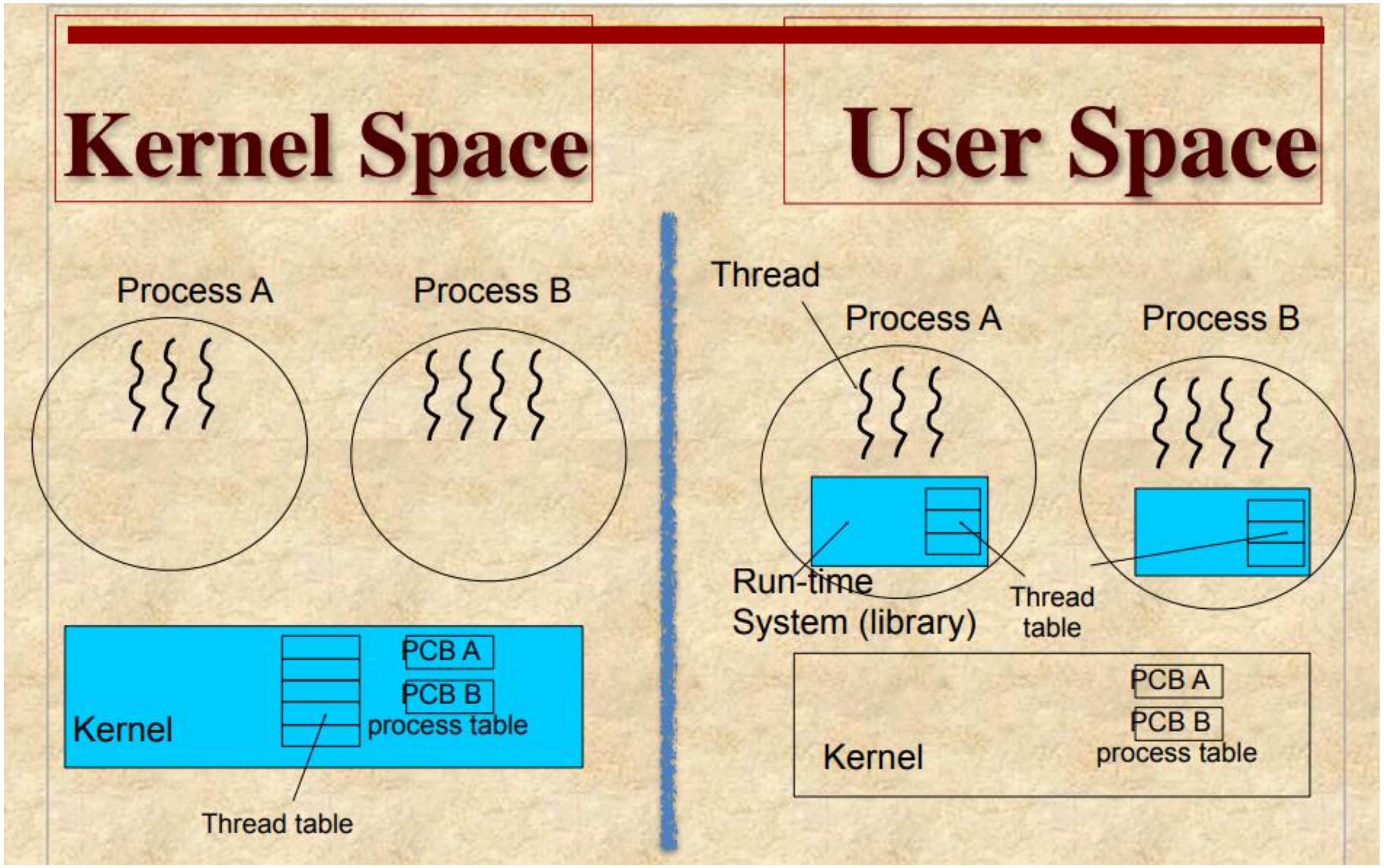
Implementing Threads in the Kernel

- Kernel knows about Processes as well as the threads.



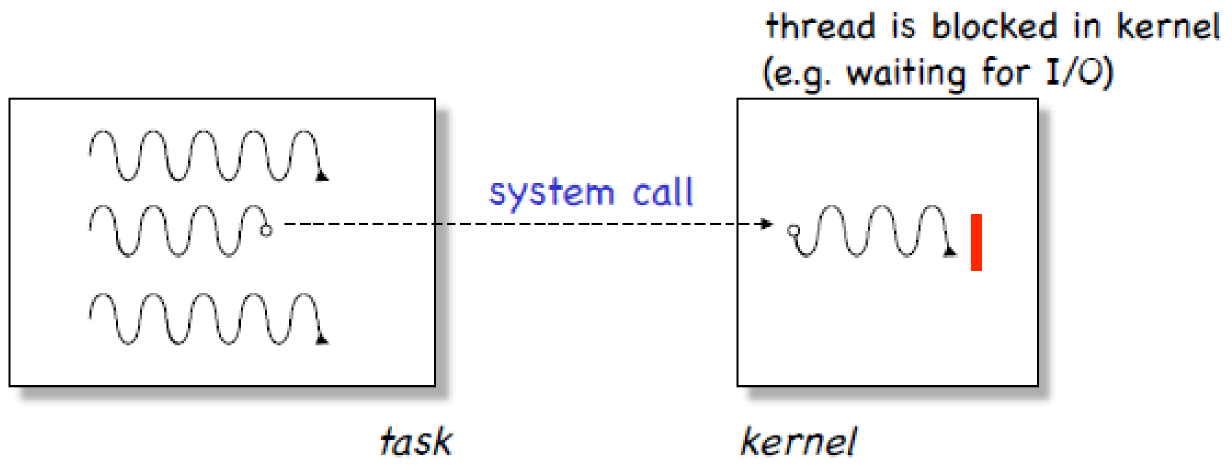
A threads package managed by the kernel

Threads at Kernel and User space



User Level Threads

- Advantages
  - cheap context switch costs!
  - User-programmable scheduling policy
  - User-level threads are OS independent
  - Thread switching does not require kernel mode privileges
  - Scheduling can be application specific
  - User Level Threads can run on any OS
- Disadvantages
  - How to deal with blocking system calls!



Kernel Level Threads

- Advantages
    - The kernel has full knowledge of all threads.
    - Scheduler may decide to give more CPU time to a process having a large number of threads.
    - Good for applications that frequently block.

- Disadvantages
    - Kernel manage and schedule all threads.
    - Significant overhead and increase in kernel complexity.
    - Kernel level threads are slow and inefficient compared to user level threads.
    - Thread operations are hundreds of times slower compared to user-level threads.