EECS 111:

System Software

Lecture: Processes and Threads

Prof. Mohammad Al Faruque

The Henry Samueli School of Engineering
Electrical Engineering & Computer Science
University of California Irvine (UCI)

Outline

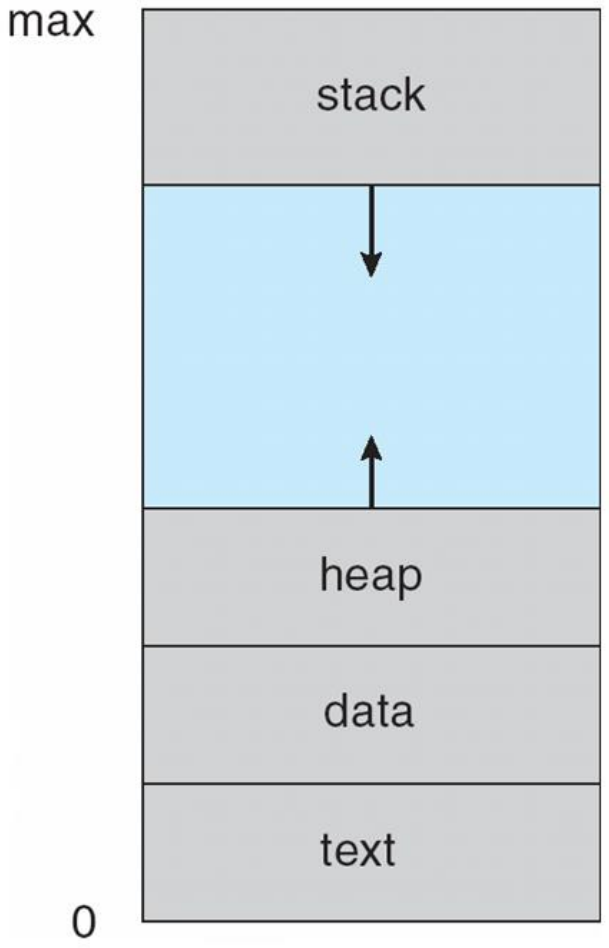
- Process Concept
- Process Scheduling
- Operations on Processes
- Threads
- Interprocess Communication
- Examples of IPC Systems

Process Concept

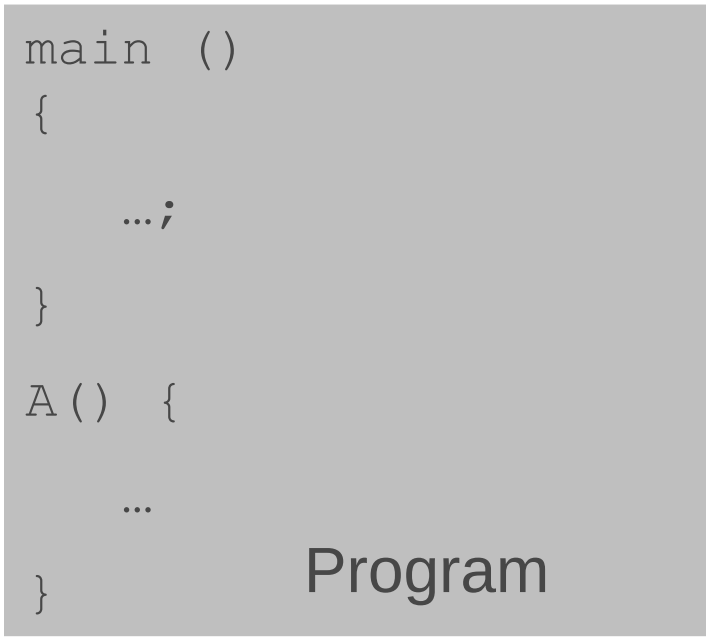
- **An operating system executes a variety of programs**
  - batch systems - jobs
  - time-shared systems - user programs or tasks
  - Textbook uses the terms job and program used interchangeably
- **Process - a program in execution**
  - process execution proceeds in a sequential fashion
- **A process contains**
  - The program code, also called text section
  - Current activity including program counter, processor registers
  - Stack containing temporary data
  - Function parameters, return addresses, local variables
  - Data section containing global variables
  - Heap containing memory dynamically allocated during run time

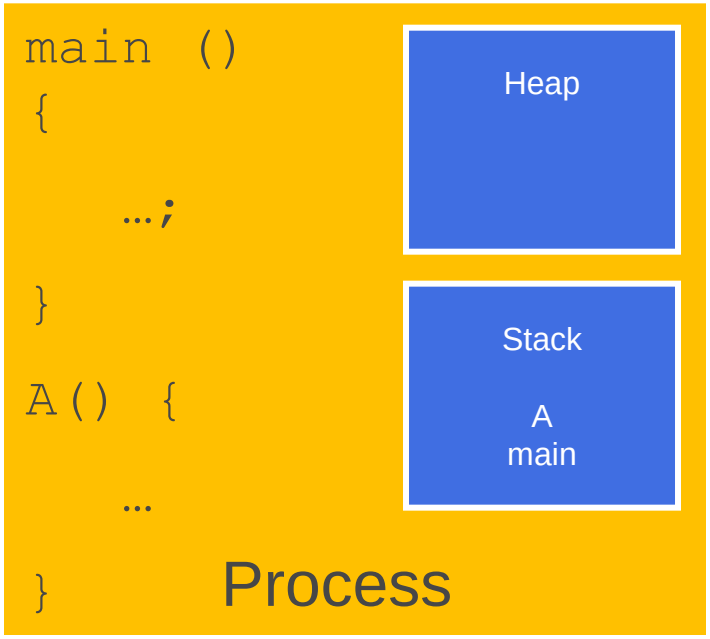
Process Concept

- Program is *passive* entity stored on disk (executable file), process is *active*
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program → can be several processes
  - Consider multiple users executing the same program



Process =? Program





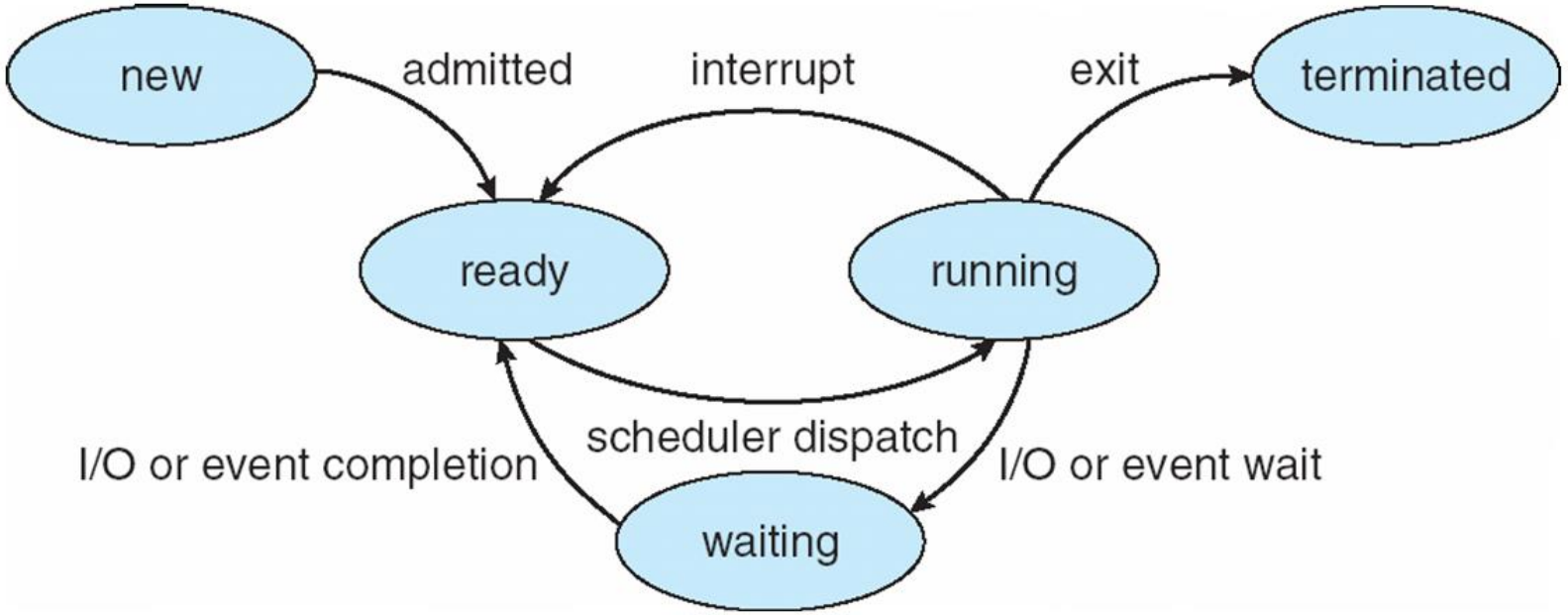
- **More to a process than just a program:**
  - **Program is just part of the process state**
  - **I run emacs/Notepad on lectures.txt, you run it on homework.java – Same program, different processes**
- **Less to a process than a program:**
  - **A program can invoke more than one process**
  - **cc/cpp starts up processes to handle different stages of the compilation process cc1, cc2, as, and ld**

Source Prof. Nalini, UCI

Process State

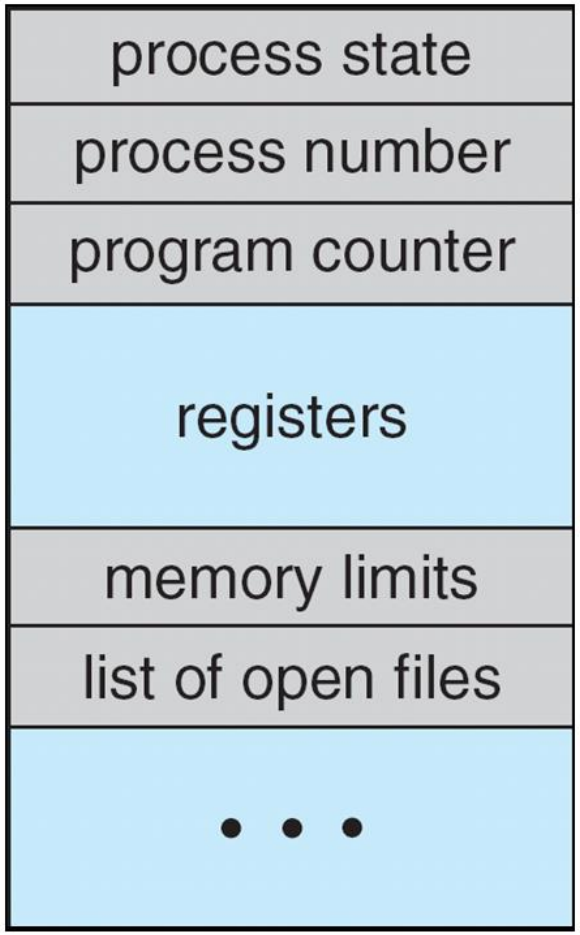
- As a process executes, it changes state
  - new: The process is being created
  - running: Instructions are being executed
  - waiting: The process is waiting for some event to occur
  - ready: The process is waiting to be assigned to a processor
  - terminated: The process has finished execution

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, page tables, value of the base, etc.
  - 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

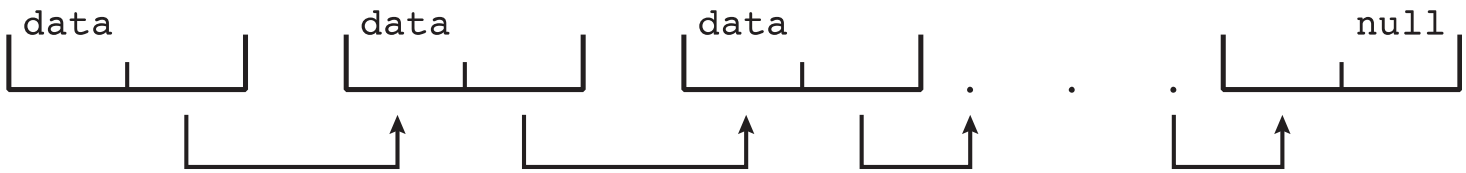


Data Structures

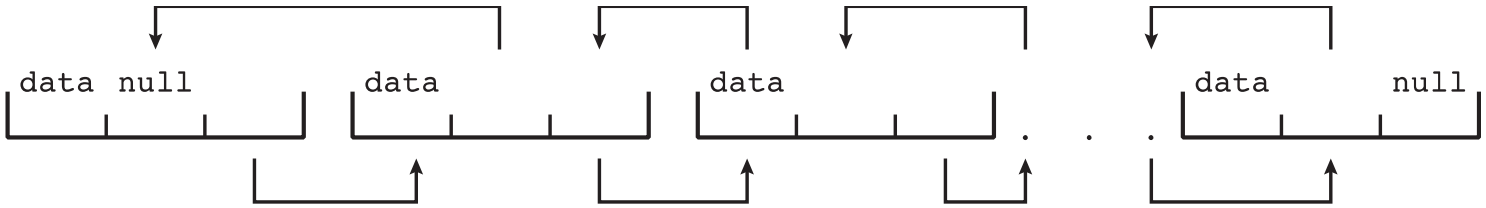
- **We need various data structures to manipulate these information efficiently.**

Data Structures

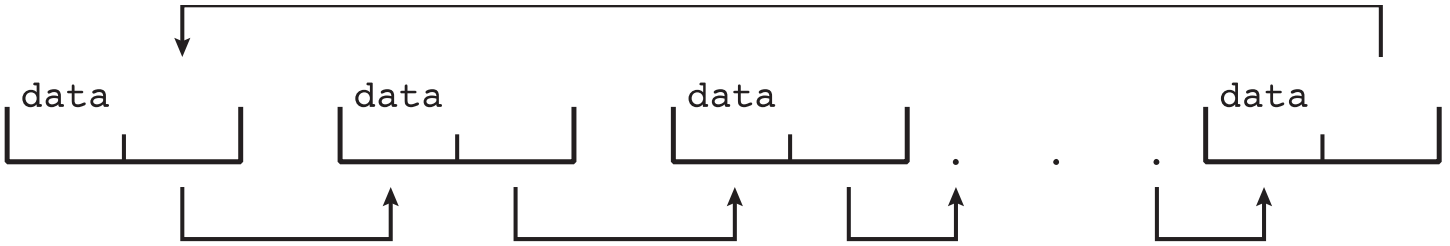
- ***Singly linked list***



- ***Doubly linked list***

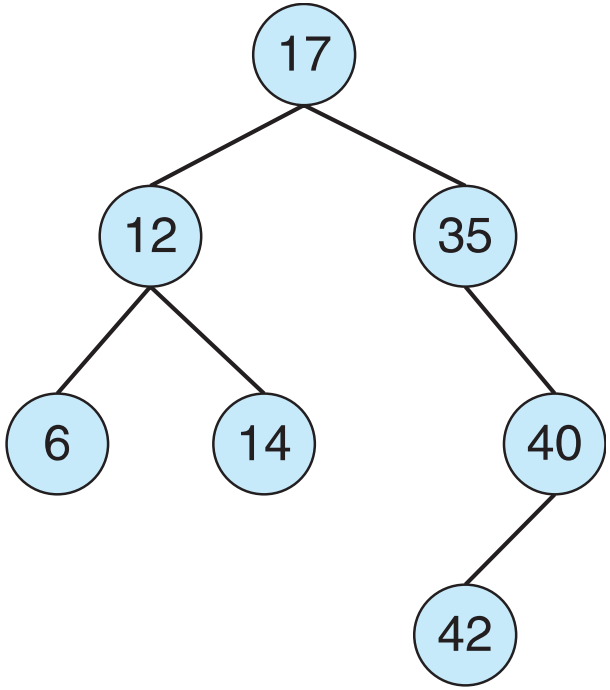


- ***Circular linked list***



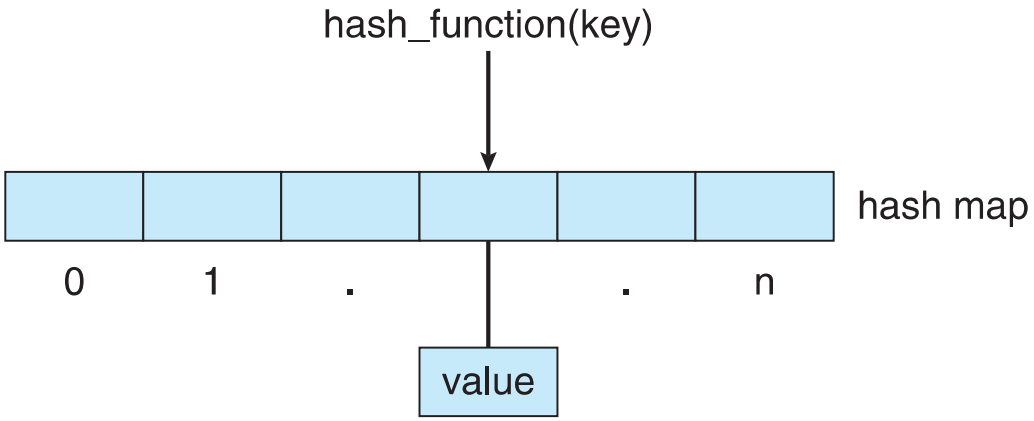
Data Structures

- **Binary search tree**
  left <= right
  - **Search performance is *O(n)***
  - **Balanced binary search tree is *O(lg n)***



Data Structures

- **Hash function can create a hash map**



- **Bitmap – string of *n* binary digits representing the status of *n* items**
- **Linux data structures defined in *include* files `<linux/list.h>`, `<linux/kfifo.h>`, `<linux/rbtree.h>`**

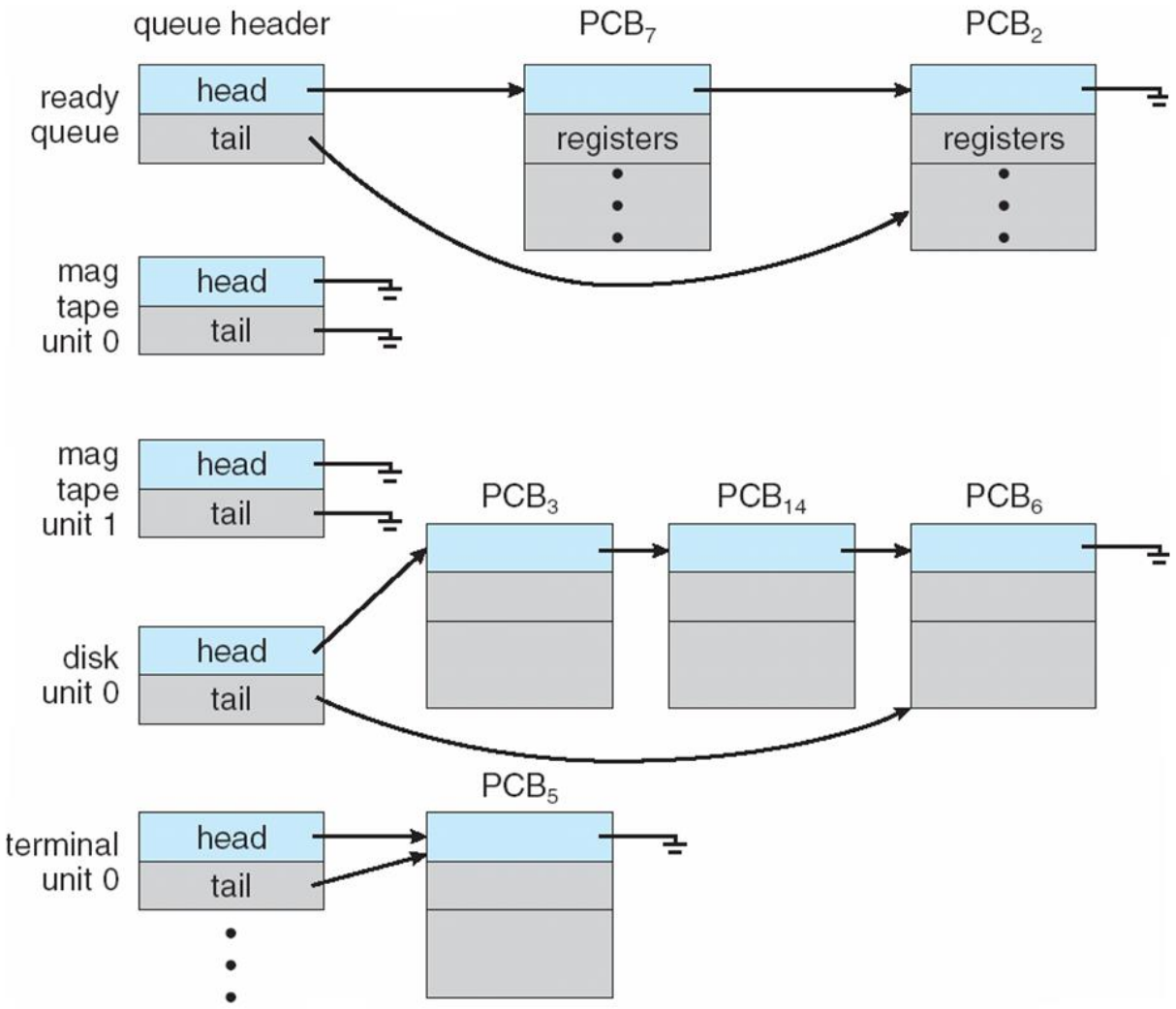
Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Threads
- Interprocess Communication
- Examples of IPC Systems

Process Scheduling Queues

- **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.**
- **Process migration between the various queues.**
- **Queue Structures - typically linked list, circular list etc.**

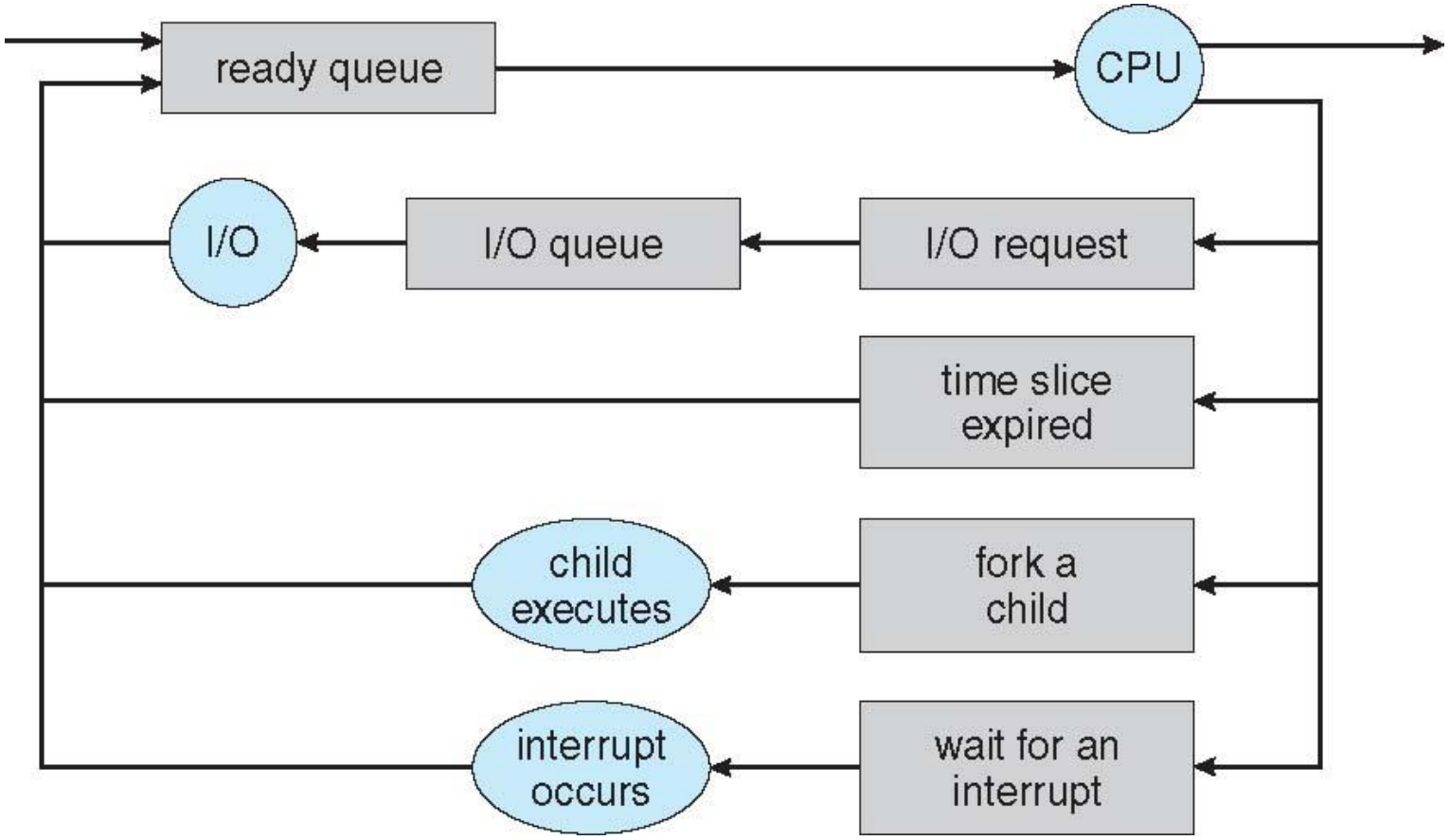
Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

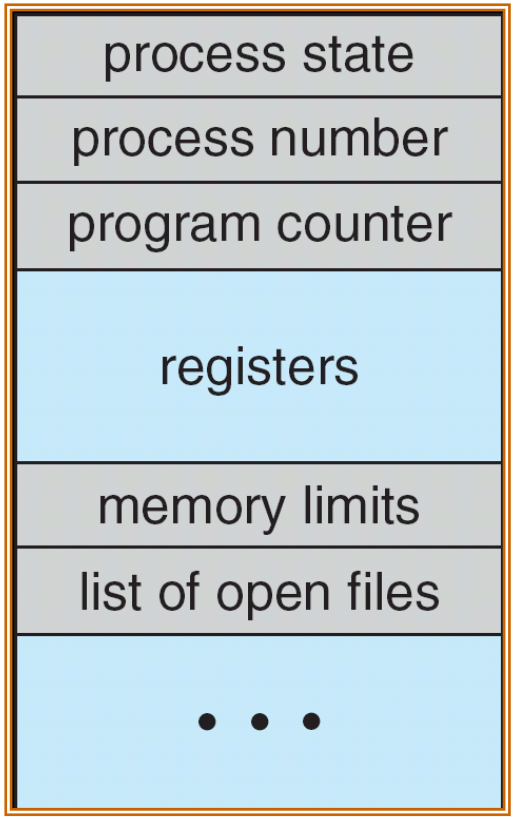
Process (PCB) moves from queue to queue

When does it move? Where? A scheduling decision



Enabling Concurrency and Protection: Multiplex processes

- **Only one process (PCB) active at a time**
  - **Current state of process held in PCB:**
    - "snapshot" of the execution and protection environment
  - **Process needs CPU, resources**
- **Give out CPU time to different processes (Scheduling):**
  - **Only one process "running" at a time**
  - **Give more time to important processes**
- **Give pieces of resources to different processes (Protection):**
  - **Controlled access to non-CPU resources**
    - E.g. Memory Mapping: Give each process their own address space

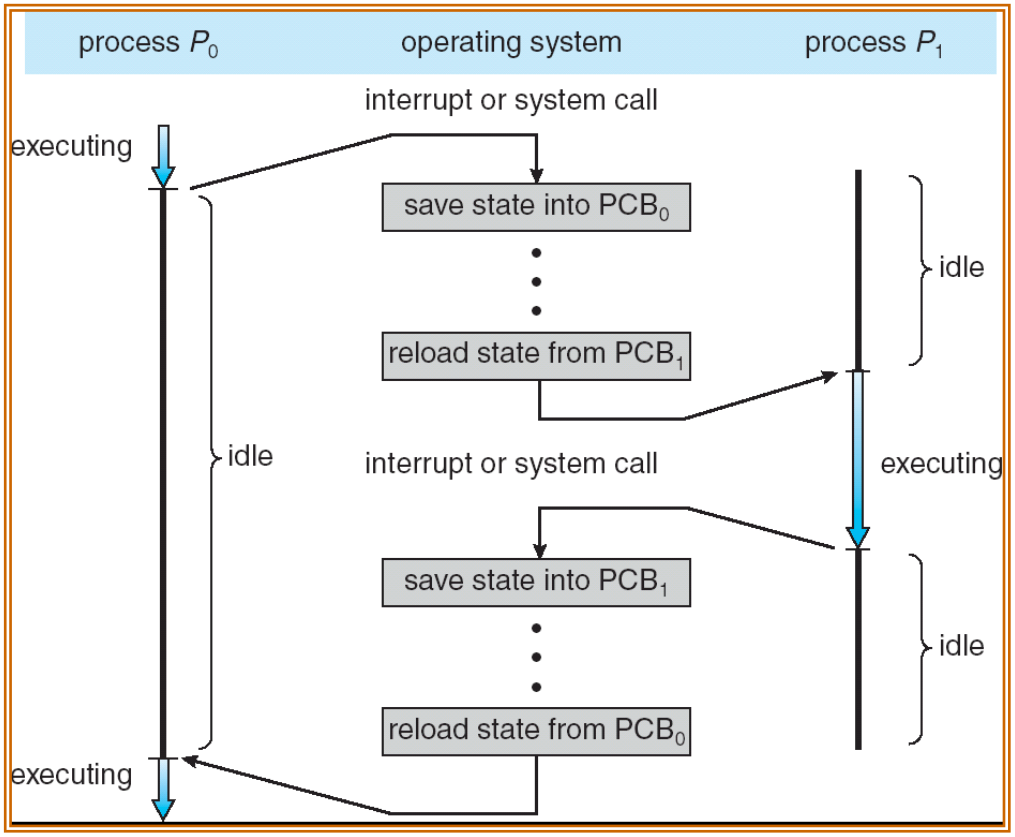


Process Control Block

Enabling Concurrency: Context Switch

- **Task that switches CPU from one process to another process**
  - the CPU must save the PCB state of the old process and load the saved PCB state of the new process.
- **Context-switch time is overhead**
  - System does no useful work while switching
  - Overhead sets minimum practical switching time; can become a bottleneck
- **Time for context switch is dependent on hardware support (1- 1000 microseconds).**

CPU Switch From Process to Process



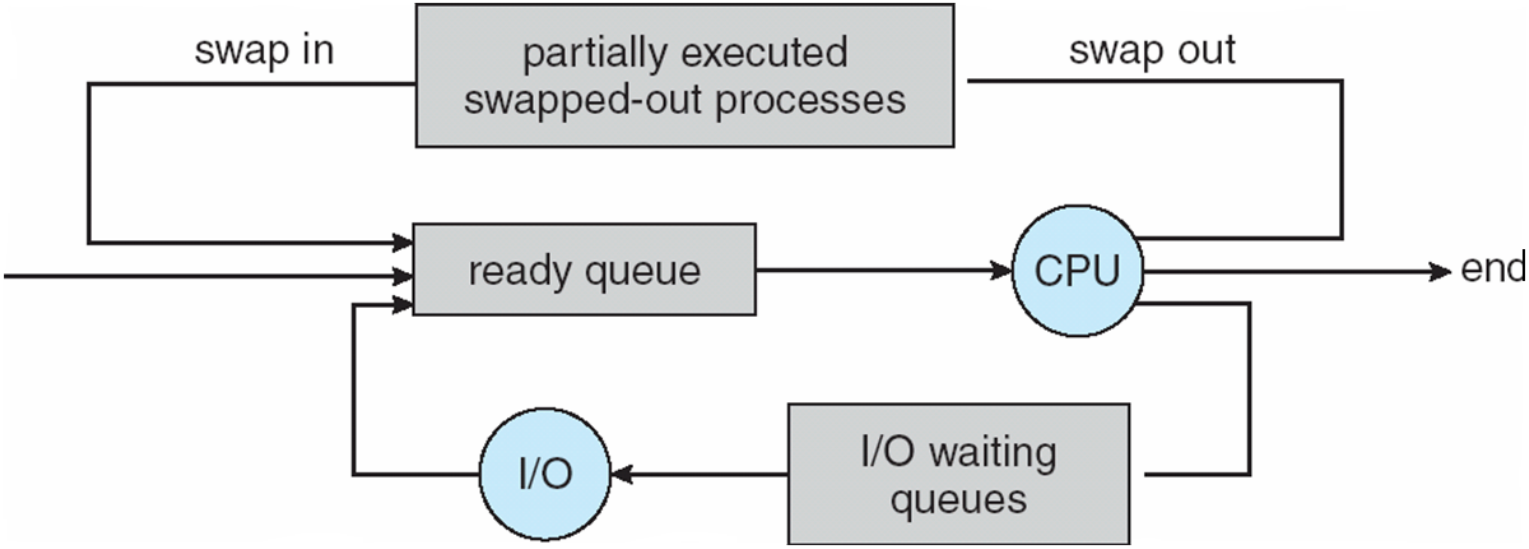
- Code executed in kernel above is overhead
  - Overhead sets minimum practical switching time

Schedulers

- **Long-term scheduler (or job scheduler) -**
  - selects which processes should be brought into the ready queue.
  - invoked very infrequently (seconds, minutes); may be slow.
  - controls the degree of multiprogramming
- **Short term scheduler (or CPU scheduler) -**
  - selects which process should execute next and allocates CPU.
  - invoked very frequently (milliseconds) - must be very fast
- **Medium Term Scheduler**
  - swaps out process temporarily → swapping
  - balances load for better throughput

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**



Process Profiles

- **I/O bound process -**
  - spends more time in I/O, short CPU bursts, CPU underutilized.
- **CPU bound process -**
  - spends more time doing computations; few very long CPU bursts, I/O underutilized.
- **The right job mix:**
  - Long term scheduler - admits jobs to keep load balanced between I/O and CPU bound processes
  - Medium term scheduler – ensures the right mix (by sometimes swapping out jobs and resuming them later)

Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Threads
- Interprocess Communication
- Examples of IPC Systems

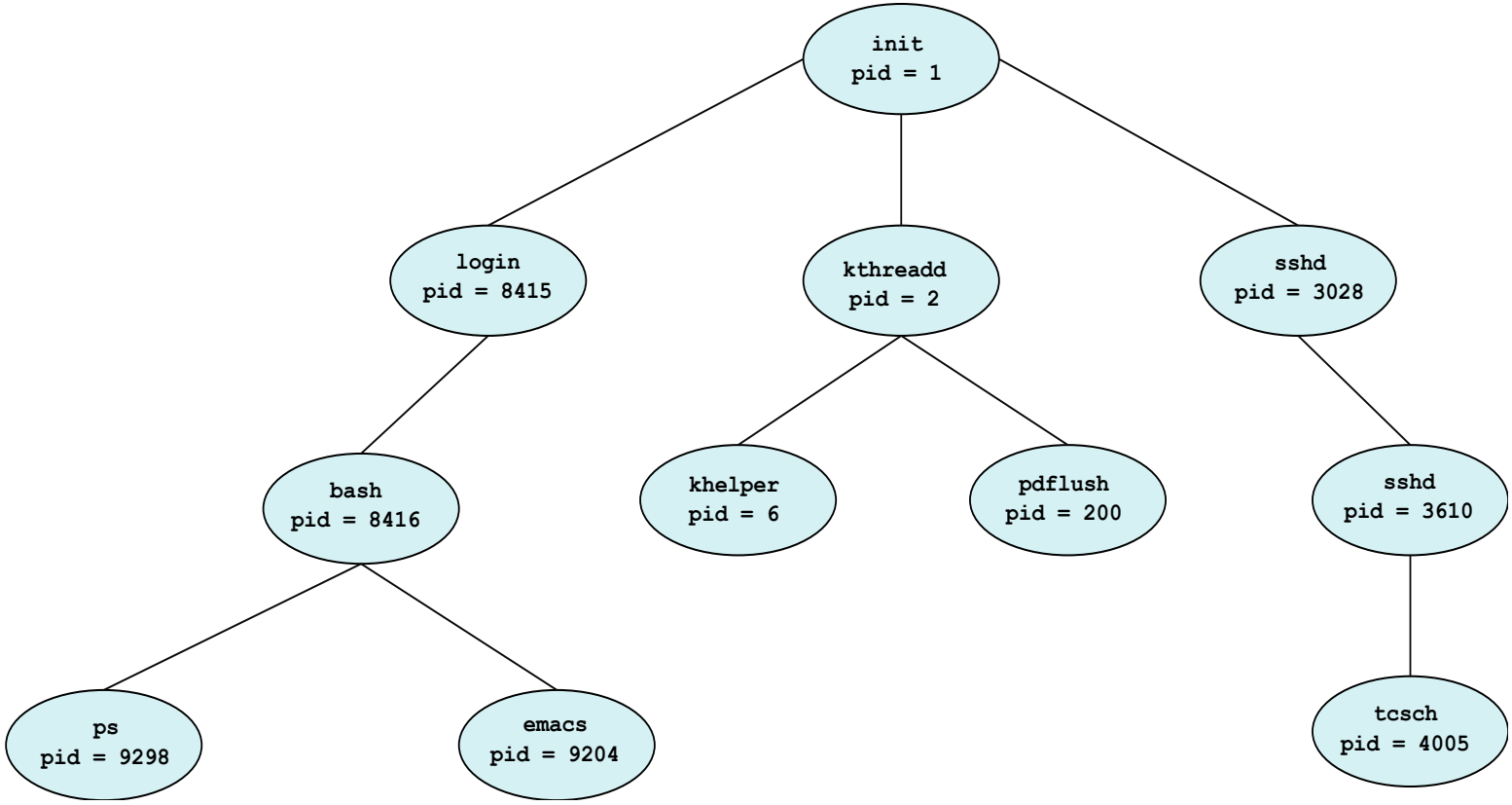
Operations on Processes

- **System must provide mechanisms for process creation, termination, and so on as detailed next**

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)**

A Tree of Processes in Linux



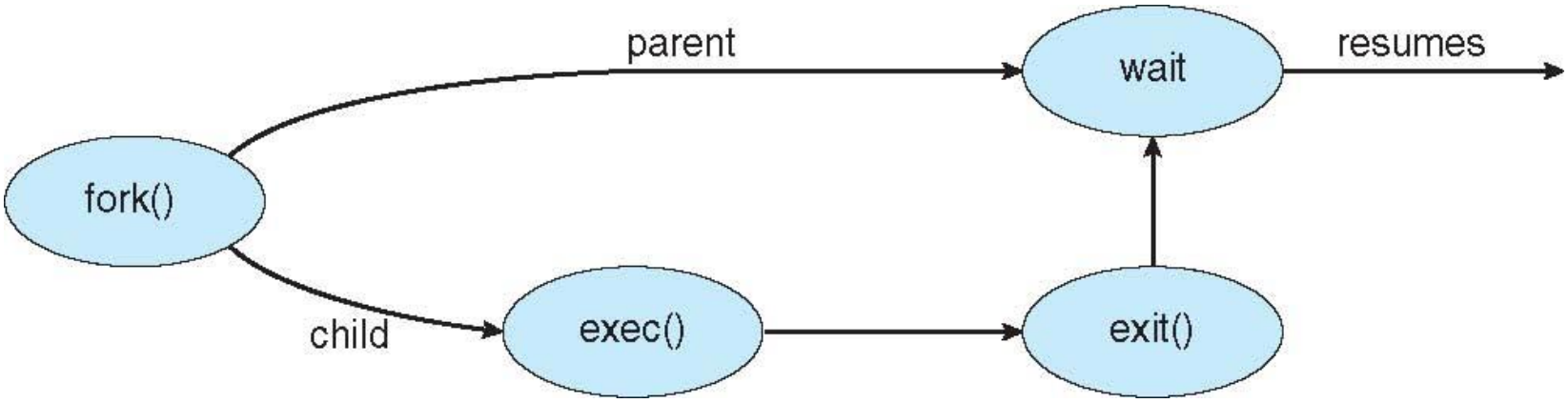
On Unix use ps –el command to list complete information of all the process currently active

Process Creation

- **Resource sharing options**
  1. Parent and children share all resources
  2. Children share subset of parent's resources
  3. Parent and child share no resources
- **Execution options**
  1. Parent and children execute concurrently
  2. Parent waits until children terminate
- **Address space**
  1. Child duplicate of parent
  2. Child has a program loaded into it

Process Creation (Cont.)

- **UNIX examples**
  - **`fork()` system call creates new process**
    - Return code for the fork () is zero for the child process
    - Non-zero process identifier is returned to the parent.
  - **`exec()` system call used after a `fork()` by one of the processes to replace the process' memory space with a new program**



C Program Forking Separate Process

```
#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 Termination

- **Process executes last statement and asks the operating system to delete it (`exit()`)**
  - **Output data from child to parent (via `wait()`)**
  - **Process' resources are deallocated by operating system**

- **Parent may terminate execution of children processes (`abort()`)**
  - **Child has exceeded allocated resources**
  - **Task assigned to child is no longer required**
  - **If parent is exiting**
    - Some operating systems do not allow child to continue if its parent terminates
      - All children terminated - **cascading termination**

Process Termination

- Wait for termination, returning the pid:

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

- If no parent waiting, then terminated process is a zombie
- If parent terminated, processes are orphans

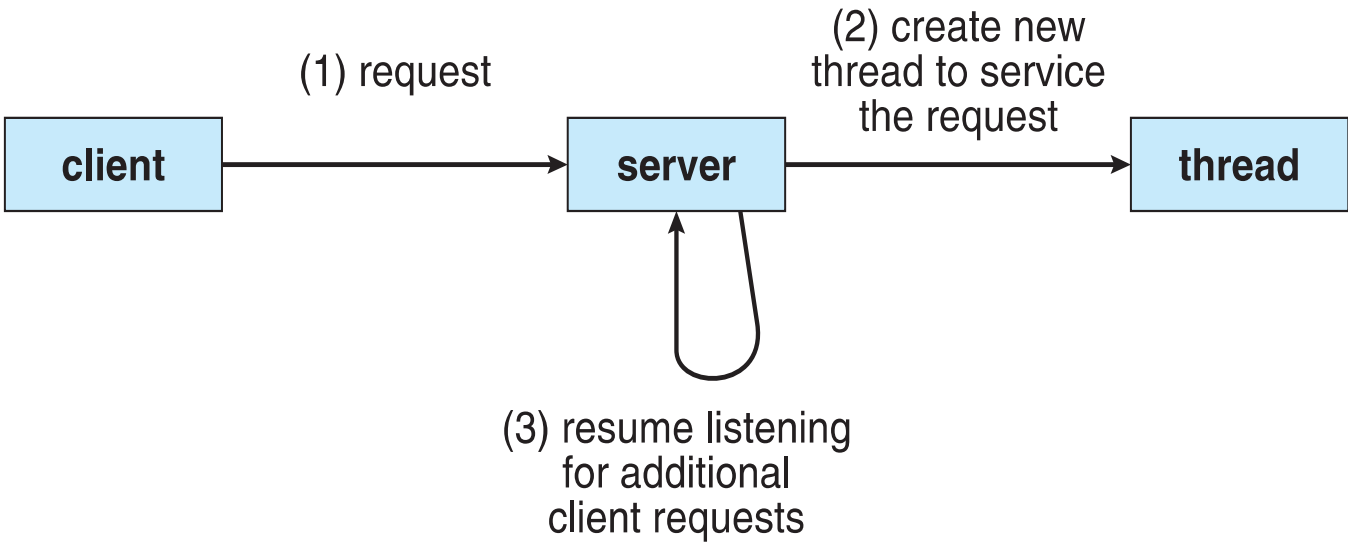
Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Threads
- Interprocess Communication
- Examples of IPC Systems

Threads

- **Processes do not share resources well**
  - high context switching overhead
- **Idea: Separate concurrency from protection**
- **Multithreading: *a single program made up of a number of different concurrent activities***
- **A thread (or lightweight process)**
  - basic unit of CPU utilization; it consists of:
    - program counter, register set and stack space
  - A thread shares the following with peer threads:
    - code section, data section and OS resources (open files, signals)
    - No protection between threads
  - Collectively called a task.
- **Heavyweight process is a task with one thread.**

Multithreaded Server Architecture



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

- **Multicore or multiprocessor systems putting pressure on programmers, challenges include:**
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**

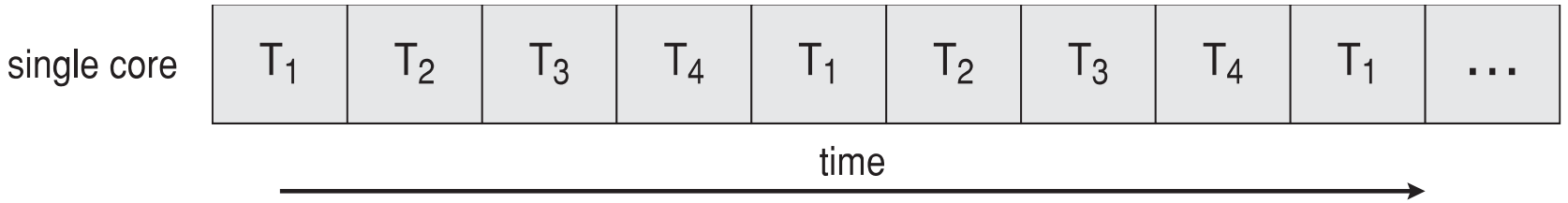
- ***Parallelism* implies a system can perform more than one task simultaneously**
- ***Concurrency* supports more than one task making progress**
  - **Single processor / core, scheduler providing concurrency**

Multicore Programming

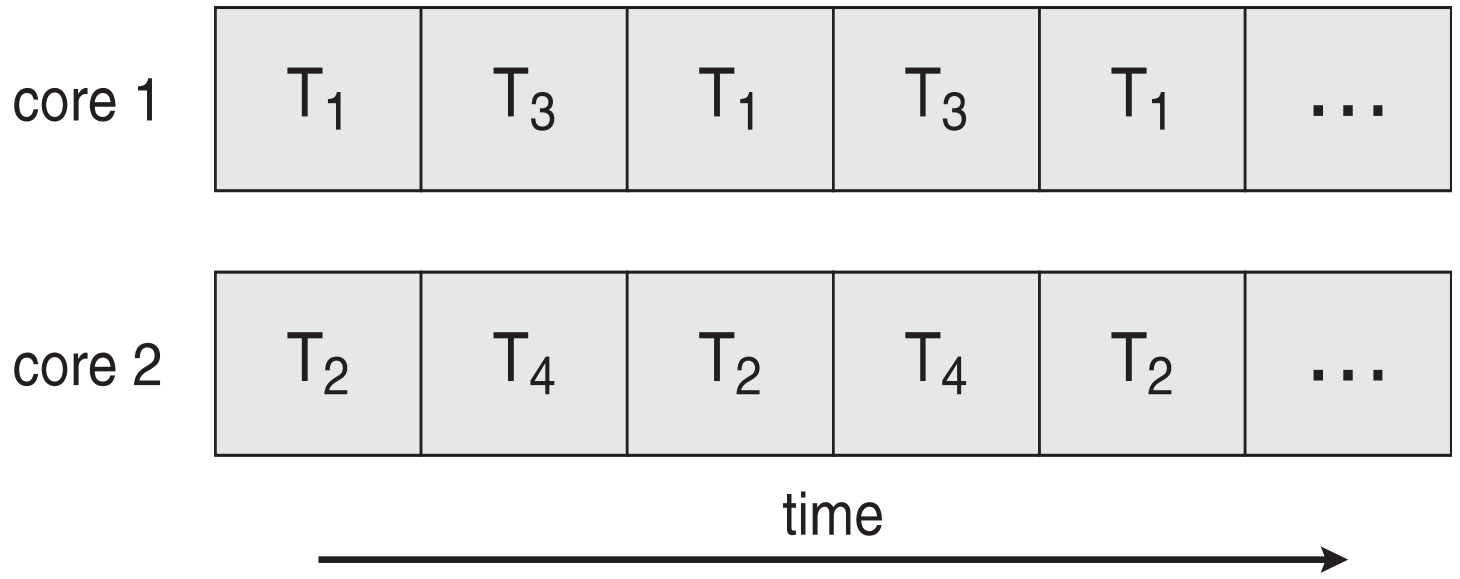
- **Types of parallelism**
  - **Data parallelism – distributes subsets of the same data across multiple cores, same operation on each**
  - **Task parallelism – distributing threads across cores, each thread performing unique operation**
- **As # of threads grows, so does architectural support for threading**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

Concurrency vs. Parallelism

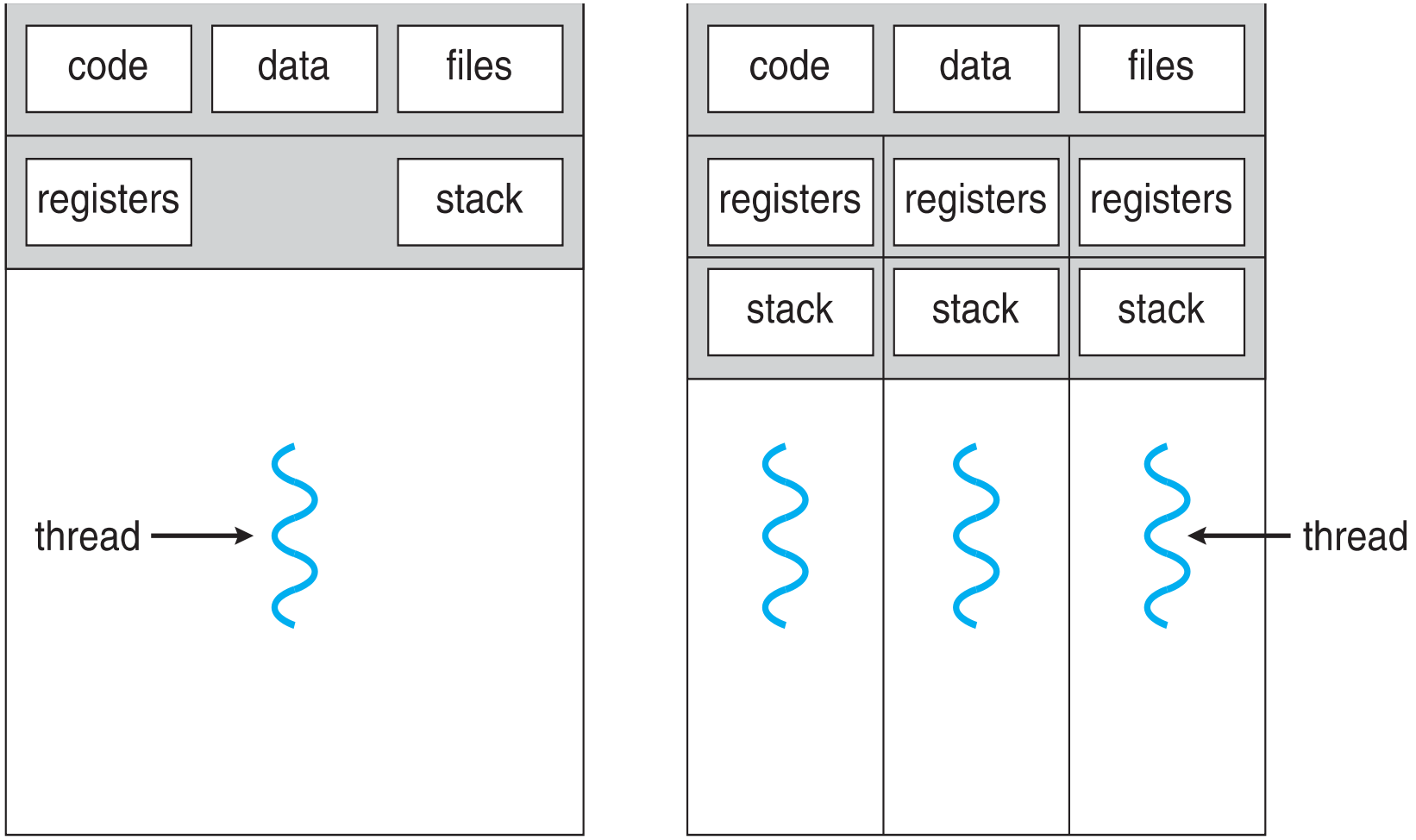
- **Concurrent execution on single-core system:**



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



Single and Multithreaded Processes



single-threaded process

multithreaded process

Threads (Cont.)

- **In a multiple threaded task, while one server thread is blocked and waiting, a second thread in the same task can run.**
  - Cooperation of multiple threads in the same job confers higher throughput and improved performance.
  - Applications that require sharing a common buffer (i.e. producer-consumer) benefit from thread utilization.
- **Threads provide a mechanism that allows sequential processes to make blocking system calls while also achieving parallelism.**

Thread State

- **State shared by all threads in process/addr space**
  - **Contents of memory (global variables, heap)**
  - **I/O state (file system, network connections, etc)**
- **State "private" to each thread**
  - **Kept in TCB ≡ Thread Control Block**
  - **CPU registers (including, program counter)**
  - **Execution stack**
    - Parameters, Temporary variables
    - return PCs are kept while called procedures are executing

Threads (cont.)

- **Thread context switch still requires a register set switch, but no memory management related work!!**
- **Thread states -**
  - *ready, blocked, running, terminated*
- **Threads share CPU and only one thread can run at a time.**
- **No protection among threads.**

Examples: Multithreaded programs

- **Embedded systems**
  - Elevators, Planes, Medical systems, Wristwatches
  - Single Program, concurrent operations
- **Most modern OS kernels**
  - Internally concurrent because have to deal with concurrent requests by multiple users
  - But no protection needed within kernel
- **Database Servers**
  - Access to shared data by many concurrent users
  - Also background utility processing must be done

More Examples: Multithreaded programs

- **Network Servers**
  - **Concurrent requests from network**
  - **Again, single program, multiple concurrent operations**
  - **File server, Web server, and airline reservation systems**
- **Parallel Programming (More than one physical CPU)**
  - **Split program into multiple threads for parallelism**
  - **This is called Multiprocessing**

threads Per AS: \ # of addr spaces:	One	Many
One	MS/DOS, early Macintosh	Traditional UNIX
Many	Embedded systems (Geoworks, VxWorks, JavaOS,etc) JavaOS, Pilot(PC)	Mach, OS/2, Linux Windows 9x??? Win NT to XP, Solaris, HP-UX, OS X

Real operating systems have either

- **One or many address spaces**
- **One or many threads per address space**

Types of Threads

- Kernel-supported threads
- User-level threads
- Hybrid approach implements both user-level and kernel-supported threads (Solaris 2).

Kernel Threads

- **Supported by the Kernel**
  - Native threads supported directly by the kernel
  - Every thread can run or block independently
  - One process may have several threads waiting on different things
- **Downside of kernel threads: a bit expensive**
  - Need to make a crossing into kernel mode to schedule
- **Examples**
  - Windows XP/2000, Solaris, Linux,Tru64 UNIX, Mac OS X, Mach, OS/2

User Threads

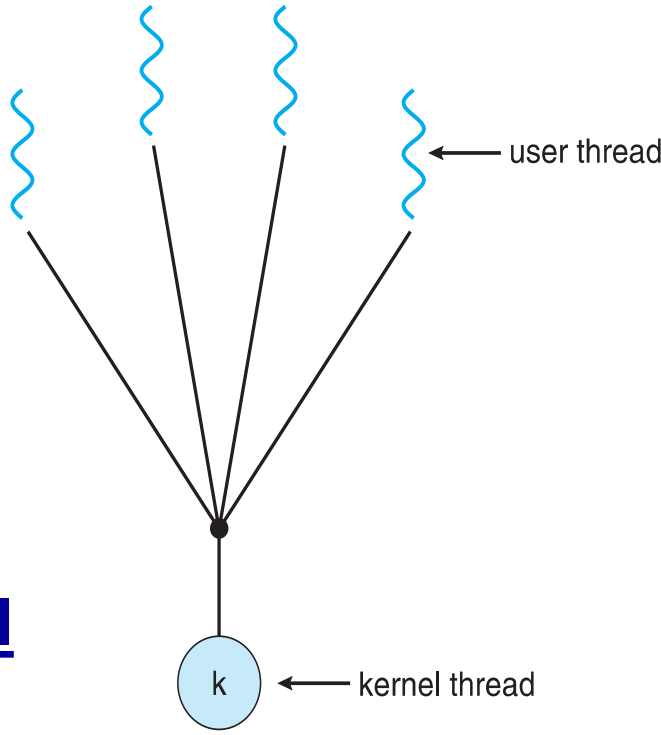
- **Supported above the kernel, via a set of library calls at the user level.**
  - Thread management done by user-level threads library
    - User program provides scheduler and thread package
  - May have several user threads per kernel thread
  - User threads may be scheduled non-preemptively relative to each other (only switch on yield())
- **Advantages**
  - **Cheap, Fast**
    - Threads do not need to call OS and cause interrupts to kernel
- **Disadvantage:**
  - **If kernel is single threaded, system call from any thread can block the entire task.**
- **Example thread libraries:**
  - **POSIX Pthreads, Win32 threads, Java threads**

Multithreading Models

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

Many-to-One

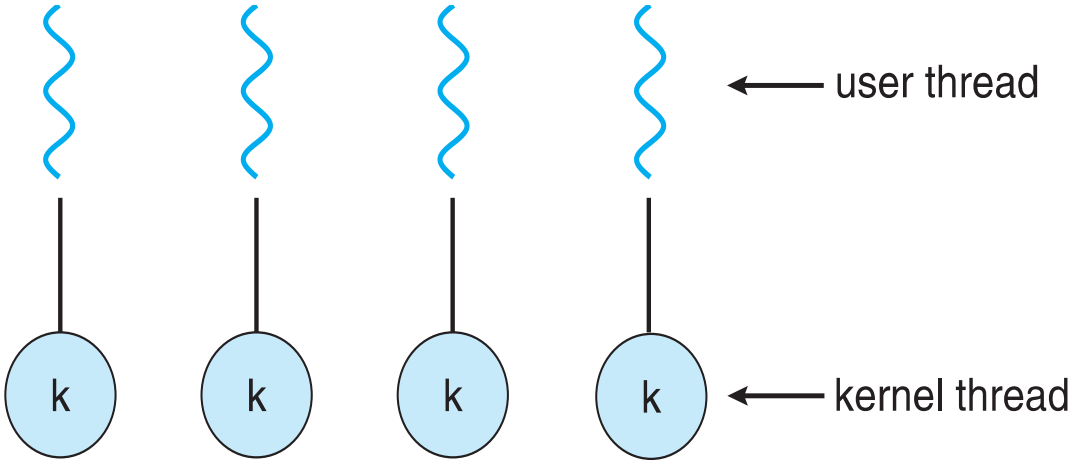
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



One-to-One

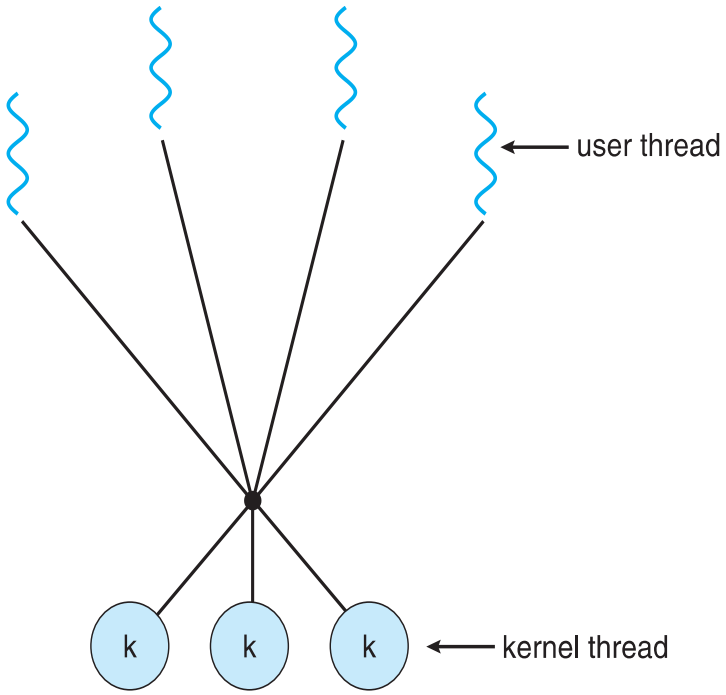
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead

- Examples
  - Windows NT/XP/2000
  - 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
  - Solaris prior to version 9
  - Windows NT/2000 with the *ThreadFiber* package



Thread Libraries

- **Thread library provides programmer with API for creating and managing threads**

- **Two primary ways of implementing**
  1. **Library entirely in user space**
  2. **Kernel-level library supported by the OS**

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 Example (synchronous fork-join)

```
#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);
}
```

Figure 4.9 Multithreaded C program using the Pthreads API.

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);
```

Figure 4.10 Pthread code for joining ten threads.

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 → Windows
  - OpenMP
  - Grand Central Dispatch →Apple→ → not discussed → self study

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
- `#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];
}
```

Run for loop 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;
}
```

Operating System Examples

- **Linux Thread**

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)
  - 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.

- `struct task_struct` points to process data structures (shared or unique)

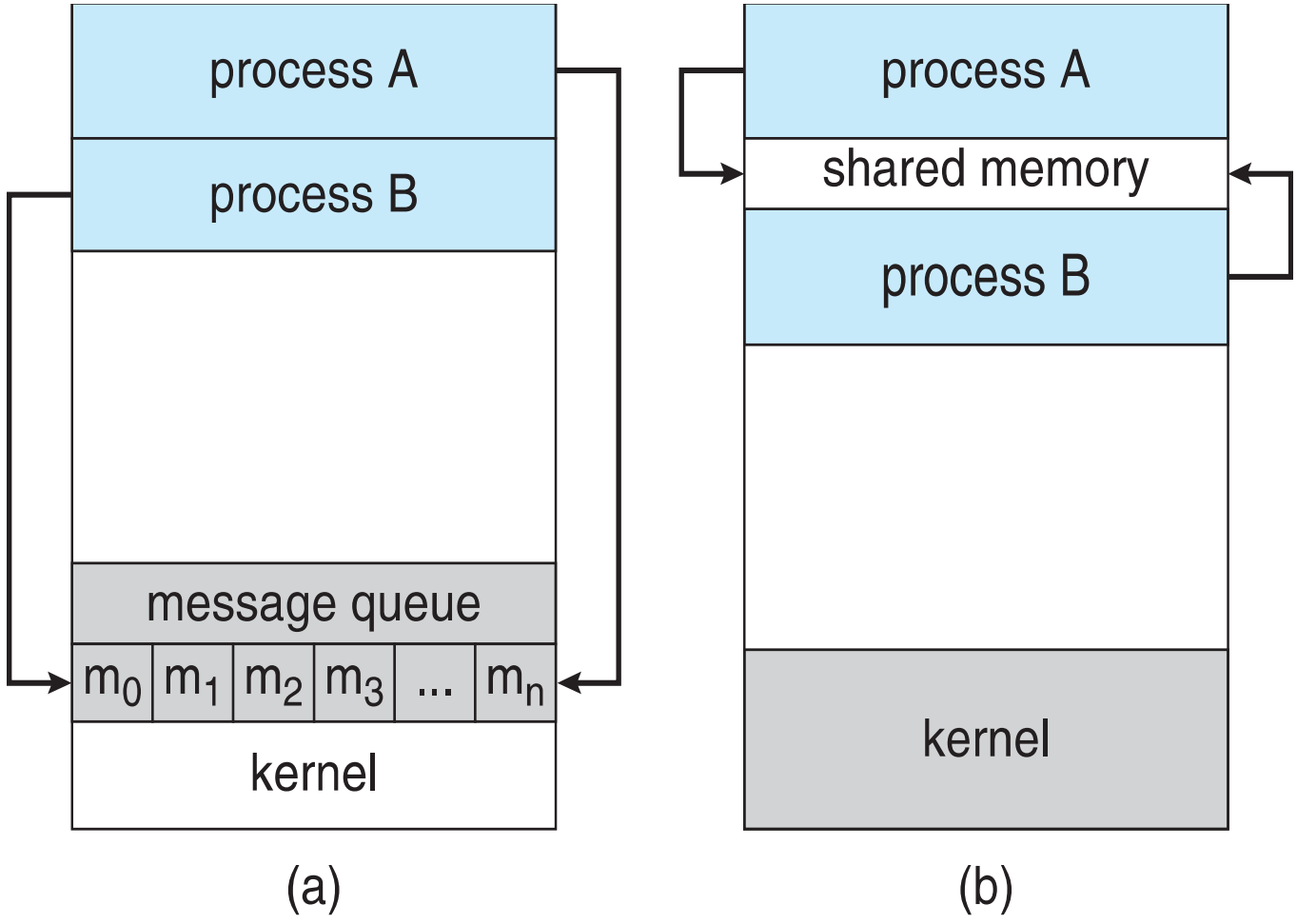
Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Threads
- Interprocess Communication
- Examples of IPC Systems

Interprocess Communication

- **Processes 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**
    - Advantages of process cooperation
      - **Information sharing**
      - **Computation speed-up**
      - **Modularity**
      - **Convenience**
  - **Cooperating processes need interprocess communication (IPC)**
- **Two models of IPC**
  1. **Shared memory**
  2. **Message passing**

Communications Models



Producer-Consumer Problem

- **Paradigm for cooperating processes → *producer* process produces information that is consumed by a *consumer* process**
  - **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 → This is a circular array**

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

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

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

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 */
}
```

- What will happen if producer process and the consumer process want to access the shared buffer concurrently?

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - `send`(*message*) – message size fixed or variable
  - `receive`(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., direct or indirect, synchronous or asynchronous, automatic or explicit buffering)

Implementation Questions????

- 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?

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
  - 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

- **Hard coding technique**

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - 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

- **Operations**
  - **create a new mailbox**
  - **send and receive messages through mailbox**
  - **destroy a mailbox**

- **Primitives are defined as:**

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

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

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**
  1. **Allow a link to be associated with at most two processes**
  2. **Allow only one process at a time to execute a receive operation**
  3. **Allow the system to select arbitrarily the receiver.**
     - Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
  - Blocking send has the sender block until the message is received
  - Blocking receive has the receiver block until a message is available
- Non-blocking is considered asynchronous
  - Non-blocking send has the sender send the message and continue
  - Non-blocking receive has the receiver receive a valid message or null

Synchronization (Cont.)

- Different combinations possible
  - If both send and receive are blocking, we have a rendezvous
- Producer-consumer becomes trivial

```
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 */
}
```

Buffering

- Queue of messages attached to the link; implemented in one of three ways

1. Zero capacity – 0 messages
   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

Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Threads
- Interprocess Communication
- Examples of IPC Systems

Examples of IPC Systems - POSIX

- **POSIX Shared Memory**
  - Process first creates shared memory segment
    ```
    shm_fd = shm_open(name, O CREAT | O RDRW, 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 <stlib.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_RDRW, 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 <stlib.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;
}
```

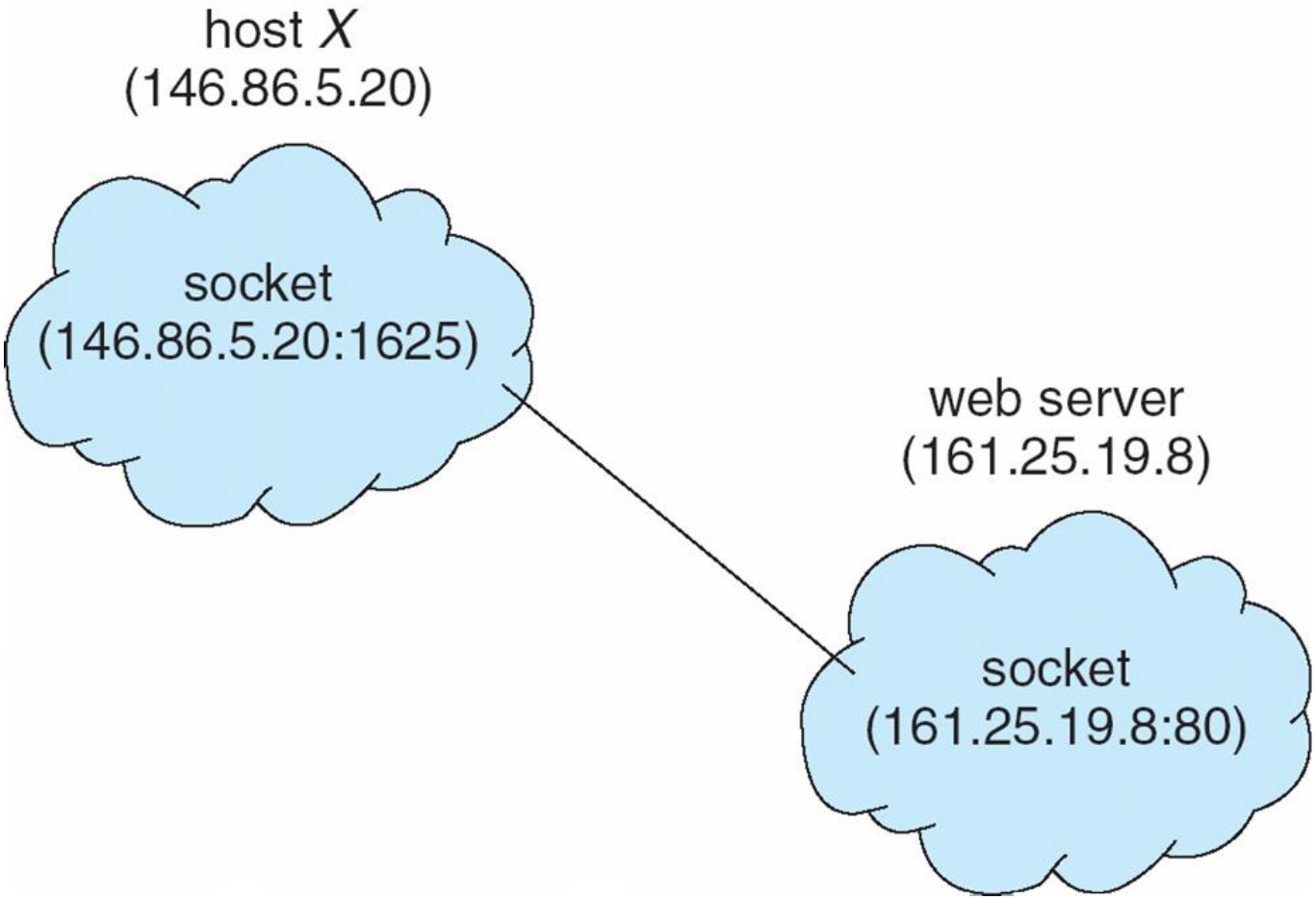
Communications in Client-Server Systems

- Sockets
- Impose structure on the data
  - Remote Procedure Calls
  - Pipes

Sockets

- A socket is defined as an endpoint for communication
- Concatenation of IP address and port – a number included at start of message packet to differentiate network services on a host
- The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Communication consists between a pair of sockets
- All ports below 1024 are *well known*, used for standard services
- Special IP address 127.0.0.1 (loopback) to refer to system on which process is running

Socket Communication



References

Part of the contents of this lecture has been adapted from the book Abraham Silberschatz, Peter B. Galvin, Greg Gagne: "Operating System Concept ", Publisher : Wiley; 9 edition (December 17, 2012), ISBN-13: 978-1118063330

Slides also contain lecture materials from John Kubiatowicz (Berkeley), John Ousterhout (Stanford), Nalini (UCI), Rainer (UCI), and others

Some slides adapted from http://www-inst.eecs.berkeley.edu/~cs162/ Copyright © 2010 UCB

Thank you for your attention