Open Source OS

Module 1: Processes Management -- Part I

Outline

- Process Concept
  - Operations on Processes
  - Interprocess Communication: Shared-Memory/Message-Passing
- Multicore Programming
- Multithreading Models
  - Thread Libraries
- Process Scheduling
  - Scheduling Algorithms
  - Thread Scheduling
  - Multi-Processor Scheduling

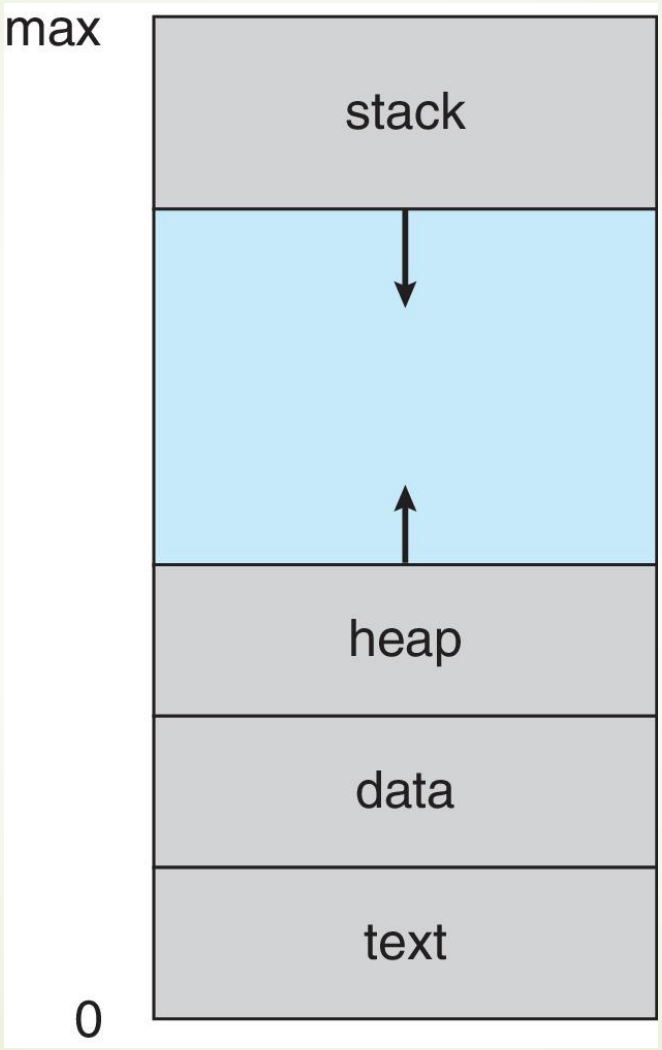
Process Concept

- An OS executes programs as a process
- **Process** – a program in execution
  - Process execution must progress in sequential fashion
  - No parallel execution of instructions of a single process
- Multiple parts
  - The program code, or **text section**
  - Current activity including **program counter**, processor registers
  - **Data section** containing global variables
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Heap** containing memory dynamically allocated during run time

Process Concept (Cont.)

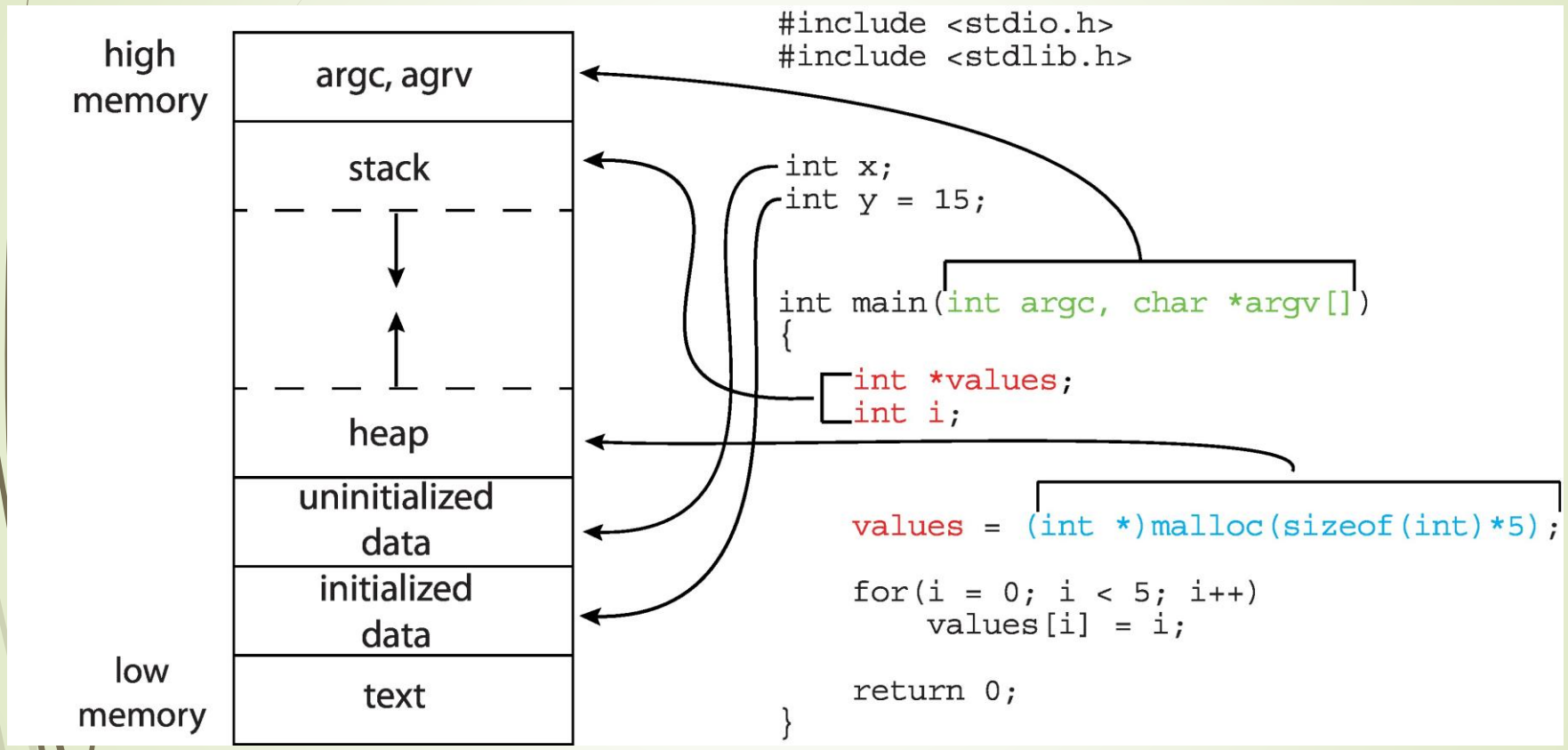
- Program is **passive** entity stored on disk (**executable file**); process is **active**
  - Program becomes process when an executable file is loaded into memory
  - Execution of program started via GUI mouse clicks, command line input of its name, etc.
- One program can be several processes
  -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

Process in Memory



[Source: Operating Systems Concepts, 10th ed.]

Memory Layout of a C Program

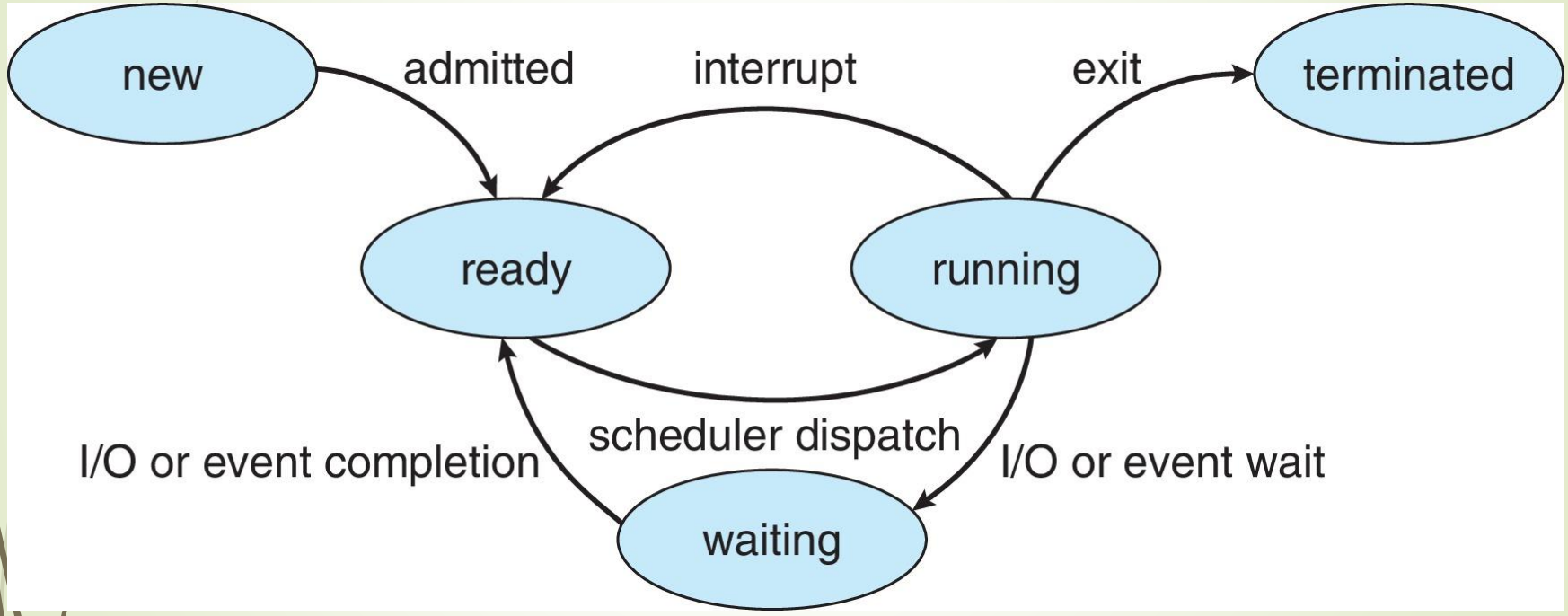


[Source: Operating Systems Concepts, 10th ed.]

Process State

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

Diagram of Process State

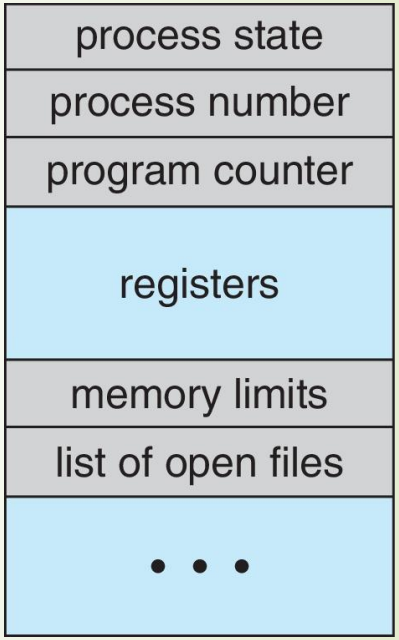


[Source: Operating Systems Concepts, 10th ed.]

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



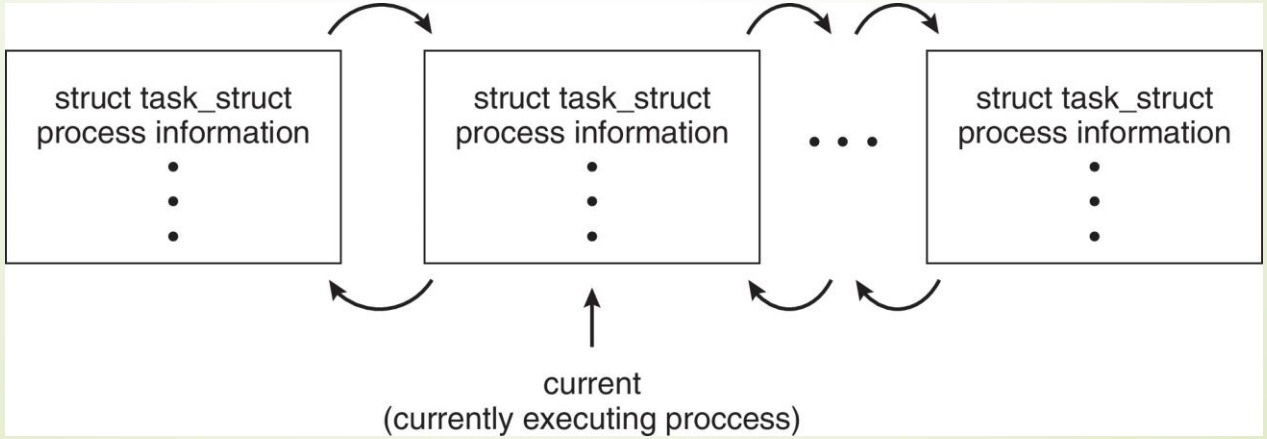
Threads

- So far, process has a single thread of execution
- 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
- More details later

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 */
```

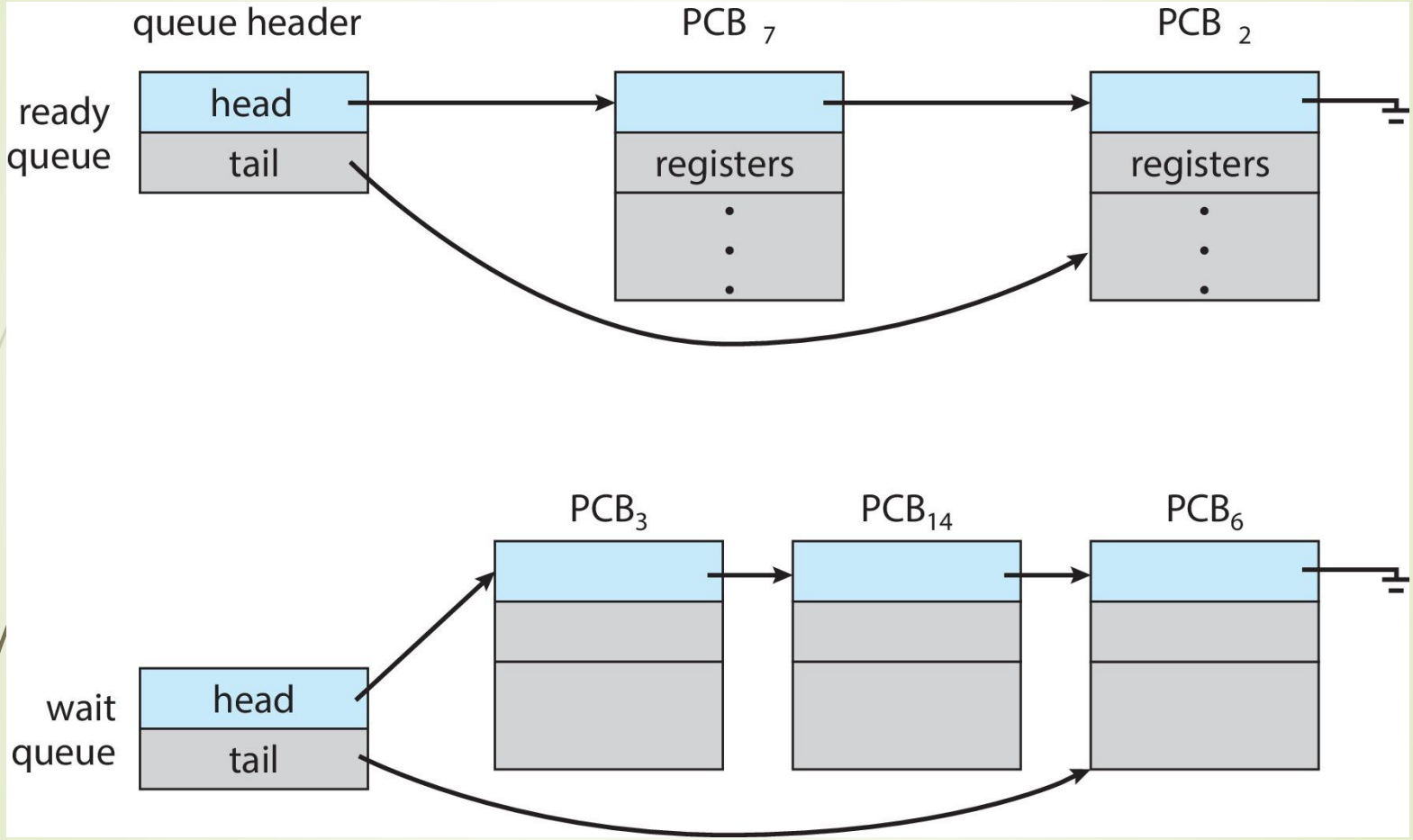


[Source: Operating Systems Concepts, 10th ed.]

Process Scheduling

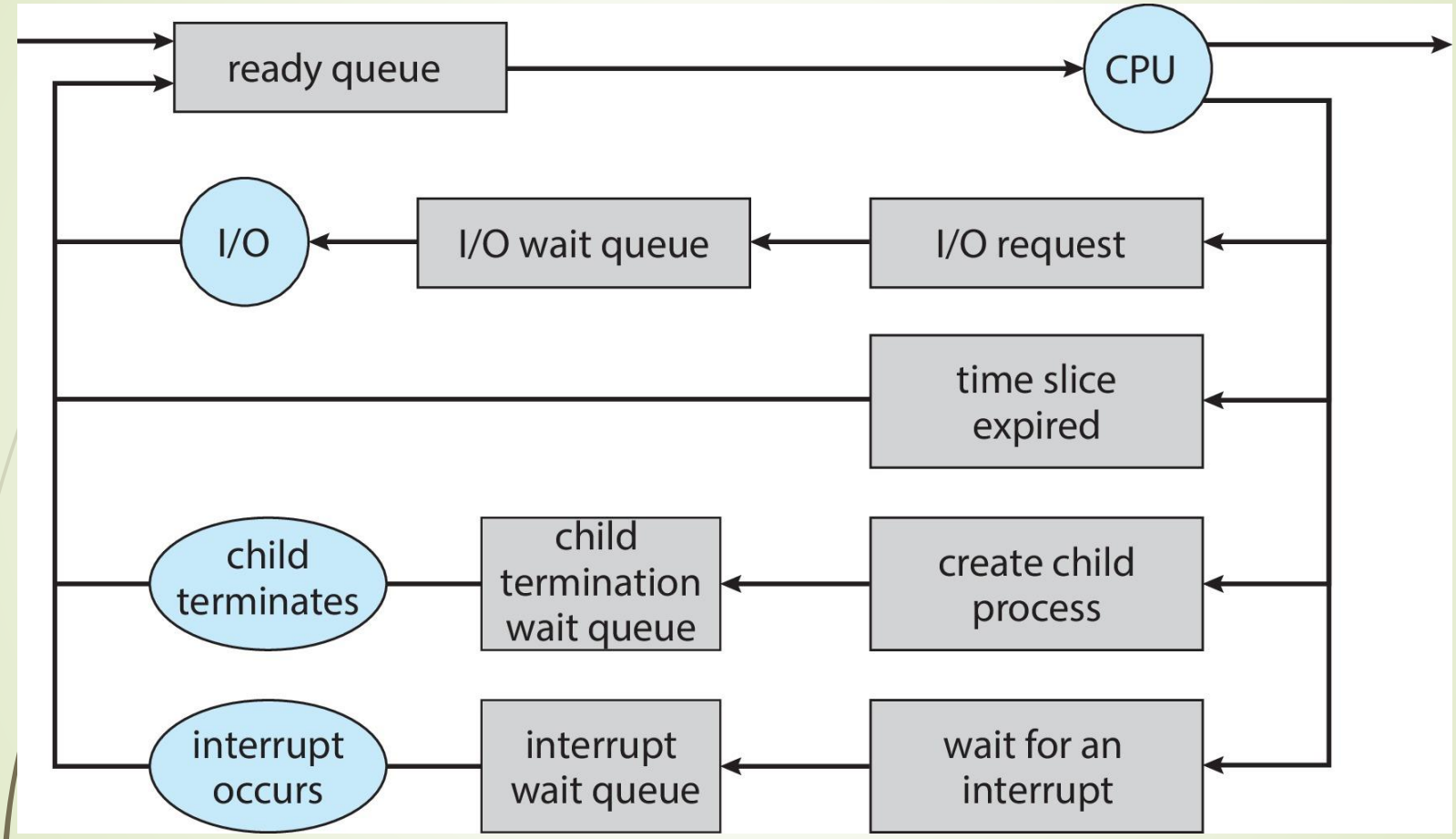
- **Process scheduler** selects among available processes for next execution on CPU core
  - Goal -- Maximize CPU use, quickly switch processes onto CPU core
- Maintains **scheduling queues** of processes
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting for an event (i.e., I/O)
  - Processes migrate among various queues

Ready and Wait Queues



[Source: Operating Systems Concepts, 10th ed.]

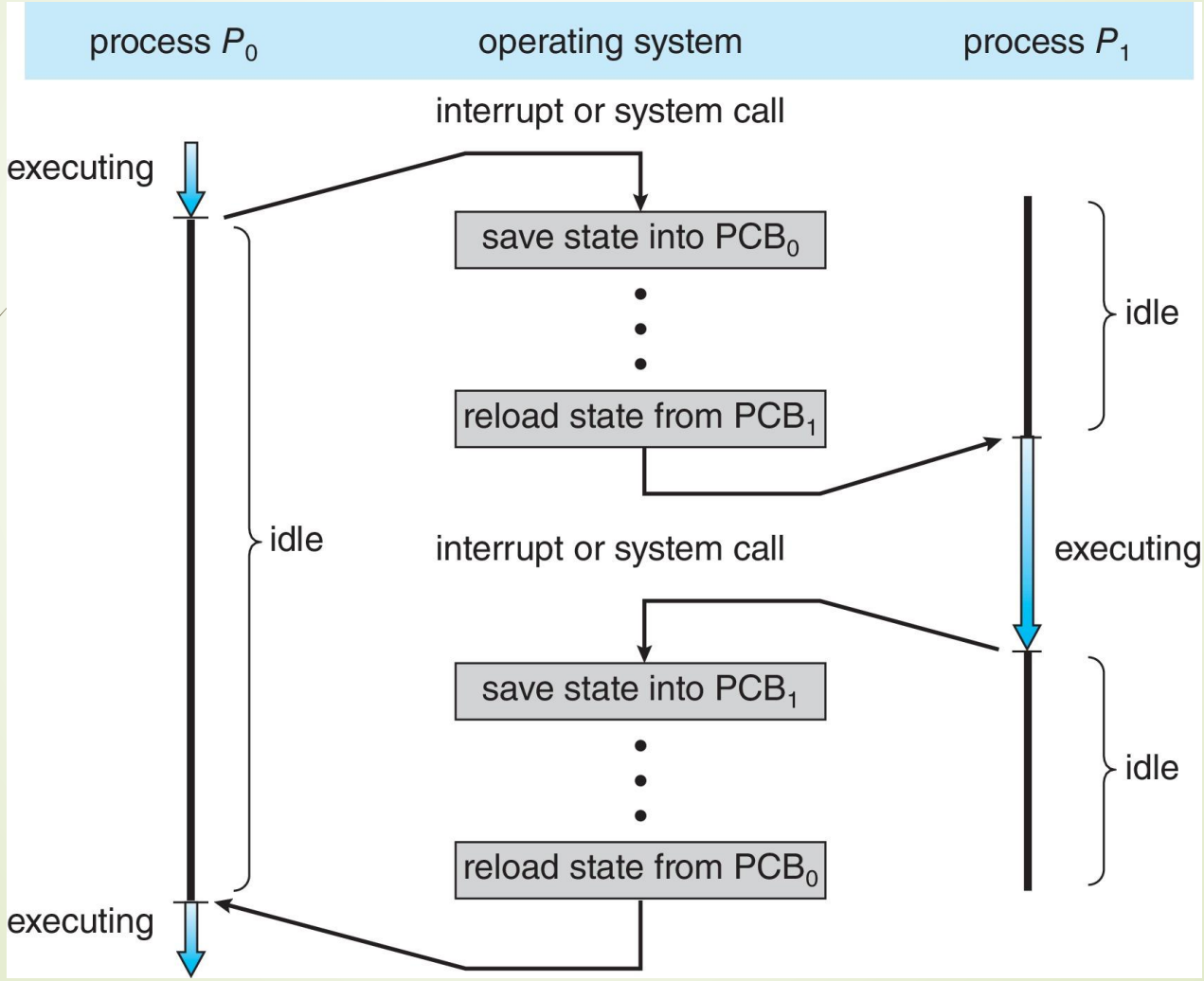
Representation of Process Scheduling



[Source: Operating Systems Concepts, 10th ed.]

CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another



[Source: Operating Systems Concepts, 10th ed.]

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 represented in the PCB
- Context-switch time is pure 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

Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits, iOS provides
  - Single **foreground** process - controlled via user interface
  - Multiple **background** processes – in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

Operations on Processes

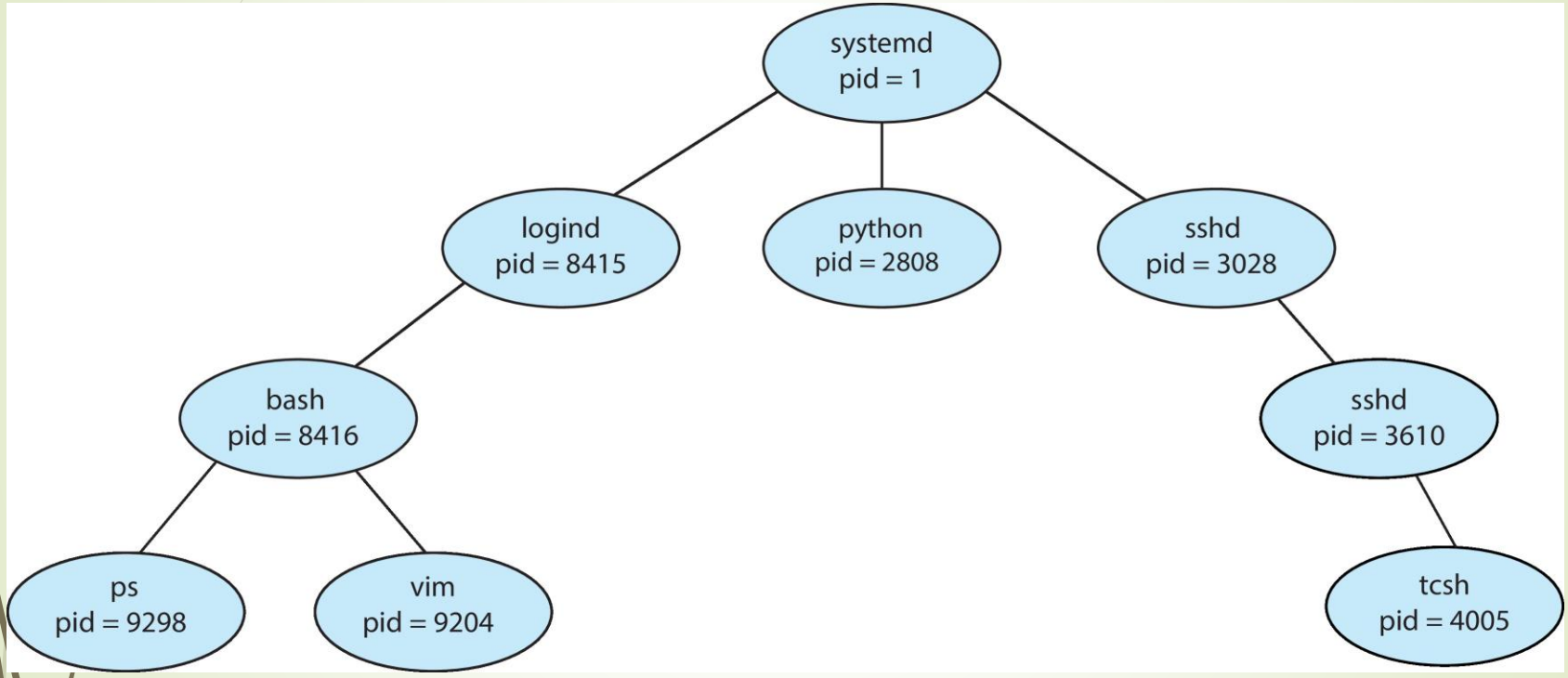
Operations on Processes

- 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**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

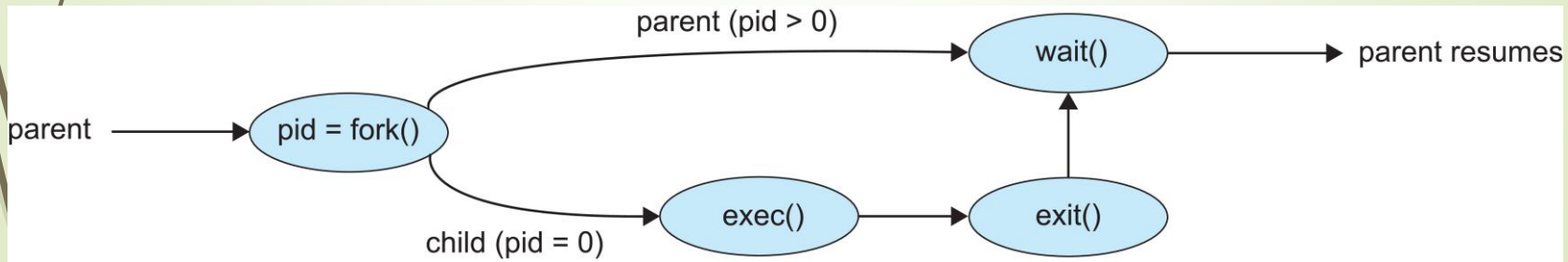
A Tree of Processes in Linux



[Source: Operating Systems Concepts, 10th ed.]

Process Creation (Cont.)

- 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
  - Parent process calls **`wait()`** waiting for the child to terminate



[Source: Operating Systems Concepts, 10th ed.]

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;
}
```

[Source: Operating Systems Concepts, 10th ed.]

Process Termination

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

Process Termination

- Some OS do not allow child to exist if its parent has terminated
  - If a process terminates, then all its children must also be terminated
    - **cascading termination -** all children, grandchildren, etc., are terminated
  - The termination is initiated by the OS
- 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);
```

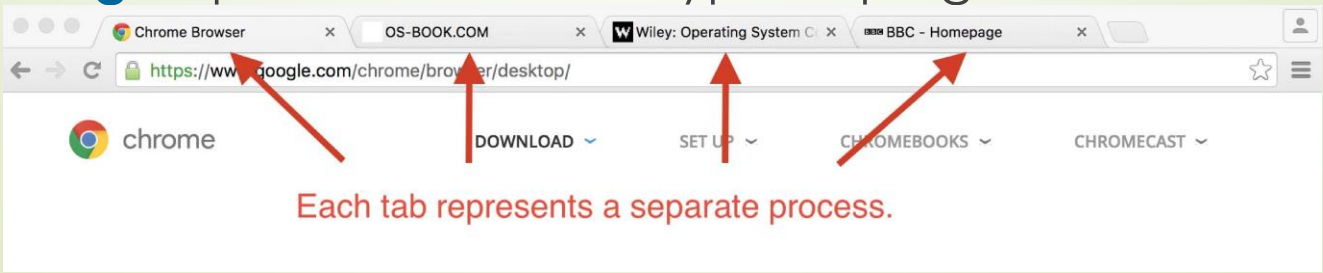
  - If no parent waiting (did not invoke **`wait()`**), process is a **zombie**
  - If parent terminated without invoking **`wait()`**, process is an **orphan**

Android Process Importance Hierarchy

- Mobile OS often have to terminate processes to reclaim system resources such as memory, from **least** to **most** important:
  - Empty process
  - Background process
  - Service process
  - Visible process
  - Foreground process

Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in



Interprocess Communications (IPC)

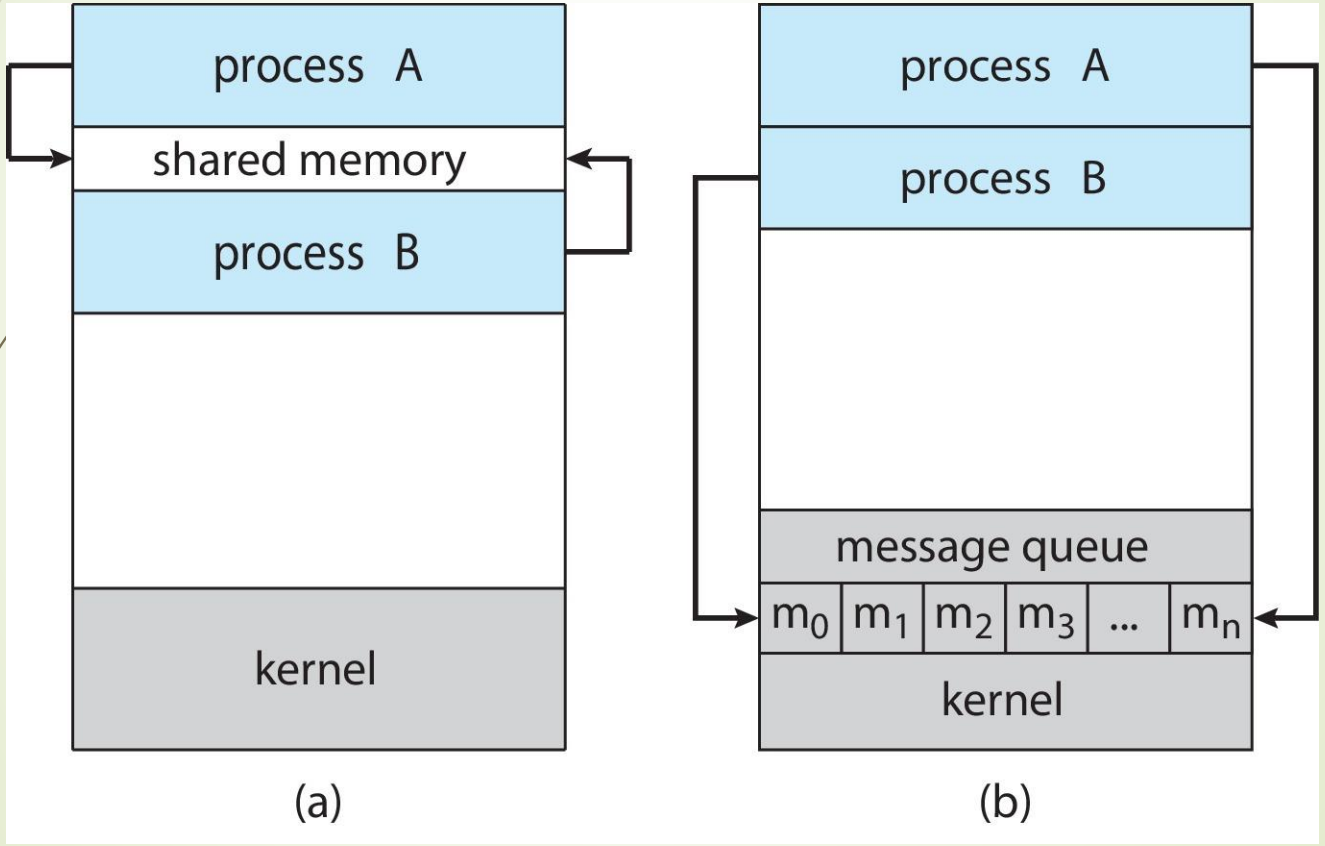
Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
  - Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

(a) Shared memory

(b) Message passing



[Source: Operating Systems Concepts, 10th ed.]

Producer-Consumer Problem

- Paradigm for cooperating processes:
  - *producer* process produces information that is consumed by a *consumer* process
- Two variations:
  - **unbounded-buffer** places no practical limit on the size of the buffer:
    - Producer never waits
    - Consumer waits if there is no buffer to consume
  - **bounded-buffer** assumes that there is a fixed buffer size
    - Producer must wait if all buffers are full
    - Consumer waits if there is no buffer to consume

IPC – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the user processes not the OS
- Major issue is to provide mechanism that will allow user processes to synchronize their actions when they access shared memory
- Synchronization is discussed in great details in Chaps. 6 & 7

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#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

Producer Process – Shared Memory

```
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;
}
```

[Source: Operating Systems Concepts, 10th ed.]

Consumer Process – Shared Memory

```
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 */
}
```

[Source: Operating Systems Concepts, 10th ed.]

What about Filling all the Buffers?

- Suppose that we want a solution to the consumer-producer problem that fills **all** the buffers
- We can do so by having an integer **`counter`** that keeps track of the number of full buffers
  - Initially, **`counter`** is set to 0
  - The integer **`counter`** is incremented by the producer after it produces a new buffer
  - The integer **`counter`** is decremented by the consumer after it consumes a buffer

Producer

```
while (true)
{
   /* produce an item in next_produced */

   while (counter == BUFFER_SIZE)
      ; /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
}
```

[Source: Operating Systems Concepts, 10th ed.]

Consumer

```
while (true)
{
   while (counter == 0)
      ; /* do nothing */
   next_consumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   counter--;
   /* consume the item in next_consumed */
}
```

[Source: Operating Systems Concepts, 10th ed.]

Race Condition

- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute **`register1 = counter`** {register1 = 5}
S1: producer execute **`register1 = register1 + 1`** {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: producer execute **`counter = register1`** {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Race Condition (Cont.)

- Question: why was there no race condition in the first solution where at most (N-1) buffers can be filled?
- More details in Process Synchronization

IPC – Message Passing

- Processes communicate with each other without resorting to shared variables
- IPC 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?

Implementation of Communication Link

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

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

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 (Cont.)

- Operations
  - Create a new mailbox (port)
  - Send and receive messages through mailbox
  - Delete 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 (Cont.)

- 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

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 possible
  - If both send and receive are blocking, we have a **rendezvous**

Buffering

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

1. Zero capacity – 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

Producer-Consumer: Message Passing

- Producer

```
message next_produced;
while (true)
{
  /* produce an item in next_produced */

  send(next_produced);
}
```

- Consumer

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

  /* consume the item in next_consumed */
}
```

[Source: Operating Systems Concepts, 10th ed.]

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
  - Set the size of the object

    `ftruncate(shm_fd, 4096);`
  - Use **`mmap()`** to memory-map a file pointer to the shared memory object
  - Reading and writing to shared memory is done by using the pointer returned by **`mmap()`**

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;
}
```

[Source: Operating Systems Concepts, 10th ed.]

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;
}
```

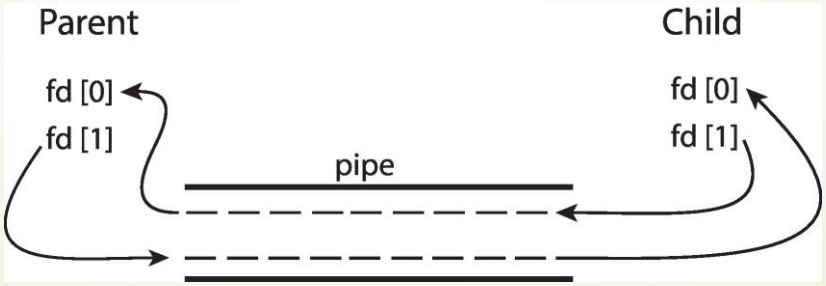
[Source: Operating Systems Concepts, 10th ed.]

Pipes

- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?
- **Ordinary pipes** – cannot be accessed from outside the process that created it
  - Typically, a parent process creates a pipe and uses it to communicate with a child process that it created
- **Named pipes** – can be accessed without a parent-child relationship

Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
  - Producer writes to one end (the **write-end** of the pipe)
  - Consumer reads from the other end (the **read-end** of the pipe)
  - Ordinary pipes are therefore unidirectional
  - Require parent-child relationship between communicating processes



- Windows calls these **anonymous pipes**

[Source: Operating Systems Concepts, 10th ed.]

Examples of Pipes in UNIX

- Ordinary Pipes in UNIX
  - fork(), read(), write(), wait(), close()
  - (See Fig. 3.25 & 3.26)
-)

Ordinary Pipes in UNIX

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

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
char write_msg[BUFFER_SIZE] = "Greetings";
char read_msg[BUFFER_SIZE];
int fd[2];
pid_t pid;

/* Program continues in Figure 3.26 */
```

Figure 3.25 Ordinary pipe in UNIX.

[Source: Operating Systems Concepts, 10th ed.]

```
   /* create the pipe */
   if (pipe(fd) == -1) {
      fprintf(stderr,"Pipe failed");
      return 1;
   }

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

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }

   if (pid > 0) { /* parent process */
      /* close the unused end of the pipe */
      close(fd[READ_END]);

      /* write to the pipe */
      write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

      /* close the write end of the pipe */
      close(fd[WRITE_END]);
   }
   else { /* child process */
      /* close the unused end of the pipe */
      close(fd[WRITE_END]);

      /* read from the pipe */
      read(fd[READ_END], read_msg, BUFFER_SIZE);
      printf("read %s",read_msg);

      /* close the write end of the pipe */
      close(fd[READ_END]);
   }

   return 0;
}
```

Figure 3.26 Figure 3.25, continued.

[Source: Operating Systems Concepts, 10th ed.]

Named Pipes

- Named Pipes are more powerful than ordinary pipes
  - Communication is bidirectional
  - No parent-child relationship is necessary between the communicating processes
  - Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

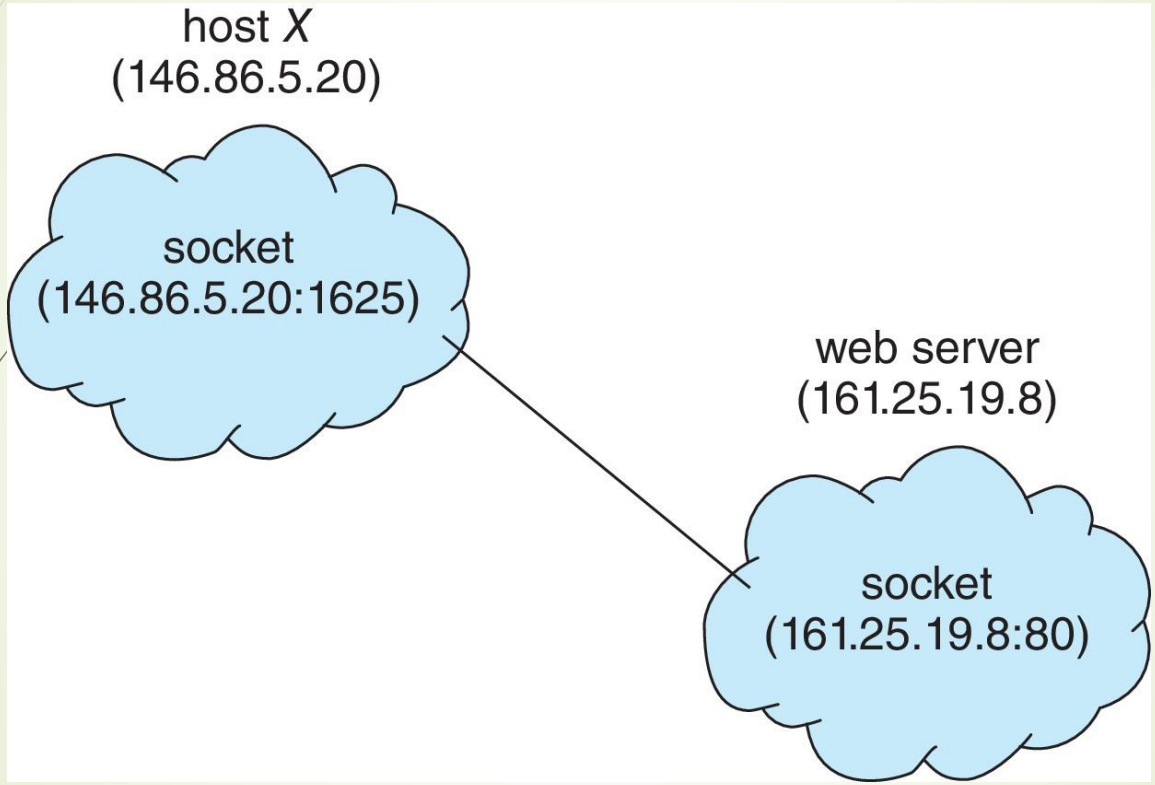
Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls

Sockets

- A **socket** is defined as an endpoint for communication
  - Concatenation of IP address and **port** – to differentiate network services on a host
  - 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
  - The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets

Socket Communication

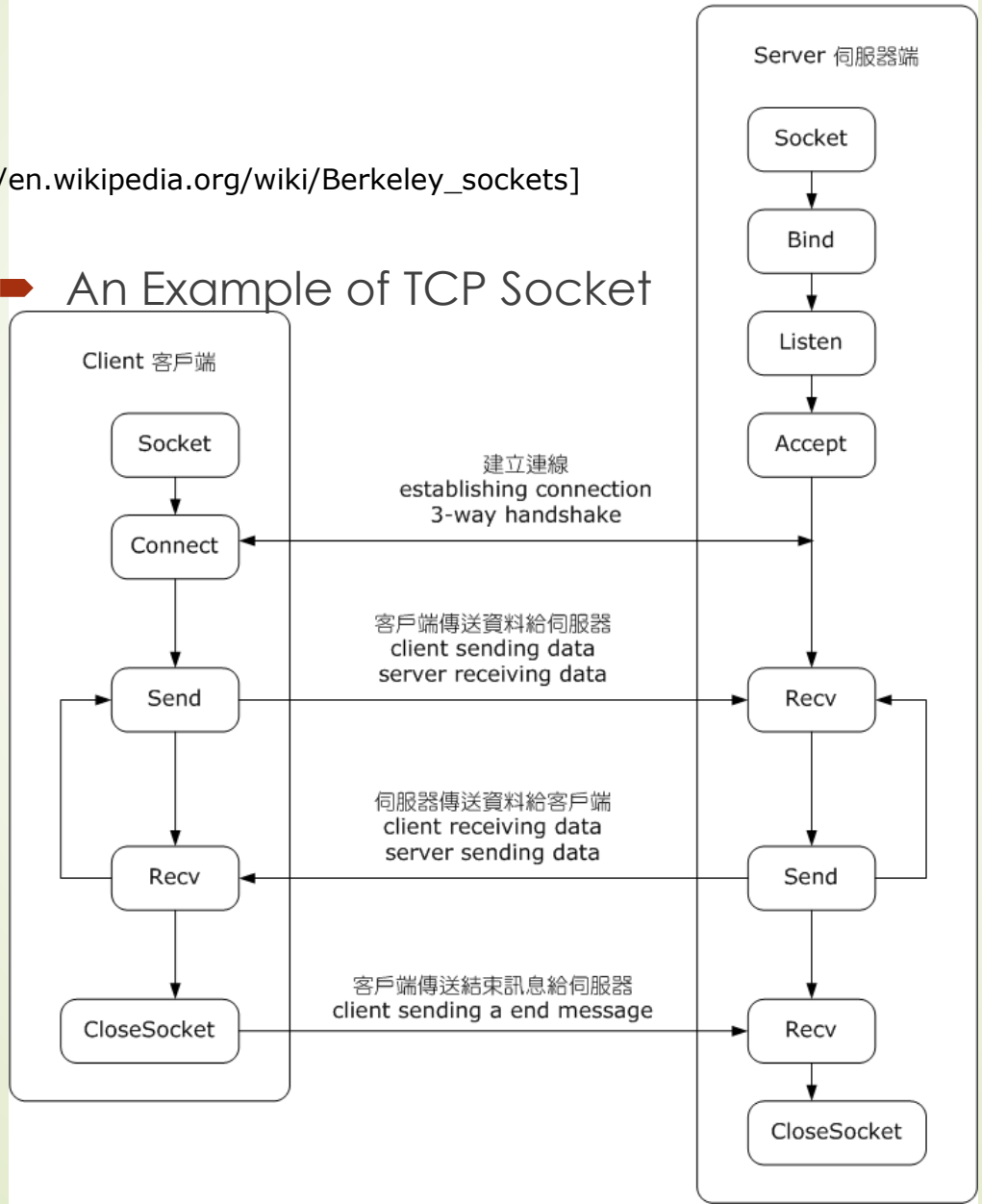


[Source: Operating Systems Concepts, 10th ed.]

Berkeley Socket API (in C)

[Source: https://en.wikipedia.org/wiki/Berkeley_sockets]

- An Example of TCP Socket



Sockets in Java

- Three types of sockets
  - **Connection-oriented (TCP)**
  - **Connectionless (UDP)**
  - **`MulticastSocket`** class – data can be sent to multiple recipients
- Consider this "Date" server in Java:

```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         /* now listen for connections */
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            /* write the Date to the socket */
            pout.println(new java.util.Date().toString());

            /* close the socket and resume */
            /* listening for connections */
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

[Source: Operating Systems Concepts, 10th ed.]

Sockets in Java

The equivalent Date client

```
import java.net.*;
import java.io.*;

public class DateClient
{
   public static void main(String[] args) {
      try {
         /* make connection to server socket */
         Socket sock = new Socket("127.0.0.1",6013);

         InputStream in = sock.getInputStream();
         BufferedReader bin = new
            BufferedReader(new InputStreamReader(in));

         /* read the date from the socket */
         String line;
         while ( (line = bin.readLine()) != null)
            System.out.println(line);

         /* close the socket connection*/
         sock.close();
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

[Source: Operating Systems Concepts, 10th ed.]

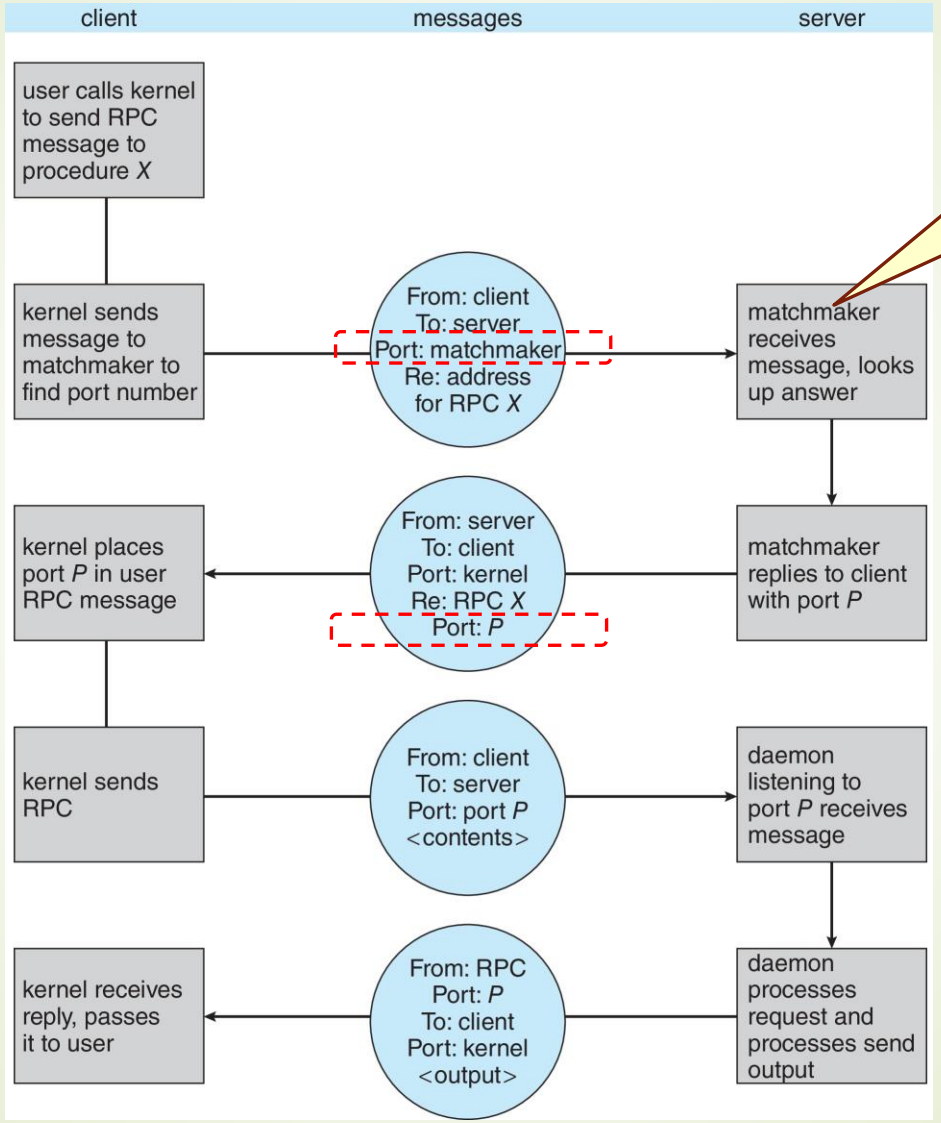
Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
  - **Stubs** – client-side proxy for the actual procedure on the server
    - The client-side stub locates the server and **marshalls** the parameters
    - The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
  - On Windows, stub code compile from specification written in **Microsoft Interface Definition Language** (**MIDL**)

Remote Procedure Calls (Cont.)

- Data representation handled via **External Data Representation** (**XDL**) format to account for different architectures
  - **Big-endian** and **little-endian**
- Remote communication has more failure scenarios than local
  - Messages can be delivered ***exactly once*** rather than ***at most once***
- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server

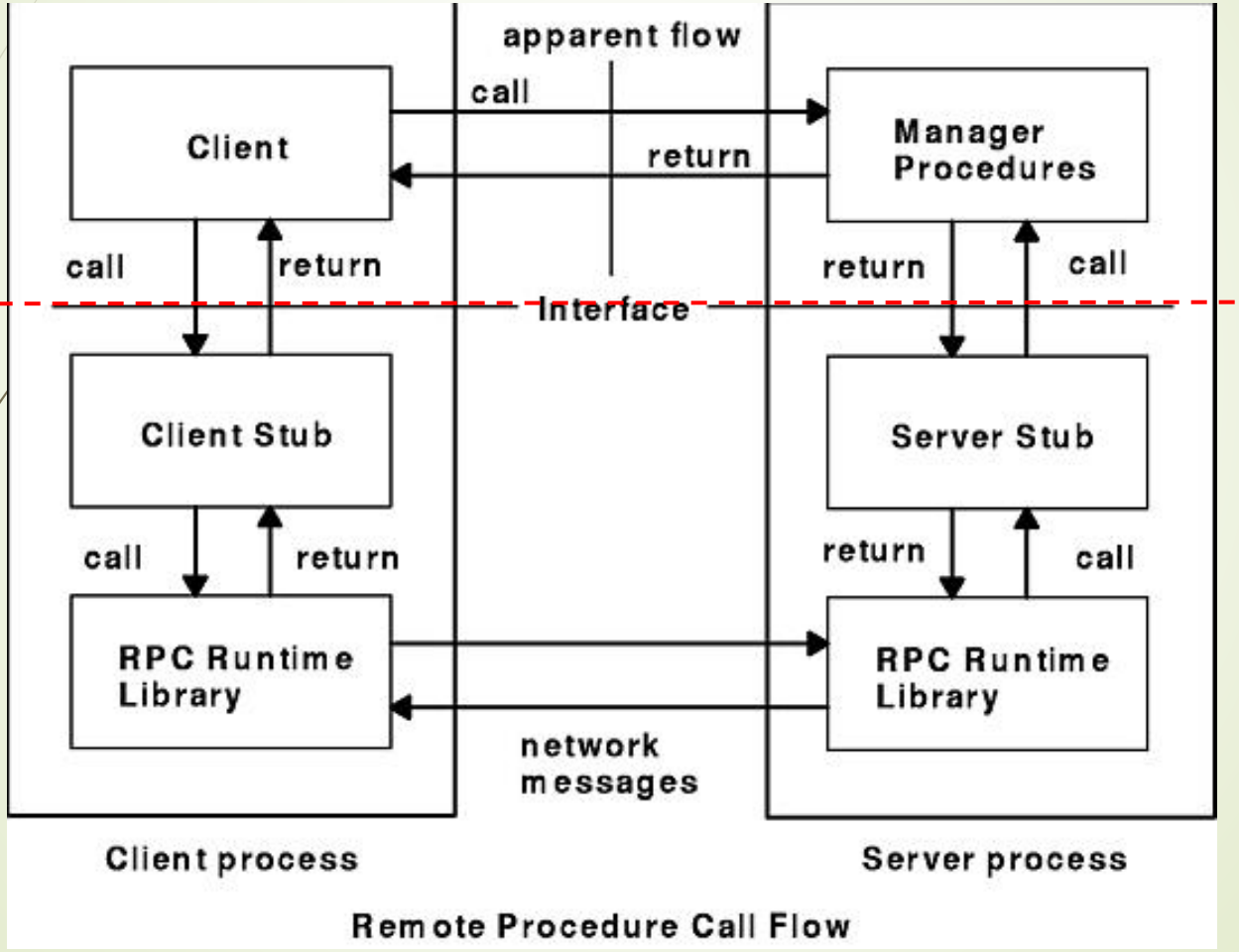
Execution of RPC



Portmapper (or rpcbind): port 111

[Source: Operating Systems Concepts, 10th ed.]

RPC Flow



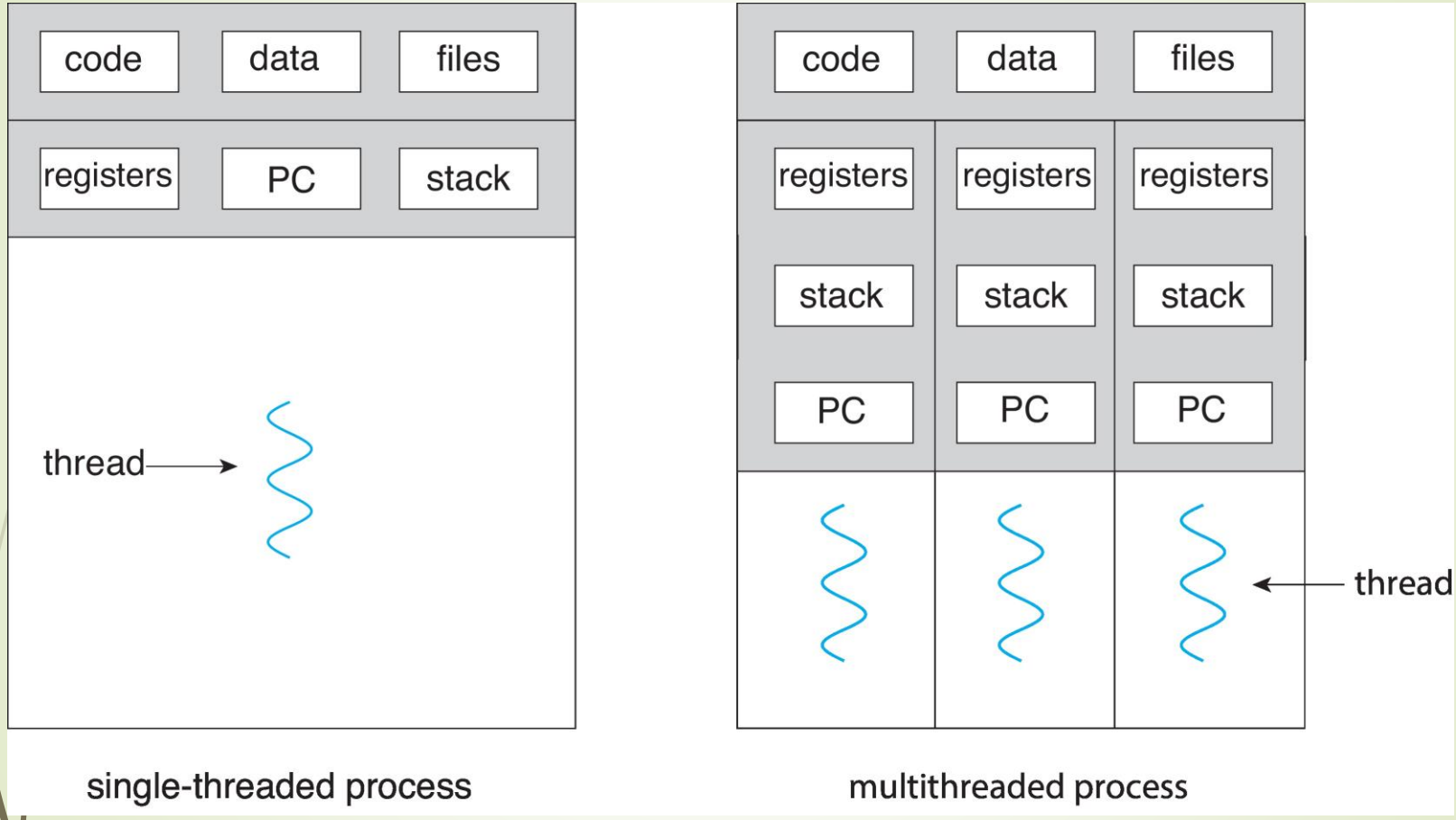
[Source: https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/commprogramming/rpc_mod.html]

Multithreading

Motivation on Multithreading

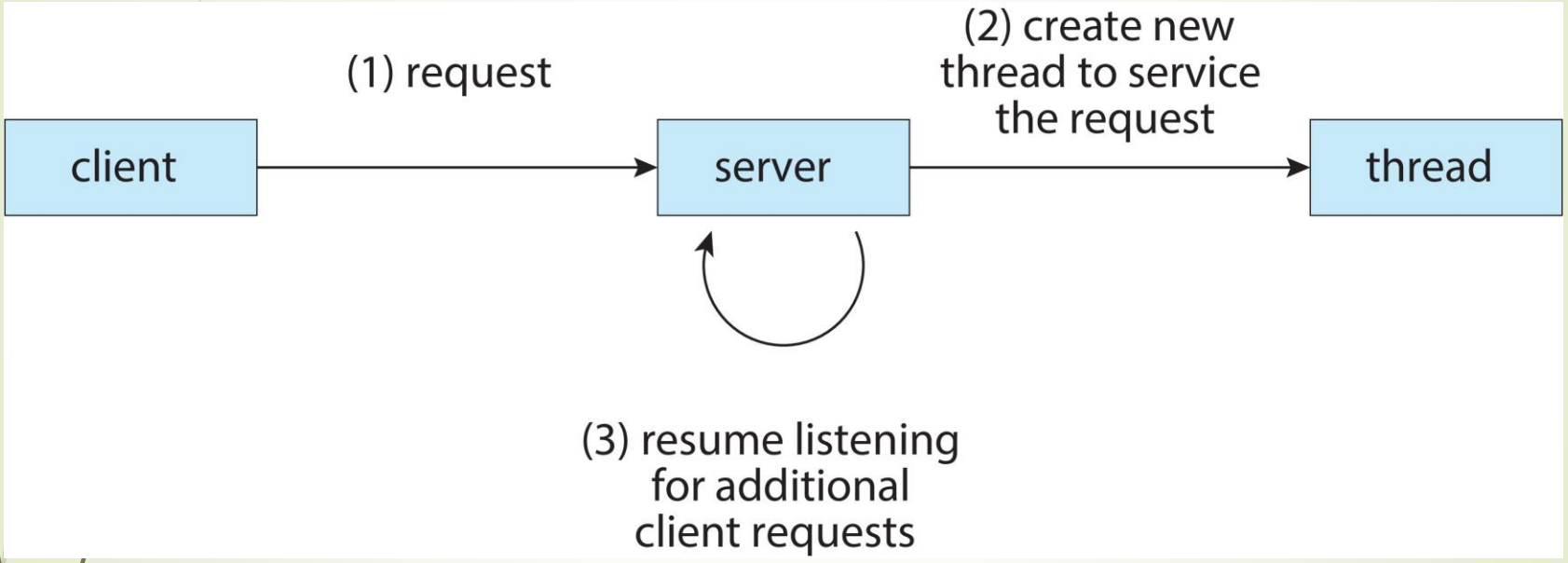
- Most modern applications are multithreaded
  - Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight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

Single and Multithreaded Processes



[Source: Operating Systems Concepts, 10th ed.]

Multithreaded Server Architecture



[Source: Operating Systems Concepts, 10th ed.]

Benefits

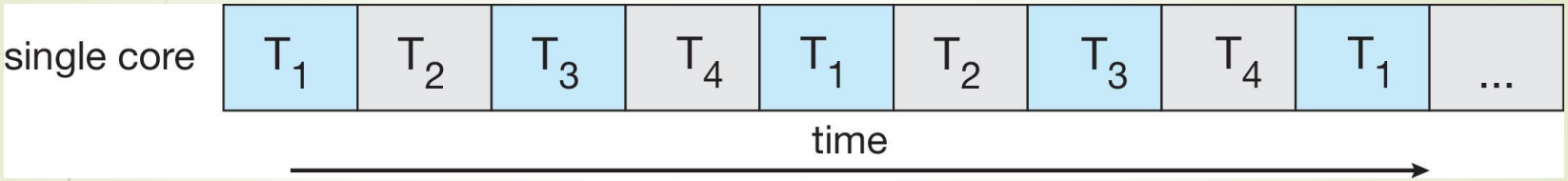
- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower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

Multicore Programming

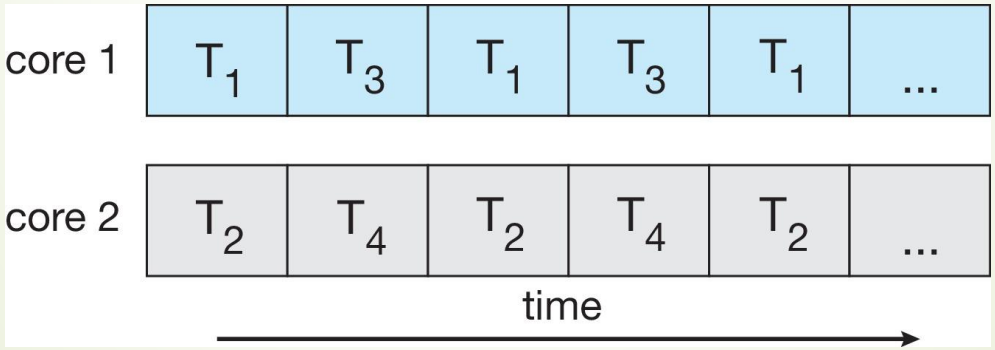
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Load balancing
  - Data splitting
  - Data dependency
  - Testing and debugging
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency***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

Concurrency vs. Parallelism

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



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

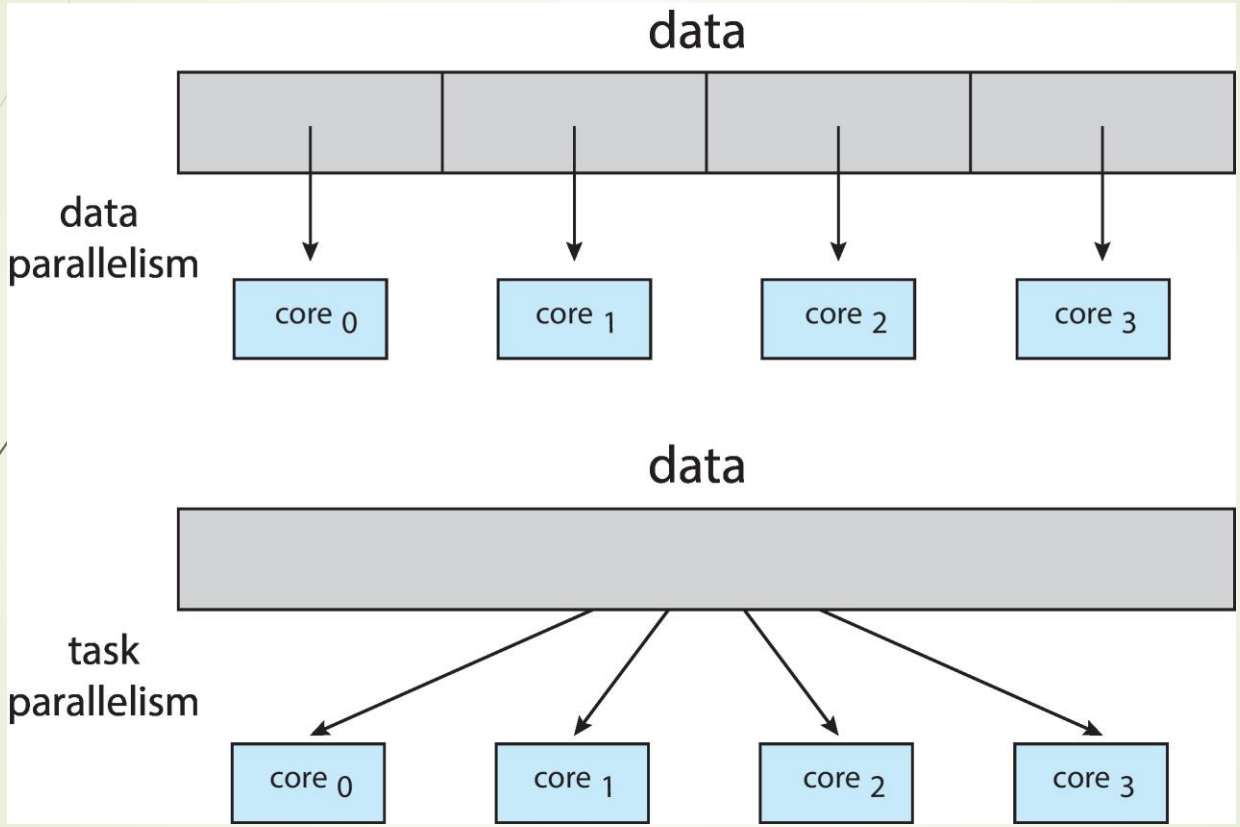


[Source: Operating Systems Concepts, 10th ed.]

Multicore Programming

- Types of parallelism
  - **Data parallelism** – distributes different subsets of the whole data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation

Data and Task Parallelism



[Source: Operating Systems Concepts, 10th ed.]

Amdahl's Law

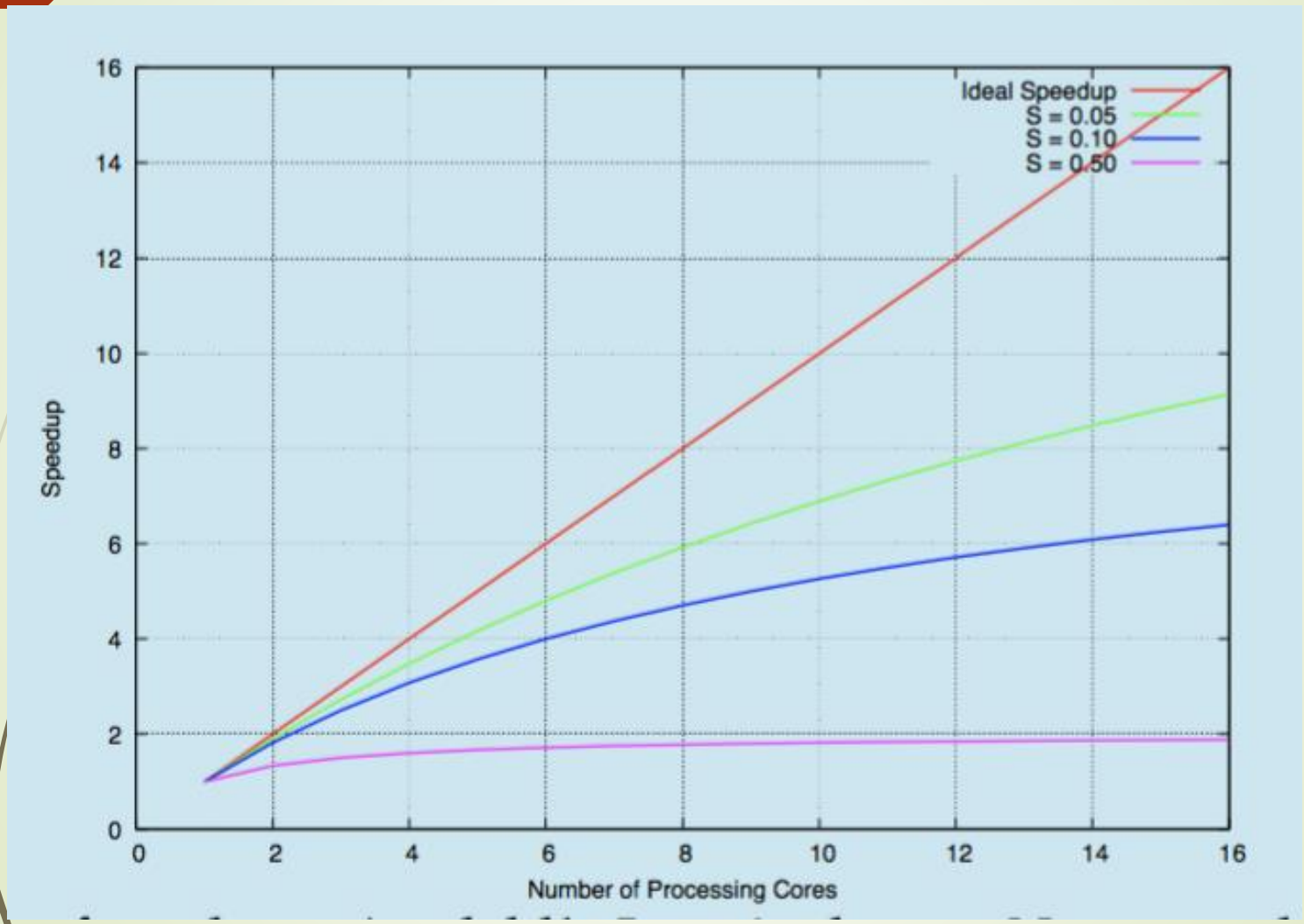
- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
  - *S* is serial portion, *N* processing cores

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

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

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

Amdahl's Law

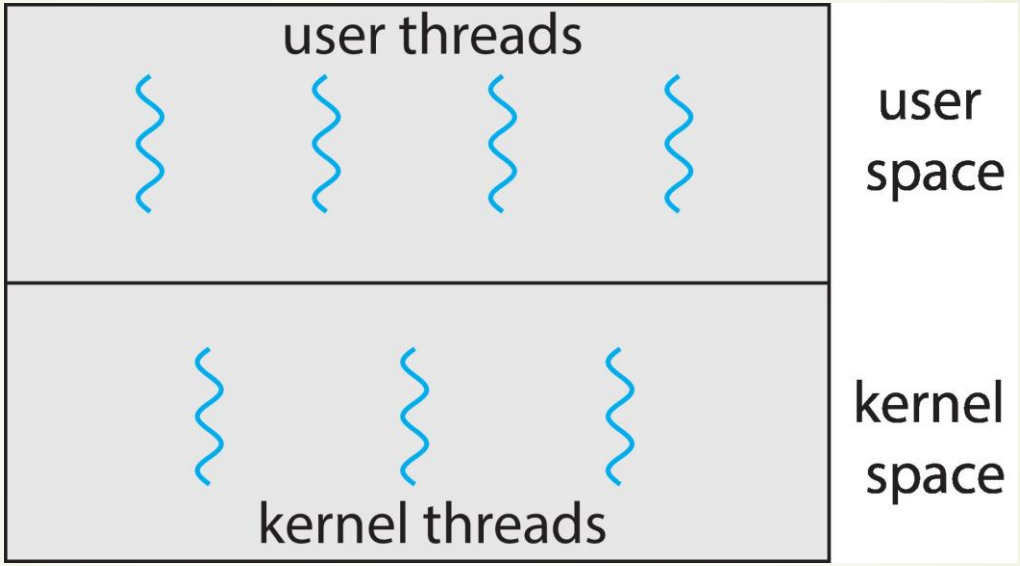


[Source: Operating Systems Concepts, 10th ed.]

User Threads and Kernel Threads

- **User threads** - managed by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general-purpose OS, including:
  - Windows
  - Linux
  - Mac OS X
  - iOS
  - Android

User and Kernel Threads



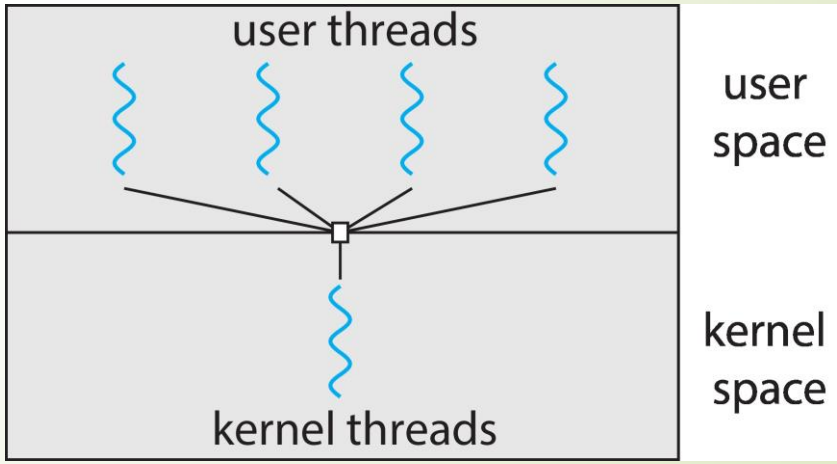
[Source: Operating Systems Concepts, 10th ed.]

Multithreading Models

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

Many-to-One

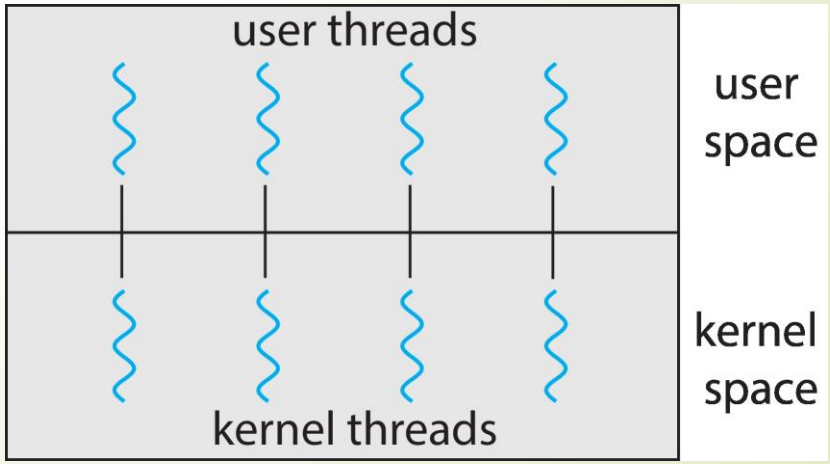
- Many user-level threads mapped to single kernel thread
  - One thread blocking causes all to block
  - Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU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



[Source: Operating Systems Concepts, 10th ed.]

One-to-One

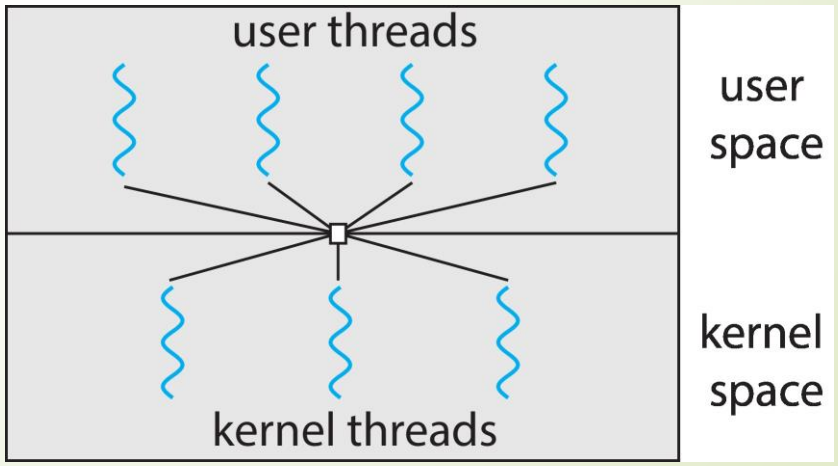
- Each user-level thread maps to a kernel thread
  - Creating a user-level thread creates a kernel thread
  - More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux



[Source: Operating Systems Concepts, 10th ed.]

Many-to-Many Model

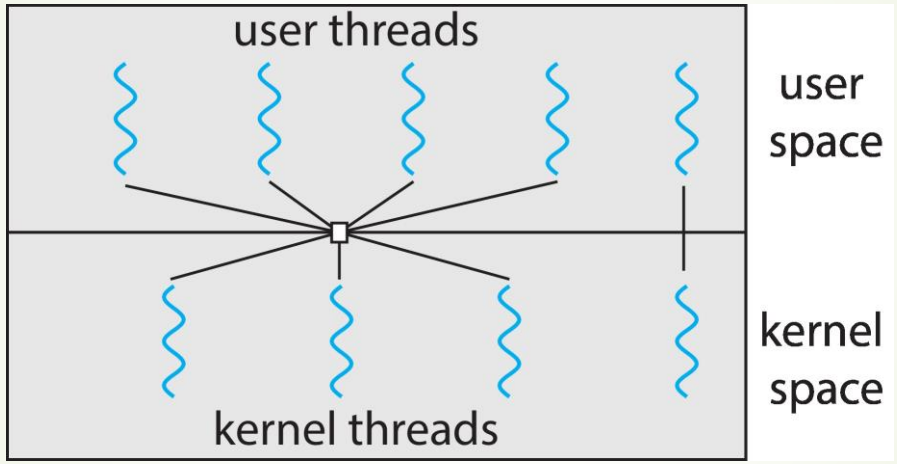
- Allows many user level threads to be mapped to many kernel threads
  - Allows the OS to create a sufficient number of kernel threads
- Windows with the *ThreadFiber* package
- Otherwise not very common



[Source: Operating Systems Concepts, 10th ed.]

Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread



[Source: Operating Systems Concepts, 10th ed.]

Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Pthreads

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

Pthreads Example

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

#include <stdlib.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 */

   /* set the default attributes of the thread */
   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);
}
```

[Source: Operating Systems Concepts, 10th ed.]

Pthreads Example (Cont.)

```
/* The thread will execute 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);
}
```

[Source: Operating Systems Concepts, 10th ed.]

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);
```

[Source: Operating Systems Concepts, 10th ed.]

Operating System Examples

- Linux Threads

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.

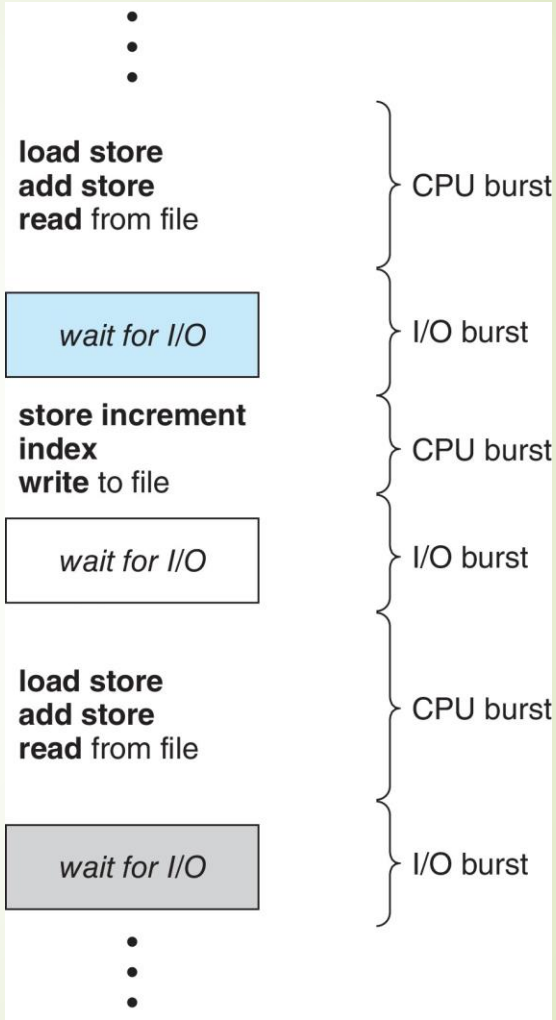
- **`struct task_struct`** points to process data structures (shared or unique)

[Source: Operating Systems Concepts, 10th ed.]

Process Scheduling

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle
  - Process execution consists of a **cycle** of CPU execution and I/O wait
  - **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern

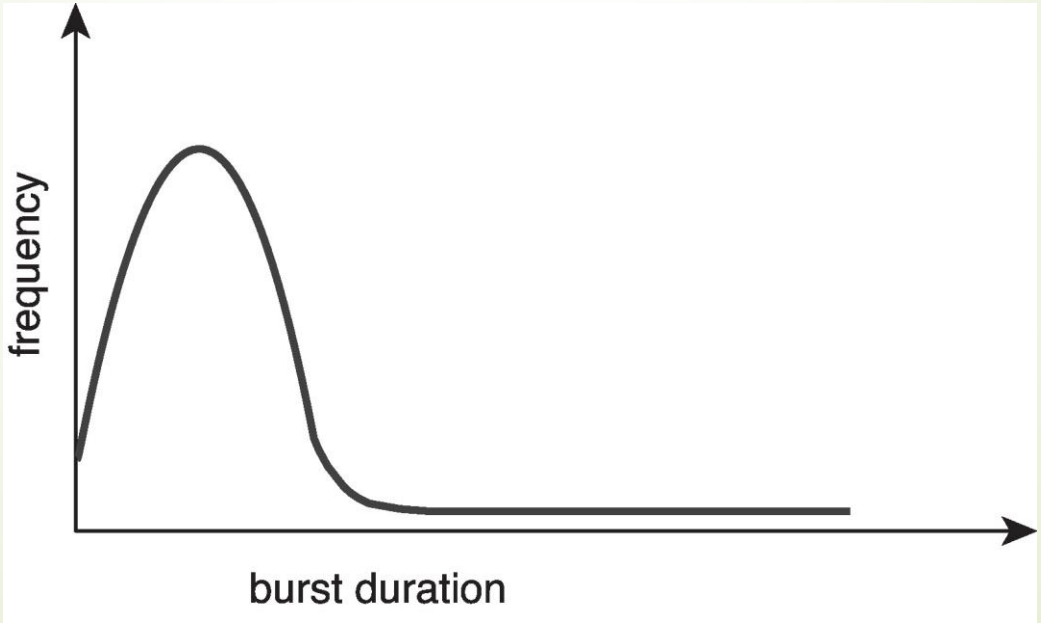


[Source: Operating Systems Concepts, 10th ed.]

Histogram of CPU-burst Times

Large number of short bursts

Small number of longer bursts



[Source: Operating Systems Concepts, 10th ed.]

CPU Scheduler

- The **CPU scheduler** selects from the processes in ready queue, and allocates a CPU core
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready state
  4. Terminates
- For situations 1 and 4, there is no choice in terms of scheduling
  - A new process (if one exists in the ready queue) must be selected for execution
- For situations 2 and 3, however, there is a choice

Preemptive and Nonpreemptive Scheduling

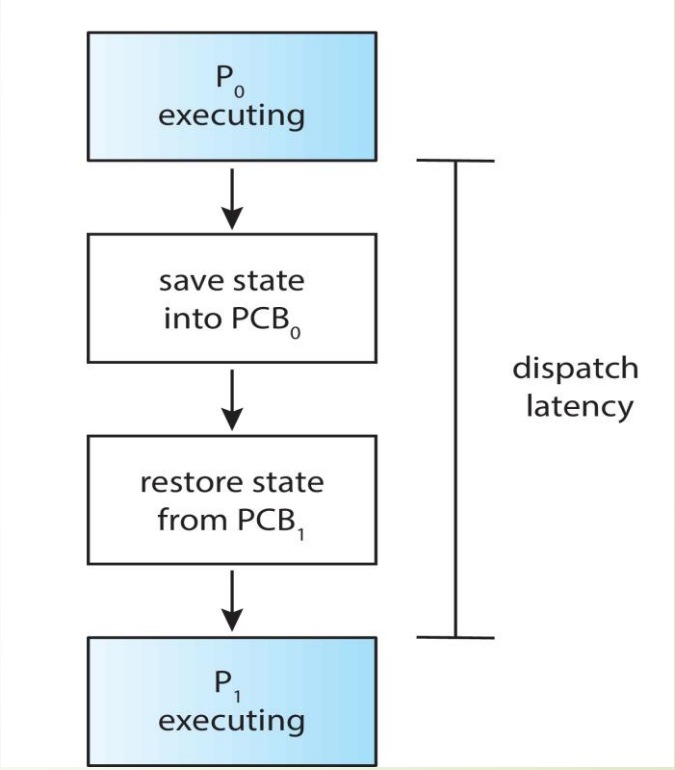
- When scheduling takes place only under situations 1 and 4, the scheduling scheme is **nonpreemptive**
  - Otherwise, it is **preemptive**
  - Under Nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state
- Virtually all modern OS including Windows, MacOS, Linux, and UNIX use preemptive scheduling algorithms

Preemptive Scheduling and Race Conditions

- Preemptive scheduling can result in race conditions when data are shared among several processes
- Consider the case of two processes that share data
  - While one process is updating the data, it is preempted so that the second process can run
  - The second process then tries to read the data, which are in an inconsistent state
- This issue will be explored in detail in Chap. 6

Dispatcher

- Dispatcher gives CPU control to the process selected by the CPU scheduler; this involves:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart it
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running



[Source: Operating Systems Concepts, 10th ed.]

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced

Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

CPU Scheduling Algorithms

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3 The Gantt Chart for the schedule is:

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
  - Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

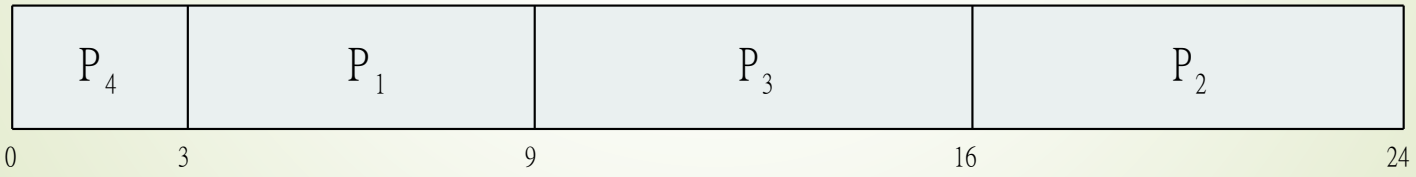
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Schedule the process with the shortest length
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user

Example of SJF

Process	Arrival e	Burst Time
P_1	0.0	6
P_2	2.0	8
P_3	4.0	7
P_4		3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Shortest-Job-First (SJF) Scheduling

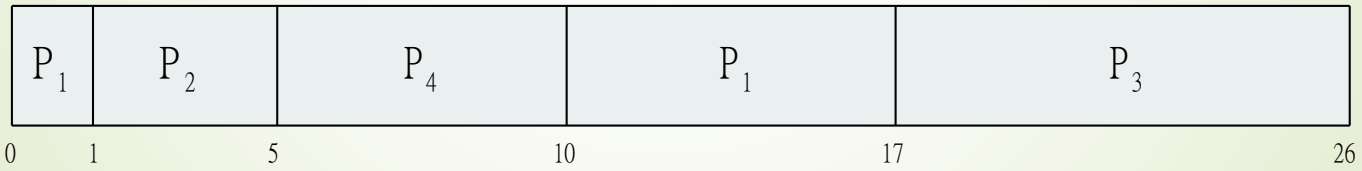
- Preemptive version of SJF is called **shortest-remaining-time-first**
- How do we determine the length of the next CPU burst?
  - Could ask the user
  - Estimate

Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5

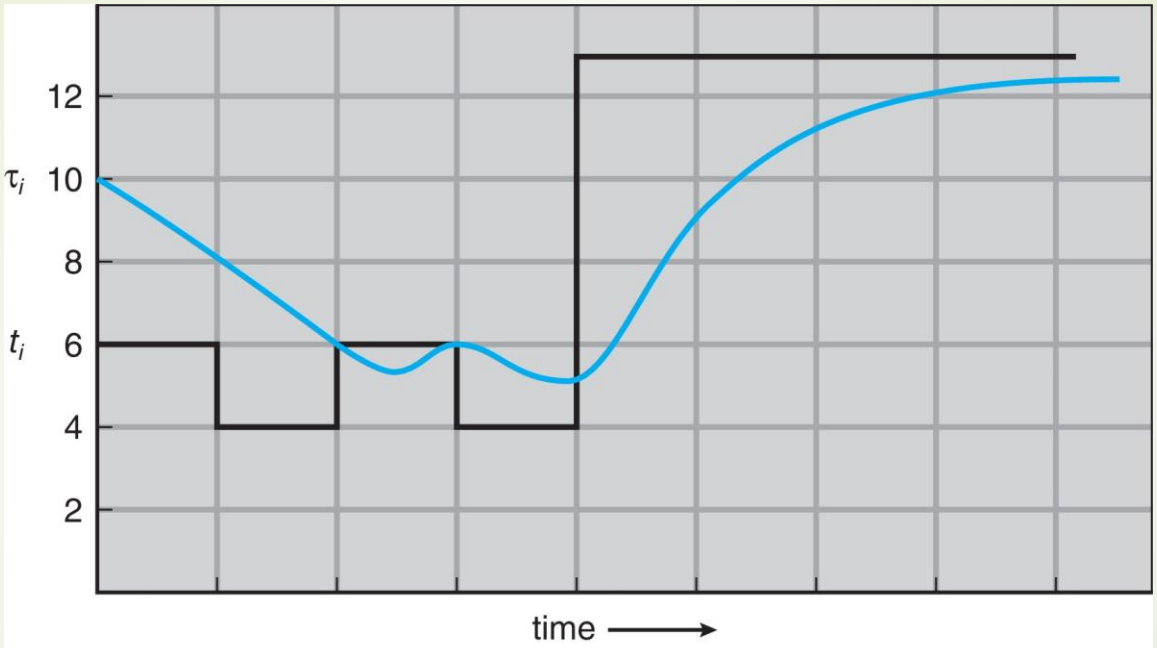
Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define : $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$

- Commonly, α set to ½

Example Prediction of the Length of the Next CPU Burst



CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

[Source: Operating Systems Concepts, 10th ed.]

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots$$
$$+ (1 - \alpha)^j \alpha\, t_{n-j} + \ldots$$
$$+ (1 - \alpha)^{n+1} \tau_0$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

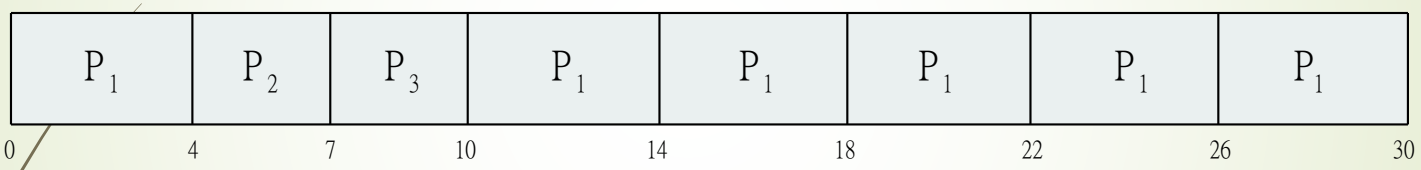
Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds
  - After this time has elapsed, the process is preempted and added to the end of the ready queue
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units
  - No process waits more than $(n-1)q$ time units
  - Timer interrupts every quantum to schedule next process
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

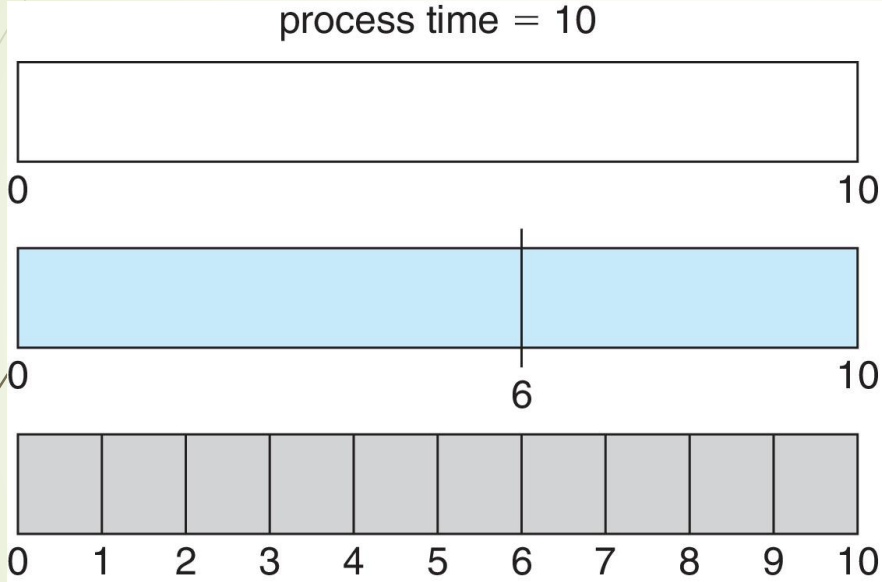
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
  - q usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds

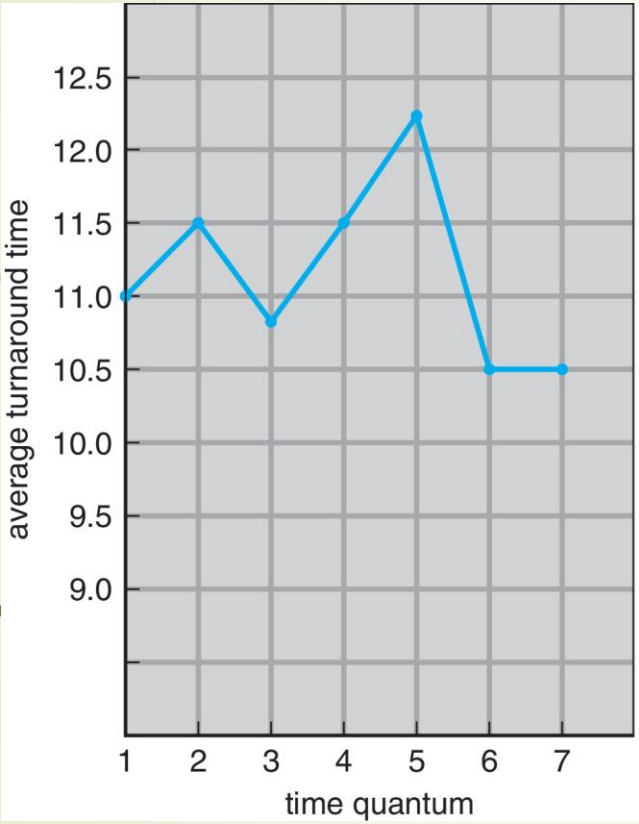
Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9

[Source: Operating Systems Concepts, 10th ed.]

Turnaround Time Varies With the Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

* Rule of thumb:
80% of CPU bursts
should be shorter than q

[Source: Operating Systems Concepts, 10th ed.]

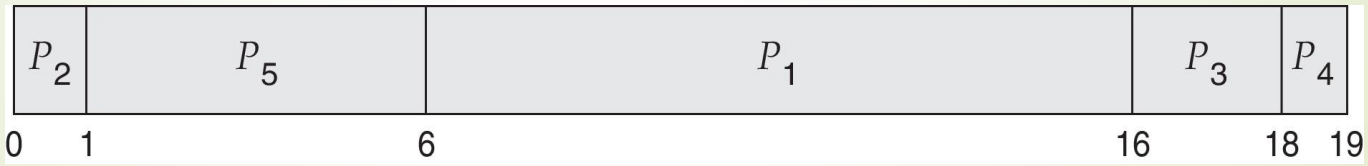
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
  - SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses the priority of the process increases

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart

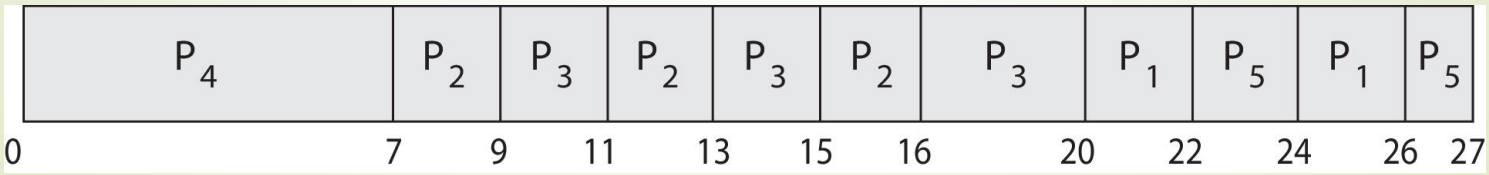


- Average waiting time = 8.2

Priority Scheduling w/ Round-Robin

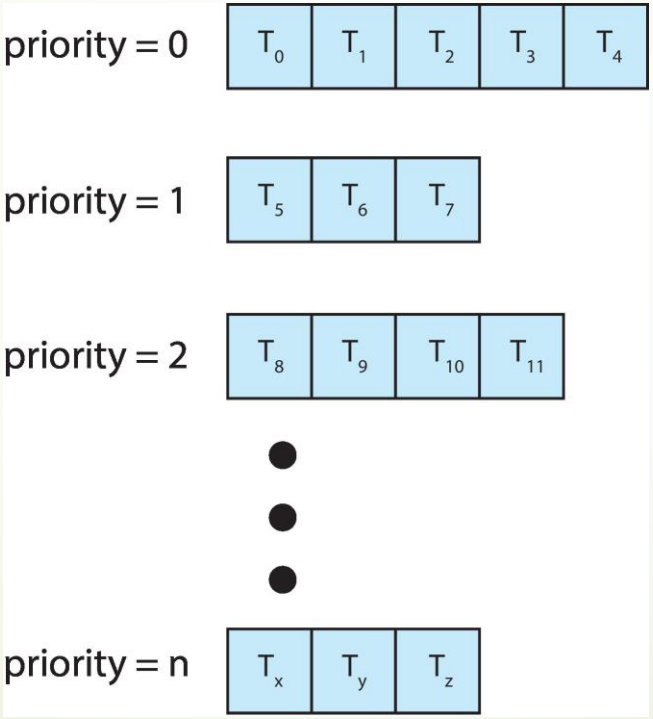
Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Run the process with the highest priority
- Processes with the same priority run round-robin
- Gantt Chart with time quantum = 2



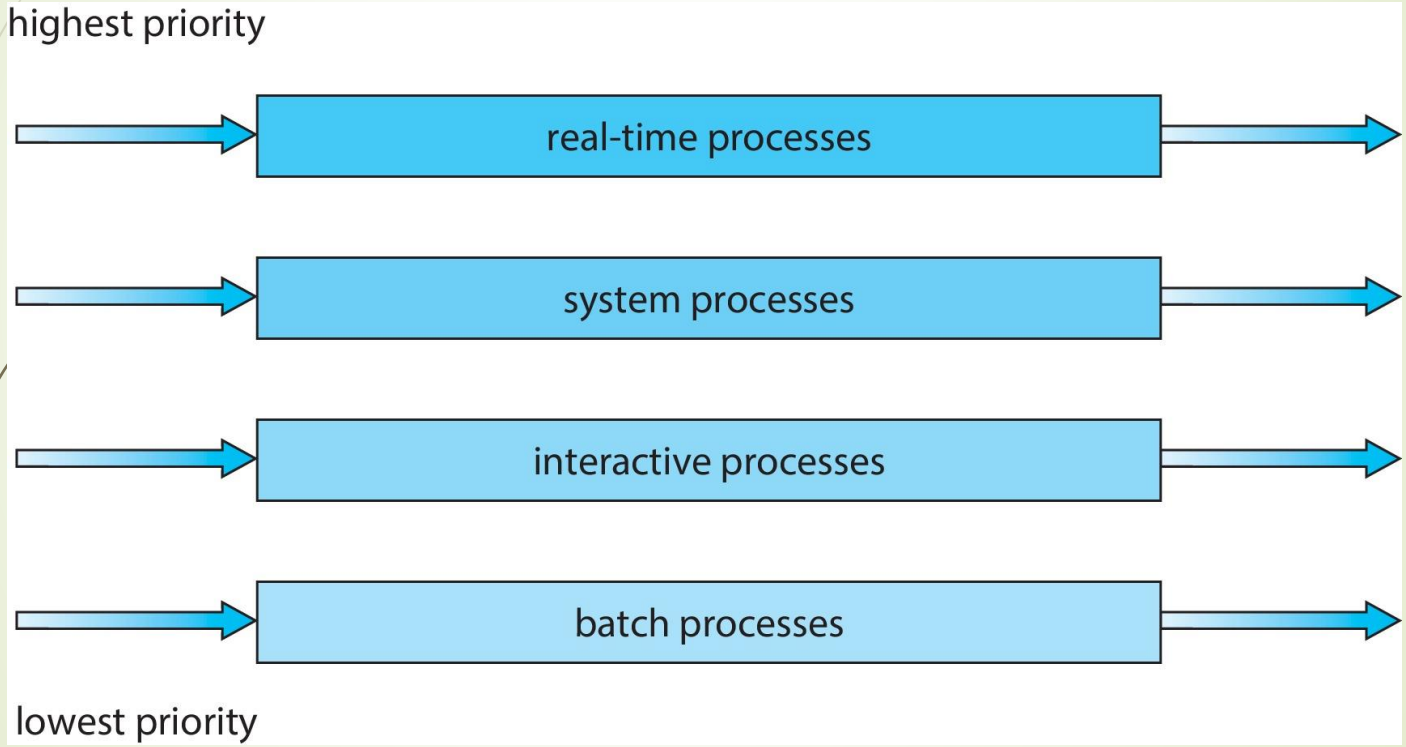
Multilevel Queue

- With priority scheduling, have separate queues for each priority
- Schedule the process in the highest-priority queue!



[Source: Operating Systems Concepts, 10th ed.]

Multilevel Queue



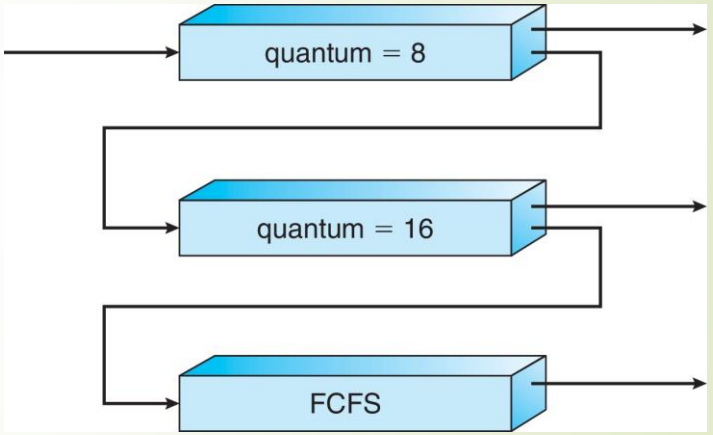
[Source: Operating Systems Concepts, 10th ed.]

Multilevel Feedback Queue

- A process can move between various queues
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine when to upgrade a process
  - Method used to determine when to demote a process
  - Method used to determine which queue a process will enter when that process needs service
- Aging can be implemented using multilevel feedback queue

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new process enters queue Q_0 which is served in RR
    - When it gains CPU, the process receives 8 milliseconds
    - If it does not finish in 8 milliseconds, the process is moved to queue Q_1
  - At Q_1 job is again served in RR and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



[Source: Operating Systems Concepts, 10th ed.]

Thread Scheduling

- Distinction between user-level and kernel-level threads
  - When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope** (**PCS**) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope** (**SCS**) – competition among all threads in system

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and macOS only allow PTHREAD_SCOPE_SYSTEM

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

[Source: Operating Systems Concepts, 10th ed.]

Pthread Scheduling API

```
   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

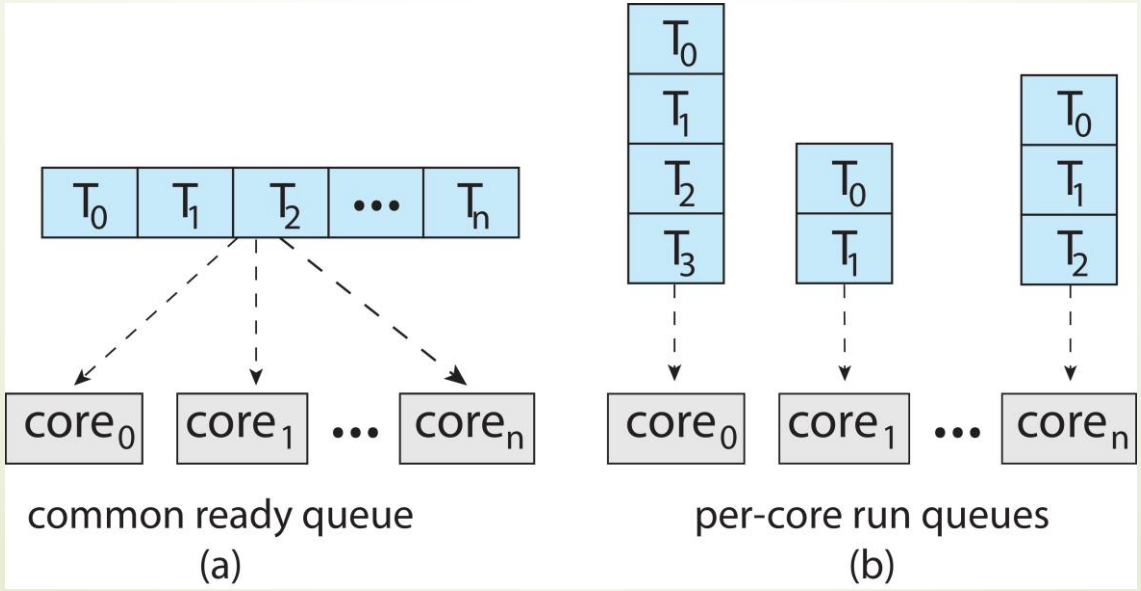
[Source: Operating Systems Concepts, 10th ed.]

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- Multiprocess may be any one of the following architectures:
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems
  - Heterogeneous multiprocessing

Multiple-Processor Scheduling

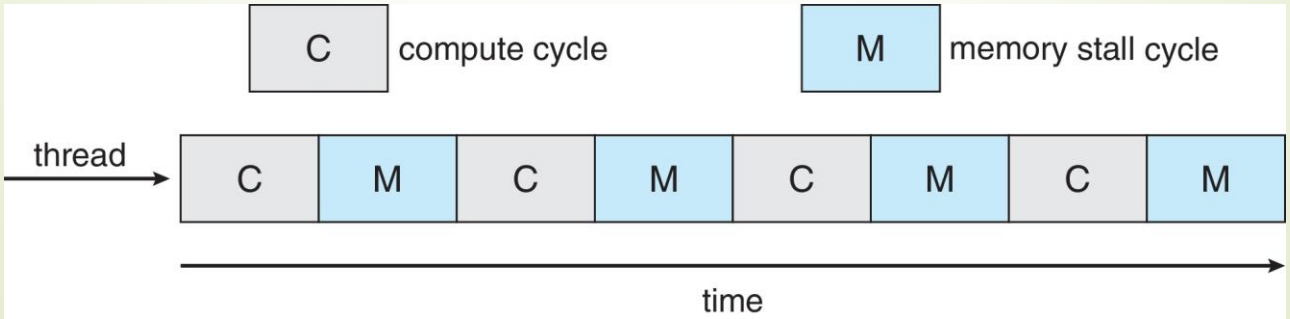
- Symmetric multiprocessing (SMP) is where each processor is self-scheduling
  - (a) All threads may be in a common ready queue
  - (b) Each processor may have its own private queue of threads



[Source: Operating Systems Concepts, 10th ed.]

Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
  - Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

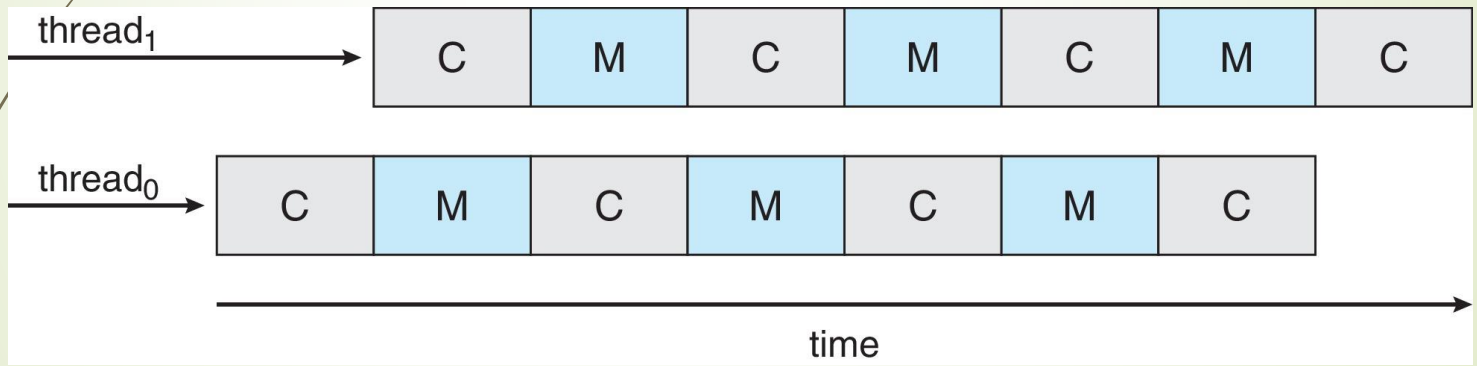


[Source: Operating Systems Concepts, 10th ed.]

Multithreaded Multicore System

Each core has > 1 hardware threads

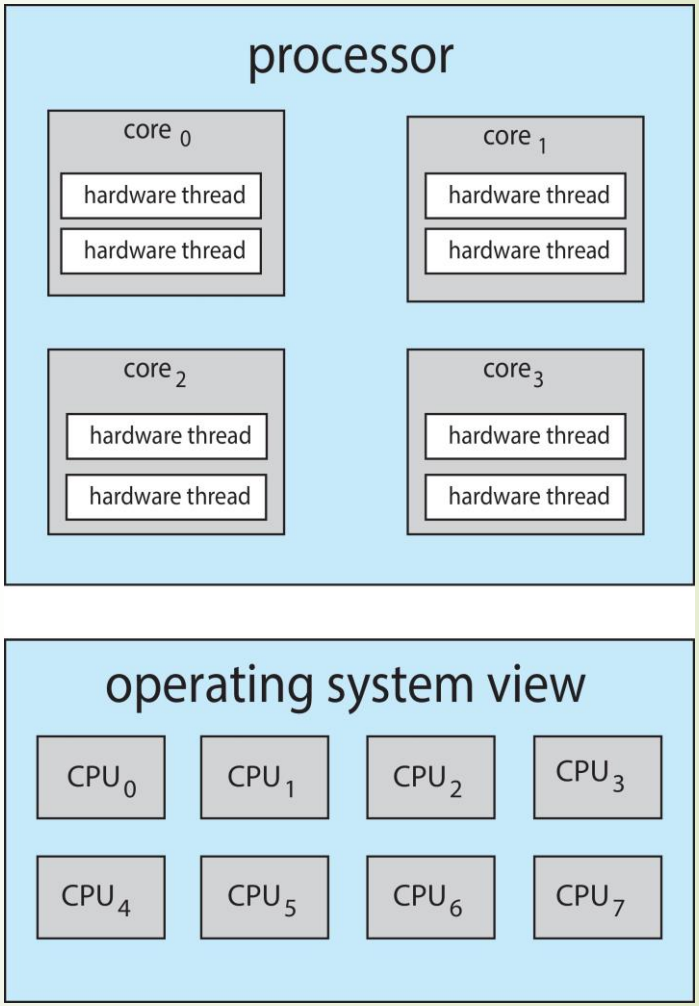
If one thread has a memory stall, switch to another thread!



[Source: Operating Systems Concepts, 10th ed.]

Multithreaded Multicore System

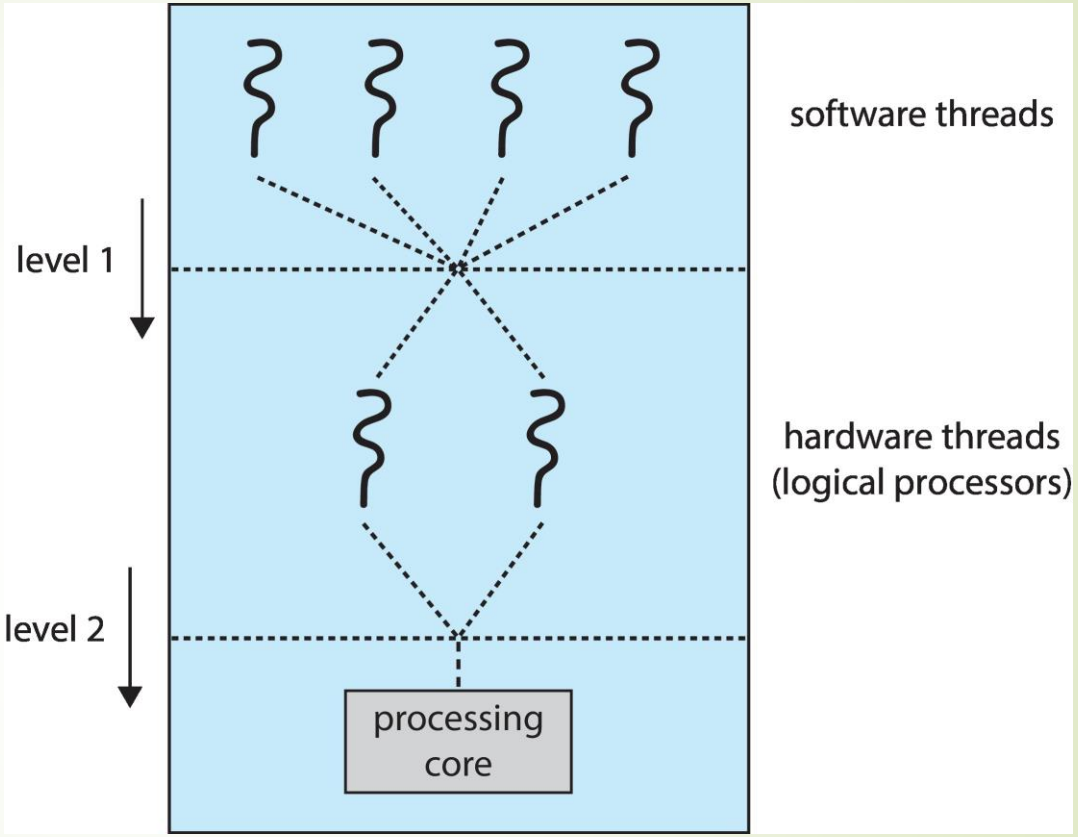
- **Chip-multithreading** (CMT) assigns each core multiple hardware threads (Intel refers to this as **hyperthreading**)
- On a quad-core system with 2 hardware threads per core, the OS sees 8 logical processors



[Source: Operating Systems Concepts, 10th ed.]

Multithreaded Multicore System

- Two levels of scheduling:
  1. The OS decides which software thread to run on a logical CPU
  2. Each core decides which hardware thread to run on the physical core



[Source: Operating Systems Concepts, 10th ed.]

Multiple-Processor Scheduling – Load Balancing

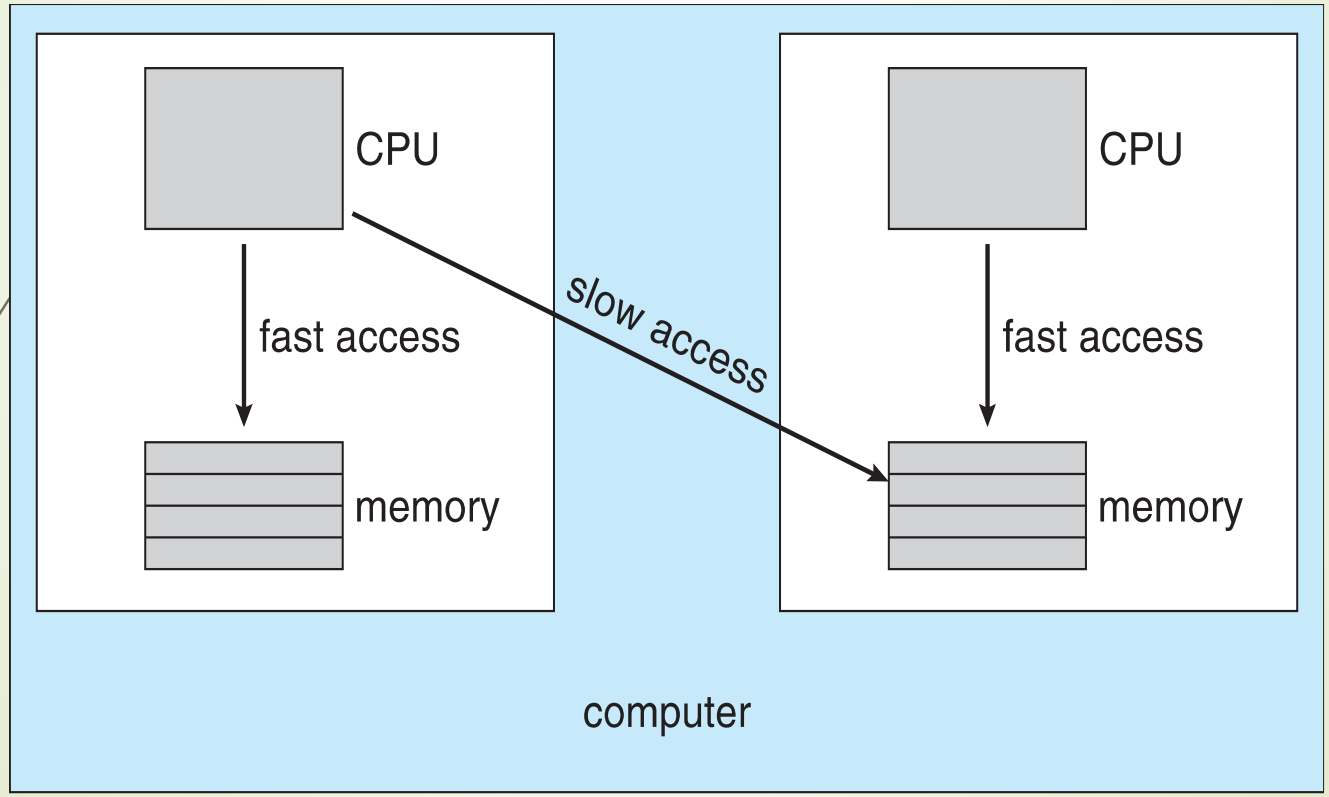
- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor

Multiple-Processor Scheduling – Processor Affinity

- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread
  - Migrating the thread to another processor has high cost, and should be avoided
  - We refer to this as a thread having affinity for a processor (i.e., "processor affinity")
- **Soft affinity** – the OS attempts to keep a thread running on the same processor, but no guarantees
- **Hard affinity** – allows a process to specify a set of processors it may run on

NUMA and CPU Scheduling

If the OS is **NUMA-aware**, it will assign memory closest to the CPU the thread is running on



[Source: Operating Systems Concepts, 10th ed.]

Operating System Examples

- Linux scheduling
- Solaris scheduling

Linux Scheduling Through Version 2.5

- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order O(1) scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
    - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
    - Map into global priority with numerically lower values indicating higher priority
  - Higher priority gets larger q
    - Task run-able as long as time left in time slice (**active**)
    - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU **runqueue** data structure
    - Two priority arrays (active, expired) indexed by priority
    - When no more active, arrays are exchanged
  - Worked well, but poor response times for interactive processes

Linux Scheduling in Version 2.6.23 +

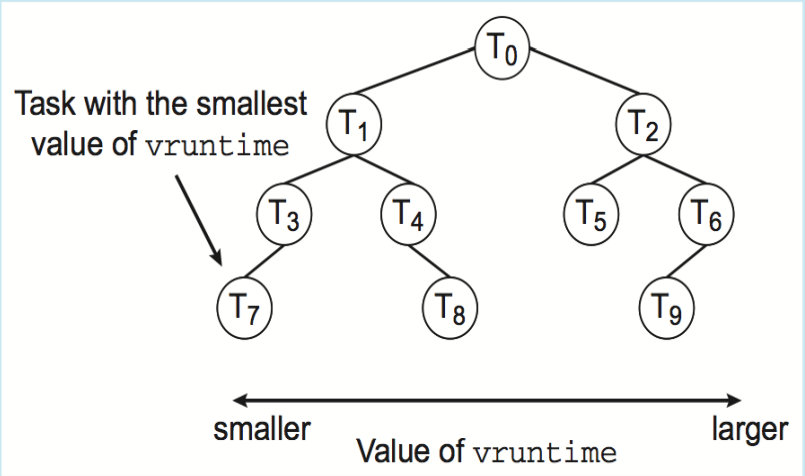
- ***Completely Fair Scheduler*** (CFS)
- **Scheduling classes**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
    - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - Two scheduling classes included, others can be added
    1. default
    2. real-time

Linux Scheduling in Version 2.6.23 + (Cont.)

- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if the number of active tasks increases
- CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**
  - Associated with decay factor based on priority of task – lower priority has higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time

CFS Performance

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:

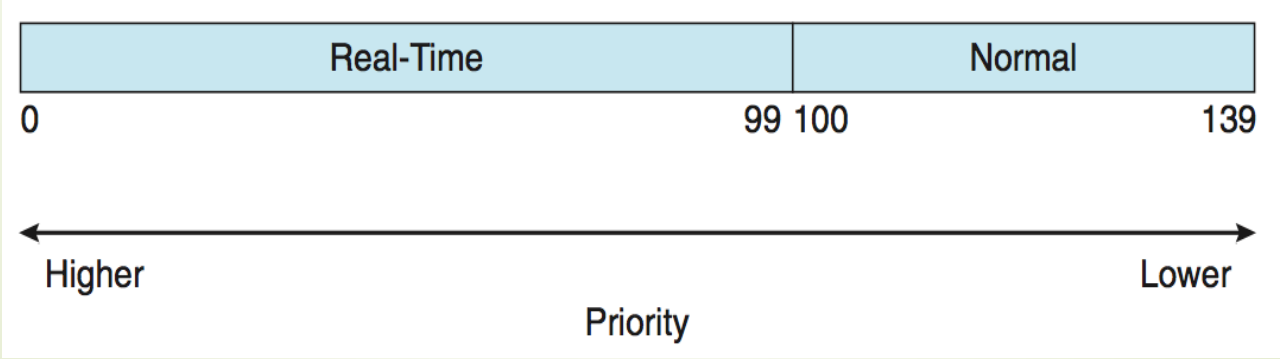


When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

[Source: Operating Systems Concepts, 10th ed.]

Linux Scheduling (Cont.)

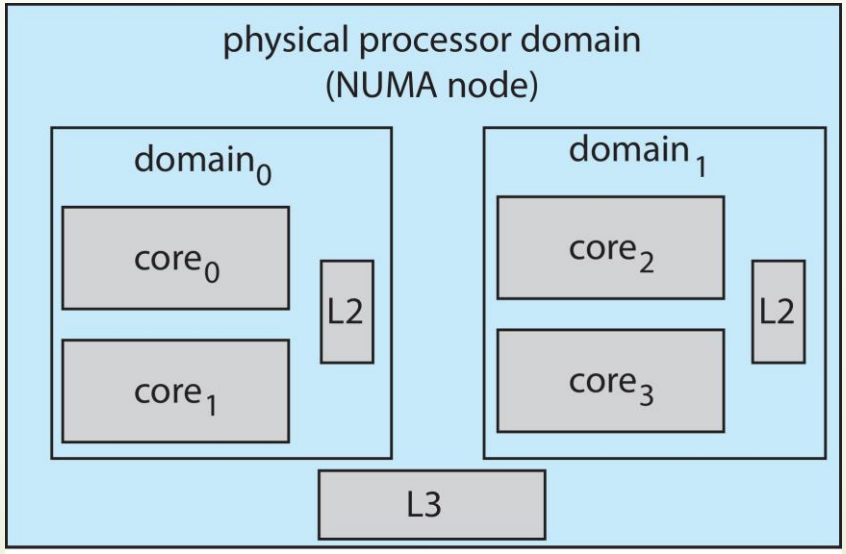
- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
  - Nice value of -20 maps to global priority 100
  - Nice value of +19 maps to priority 139



[Source: Operating Systems Concepts, 10th ed.]

Linux Scheduling (Cont.)

- Linux supports load balancing, but is also NUMA-aware
- **Scheduling domain** is a set of CPU cores that can be balanced against one another
- Domains are organized by what they share (i.e., cache memory) Goal is to keep threads from migrating between domains



[Source: Operating Systems Concepts, 10th ed.]

Solaris

- Priority-based scheduling
- Six classes available
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
  - Loadable table configurable by sysadmin

Solaris Dispatch Table

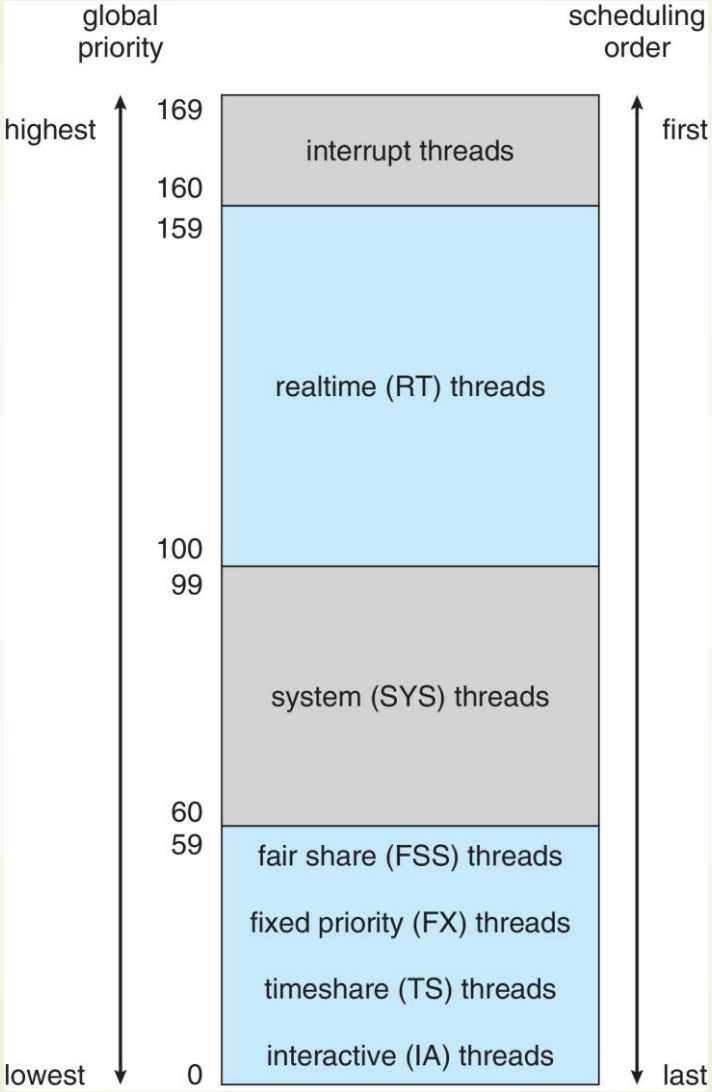
low

high

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

[Source: Operating Systems Concepts, 10th ed.]

Solaris Scheduling



[Source: Operating Systems Concepts, 10th ed.]

Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
  - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR

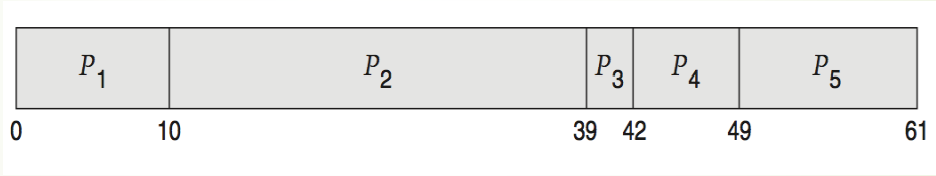
Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
  - Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

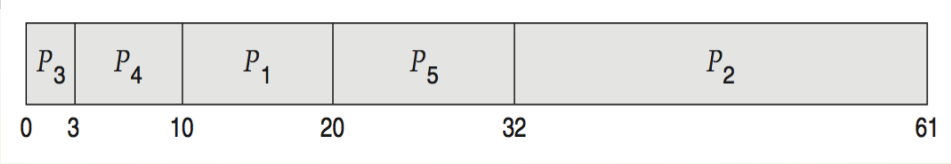
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Evaluation

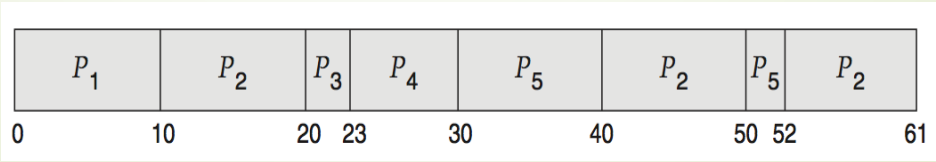
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:



  - Non-preemptive SFJ is 13ms:



  - RR is 23ms:



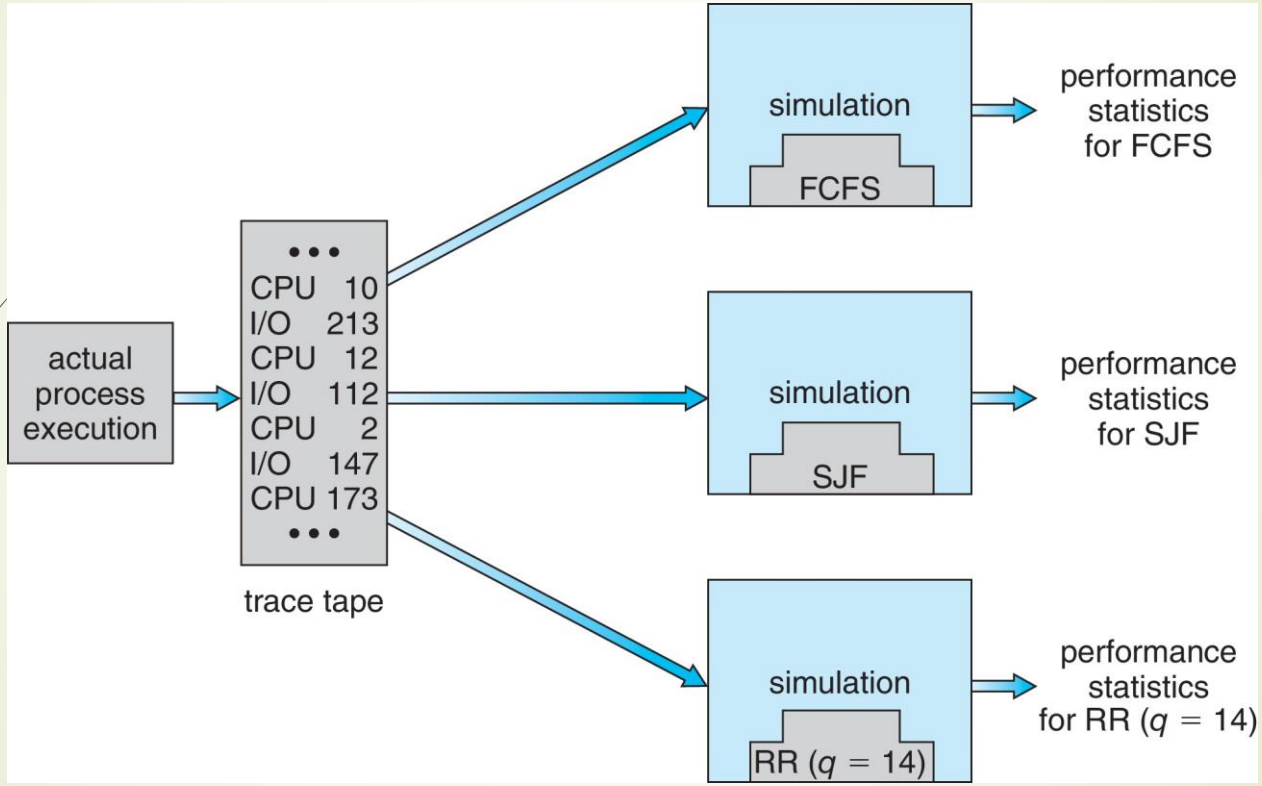
Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc

Simulations

- Queueing models limited
- **Simulations** more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems

Evaluation of CPU Schedulers by Simulation



[Source: Operating Systems Concepts, 10th ed.]

Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary

End of Module 1 – Part I