VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

SWE3001-Operating Systems

Prepared By
Dr. L. Mary Shamala
Assistant Professor
SCOPE/VIT

Module 2: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Threads- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues

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 scheduling, creation and termination, and communication
- To explore interprocess communication using shared memory and message passing
- To describe communication in client-server systems

Process Concept

- A process is the unit of work in a modern time-sharing system.
- A system consists of a collection of processes:
  - Operating system processes executing system code
  - User processes executing user code.
- All these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them.

The Process

- A process is a program in execution
- An instance of a program running on a computer
- An entity that can be assigned to and executed on a process
- Essential elements of a process
  - Text section
  - Data section
  - Stack
  - Heap
- Program is a passive entity
- Process is an active entity

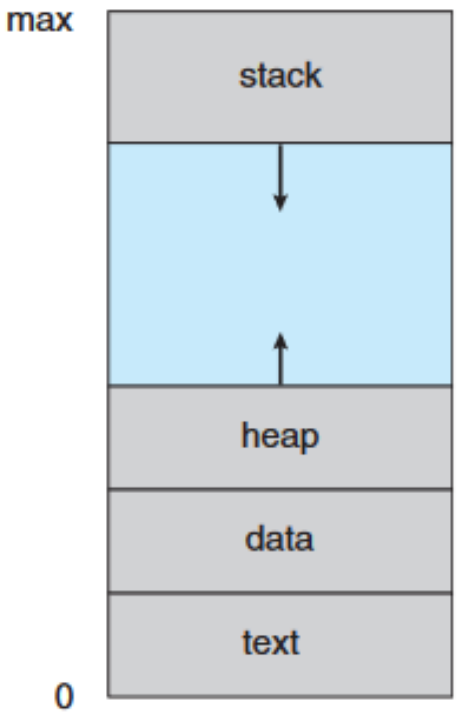
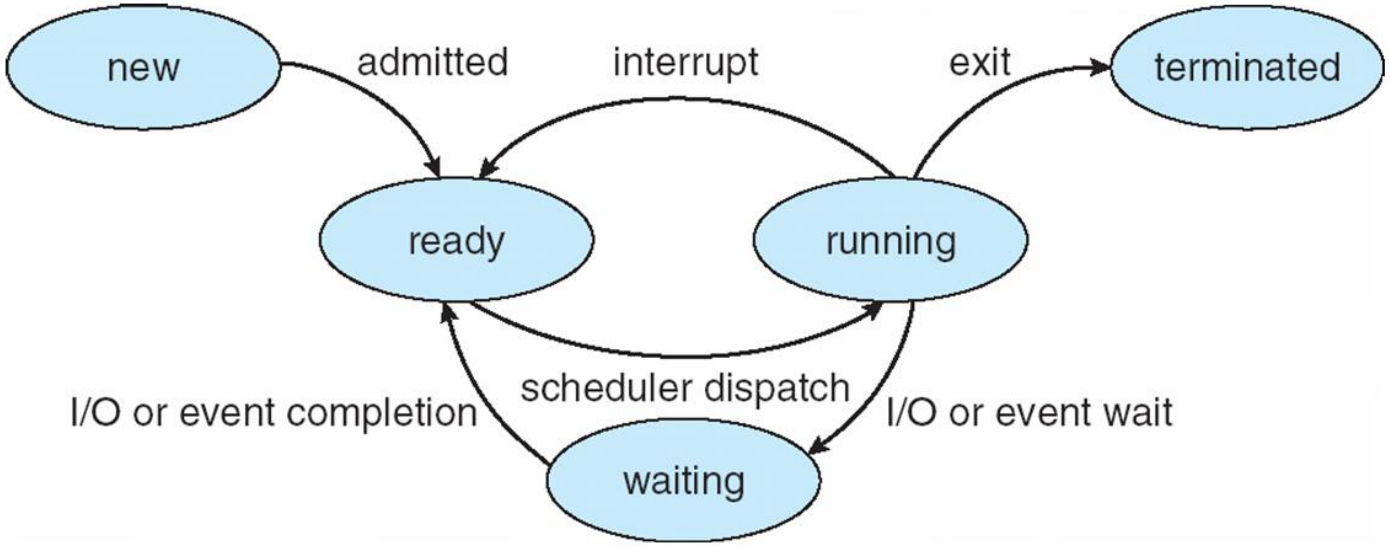


Figure 3.1 Process in memory.

Process State

- The **state** of a process denotes the current activity of that process.
- As a process executes, it changes the state
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process has finished execution

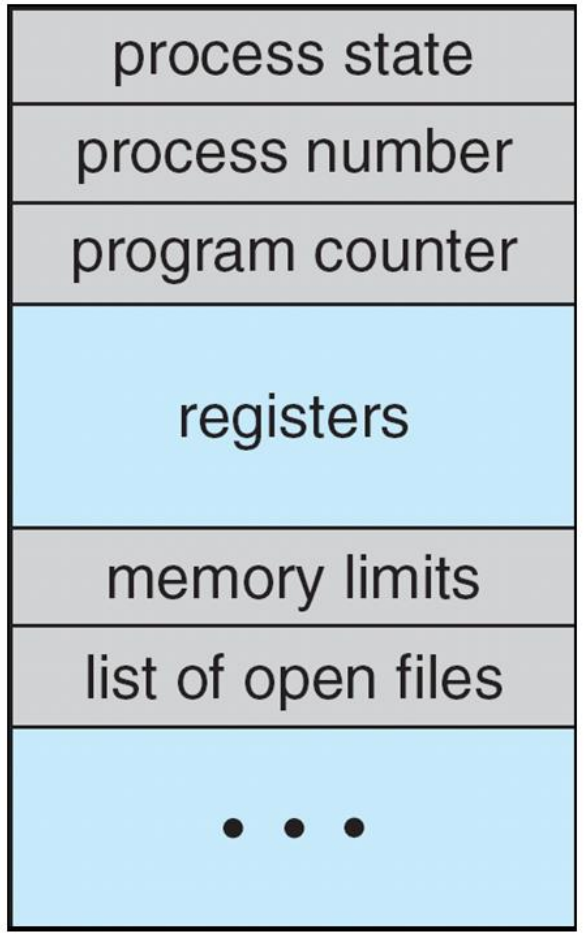
Diagram of Process State



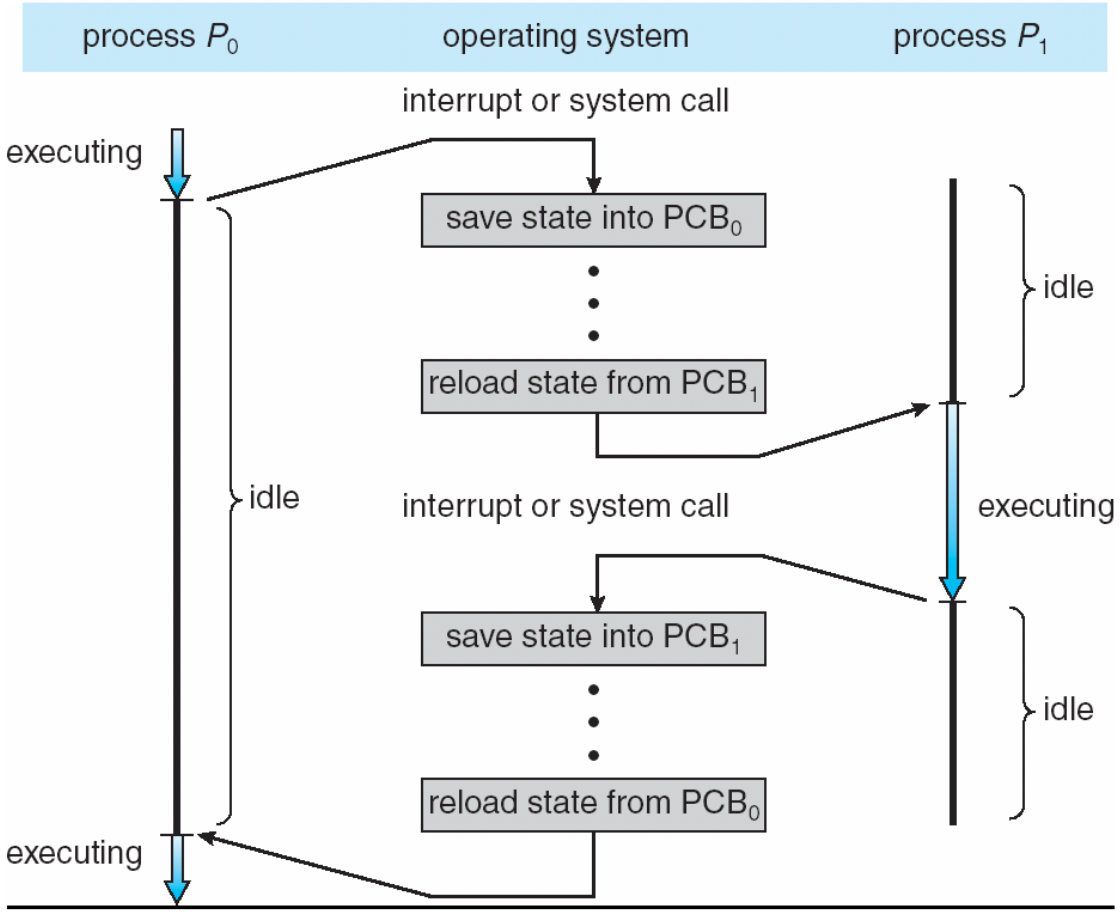
Process Control Block (PCB)

Information associated with each process
(also called **task control block**)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



CPU Switch From Process to Process



The Process Model

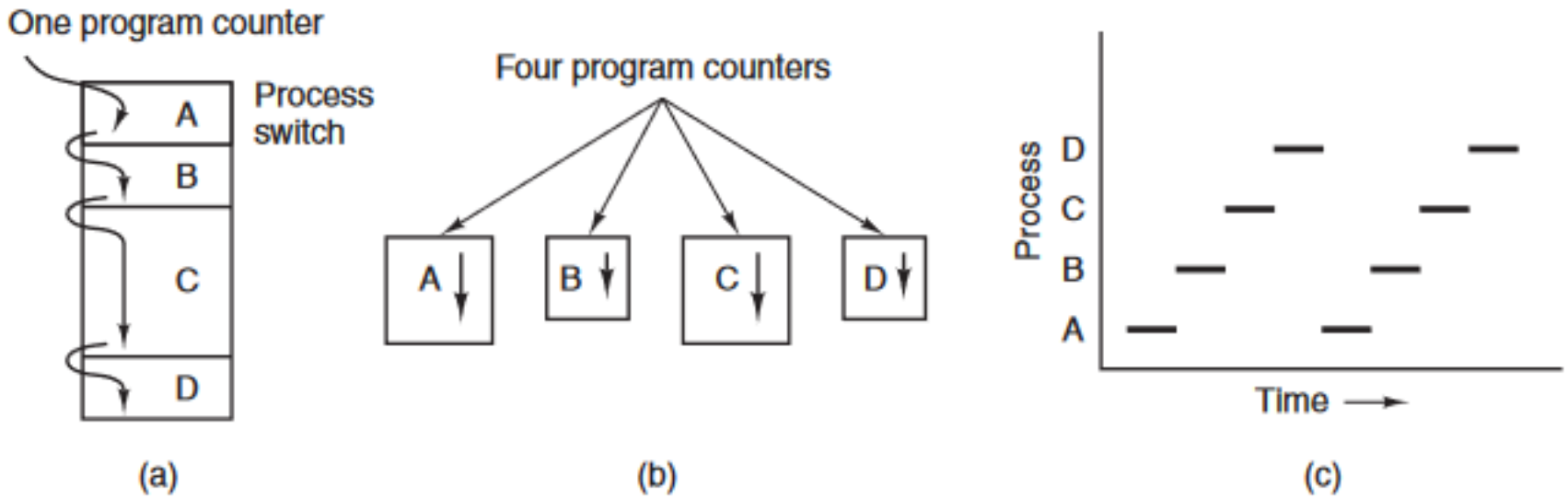


Figure 2-1. (a) Multiprogramming four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

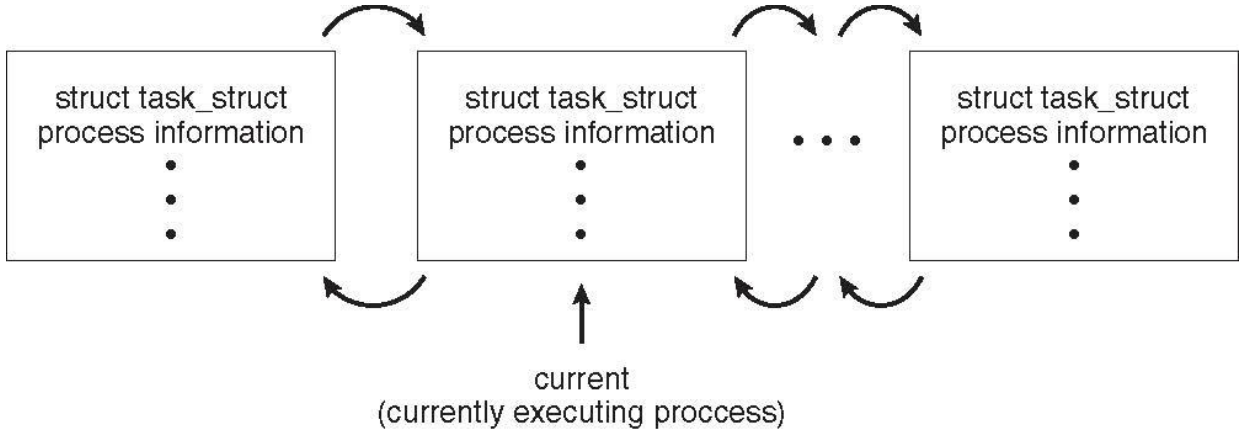
Threads

- So far, process has a single thread of execution.
- This single thread of control allows the process to perform only one task at a time.
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- See next chapter

Process Representation in Linux

Represented by the C structure `task_struct`

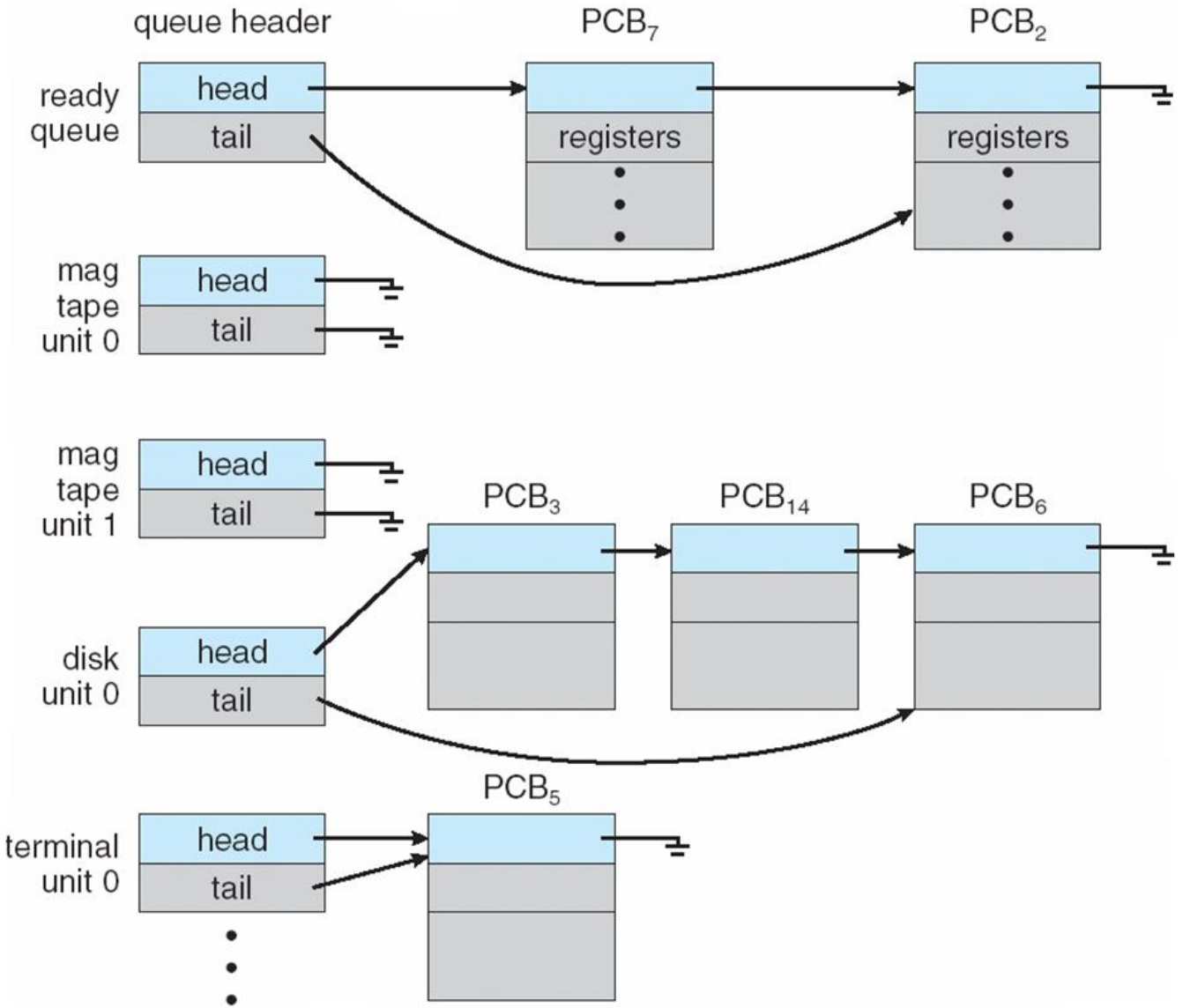
```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



Process Scheduling

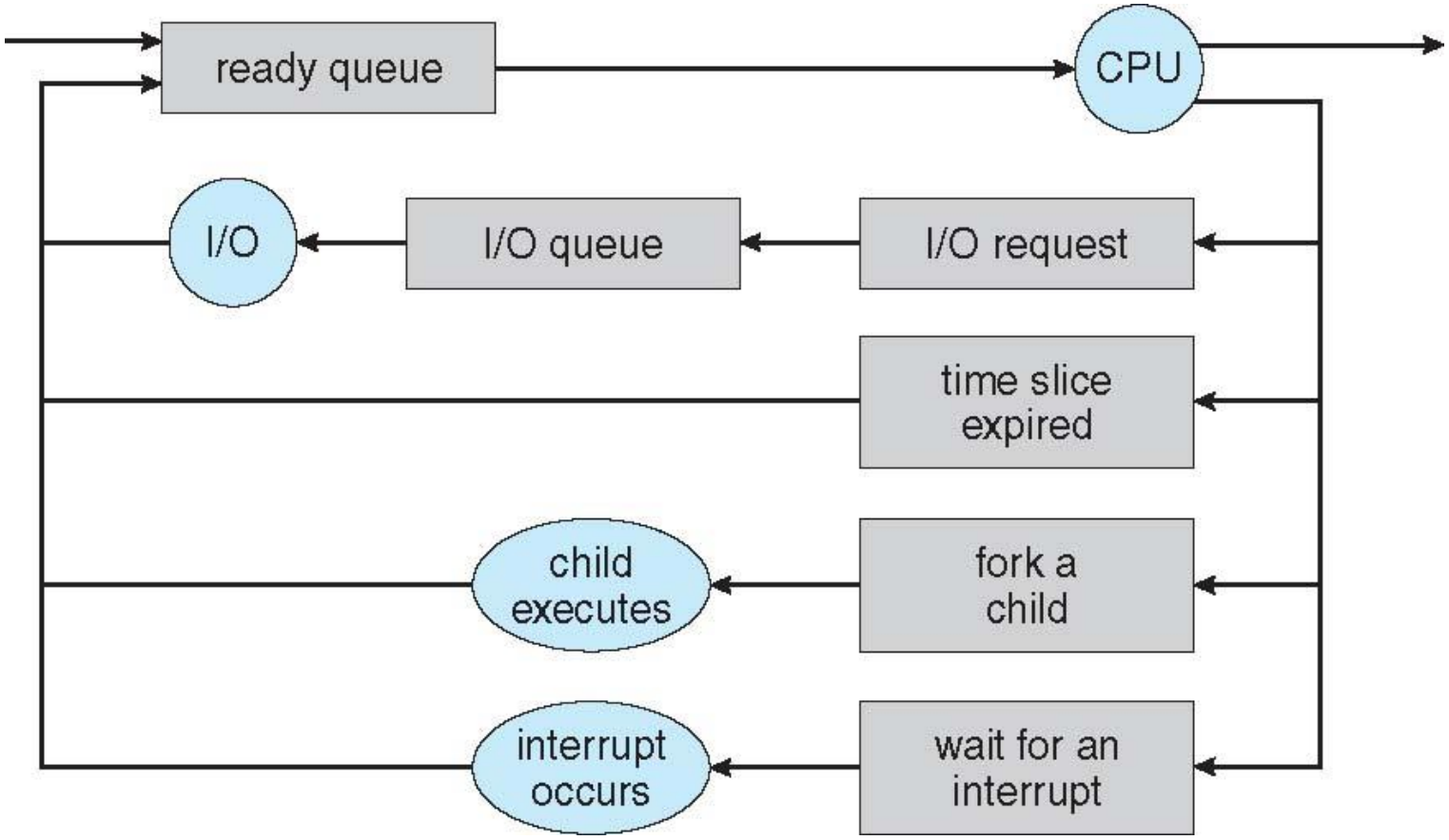
- Maximize CPU use for multiprogramming, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set 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

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

- **Queueing diagram** represents queues, resources, flows
- Two types of queues are present: the **ready queue** and a set of **device queues**.

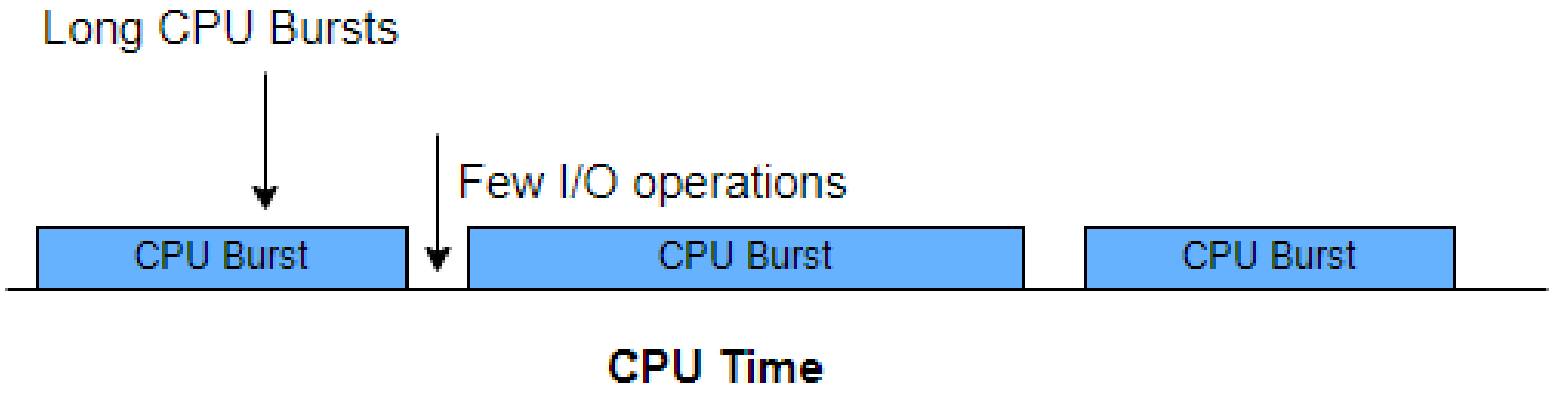


Schedulers

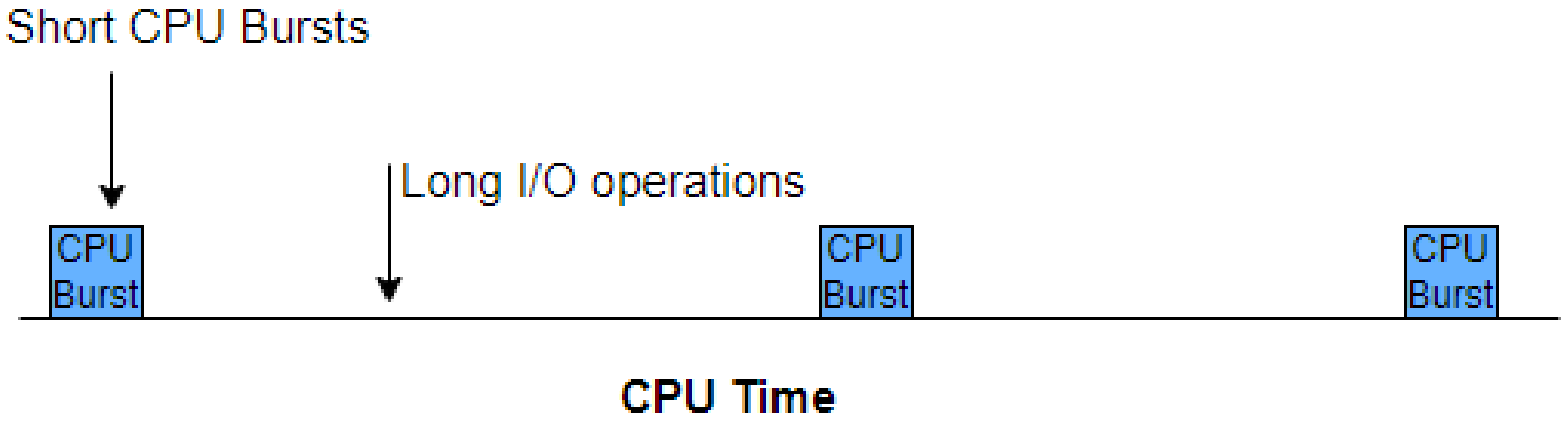
- **The scheduler** selects processes from the queues in some fashion
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- 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 strives for good ***process mix***

CPU Bound Vs IO Bound Processes

- CPU bound Process

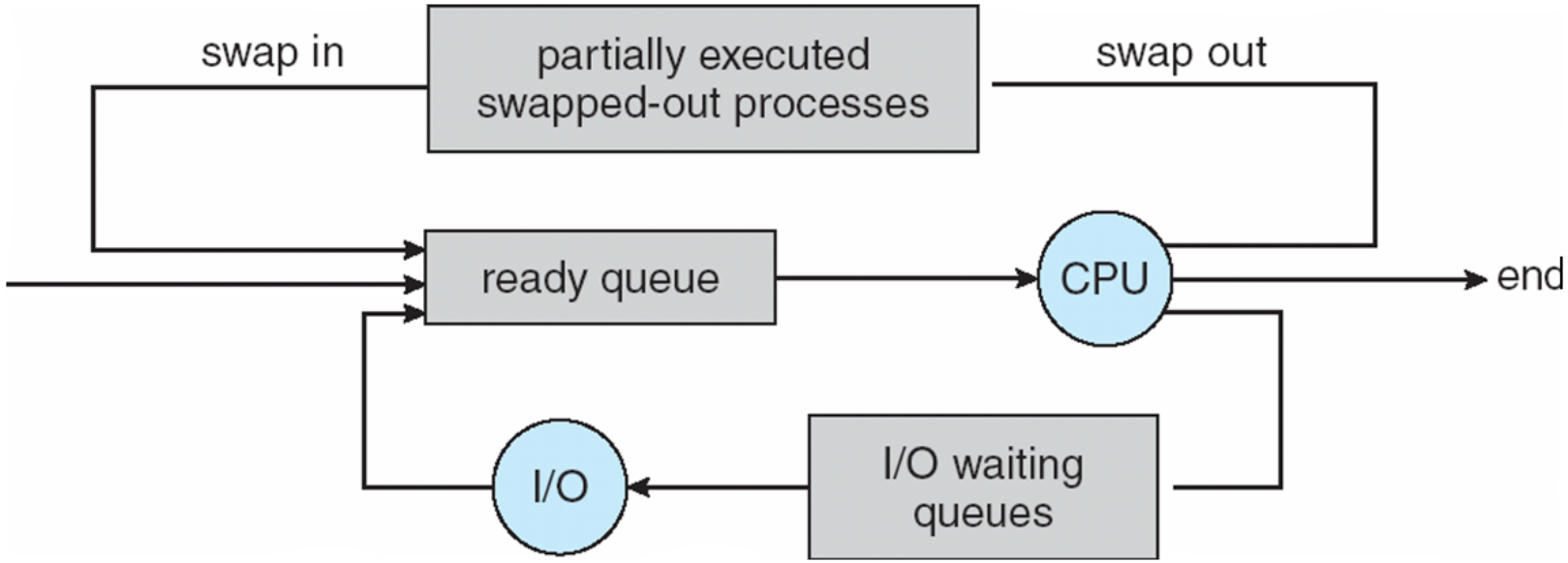


- IO Bound Process



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**



Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process is represented in the PCB
- Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **context switch**
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once

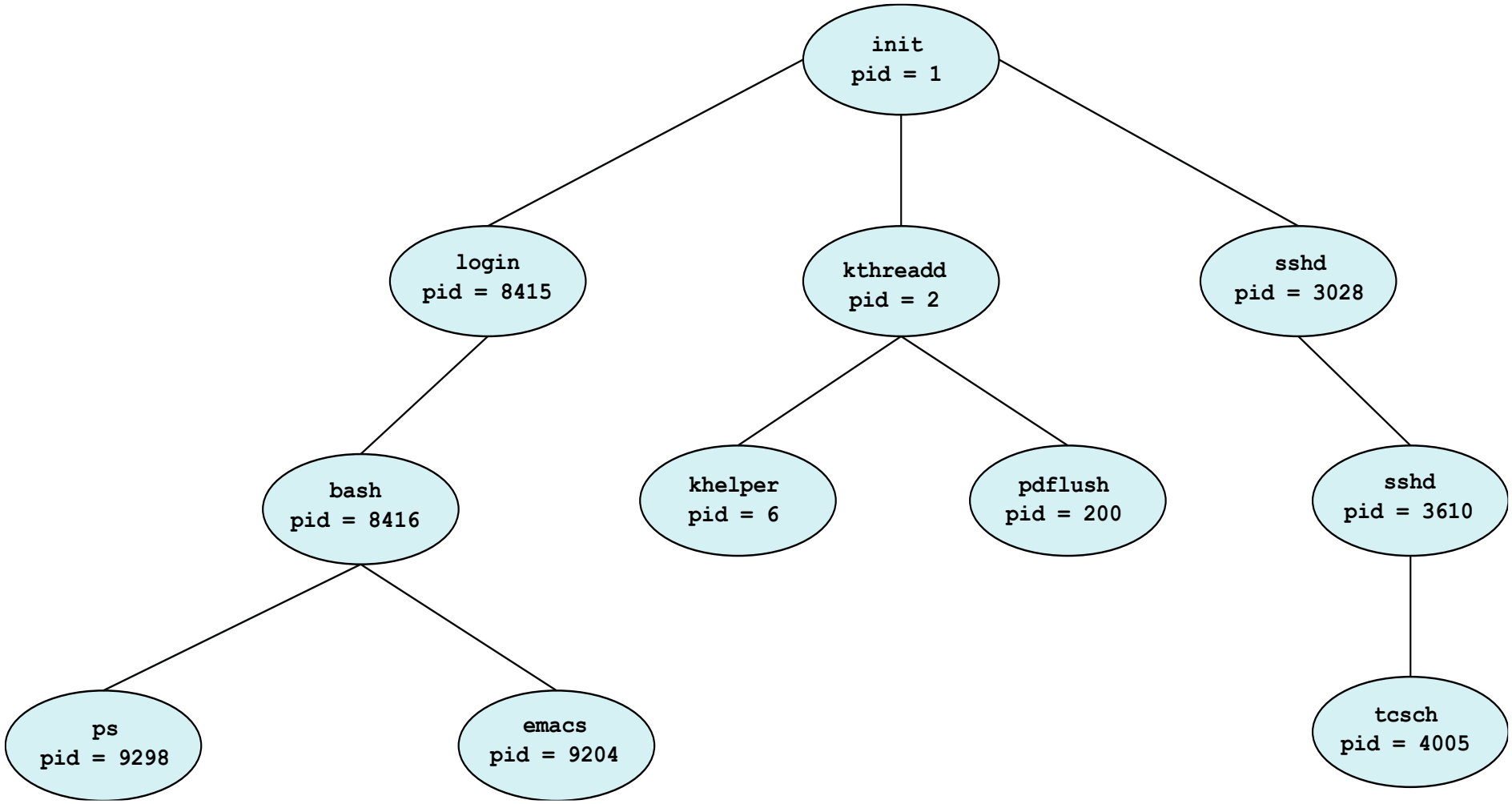
Operations on Processes

- System must provide mechanisms for:
  - Process creation
  - Process termination

Process Creation

- **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**)
- The pid is an integer, provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources. Child obtains resources directly from OS
- The parent process may pass along initialization data (input) to the child process.

A Tree of Processes in Linux



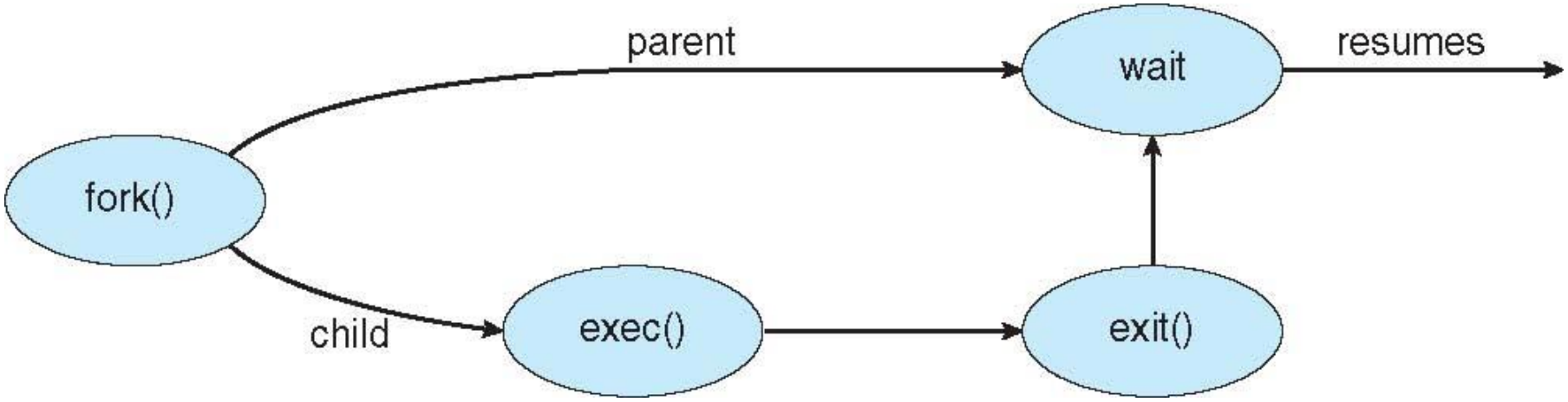
Process Creation (Cont.)

- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program

C Program Forking Separate Process

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
    }
    else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
    }

    return 0;
}
```

Process creation using the fork() system call



Creating a Separate Process via Windows API

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

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
       fprintf(stderr, "Create Process Failed");
       return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
- This phenomenon, referred to **cascading termination.**
  - All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **`wait()`** system call **.** The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

- When a process terminates, its resources are deallocated by the operating system

Process Termination…

- If no parent waiting (did not invoke **`wait()`**) process is a **zombie**
  - All processes transition to this state when they terminate, but generally they exist as zombies only briefly.
  - Once the parent calls wait(), the process identifier of the zombie process and its entry in the process table are released.

- If parent terminated without invoking **`wait`**, process is an **orphan**
  - Example: Linux and UNIX, the init process is assigned as the new parent to orphan processes.
  - Allows the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry

Interprocess Communication

Cooperating Processes

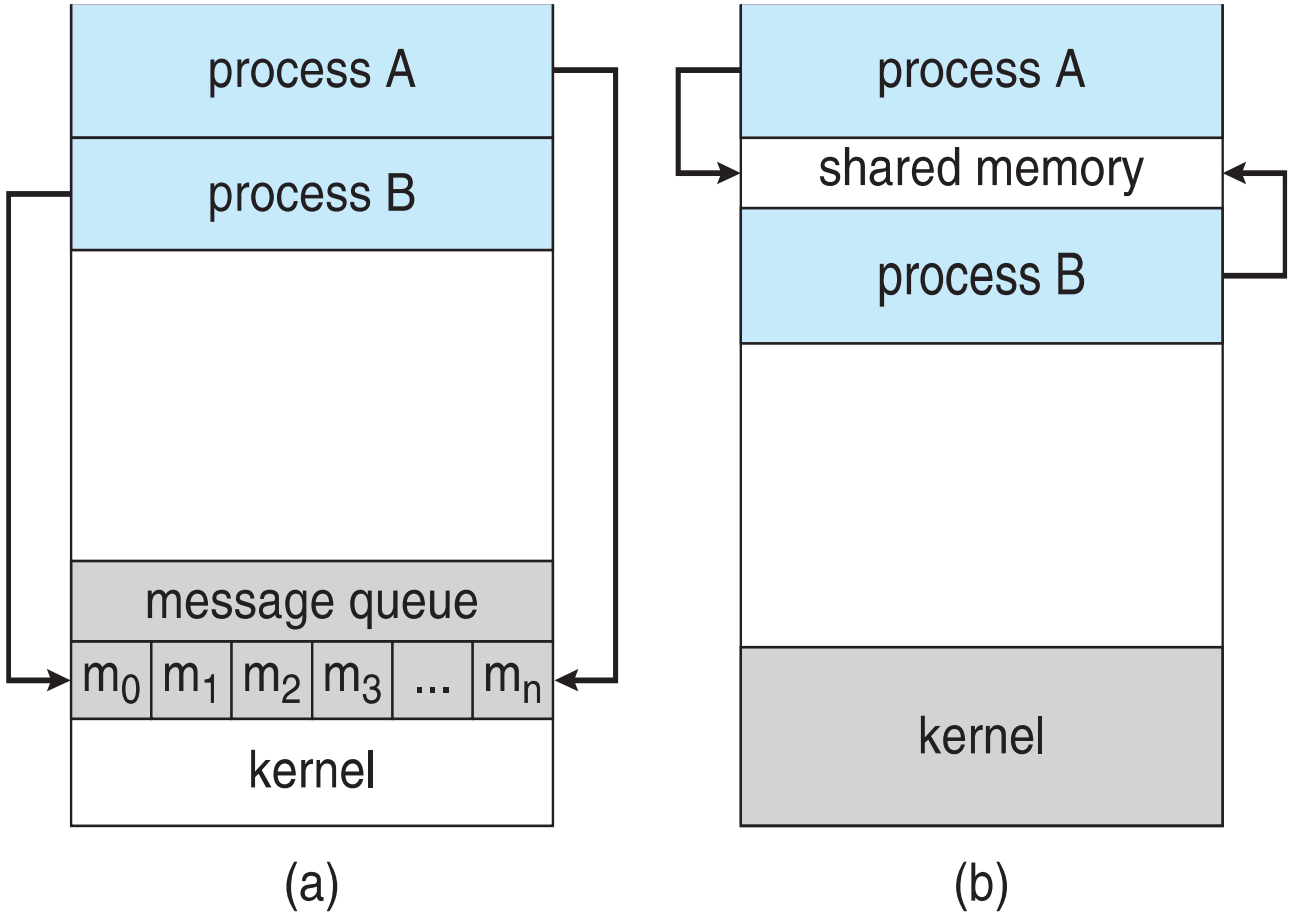
- Processes executing concurrently within a system may be ***independent*** or ***cooperating***
- ***Independent*** process cannot affect or be affected by the execution of another process
- ***Cooperating*** process can affect or be affected by the execution of another process, including sharing data
- Reasons(Advantages) for cooperating processes:
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Interprocess Communication

- Cooperating processes need **interprocess communication** (**IPC**) to exchange data and information
- Two models of IPC
  - **Shared memory**
    - a region of memory that is shared by cooperating processes is established.
    - Processes can then exchange information by reading and writing data to the shared region.
  - **Message passing**
    - communication takes place by means of messages exchanged between the cooperating processes.

Communications Models

(a) Message passing. (b) shared memory.



Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
- One solution to the producer–consumer problem uses shared memory.
- To allow producer and consumer processes to run concurrently, we must have a buffer of items
- The buffer will reside in a region of memory that is shared by the producer and consumer processes.
- A producer can produce one item while the consumer is consuming another item.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.
- Two types of buffers can be used:
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Bounded-Buffer – Producer

```
item next_produced;
while (true) {
      /* produce an item in next produced */
      while (((in + 1) % BUFFER_SIZE) == out)
            ; /* do nothing */
      buffer[in] = next_produced;
      in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer – Consumer

```
item next_consumed;
while (true) {
      while (in == out)
            ; /* do nothing */
      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;

      /* consume the item in next consumed */
}
```

Solution is correct, but can only use BUFFER_SIZE-1 elements

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without sharing the same address space
- Useful in a distributed environment
- A message-passing facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Message Passing (Cont.)

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

1.Naming -Direct Communication

- Processes that want to communicate must have a way to refer to each other.
- They can use either direct or indirect communication
- In direct communication, processes must name each other explicitly:
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically between every pair of processes.
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Direct Communication…

- The previous scheme exhibits **symmetry** in addressing
- **Asymmetry** in addressing- only the sender names the recipient; the recipient is not required to name the sender
  - **`send(P, message)`** –Send a message to process P
  - **`receive(id, message)`** –Receive a message from any process
- The disadvantage of direct communication
  - limited modularity of the resulting process definitions

1. Naming -Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
- A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication

- Primitives are defined as:

  `send`(*A, message*) – send a message to mailbox A

  `receive`(*A, message*) – receive a message from mailbox A

- A mailbox may be owned by a process or operating system.
- If owned by OS, then must allow the process to do the following operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox

Indirect Communication…

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

2.Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives a valid message or Null message
- Different combinations of send() and receive() are possible
- If both send and receive are blocking, we have a **rendezvous** between the sender and the receiver.

Synchronization (Cont.)

- Solution to Producer-consumer

```
message next_produced;
while (true) {
    /* produce an item in next produced */
send(next_produced);
}
```

```
message next_consumed;
while (true) {
   receive(next_consumed);

   /* consume the item in next consumed
*/
}
```

3.Buffering

- Queue of messages attached to the link.
- Implemented in one of three ways
  1. Zero capacity – The queue has maximum length of zero; no messages are queued on a link. Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages. Sender must wait if link full
  3. Unbounded capacity – infinite length .Sender never waits
- The zero-capacity case is sometimes referred to as a **message system with no buffering.**
- The other cases are referred to as systems with **automatic buffering**.

Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment
    ```
    shm_fd = shm_open(name, O CREAT | O RDWR, 0666);
    ```
  - Also used to open an existing segment to share it
  - Set the size of the object
    ```
    ftruncate(shm fd, 4096);
    ```
  - Now the process could write to the shared memory
    ```
    sprintf(shared memory, "Writing to shared memory");
    ```

IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

IPC POSIX Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

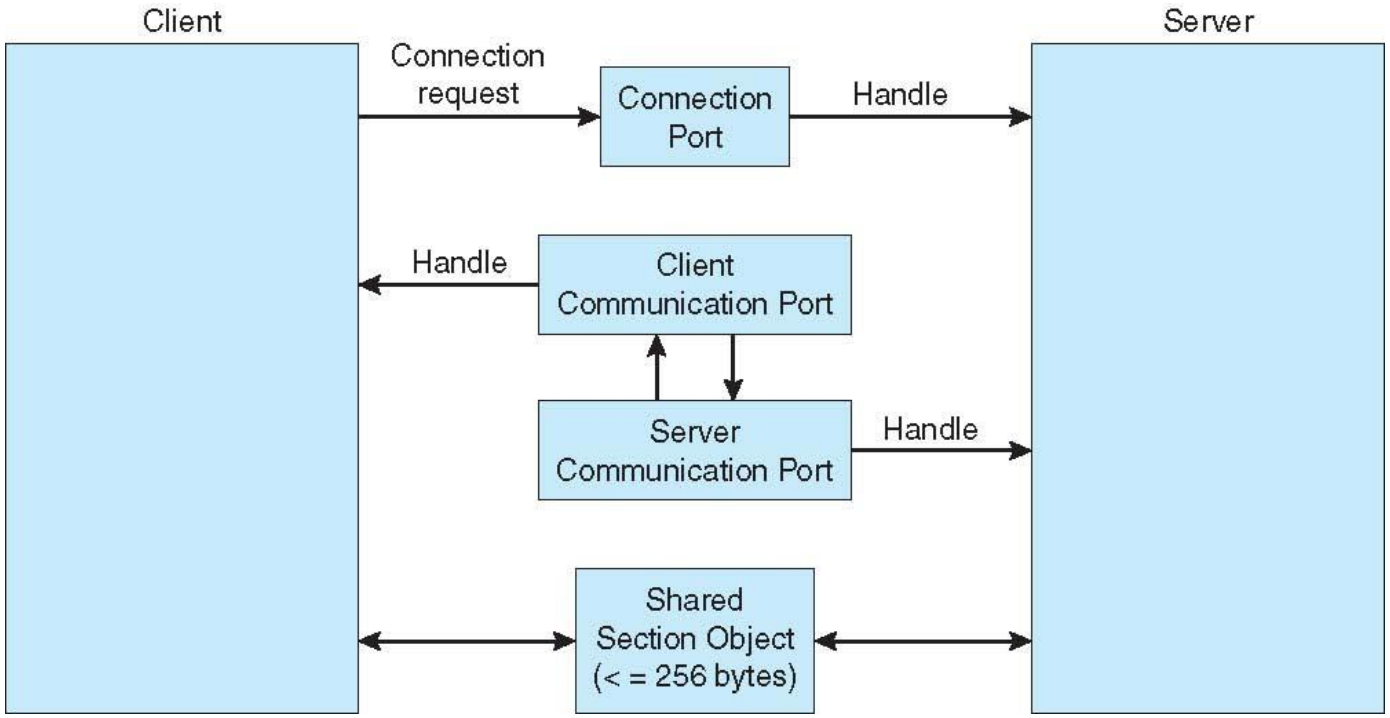
Examples of IPC Systems - Mach

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation- Kernel and Notify
  - Only three system calls needed for message transfer
    `msg_send(), msg_receive(), msg_rpc()`
  - Mailboxes needed for commuication, created via
    `port_allocate()`
  - Send and receive are flexible, for example four options if mailbox full:
    - Wait indefinitely
    - Wait at most n milliseconds
    - Return immediately
    - Temporarily cache a message

Examples of IPC Systems – Windows

- Message-passing centric via **advanced local procedure call** (**LPC**) facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - The client opens a handle to the subsystem's **connection port** object.
    - The client sends a connection request.
    - The server creates two private **communication ports** and returns the handle to one of them to the client.
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Local Procedure Calls in Windows



Threads

Overview

Multicore Programming

Multithreading Models

Thread Libraries

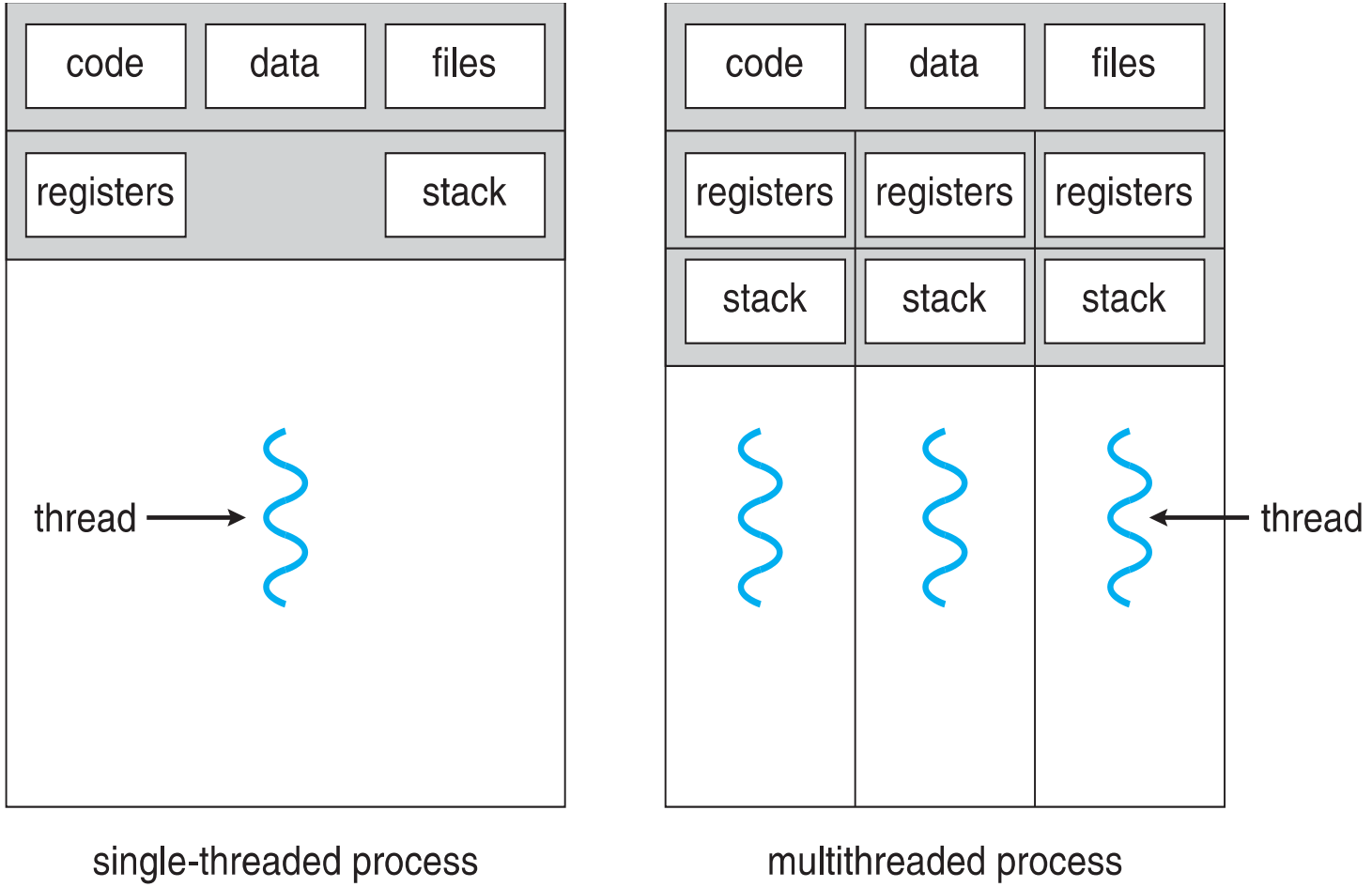
Implicit Threading

Threading Issues

Thread

- A **thread** is a flow of control within a process.
- A thread sometimes called **Lightweight Process** is a basic unit of CPU utilization
- It comprises a thread ID, a program counter, a register set, and a stack.
- It shares its code section, data section, and other operating-system resources with other threads belonging to the same process.
- A traditional (or heavyweight) process has a single thread of control.
- A process with multiple threads of control can perform more than one task at a time.

Single and Multithreaded Processes



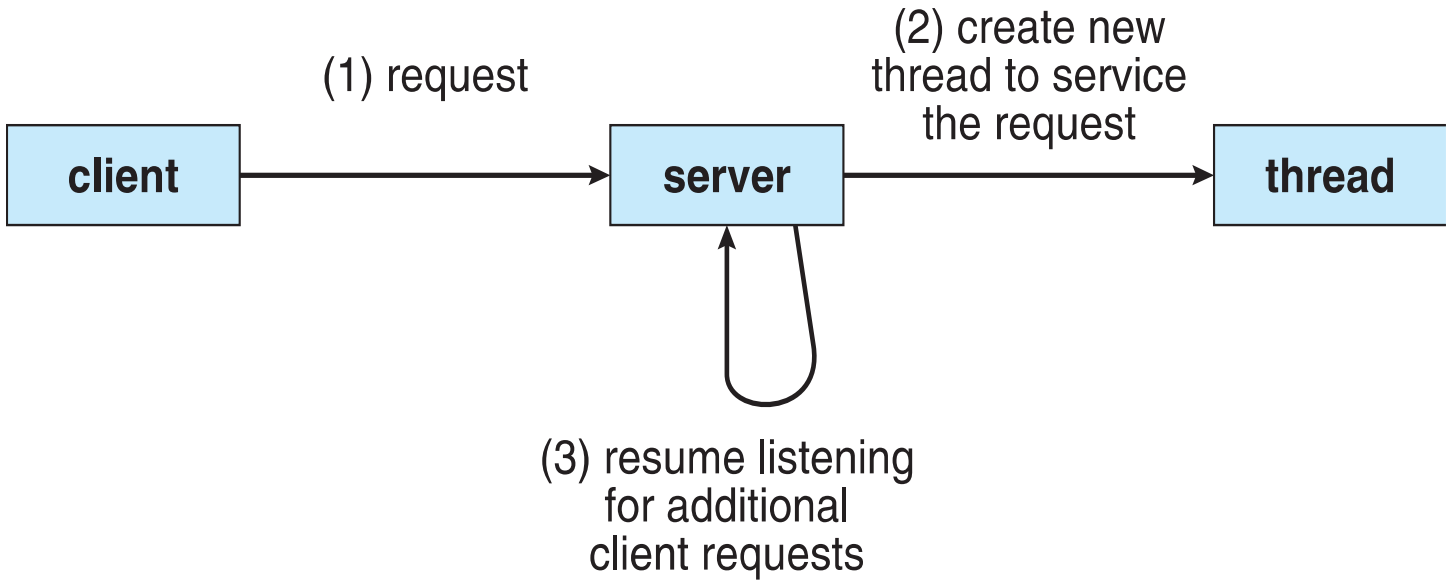
single-threaded process

multithreaded process

Motivation

- Most modern applications are multithreaded
- An application is implemented as a separate process with several threads of control.
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Applications are designed to leverage processing capabilities on multicore systems.
- A single application may be required to perform several similar tasks.
- Kernels are generally multithreaded

Multithreaded Server Architecture



- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency

Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

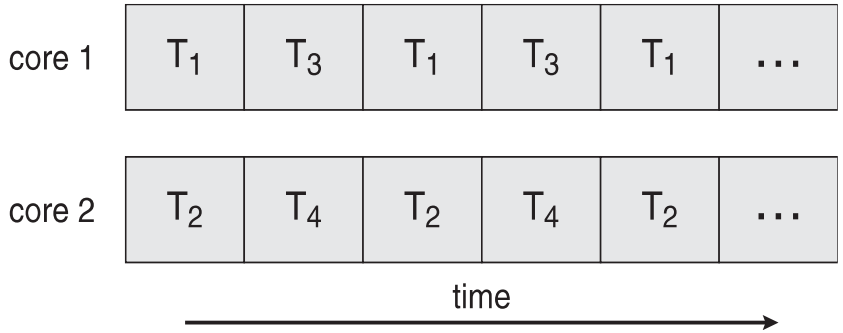
Multicore Programming

- A more recent in system design is to place multiple computing cores on a single chip.
- Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency.
- ***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
- As the number 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

Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**

- **Parallelism on a multi-core system:**



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*

The serial portion of an application has a disproportionate effect on performance gained by adding additional cores

Multicore Programming (Cont.)

- **Multicore** or **multiprocessor** systems are putting pressure on programmers
- For application programmers, the challenge is to modify existing programs as well as design new programs that are multithreaded.
- Challenges include:
  - **Identifying tasks/dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**

Multicore Programming (Cont.)

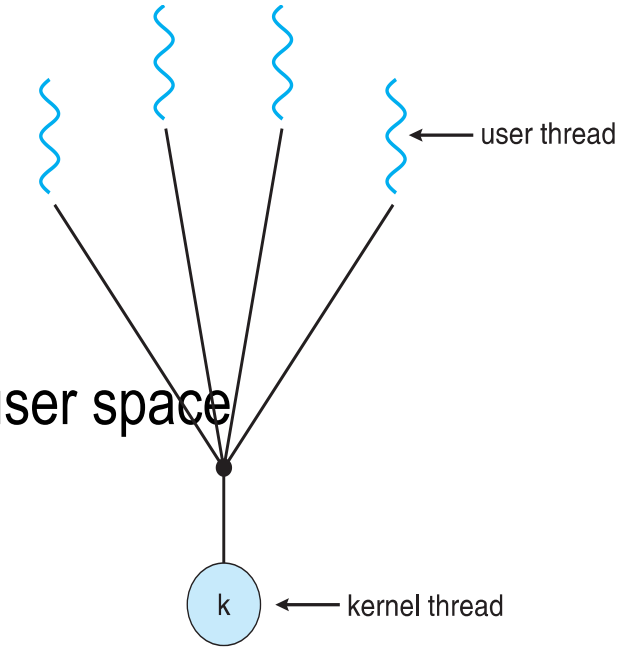
- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- In most instances, applications use a hybrid of these two strategies.

User Threads and Kernel Threads

- Threads may be implemented in two ways:
  - **User threads** are threads that are visible to the programmer and are unknown to the kernel
  - **Kernel threads** are supported and managed directly by the Kernel

User threads

- Supported above the kernel and management done by user-level threads library
- Examples:
  - POSIX Pthreads
  - Mach C-threads
  - Solaris 2 UI-threads
- Advantages
  - Efficient as thread creation and scheduling are done in user space
  - Faster to create and manage
- Limitation
  - One thread blocking causes all to block



Kernel Threads

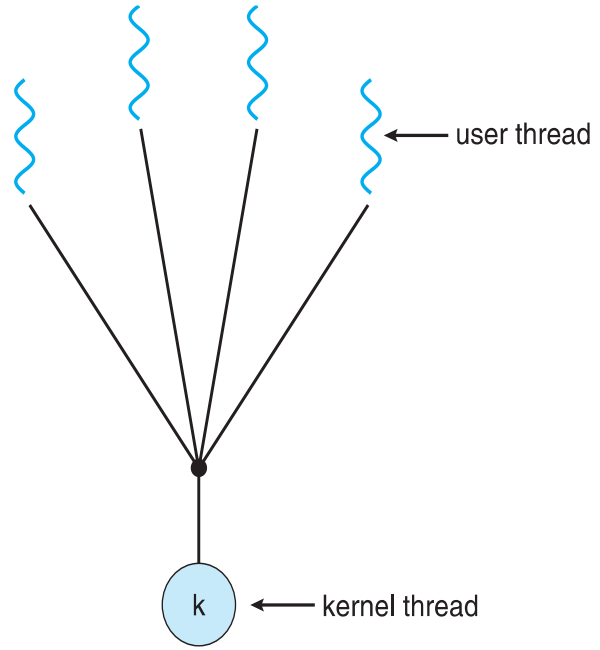
- Supported directly by the OS
- The kernel performs thread creation, management and scheduling in kernel space
- Examples: All contemporary operating systems—including Windows, Linux, Mac OS X , and Solaris— support kernel threads.
- Advantages
  - A blocking system call will not block other threads
  - In multiprocessor environment, kernel can schedule threads on different processors
- Limitation
  - Slower to create and manage

Multithreading Models

- Three different types of models relate user and kernel threads:
  - Many-to-One
  - One-to-One
  - Many-to-Many

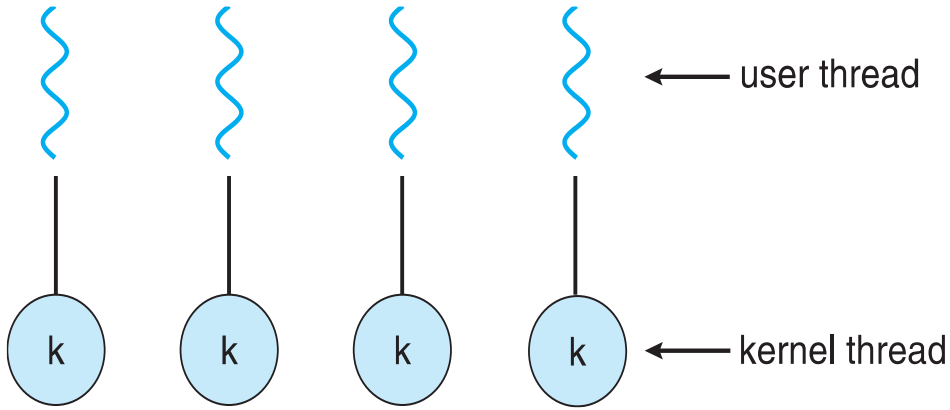
Many-to-One

- Many user-level threads mapped to single kernel thread
- Thread management is done by the thread library in user space,so it is efficient
- One thread blocking causes all to block
- Multiple threads may not run in parallel on a muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Example:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



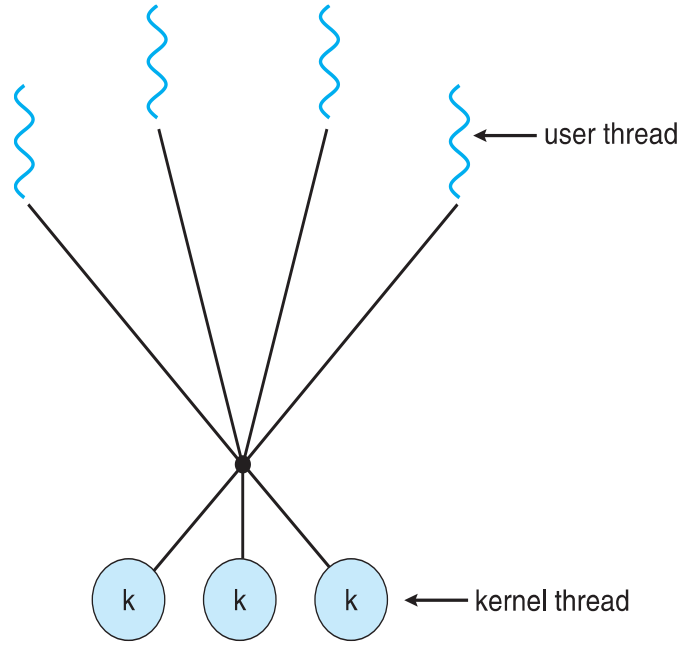
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



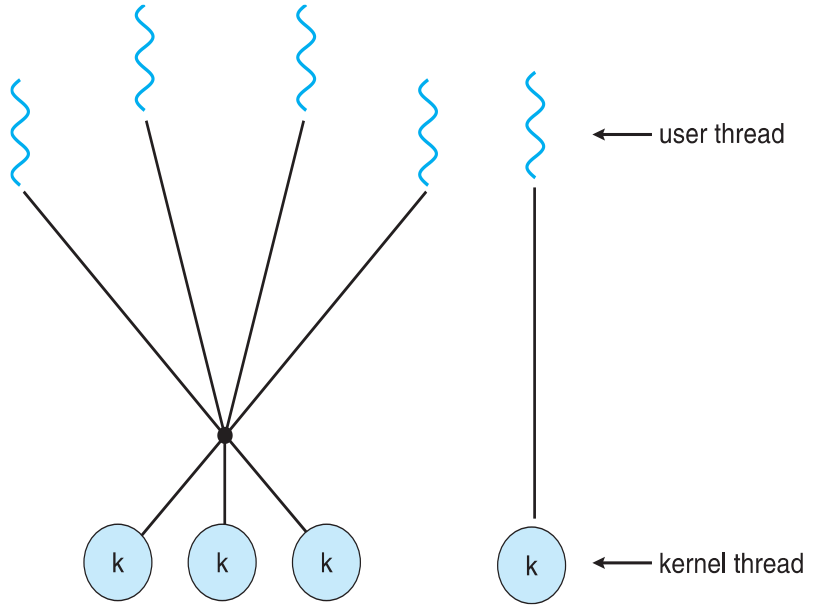
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
- When a thread performs a blocking system call, the kernel can schedule another thread for execution.
- Examples
  - Solaris prior to version 9
  - Windows with the *ThreadFiber* package



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing thread library:
  - Library entirely in user space
  - Kernel-level library supported by the OS
- Three main thread libraries are in use today:
  - POSIX Pthreads
  - Windows
    Java.
- Two general strategies for creating multiple threads:
  - asynchronous threading
  - synchronous threading

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads…

- **Thread Creation**
- Use the **pthread_create** function to create a new thread.
- The **pthread.h** header file includes its signature definition
- Syntax

```
int pthread_create(pthread_t *tid,   const pthread_attr_t
*attr,   void *(*func)(void *), void *arg)
```

  - The first argument is the variable where its thread ID will be stored
  - The second argument contains attributes describing the thread. You can usually just pass a NULL pointer.
  - The third argument is a pointer to the function you want to run as a thread.
  - The final argument is a pointer to data you want to pass to the function.

Pthreads…

- To exit from a thread, you can use the **pthread_exit** function.
  - Syntax: void pthread_exit(void *status)
- Use **pthread_join** function, to wait for a thread to terminate
  - Syntax: **int pthread_join(pthread_t tid, void **status)**
    - The first argument is the thread ID.
    - The second argument is a pointer to the data your thread function returned.

Pthreads…

- If you do not care about the return value, you can call the **pthread_detach** function with its thread ID as the only parameter to tell the system to discard the return value.
  - Syntax: int pthread_detach(pthread_t thread);
- Your thread function can use the **pthread_self** function to return its thread ID.
- If you don't want the return value, you can call **pthread_detach(pthread_self())** inside your thread function.

Example Program

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>.
#include <pthread.h>
void *myThreadFun(void *vargp)
{
        sleep(1);
        printf("Hello from Thread\n");
        return NULL;
}
int main()
{
        pthread_t thread_id;
        printf("Before Thread\n");
        pthread_create(&thread_id, NULL, myThreadFun, NULL);
        pthread_join(thread_id, NULL);
        printf("After Thread\n");
        exit(0);
}
```

Example Program…

- To compile a multithreaded program using gcc, we need to link it with the pthreads library.
- You can instruct the compiler to link to the library using the **-l** option as:

```
gcc multithread.c -lpthread
```

Pthreads Example

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

Pthreads Example (Cont.)

```
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Windows Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }
```

Windows Multithreaded C Program (Cont.)

```
   /* create the thread */
   ThreadHandle = CreateThread(
      NULL, /* default security attributes */
      0, /* default stack size */
      Summation, /* thread function */
      &Param, /* parameter to thread function */
      0, /* default creation flags */
      &ThreadId); /* returns the thread identifier */

   if (ThreadHandle != NULL) {
       /* now wait for the thread to finish */
      WaitForSingleObject(ThreadHandle,INFINITE);

      /* close the thread handle */
      CloseHandle(ThreadHandle);

      printf("sum = %d\n",Sum);
   }
}
```

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:

```
public interface Runnable
{
    public abstract void run();
}
```

  - Extending Thread class
  - Implementing the Runnable interface

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.)

```
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 {
         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>"); }
}
```

Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Three methods explored
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch
- Other methods include Microsoft Threading Building Blocks (TBB), **`java.util.concurrent`** package

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e.Tasks could be scheduled to run periodically
- The number of threads in the pool can be set heuristically based on factors:
  - number of CPUs in the system
  - the amount of physical memory
  - the expected number of concurrent client requests

Thread Pools…

- Windows API supports thread pools:

```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
    * this function runs as a separate thread.
    */
}
```

- One member is **QueueUserWorkItem()** function, which is passed three parameters:
  - LPTHREAD START ROUTINE Function —a pointer to the function that is to run as a separate thread
  - PVOID Param —the parameter passed to Function
  - ULONG Flags —flags indicating how the thread pool is to create and manage execution of the thread
- An example of invoking a function is the following:

  QueueUserWorkItem(&PoolFunction, NULL, 0);

OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

```
#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;
}
```

OpenMP…

- Provides directives for parallelization

```
#pragma omp parallel
```

Create as many threads as there are cores

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

- Allows developers to choose among several levels of parallelism.
- Open MP is available on several open-source and commercial compilers for Linux, Windows, and Mac OS X systems

Grand Central Dispatch (GCD)

- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++ languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- GCD identifies extensions to the C and C++ languages known as **blocks**.
- Block is in "^{ }" - `^{ printf("I am a block"); }`
- GCD schedules blocks for run-time execution by placing them on a **dispatch queue**
  - Assigned to available thread in thread pool when removed from queue

Grand Central Dispatch

- Two types of dispatch queues:
  - serial – blocks removed in FIFO order
    - Each process has its own serial queue called **main queue**
    - Programmers can create additional serial queues within program
    - useful for ensuring the sequential execution of several tasks
  - concurrent – removed in FIFO order but several may be removed at a time; allowing multiple blocks to execute in parallel
    - Three system wide queues with priorities low, default, high

```
dispatch_queue_t queue = dispatch_get_global_queue
   (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_async(queue, ^{ printf("I am a block."); });
```

- GCD ’s thread pool is composed of POSIX threads;actively manages the pool

Other Approaches

- Parallel and concurrent libraries
  - Intel's Threading Building Blocks (TBB) and several products from Microsoft.
- The Java language and API have seen significant movement toward supporting concurrent programming as well.
  - Example: java.util.concurrent package, which supports implicit thread creation and management.

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

Semantics of fork() and exec()

- The semantics of the fork() and exec() system calls change in a multithreaded program
- Does **`fork()`** duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
- **`exec()`** usually works as normal – replace the running process including all threads
- the versions of fork() to use depends on the application.
  - If exec() is called immediately after forking, then duplicate only the calling thread
  - If the process does not call exec() after forking, the process should duplicate all threads.

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A signal may be received either synchronously or asynchronously
  - depends on the source of and the reason for the event being signaled
- All signals follow the same pattern:
  1. A Signal is generated by a particular event
  2. The Signal is delivered to a process
  3. Once delivered, the signal must be handled

Signal Handling…

- Synchronous signals
  - When a signal is generated by a running process
  - Examples of synchronous signals include illegal memory access and division by 0.
  - Delivered to the same process that performed the operation that caused the signal
- Asynchronous signals
  - When a signal is generated by an event external to a running process, that process receives the signal asynchronously.
  - Examples of such signals include terminating a process with specific keystrokes (such as < control >< C >) and having a timer expire.
  - An asynchronous signal is sent to another process.

Signal Handling…

- A **signal handler** is used to process signals
- Signal is handled by one of two signal handlers:
  - default
  - user-defined
- Every signal has **default handler** that kernel runs when handling signal
- **User-defined signal handler** can override default
- Signals are handled in different ways.
  - Some signals (such as changing the size of a window) are simply ignored;
  - others (such as an illegal memory access) are handled by terminating the program
- For single-threaded, the signal is delivered to process

Signal Handling (Cont.)

- Where should a signal be delivered for multi-threaded?
  - 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
- The method for delivering a signal depends on the type of signal generated.
  - Synchronous signals need to be delivered to the thread causing the signal and not to other threads in the process
  - Some asynchronous signals, such as a signal that terminates a process should be sent to all threads

Signal Handling (Cont.)

- The standard UNIX function for delivering a signal is
  kill(pid t pid, int signal)
- Pthreads provides the following function:
  pthread kill(pthread t tid, int signal)
- Windows allows us to emulate signals using **Asynchronous Procedure Calls (APCs)**
  - An APC is roughly equivalent to an asynchronous signal in UNIX
  - APC is delivered to a particular thread rather than a process.

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Canceling a thread asynchronously may not free a necessary system-wide resource
- With deferred cancellation, cancellation occurs only after the target thread has checked a flag to determine whether or not it should be canceled.
  - The thread can perform this check at a point at which it can be canceled safely.

Thread Cancellation…

- In Pthreads, thread cancellation is initiated using the **pthread cancel()** function.
- The identifier of the target thread is passed as a parameter to the function.
- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state
- Pthreads supports three cancellation modes.

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

```
while (1) {
   /* do some work for awhile */
   /* . . . */

   /* check if there is a cancellation request */
   pthread_testcancel();
}
```

- Pthreads allows threads to disable or enable cancellation
- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - I.e. **`pthread_testcancel()`**
    - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals

Thread-Local Storage

- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- To associate each thread with its unique identifier, we could use thread-local storage
- Different from local variables
  - Local variables are visible only during single function invocation
  - TLS are visible across function invocations
- Similar to `static` data
  - TLS is unique to each thread
- Most thread provide some form of support for thread-local storage

Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process** (**LWP**)
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads

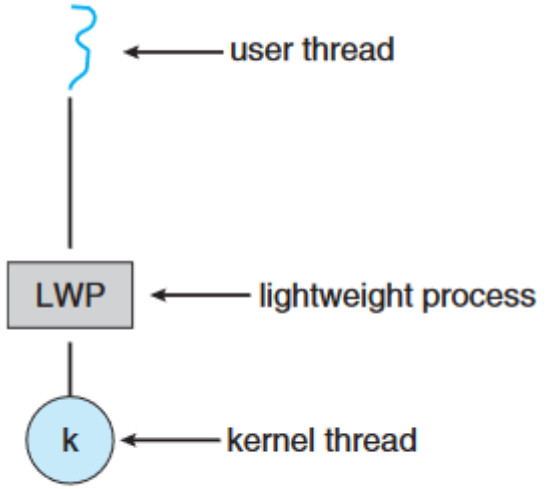
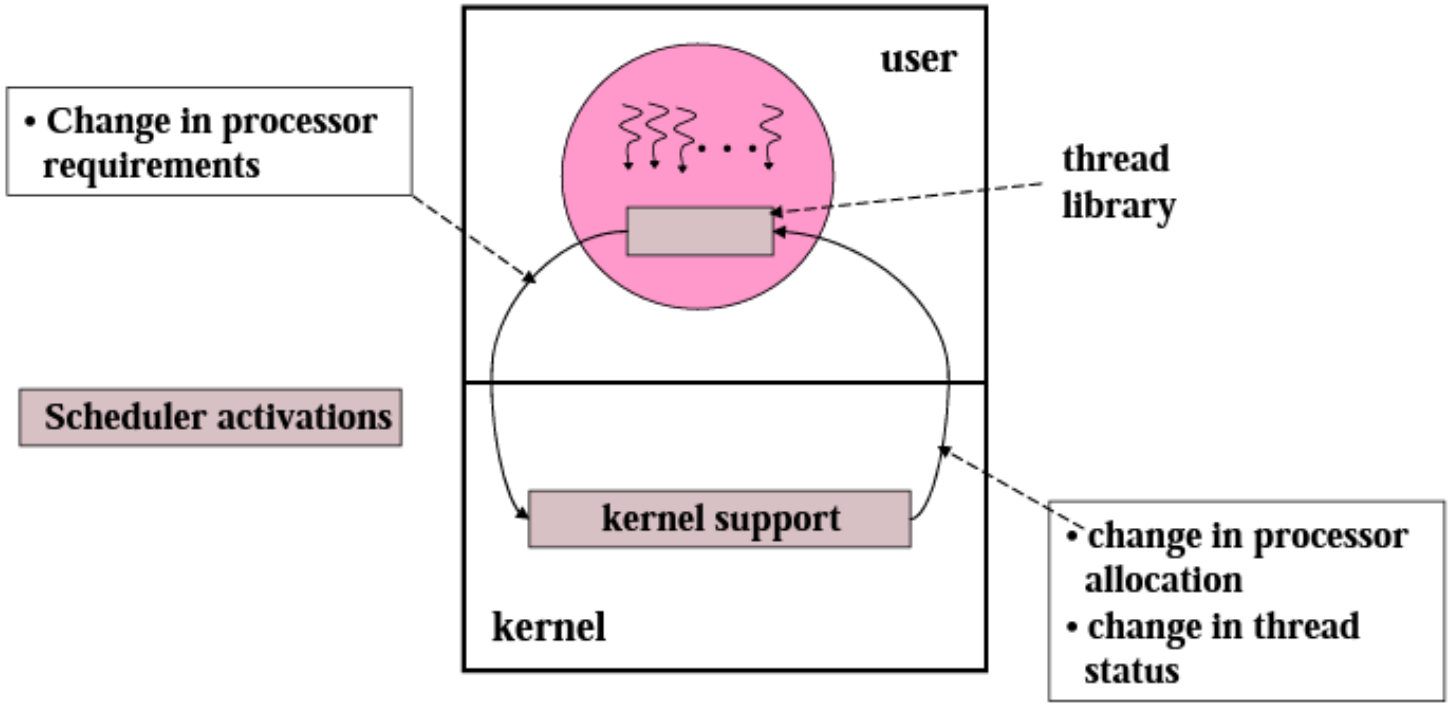
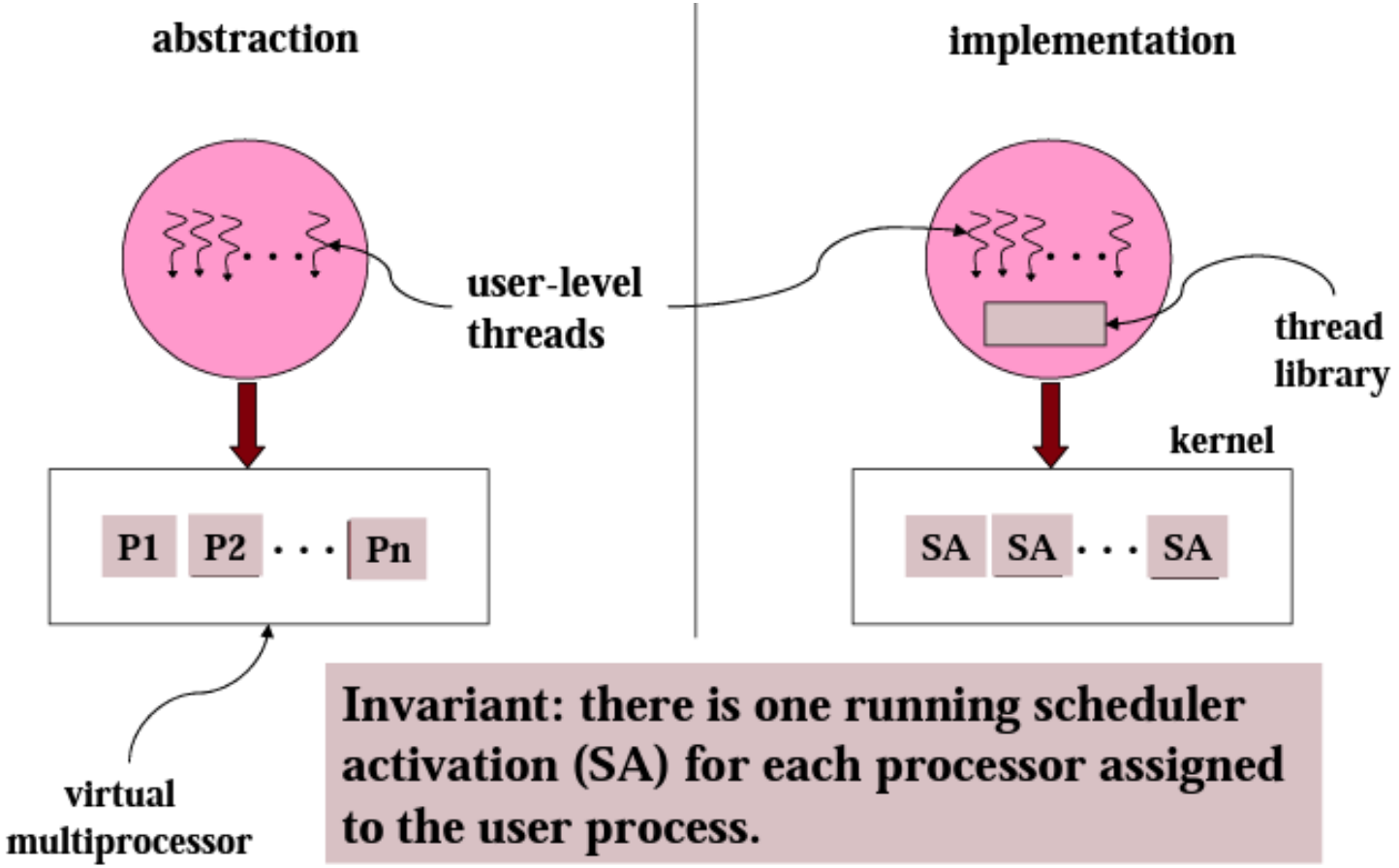


Figure 4.13 Lightweight process (LWP).

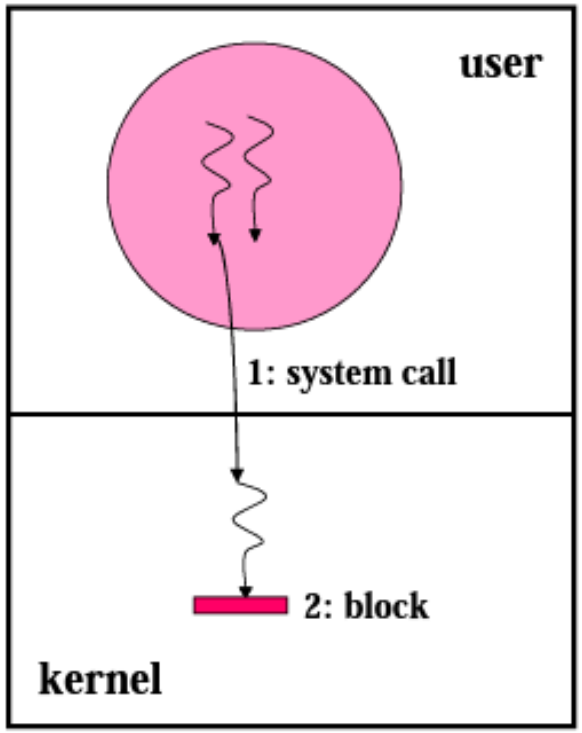
Scheduler Activation: Structure



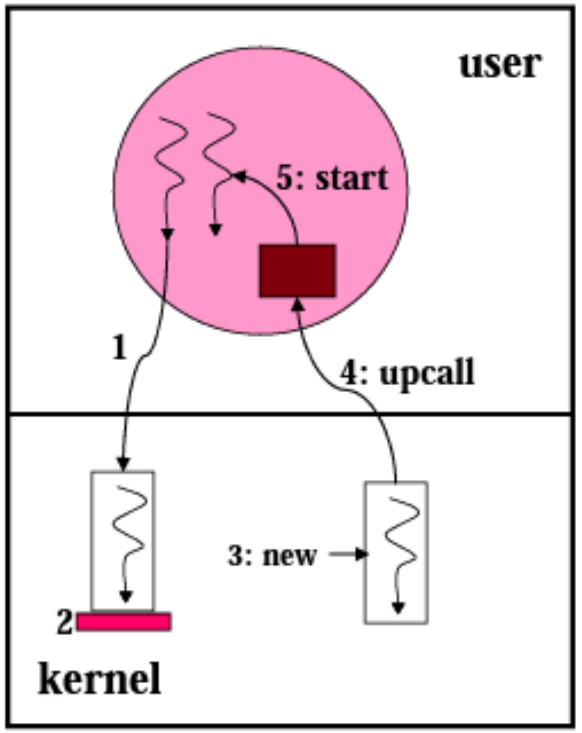
Role of Scheduler Activations



Avoiding Effects of Blocking

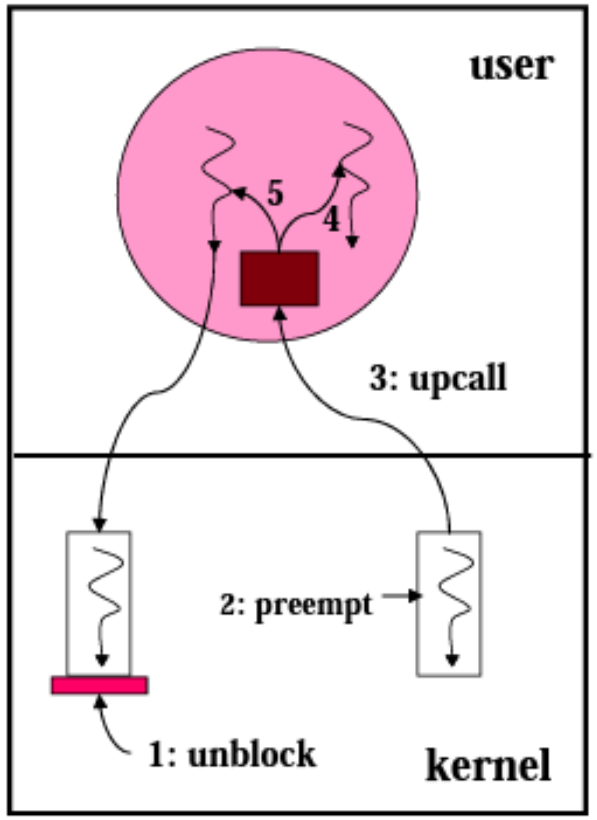


Kernel threads



Scheduler Activations

Resuming Blocked Thread



4: preempt
5: resume

Operating System Examples

- Windows Threads
- Linux Threads

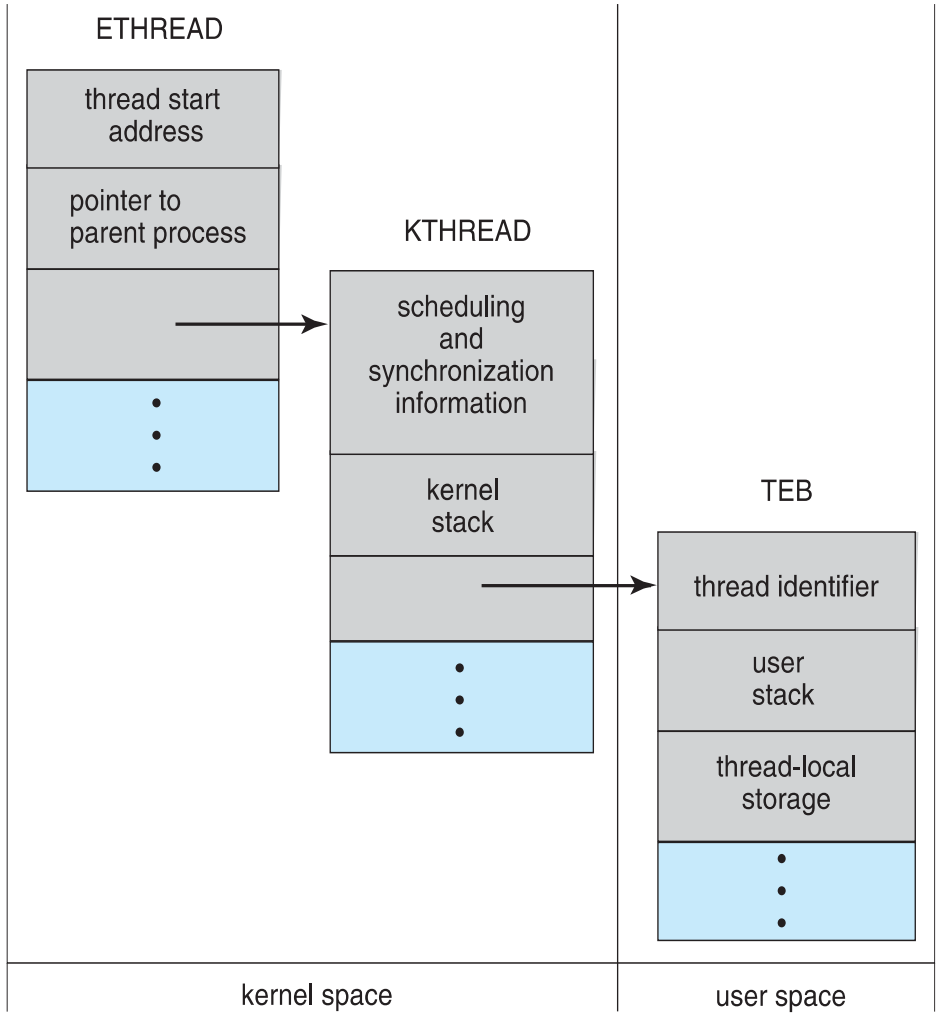
Windows Threads

- Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7
- A Windows application runs as a separate process, and each process may contain one or more threads.
- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set representing state of processor
  - Separate user and kernel stacks for the thread that runs in user mode or kernel mode
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context** of the thread

Windows Threads (Cont.)

- The primary data structures of a thread include:
  - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
  - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
  - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space

Windows Threads Data Structures



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)
  - When clone() is invoked, it is passed a set of flags that determine how much sharing is to take place between the parent and child tasks.
  - Flags control behavior

flag	meaning
`CLONE_FS`	File-system information is shared.
`CLONE_VM`	The same memory space is shared.
`CLONE_SIGHAND`	Signal handlers are shared.
`CLONE_FILES`	The set of open files is shared.

- Varying level of sharing is possible
- A unique kernel data structure **`struct task_struct`** exists for each task in the system
  - points to process data structures (shared or unique)

Review Questions

- Describe the differences among short-term, medium-term, and long-term scheduling.
- Describe the actions taken by a kernel to context-switch between processes
- Write about the following aspects of interprocess communication.
  a. Synchronous and asynchronous communication
  b. Automatic and explicit buffering
  c. Send by copy and send by reference
  d. Fixed-sized and variable-sized messages
- Give the differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?
- Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system? Explain.
- Write a multithreaded program that outputs prime numbers.