SWENG3043

Operating System

Software Engineering Department

AASTU

October 2018

Chapter Two

Processes and Threads

Chapter Objectives

- To introduce the notion of a process- a program in execution, which forms the basis of all computation.
- To describe the various features of processes, including creation, scheduling and termination.
- To introduce the notion of a thread- a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems.

Outline

- The Process Concept
- Process States
- Process Control
- Threads

The Process Concept

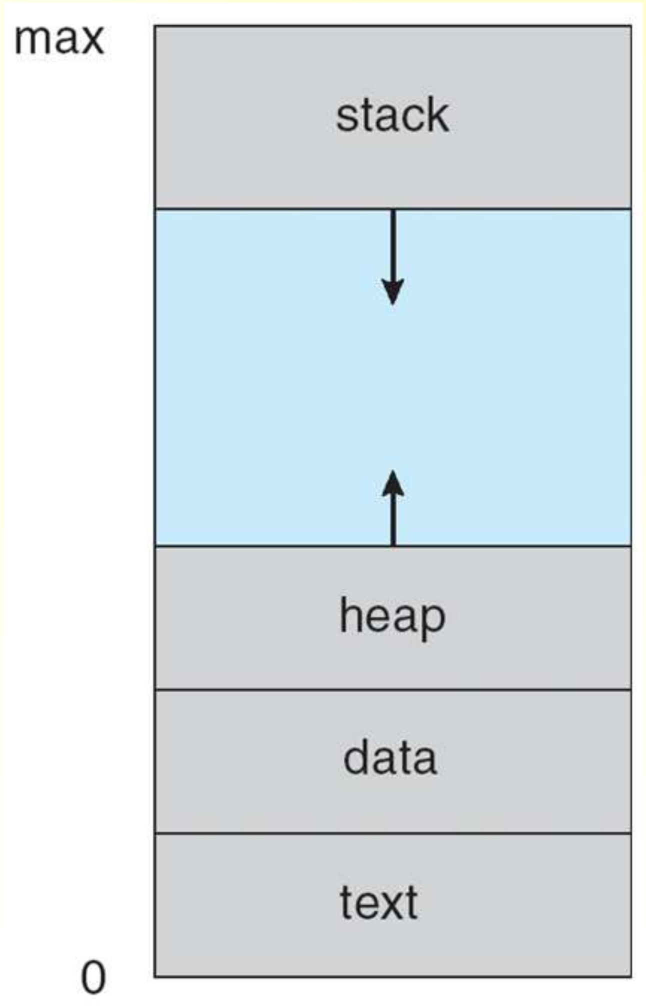
Objectives

- After completing this topic you should be able to:
  - Define process
  - Differentiate program and process
  - Explain different types of processes
  - Explain different states of process
  - Understand how OS manages processes

1 Process Concept

- Process vs. Program
  - Program
    - It is sequence of instructions defined to perform some task
    - It is a passive entity, a file containing a list of instructions stored on disk(often called an executable file).
  - Process, also called job
    - It is a program in execution. A process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task.
    - It is an instance of a program running on a computer.
    - It is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources.
      - Program becomes process when executable file loaded into memory
  - A process includes:
    - The program code, also called **text section**
    - **program counter**: contains the address of the next instruction to be fetched
    - **Stack**: contains temporary data (such as function parameters, return addresses, and local variables).
    - **data section**: contains global variables.
    - **heap**: memory that is dynamically allocated during process run time

Process in Memory



Process concept(cont'd)

- **Real life Example:** Consider a computer scientist who is baking a birthday cake for her daughter and who is interrupted by her daughter's bleeding accident
- Analysis

Processes	Baking Cake	First Aid
Processor	Computer Scientist	Computer Scientist
Program	Recipe	First Aid Book
Input	Ingredients	First Aid Kit
Output	Cake	First Aid Service
Priority	Lower	Higher
States	Running, Idle	Running, Idle

Process concept(cont'd)

- Sequence of actions
  - Bringing ingredients i.e. flour, sugar, eggs, etc
  - Placing the mixture into the oven
  - Following the baking processes
  - Hearing a cry and analyzing it to be because of bleeding
  - **Recording** baking processes **state** and **switching** to provide first aid service
  - Providing first aid service
  - **Coming back and resuming** the baking process
- The key idea here is that a process is an activity of some kind. It has a program, input, output, and a state.

2. Types of Processes

- There are two types of processes
  - Sequential Processes
  - Concurrent Processes
- Sequential Processes
  - Execution progresses in a sequential fashion.
  - At any point in time, at most one process is being executed
- Concurrent Processes - two types of concurrent processes
  - ***True Concurrency*** (Multiprocessing)
    - Two or more processes are executed simultaneously in a multiprocessor environment
    - Supports real parallelism

Types of Processes(cont’d)

- ***Apparent Concurrency*** (Multiprogramming)
  - Two or more processes are executed in parallel in a uniprocessor environment by switching from one process to another
  - Supports pseudo parallelism, i.e. the fast switching among processes gives illusion of parallelism

3. Process Creation

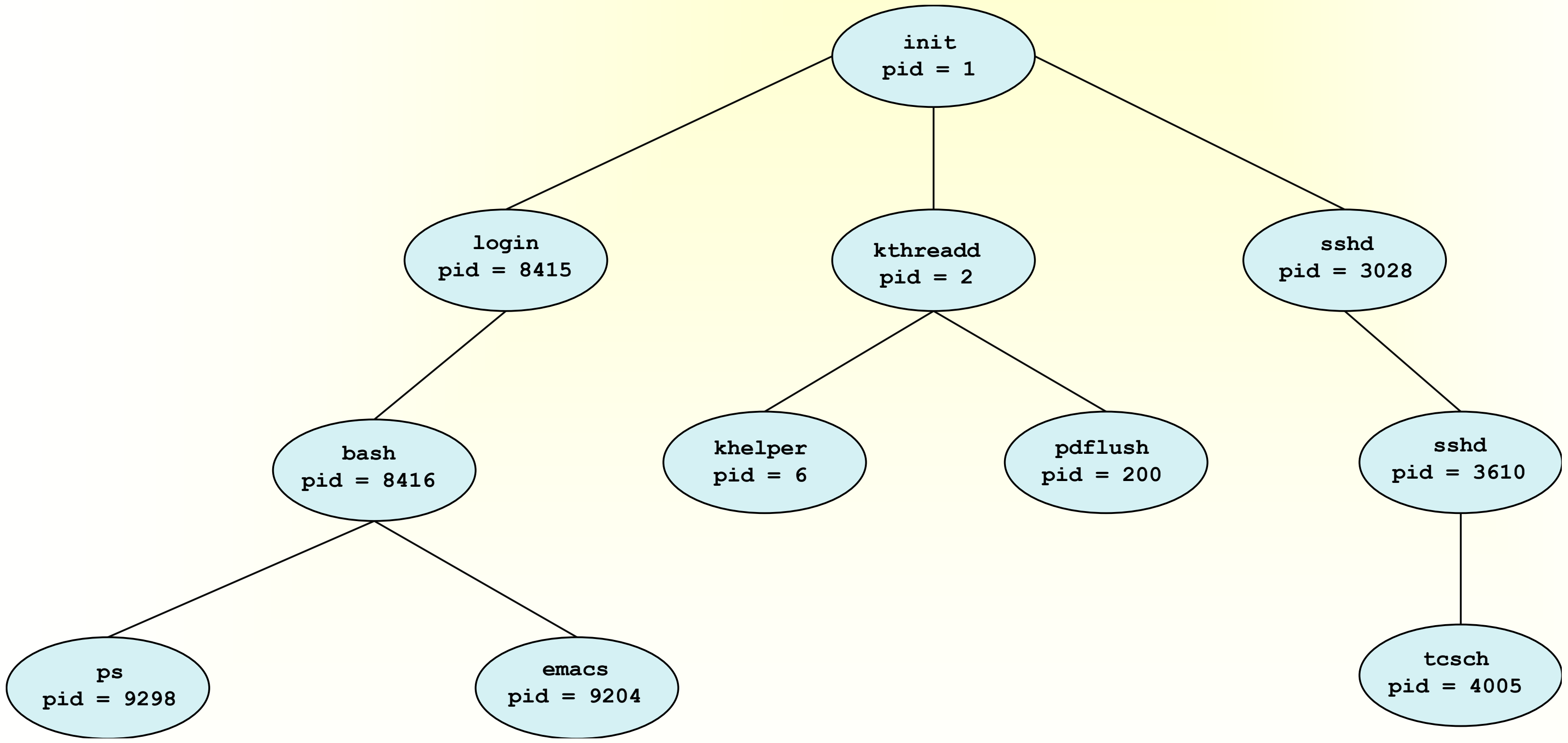
- Principal events that cause process creation:
  - System initialization.
    - When an operating system is booted, often several processes are created. Some of them are foreground and others are background.
      - Foreground processes: processes that interact with (human) users and perform work for them.
      - Background processes, which are not associated with particular users, but instead have some specific function. For example, a background process may be designed to accept incoming requests for web pages hosted on that machine.
  - Execution of a process creation system call by a running process.
    - a running process will issue system calls to create one or more new processes to help it do its job.
  - A user request to create a new process. E.g. open a document
  - Initiation of a batch job(on large mainframes).

Process Creation(cont'd)

- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via **a process identifier** (**pid**)
- Resources required when creating process to accomplish its task
  - CPU time, files, memory, I/O devices etc.
- Resource sharing options
  - Child process may be able to obtain its resources directly from the OS
  - Children share subset of parent's resources
  - Parent may have to partition its resources among its children
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Two address-space possibilities for the new process:.
  - The child process is a duplicate of the parent process (it has the same program and data as the parent).
  - The child process has a new program loaded into it.

[Read the detail from Operating Systems Concepts - Section 3.3.1]

A Tree of Processes in Linux



4. Process State

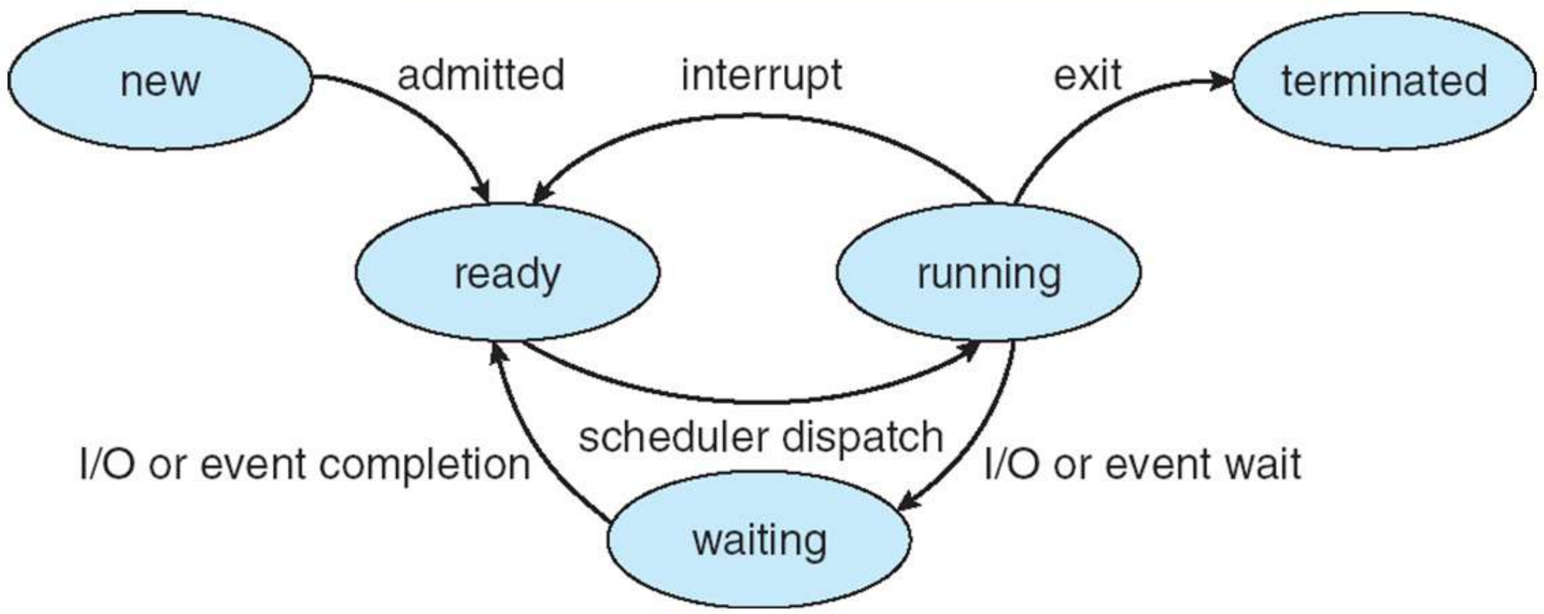
- During its lifetime, a process passes through a number of states.
- The most important states are: Ready, Running, Blocked (waiting)
- **New**
  - A process that has just been created but has **not yet been admitted to the pool of executable processes** by the operating system
  - Information concerning the process is already maintained in memory but the **code is not loaded and no space has been allocated** for the process
- **Ready**
  - A process that is not currently executing but that is ready to be executed as soon as the operating system dispatches it

Process State(cont'd)

- **Running**
  - A process that is currently being executed
- **Blocked (Waiting)**
  - A process that is waiting for the completion of some event, such as completion of an I/O operation
- **Exit (Terminated)**
  - A process that has been released from the pool of executable processes by the operating system, either because it halted or because it aborted for some reason.
  - Conditions:
    - Normal exit: processes terminate because they have done their work

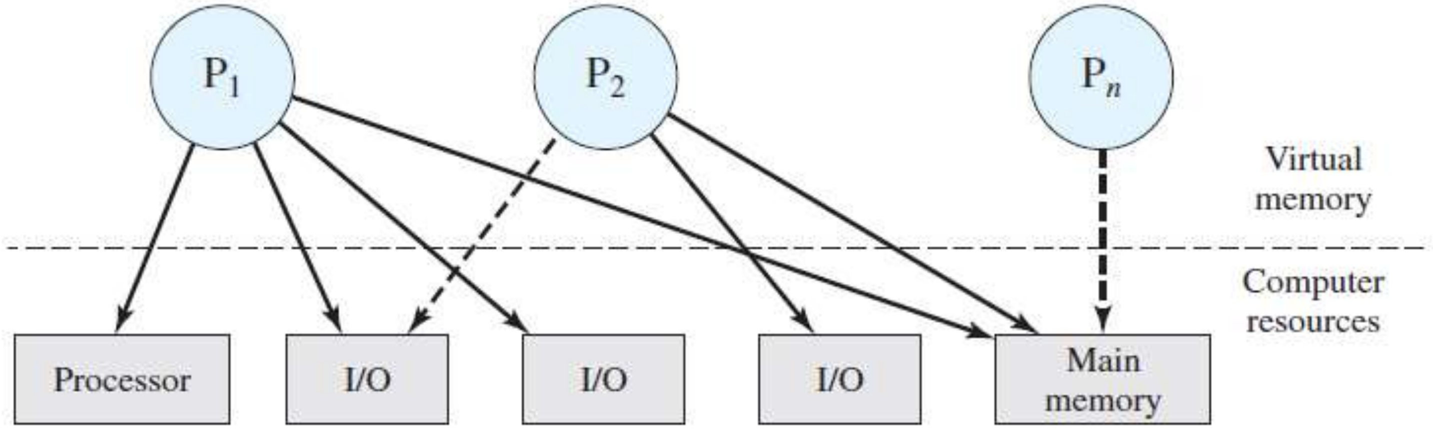
Process State(cont'd)

- Error exit: due to a program bug. Examples include executing an illegal instruction, referencing nonexistent memory, or dividing by zero.
- Fatal error: For example, if a user types the command *cc foo.c* to compile the program foo.c and no such file exists, the compiler simply exits.
- Killed by another process



4. Control Table

We can think of the OS as that entity that manages the use of system resources by processes.



Processes and Resources

Control Table(cont'd)

- **Control Tables**
  - If the operating system is to manage processes and resources, it must have information about the **current status of each process and resources**
  - The operating system constructs and maintains tables of information, called control tables.
  - There are four major control tables:
    - Memory tables
    - I/O tables
    - File tables
    - Process tables

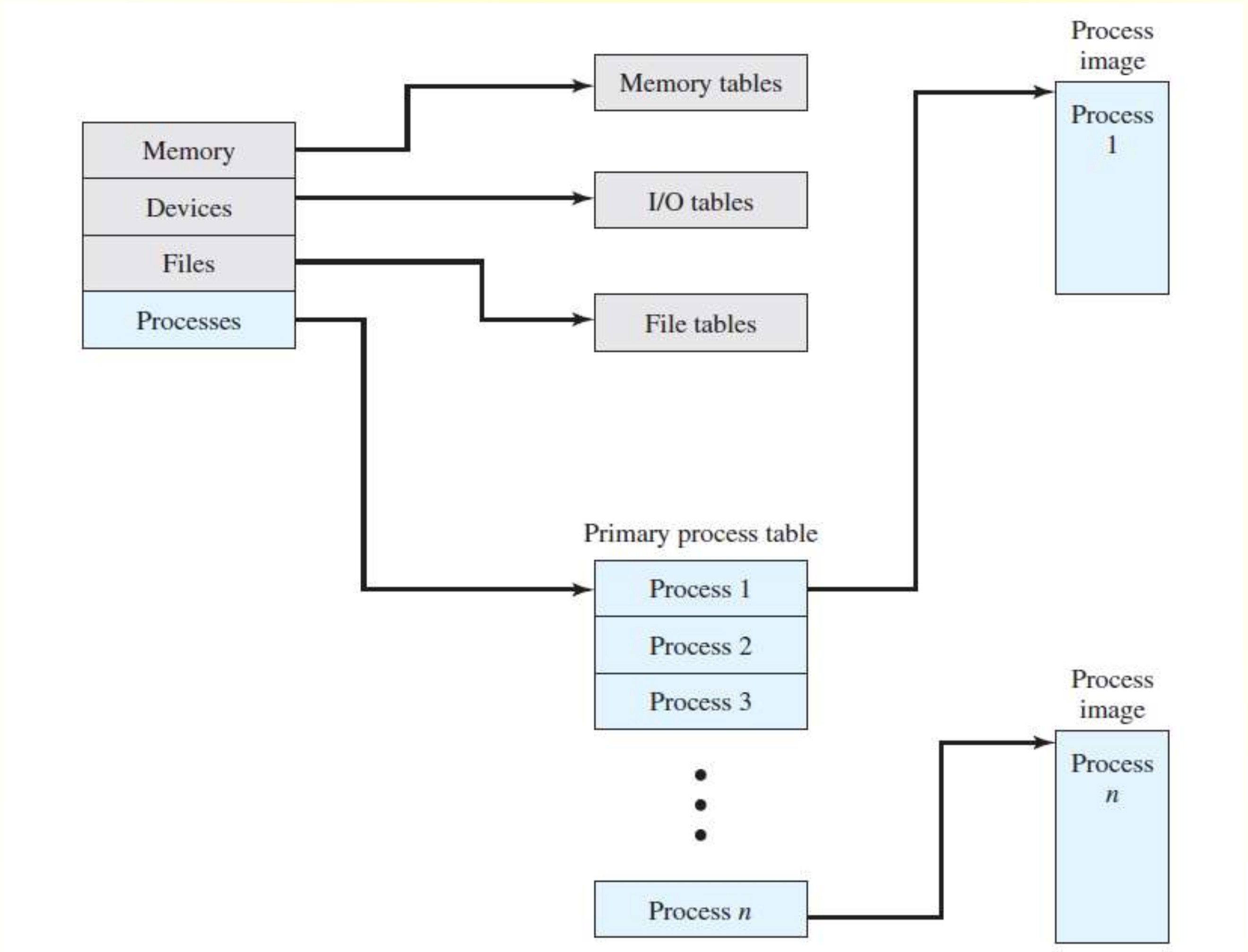
Control Table(cont'd)

- **Memory tables**
  - Are used to keep track of both main and secondary (virtual) memory. Some of main memory is reserved for use by the OS; the remainder is available for use by processes.
  - The memory tables must include the following information:
    - Allocation of main memory to processes
    - Allocation of secondary memory to processes
    - Any information needed to manage virtual memory
- **I/O tables**
  - Used by the OS to manage the I/O devices.
  - If an I/O operation is in progress, the OS needs to know the status of the I/O operation and the location in main memory being used as the source or destination of the I/O transfer.

Control Table(cont'd)

- **File tables**
  - These tables provide information about the existence of files, their location on secondary memory, their current status, and other attributes.
- **Process tables**
  - Are used to manage processes
  - OS must know the location of each process and the attributes of each process(process ID and process state)
- The tables must be linked or cross-referenced in some fashion.
- To create the tables, the operating system must have some knowledge of the basic environment, such as how much main memory exists, what are the I/O devices and what are their identifiers, and so on.

Control Table(cont'd)



General Structure of Operating System Control Tables

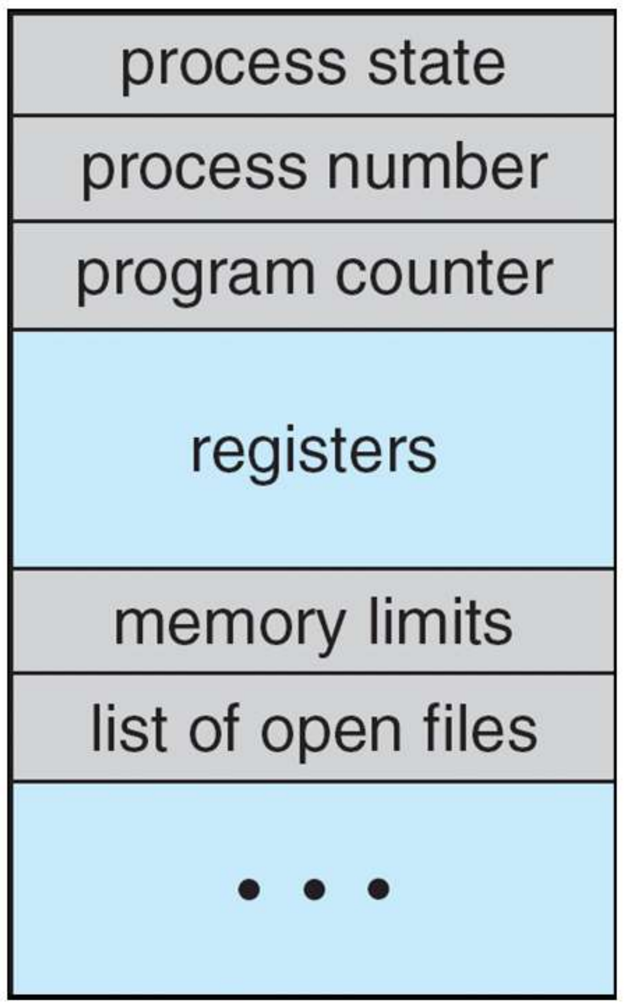
Process Control Block

- Each process is represented in the OS by process control block(PCB).
- The PCB simply serves as the repository for any information that may vary from process to process.
- Information associated with each process:
  - Process state. The state may be new, ready running, waiting, halted, and so on.
  - Program counter. The counter indicates the address of the next instruction to be executed for this process.
  - CPU registers. Along with the program counter, registers state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward

Process Control Block(cont'd)

- CPU-scheduling information
  - This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- Memory-management information
  - This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
- Accounting information
  - This information includes the amount of CPU and clock time used, time limits, job or process numbers, and so on.
- I/O status information
  - This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

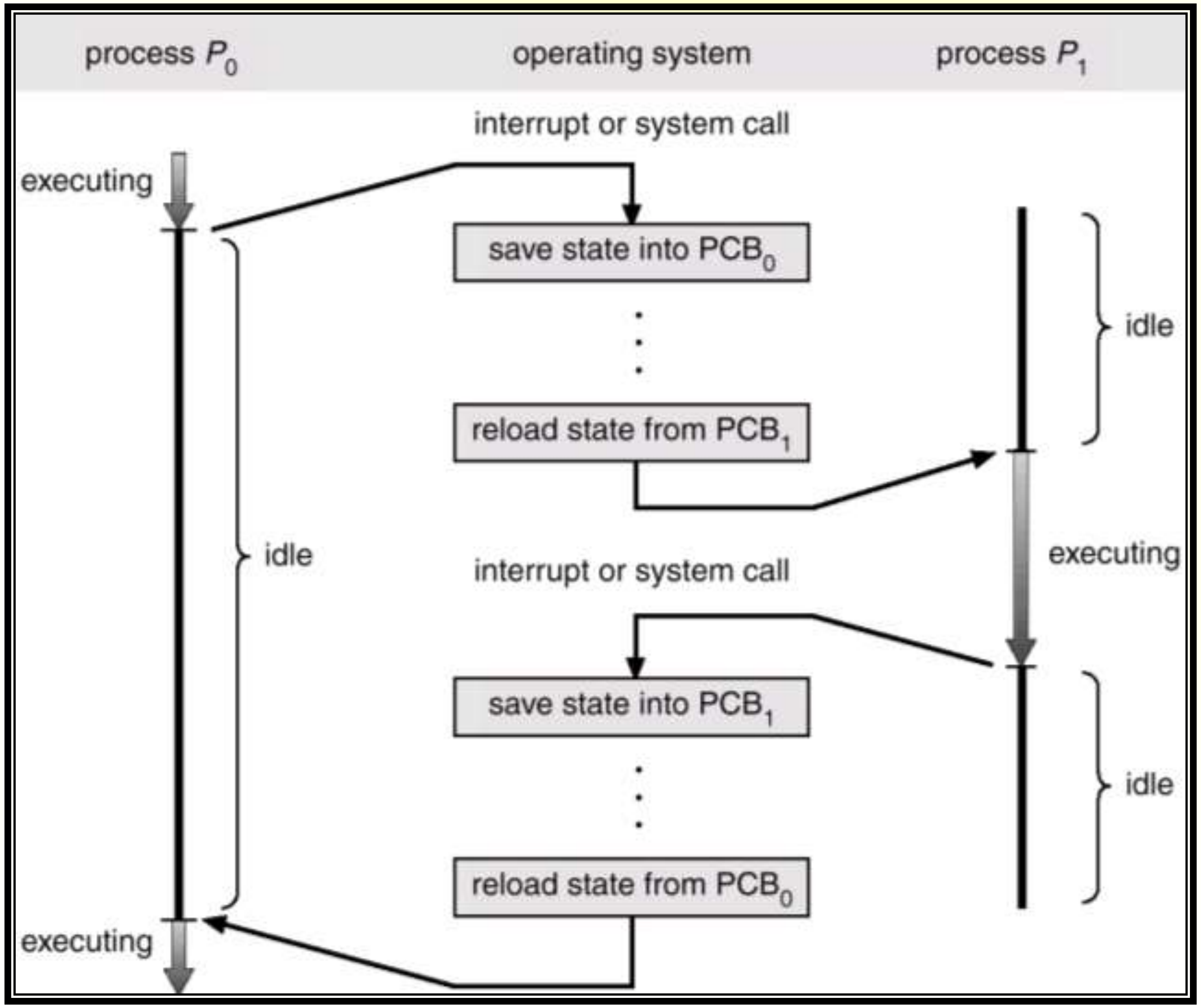
Process Control Block(cont’d)



5. **Process Switching**

- Process switch occurs when a running process is interrupted and the operating system assigns another process to the running state and turns control over to that process
- Process switch is sometimes called context switch
- A running process may be interrupted by
  - **Interrupt**: An event that occurs outside the process and that is recognized by the processor (clock interrupt, I/O interrupt)
  - **Supervisor Call**: A procedure call used by user programs to call an operating system function.
  - **Trap**: An error or exception condition generated within the currently running process (illegal file access attempt)
- Switching requires performing a state save of the current process and a state restore of a different process
- Context-switch time is overhead;
  - the system does no useful work while switching

Process Switching(cont'd)

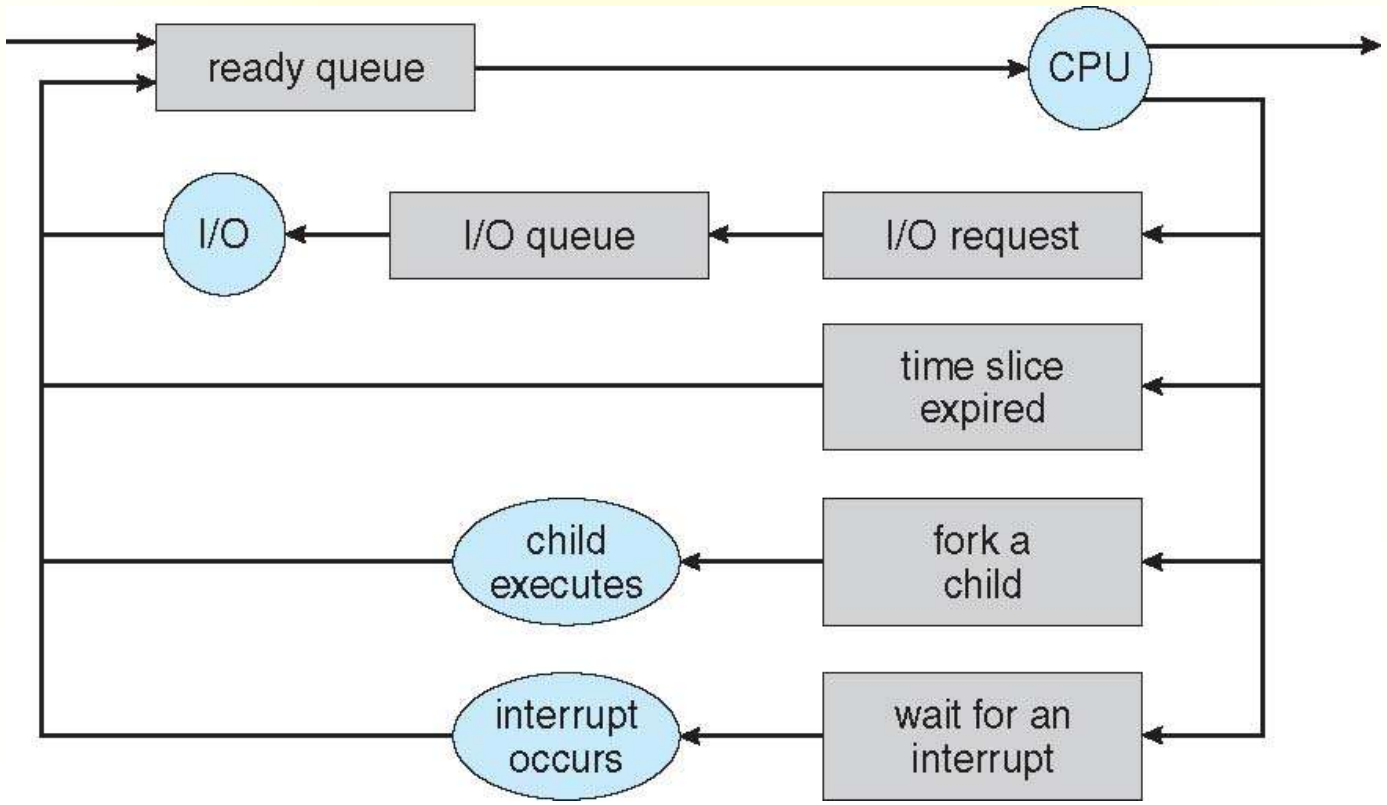


Process Scheduling

- The Objective of **multiprogramming** is to have some process running at all times (to maximize CPU utilization.)
- The objective of **time sharing** is to switch the CPU among processes so frequently
- To meet these objectives **Process scheduler** selects among available processes for next execution on CPU
- Process scheduler maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

Representation of Process Scheduling

- **Queuing diagram** is a common representation of process scheduling
  - **Rectangles** represents queues (ready and device queues),
  - **Circles** represents resources that serve queues,
  - **Arrows** indicates flows of process in the system



Schedulers

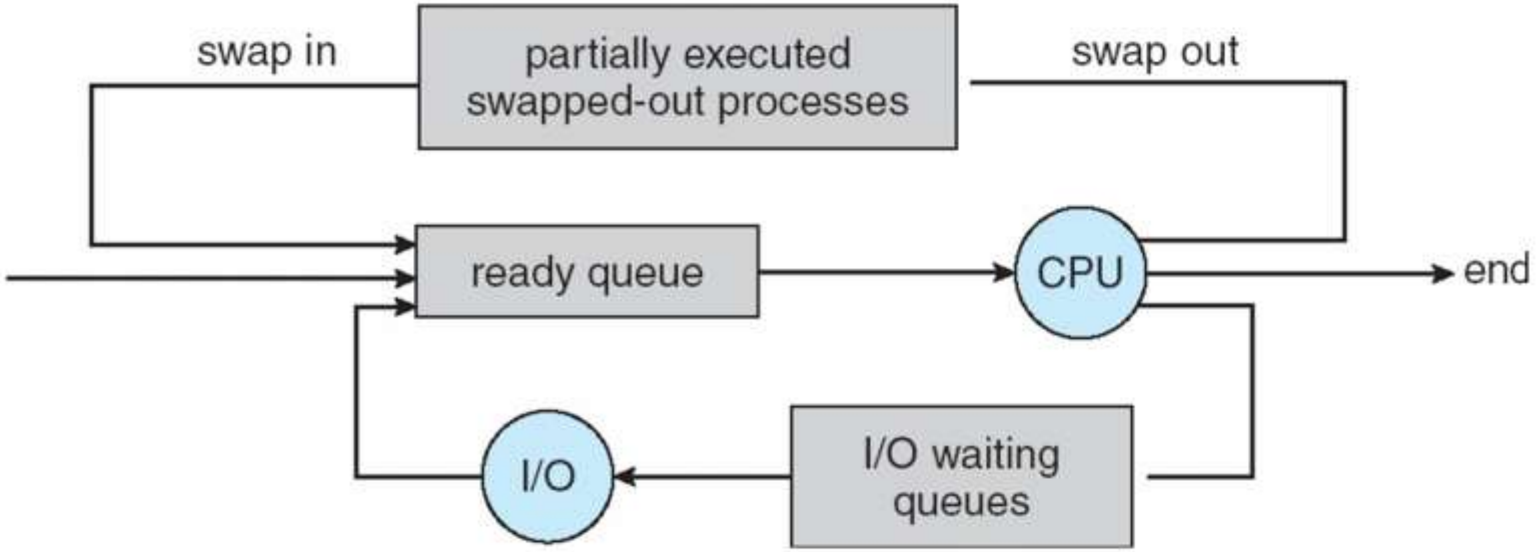
- **Long-term scheduler** (or job scheduler) – selects which processes should be **brought into the ready queue.**
- **Short-term scheduler** (or CPU scheduler) – selects which process should be **executed** next and allocates CPU
  - Sometimes the only scheduler in a system
- Short-term scheduler is invoked very frequently (milliseconds) $\Rightarrow$ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
- The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory)

Schedulers(cont’d)

- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- long-term scheduler select a good ***process mix*** of I/O-bound and CPU-bound processes.

Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**
    - Swapping may be necessary **to improve the process mix** or when **additional memory is required**



Process Termination

- A process terminates when it finishes executing its last statement and asks the operating system to delete it (using **exit()**)
  - process may return a status value (typically an integer) to its parent process (via the wait() system call)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes for the following reasons:
  - Child has exceeded allocated resources(the parent must have a mechanism to inspect the state of its children)
  - Task assigned to child is no longer required
  - The parent is exiting
    - Some operating systems do not allow child to continue if its parent terminates
      - All children terminated - **cascading termination**

The Thread Concept

Objectives

- After completing this topic you should be able to:
  - Define thread
  - Differentiate process and thread
  - Understand thread management

Processes and Threads

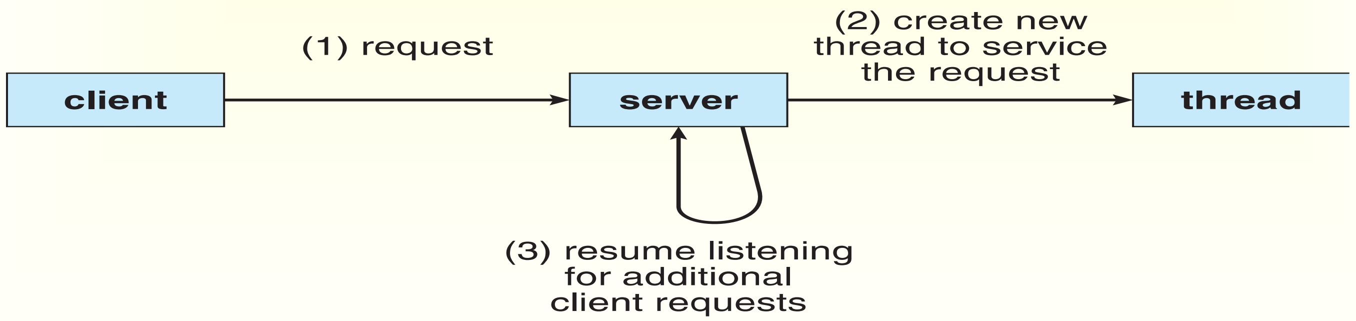
- Process holds two characteristics:
  - **A. Resource ownership:** A process includes a virtual address space to hold the process image; collection of program, data, stack, and attributes defined in the process control block.
  - **B. Scheduling/execution:** a process has an execution state (Running, Ready, etc.) and a dispatching priority and is the entity that is scheduled and dispatched by the OS.
  - These two characteristics are independent and could be treated independently by the OS.
    - Processes are used to group resources together, threads are the entities scheduled for execution on the CPU.

Processes and Threads(cont'd)

- The unit of dispatching is usually referred to as a **thread** or **lightweight process**, while the unit of resource ownership is usually still referred to as a **process** or **task.**
- A process is a collection of one or more threads and associated system resources.
- Traditional operating systems are single-threaded systems. Modern operating systems are multithreaded systems

Multithreading

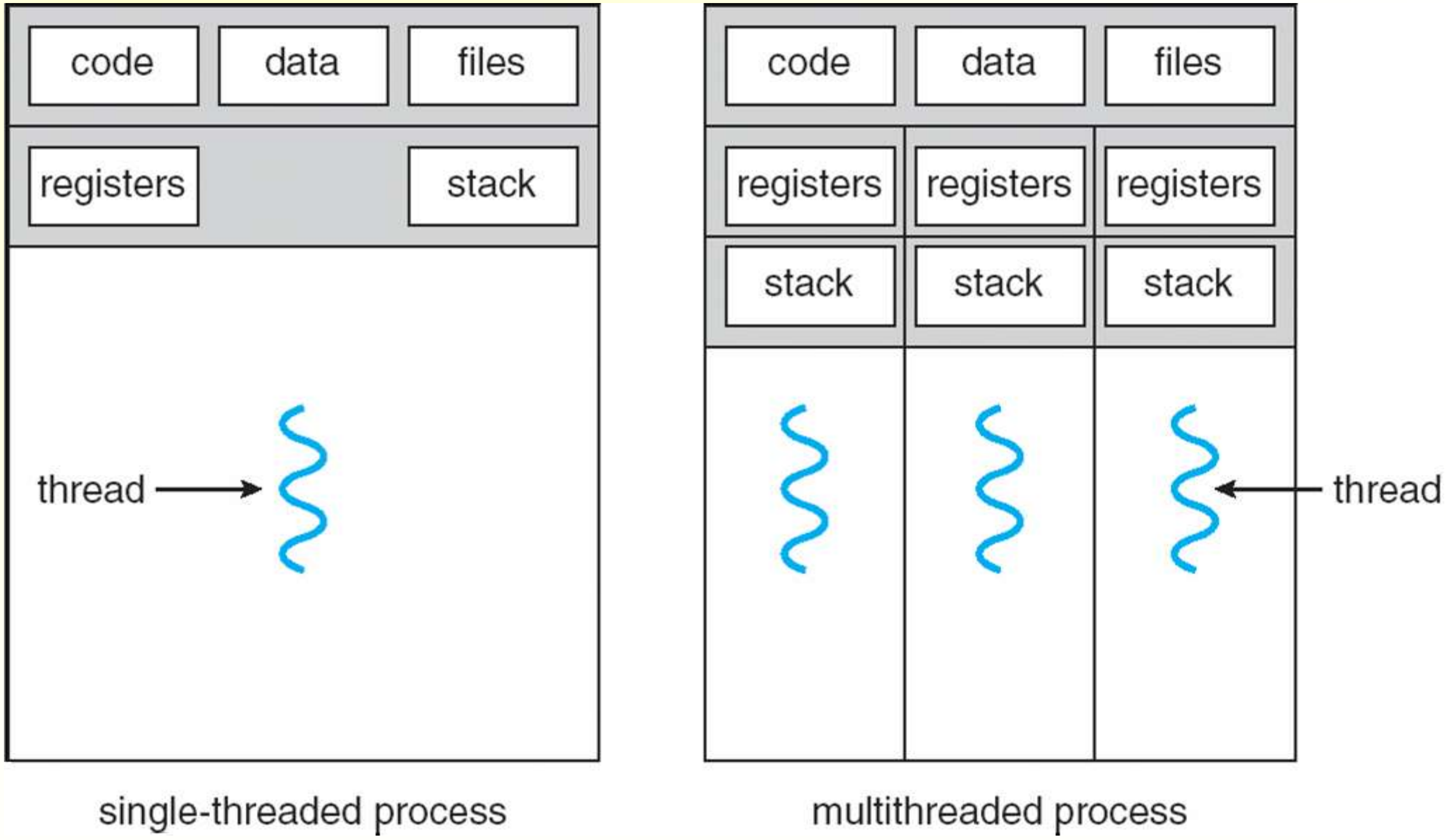
- Multithreading is a technique in which a process, executing an application, is divided into threads that can run concurrently.
  - Handles several independent tasks of an application
  - Multithreading can make your program more responsive and interactive, as well as enhance performance. Example:
    - A good word processor lets you print or save a file while you are typing.
    - When downloading a large file, we can put multiple threads to work—one to download the clip, and another to play it.
    - web server - Multiple threads allow for multiple requests to be satisfied simultaneously



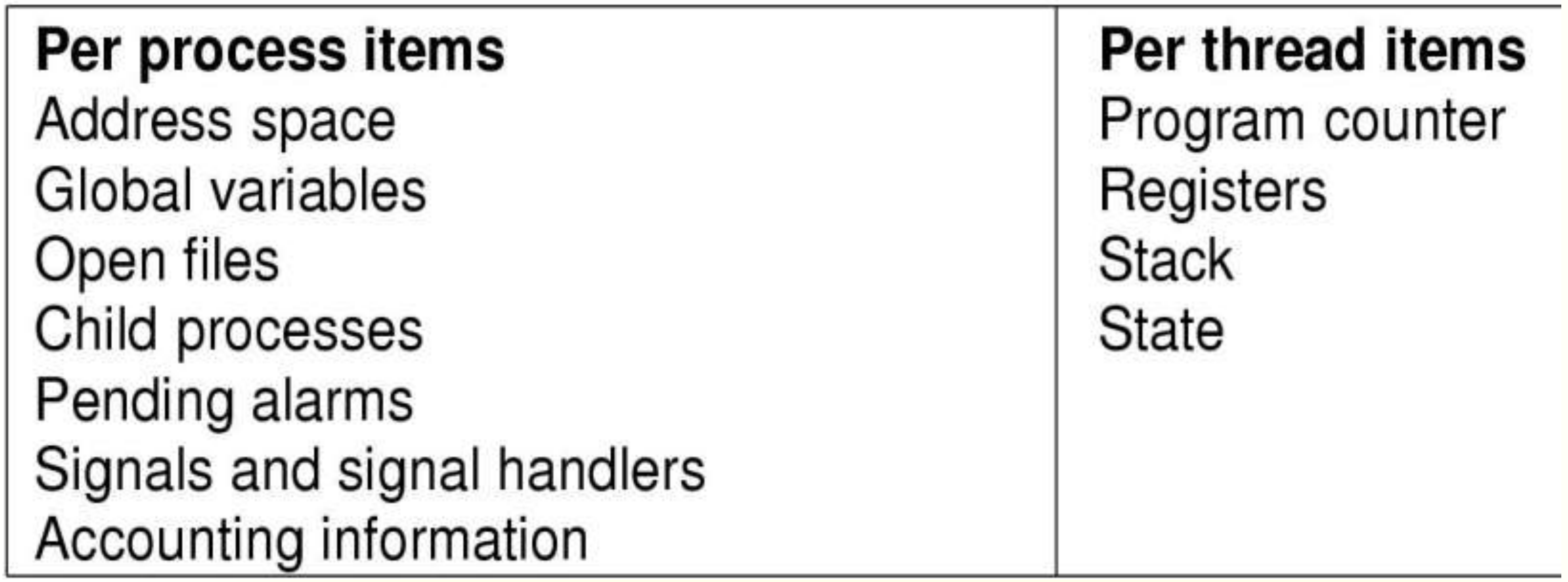
(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

- A single thread of execution per process is referred to as a single-threaded approach. E.g. MS-DOS.

Single and Multithreaded Processes



Shared and Private Items



Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

A(Items shared by all threads in a process)

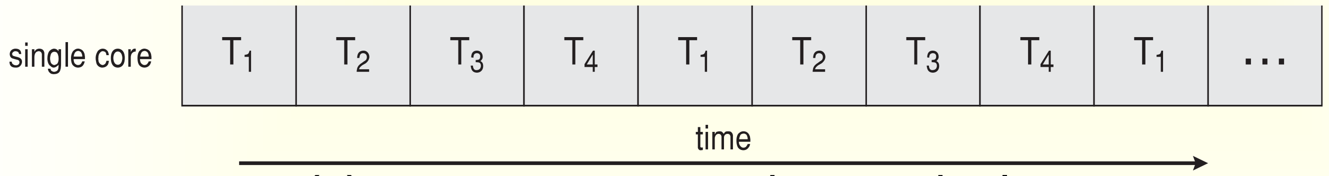
B(Items private to each thread)

Benefits of Threads

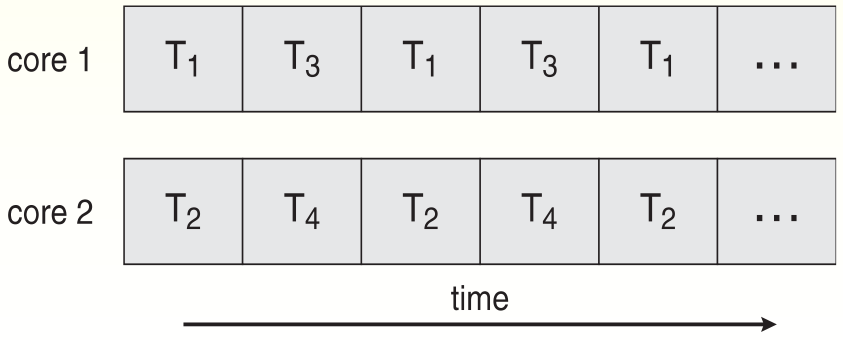
- The key benefits of threads derive from the performance implications:
  - It takes far less time to create a new thread in an existing process than to create a brand-new process.
    - E.g. In Solaris, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower.
  - It takes less time to terminate a thread than a process.
  - It takes less time to switch between two threads within the same process than to switch between processes.
  - Threads enhance efficiency in communication between different executing programs.
    - Threads within the same process share memory and files(If one thread opens a file with read privileges, other threads in the same process can also read from that file), they can communicate with each other without invoking the kernel.

Multicore Programming

- Recent systems have multiple computing cores on a single chip. These systems is called multicore or multiprocessor.
  - Each core appears as a separate processor to the operating system.
- Multithreaded programming provides a mechanism for more efficient use of multiple computing cores and improved concurrency.



(a) Concurrent execution on single-core system


core 1
T_1 T_3 T_1 T_3 T_1 ...
core 2
T_2 T_4 T_2 T_4 T_2 ...
time

(b) Parallel execution on a multicore system

Multicore Programming(cont'd)

- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task by allowing all the tasks to make progress
  - It is possible to have concurrency without parallelism
- Multicore systems putting pressure on programmers to make better use of the multiple computing cores, challenges include:
  - Dividing tasks: involves examining applications to find areas that can be divided into separate, concurrent tasks
  - Balance: ensure that the tasks perform equal work of equal value
  - Data splitting: data accessed and manipulated by the tasks must be divided to run on separate cores
  - Data dependency: When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized
  - Testing and debugging: many different execution paths are possible, testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications.
- Multicore systems will require an entirely new approach to designing software systems

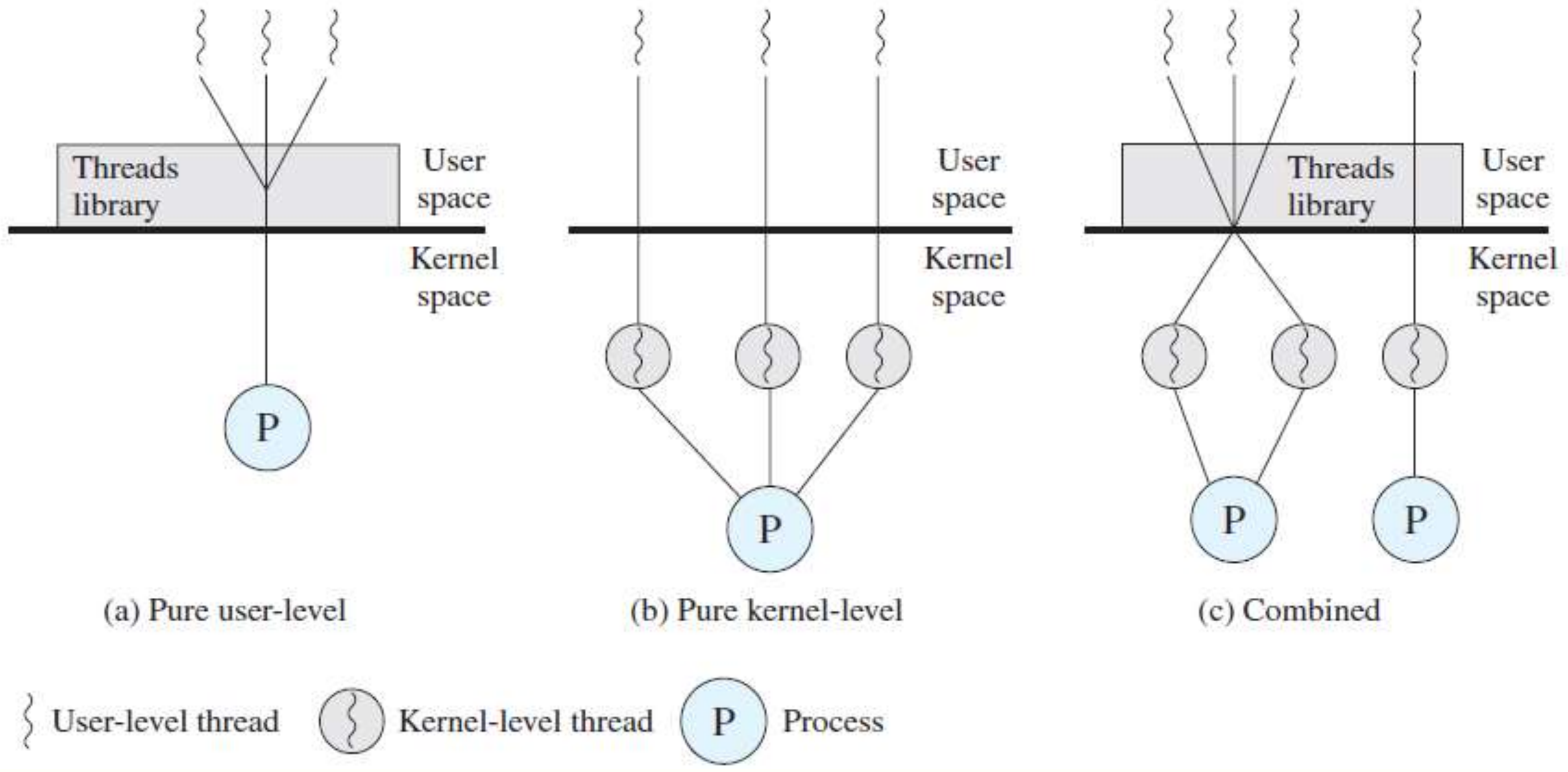
User-Level and Kernel-Level Threads

- Threads can be managed in the kernel or user space
  - Kernel Space
    - thread management is done by the kernel
    - Advantage:
      - The operating system is aware of the existence of multiple threads. When a thread blocks, the operating system chooses the next one to run, either from the same process or a different one
      - Each kernel-level thread can run in parallel on a multiprocessor system
    - Disadvantage:
      - Thread management is slow (requires a mode switch to the kernel)
      - Creating a user thread requires creating the corresponding kernel, has effect on performance, restrict the number of threads supported by the system
    - Virtually all contemporary operating systems — including Windows, Linux, Mac OS X, and Solaris— support kernel threads
  - User Space
    - All of the work of thread management is done by the application and the kernel is not aware of the existence of multiple threads
    - The user space has a thread table
    - Thread management is much faster (specially switching)

User-Level and Kernel-Level Threads(cont'd)

- Advantage of ULT
  - Can be implemented on an operating system that does not support threads.
  - Allow each process to have its own customized scheduling algorithm.
- Disadvantages of ULTs
  - when a thread is blocked, all of the threads within that process are blocked
  - If a thread starts running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU
  - cannot take advantage of multiprocessing
    - kernel assigns one process to only one processor at a time
- Combined Approach
  - Thread creation is done completely in user space
  - Multiple ULTs from a single application are mapped onto some (smaller or equal) number of KLTs. The kernel is aware of only the kernel-level threads and schedules those(user can create any number of user-level threads).
  - Combine the advantages of the pure ULT and KLT approaches while minimizing the disadvantages
  - E.g. Solaris prior to version 9

User-Level and Kernel-Level Threads(cont'd)

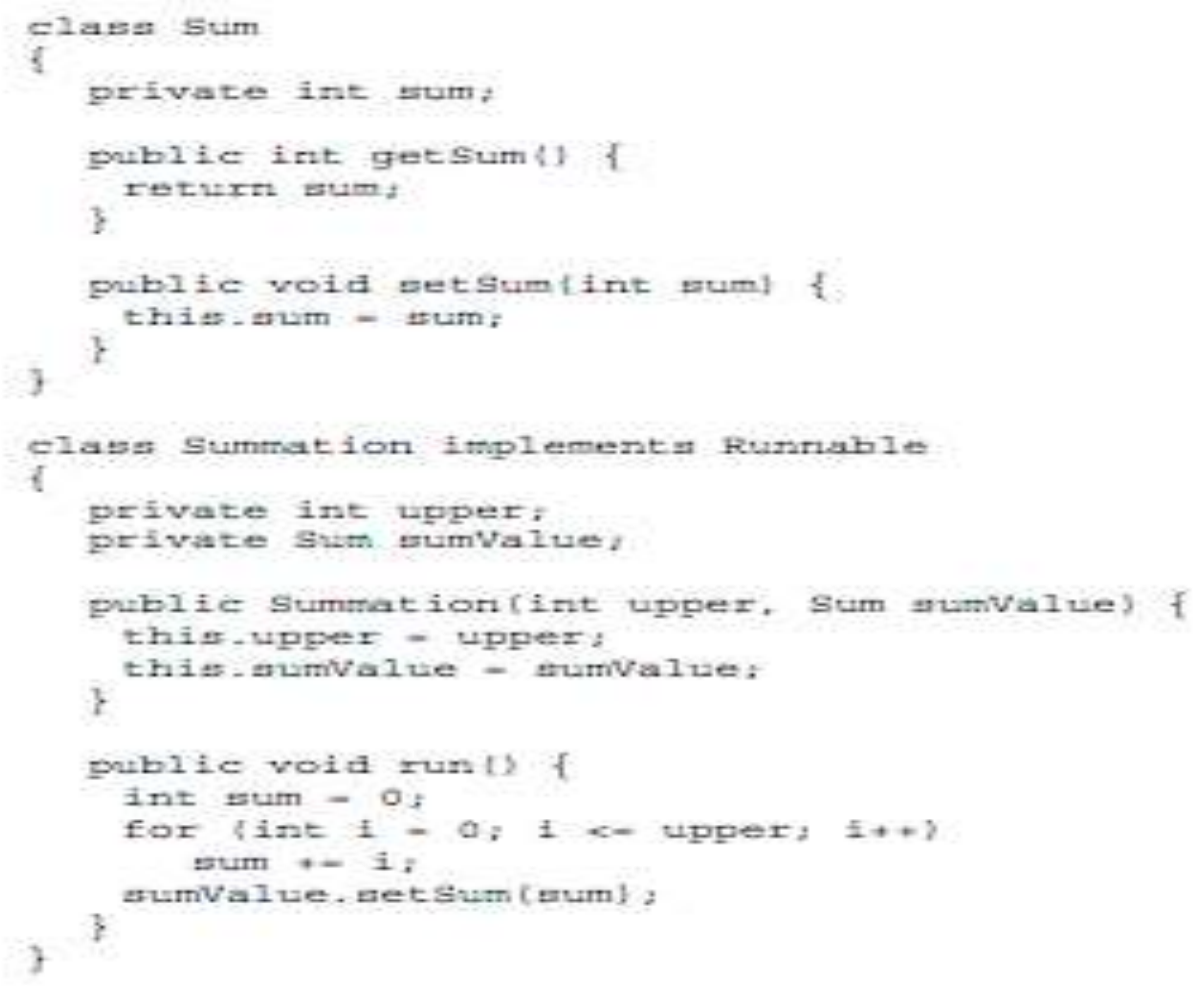


User-Level and Kernel-Level Threads

Thread Libraries

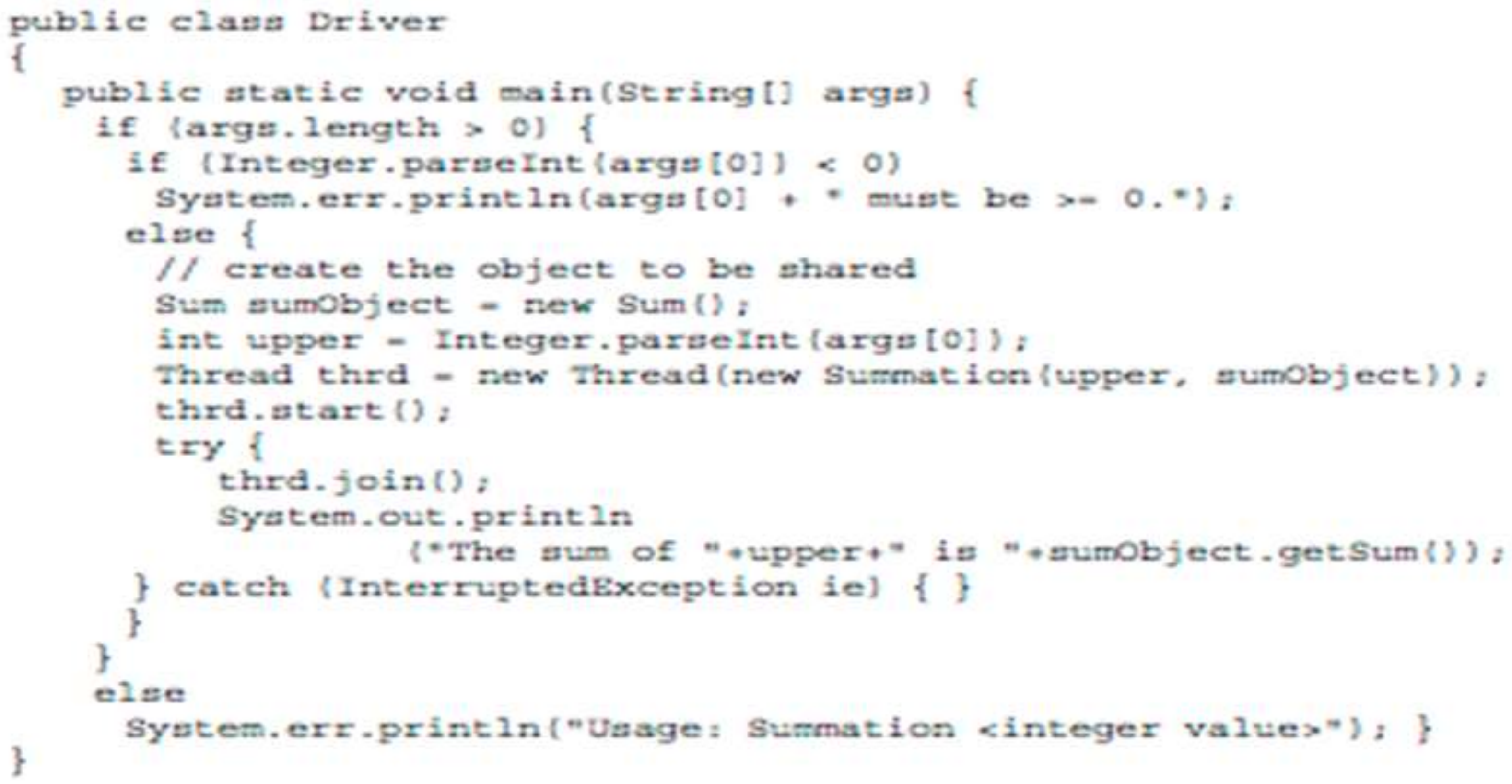
- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space with no kernel support
    - All code and data structures for the library exist in user space.
    - Invoking a function in the library results in a local function call in user space and not a system call.
  - Kernel-level library supported by the OS
    - Code and data structures for the library exist in kernel space.
    - Invoking a function in the API for the library typically results in a system call to the kernel.
- Three main thread libraries are:
  - POSIX Pthreads (provided as either a user-level or a kernel-level library)
  - Windows (a kernel-level library available on Windows systems)
  - Java (implemented using a thread library available on the host system: on Windows- using Windows API; on UNIX-using Pthreads.)

Java Multithreaded Program



```
class Sum
{
   private int sum;

   public int getSum() {
     return sum;
   }

   public void setSum(int sum) {
     this.sum = sum;
   }
}

class Summation implements Runnable
{
   private int upper;
   private Sum sumValue;

   public Summation(int upper, Sum sumValue) {
     this.upper = upper;
     this.sumValue = sumValue;
   }

   public void run() {
     int sum = 0;
     for (int i = 0; i <= upper; i++)
         sum += i;
     sumValue.setSum(sum);
   }
}
```

Java Multithreaded Program (Cont'd)



```
public class Driver
{
  public static void main(String[] args) {
    if (args.length > 0) {
     if (Integer.parseInt(args[0]) < 0)
      System.err.println(args[0] + " must be >= 0.");
     else {
      // create the object to be shared
      Sum sumObject = new Sum();
      int upper = Integer.parseInt(args[0]);
      Thread thrd = new Thread(new Summation(upper, sumObject));
      thrd.start();
      try {
         thrd.join();
         System.out.println
                ("The sum of "+upper+" is "+sumObject.getSum());
     } catch (InterruptedException ie) { }
     }
    }
    else
     System.err.println("Usage: Summation <integer value>"); }
}
```

Java program for the summation of a non-negative integer.

Implicit Threading

- Multicore processing changed the design of programs - programmers create a program with thousands of threads – this is called **explicit threading**
- **Implicit threading-** creation and management of threads done by compilers and run-time libraries rather than programmers
- Three alternative approaches:
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch Intel's Threading Building Blocks (T
several products from Microsoft, java.util.concurrent packa

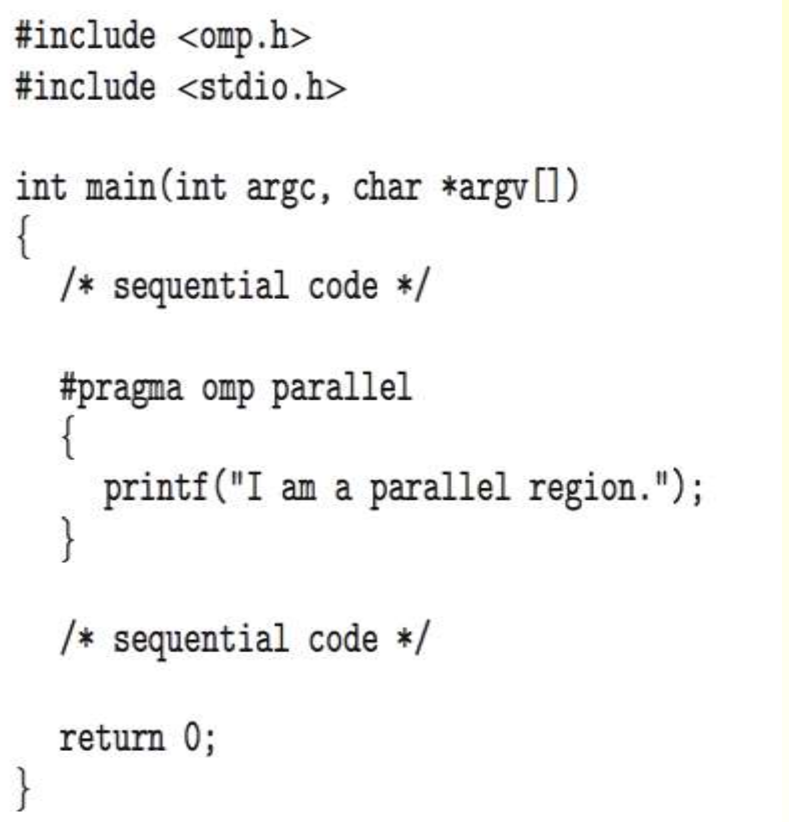
Thread Pools

- Creating a new thread for each request has potential problems:
  - the amount of time required to create the thread, thread will be discarded once it has completed its work
  - Can also lead to a very large (unlimited) number of threads being created: could exhaust system resources.
- Solution:
  - Create a number of threads at the process startup and place them into a pool, where they sit and wait for work
- Advantages:
  - Servicing a request with an existing thread is faster than waiting to create a thread.
  - Allows the number of threads in the application(s) to be bound to the size of the pool. This is important on systems that cannot support a large number of concurrent threads.

OpenMP

- OpenMP is a set of compiler directives as well as API for programs written in C, C++, or FORTRAN that provides support for parallel programming
- Application developers insert compiler directives into their code at parallel regions, and these directives instruct the OpenMP run-time library to execute the region in parallel.
- When OpenMP encounters the directive “#pragma omp parallel”, it creates as many threads are there are processing cores in the system(for a dual-core system, two threads; for a quad-core system, four).

OpenMP- Example



```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

(a) Threads execute the parallel region containing the printf() statement

```
#pragma omp parallel for
for(i=0;i<N;i++) {
    c[i] = a[i] + b[i];
}
```

(b) Parallel task in a loop- sum contents of array a & b and place in c using threads

Grand Central Dispatch

- GCD allows to identify sections of code to run in parallel.
  - A technology for Apple's Mac OS X and iOS operating systems
- GCD identifies blocks- a self-contained unit of work.
  - Specified by a caret ^ inserted in front of { }
  - E.g. `^{ printf("I am a block"); }`
- GCD schedule blocks by placing them in one of the dispatch queues
  - Assigned to available thread in thread pool when removed from queue

Grand Central Dispatch(cont'd)

- Two types of dispatch queues:
  - serial – blocks removed in FIFO order, the removed block must complete execution before another block is removed
    - Each process has its own serial queue (known as its main queue)
    - Blocks placed on a serial queue are removed one by one
    - The next block cannot be removed for scheduling until the previous block has completed.
    - are useful for ensuring the sequential execution of several tasks
  - concurrent – removed in FIFO order but several blocks may be removed at a time, allow multiple blocks to execute in parallel
    - Three system wide queues with priorities low, default, high

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations