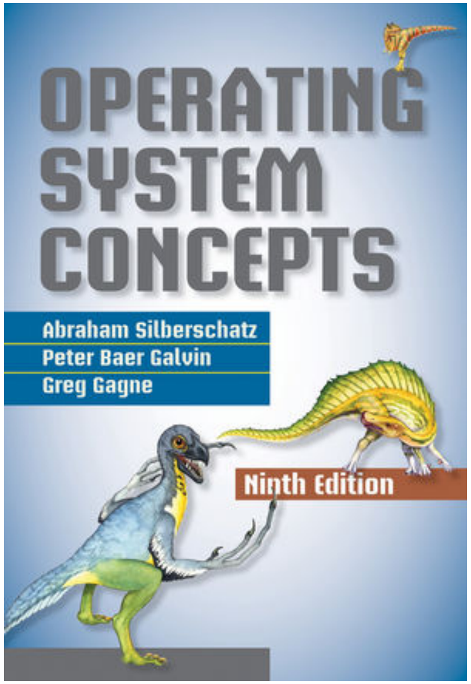
3 – Processes and Threads

EECE 315 (101)
ECE – UBC
2013 W2



Acknowledgement: This set of slides is partly based on the PPTs provided by the Wiley's companion website (including textbook images, when not explicitly mentioned/referenced).

Lecture Outline

- **Processes and Scheduling**
  - What is a process ?
  - How are processes created ?

- Inter-process Communication (IPC)
  - Examples of IPC Systems

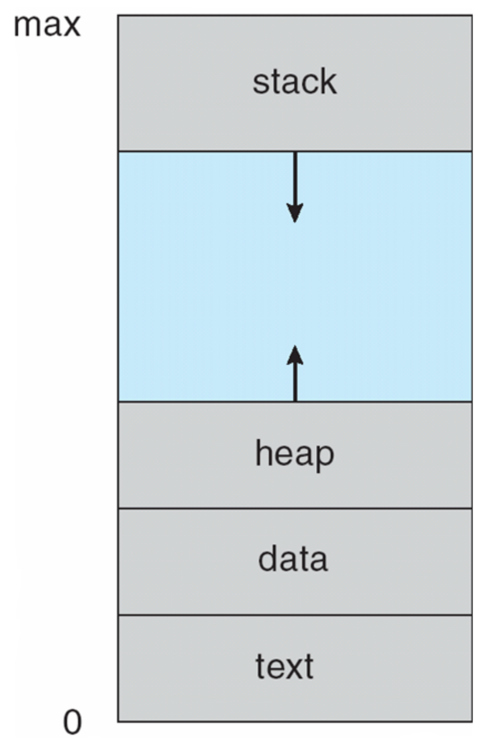
- Threads and multi-threading

What's a Process ?

- A Process is
  - a program in execution
  - the unit of work in a modern time-sharing system
    - (To be compared with threads in Ch 4)
- A program by itself is not a process; a program is a *passive* entity (e.g. a file on disk), whereas a process is an *active* entity.
- Necessary for multi-programming and multi-tasking

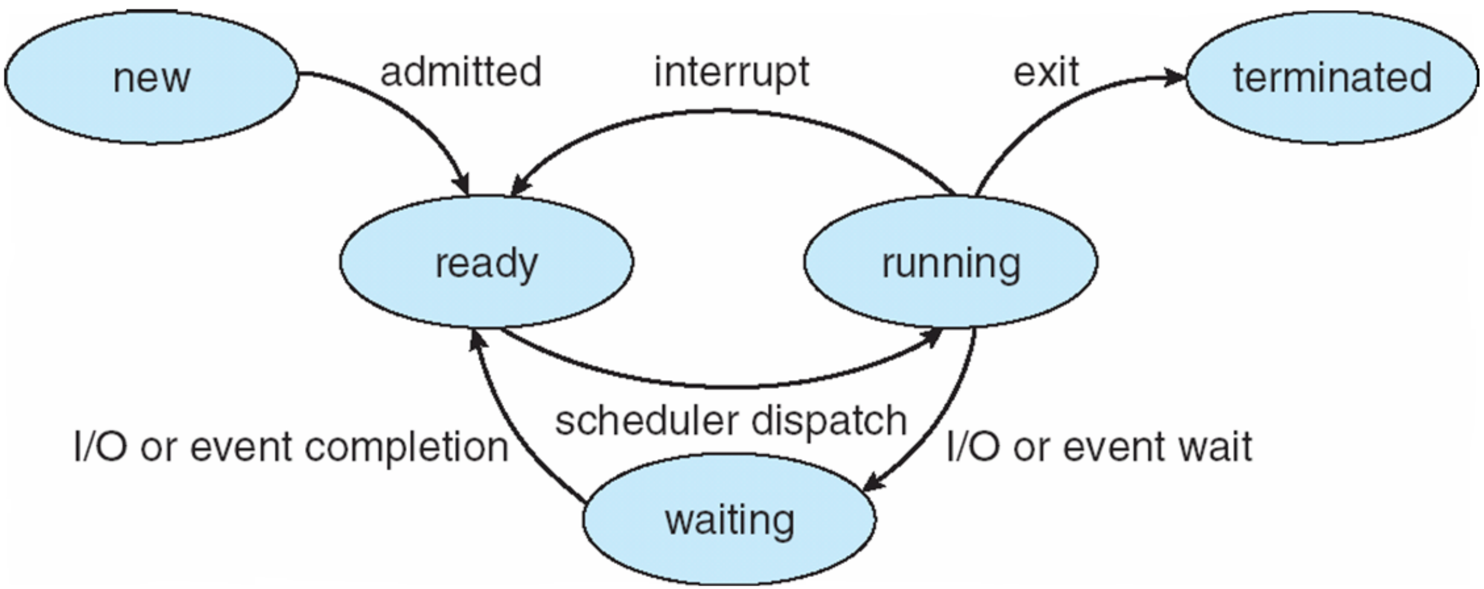
What's a Process (cont)

- A process execution must progress in sequential fashion
- A process is more than the program code (***text section***)
- A process also includes:
  - the *current activity*
    - program counter
    - contents of the processor's registers
  - generally the ***stack*** (temporary data such as function parameters, return addresses, …) and a ***data section*** (e.g. global variables)
  - and it may also include a ***heap*** (dynamically allocated memory at run-time)



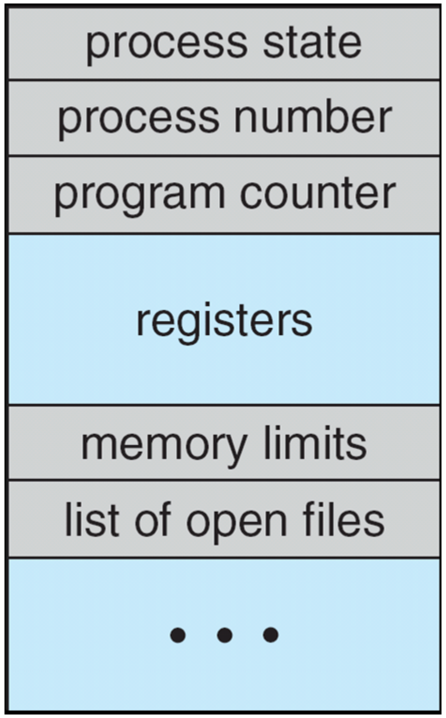
Process State

- The state of a process is defined by the current activity of that process. The states can be shown by a process state diagram.
- As a process executes, it changes *state* (note that these names are generic)
  - **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



Process Control Block (PCB)

- Each process is represented in the OS by a process control block (PCB).
- The PCB contains <u>many pieces of information</u> associated with a specific process:
  - Process state
    - e.g. new, ready, running, halted, …
  - Process number
  - Program counter
    - address of the next instruction to be executed
  - CPU registers
  - CPU scheduling information
    - e.g. priority and other sch. parameters
  - Memory-management information
  - Accounting information
  - I/O status information
    - e.g. list of open files, I/O devices allocated

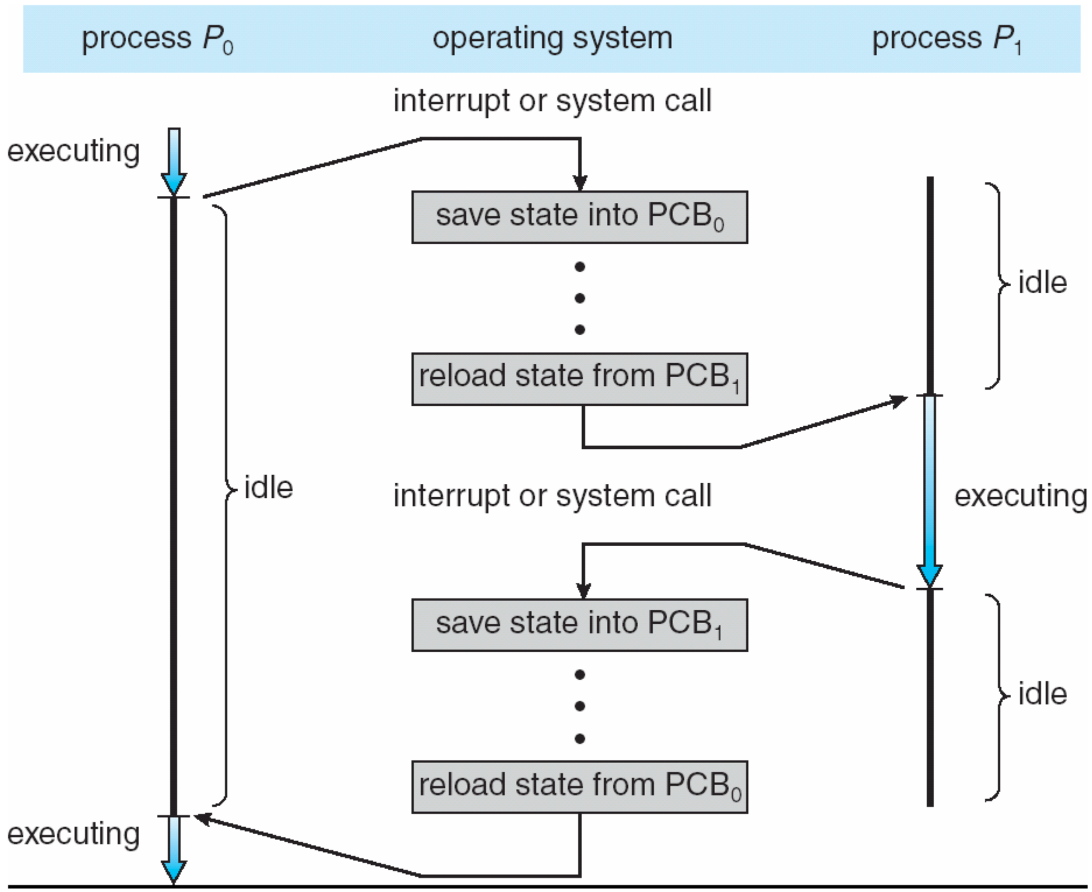


Context Switch

- Interrupts cause the OS to change a CPU from its current task and to run a kernel routine.
- When CPU switches to another process, the system must save the state of the old process (*state save*) and load the saved state for the new process (*state restore*) via a context switch
  - Context of a process represented in the PCB
    - including CPU registers, process state, and memory management information
- Context-switch time is overhead
  - The system does no useful work while switching
  - Time dependent on hardware support

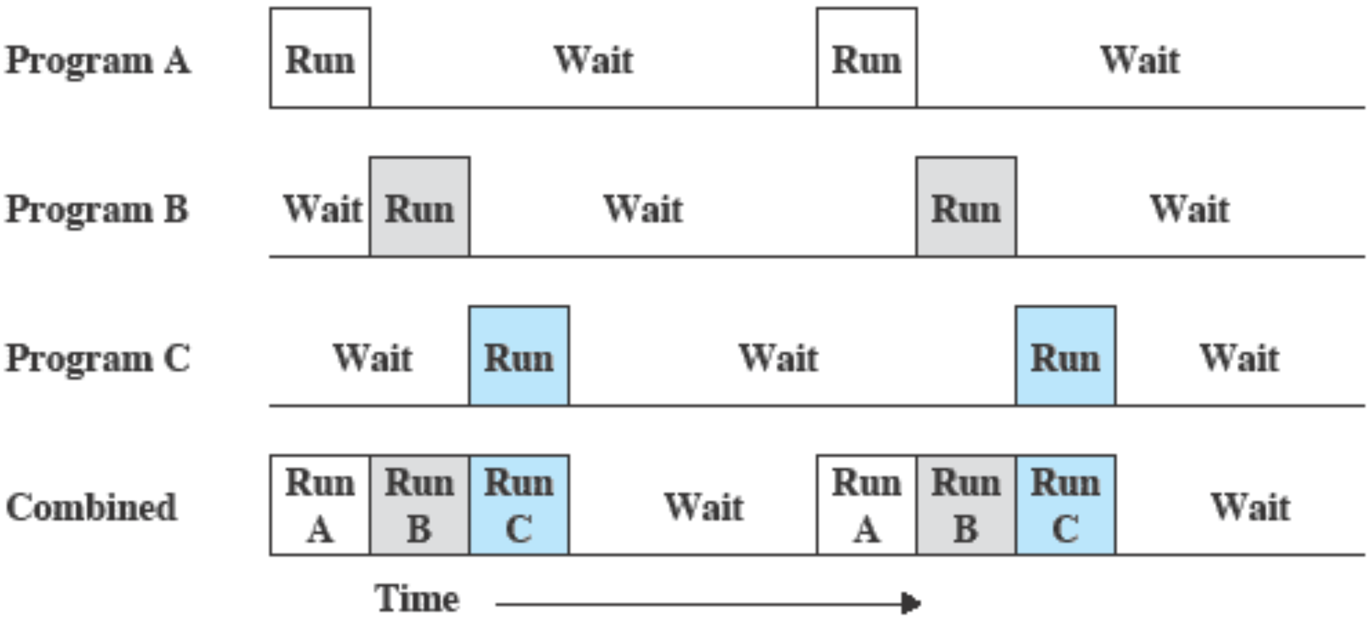
CPU Switch From Process to Process

- When the CPU switches from a process to another one (e.g. when an interrupt occurs), the state information (CPU registers, PC, …) must be saved to allow the process to be continued correctly afterward.



Process Scheduling

- *Objective of multiprogramming*: to have some process running at all times (maximization of CPU utilization)
- *Objective of time sharing*: to switch the CPU among processes so frequently that the users can interact with each program.

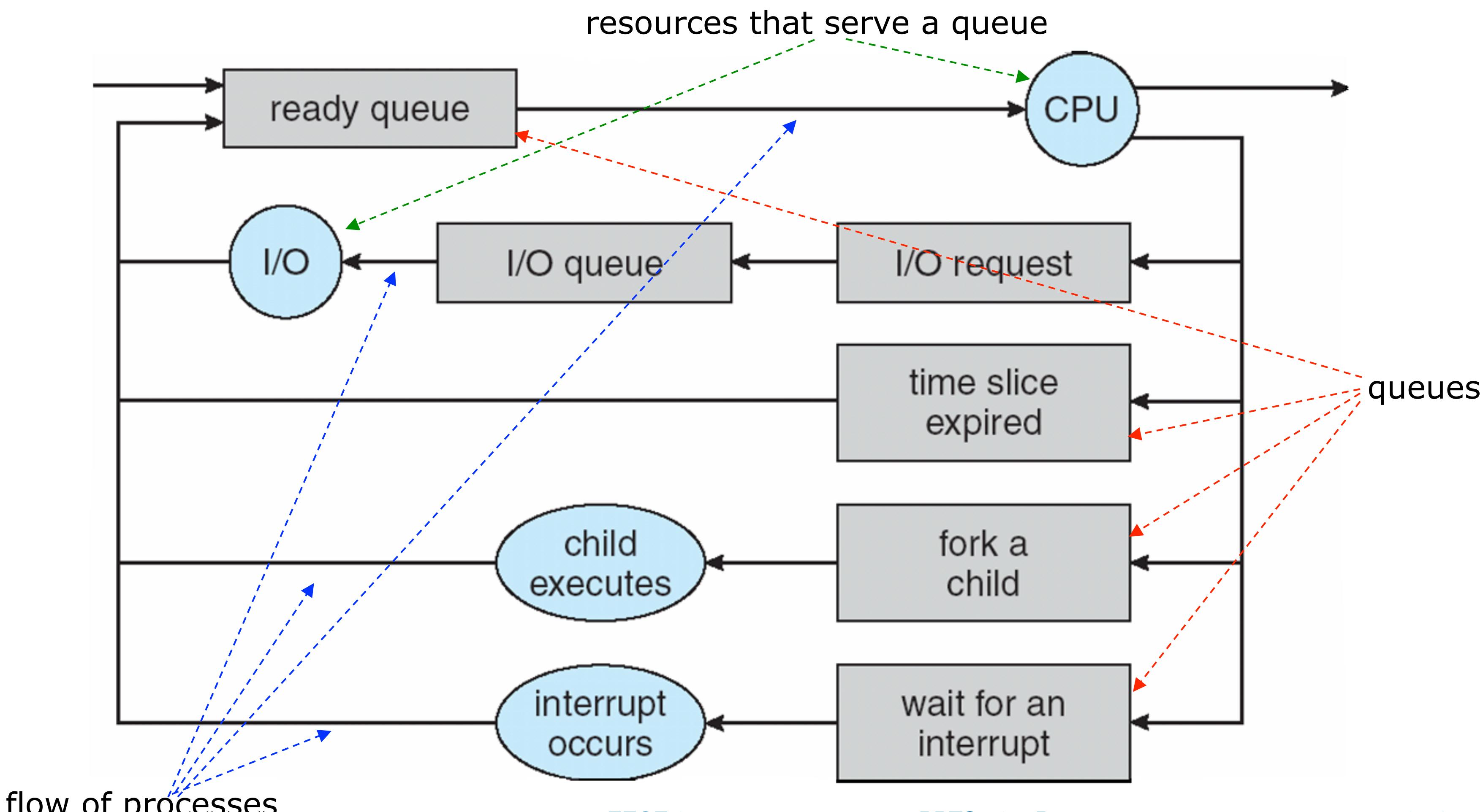


(c) Multiprogramming with three programs

source: Stallings'

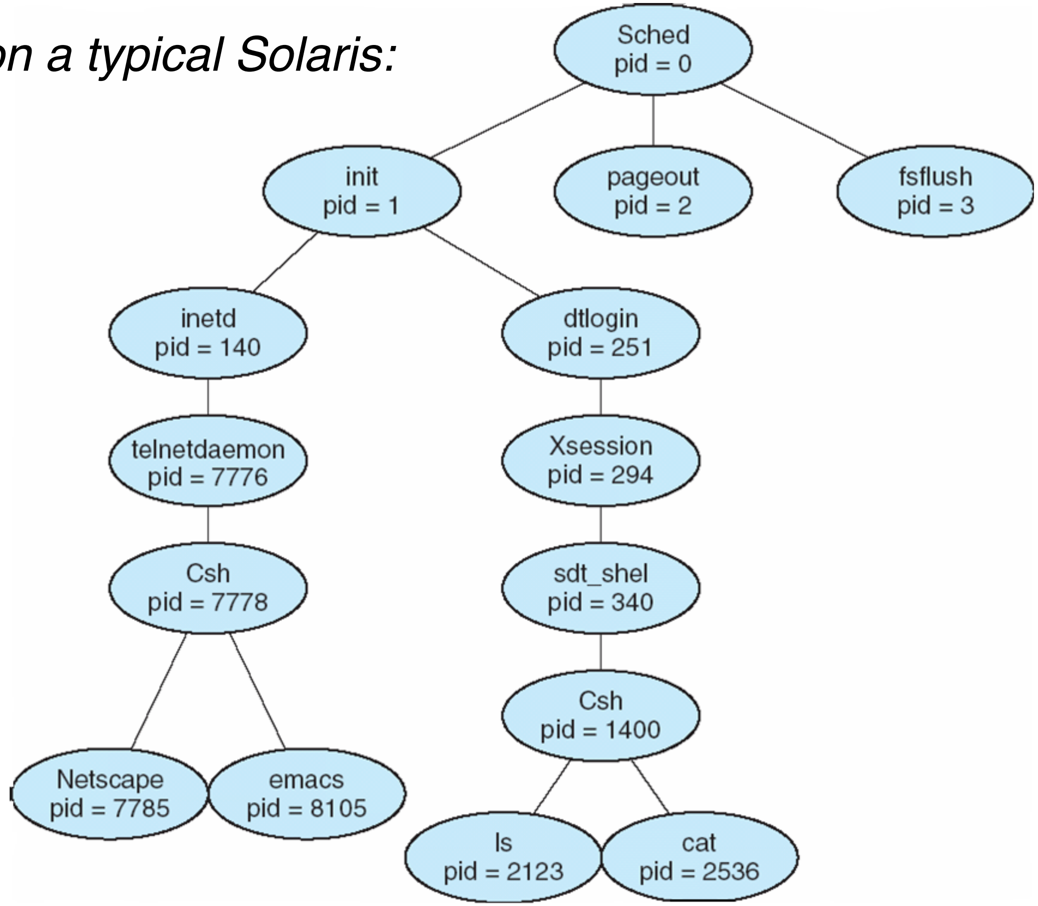
Representation of Process Scheduling

- A common representation of process scheduling is a *queuing diagram*.
- A new process is initially put in the ready queue. It waits there to be selected for execution (dispatched), then:



Process Creation

- **Parent** process create **children** processes, which, in turn may create other processes, forming a *tree of processes*
  - *e.g. a tree of processes on a typical Solaris:*



- *e.g. use "ps –el" command in UNIX*

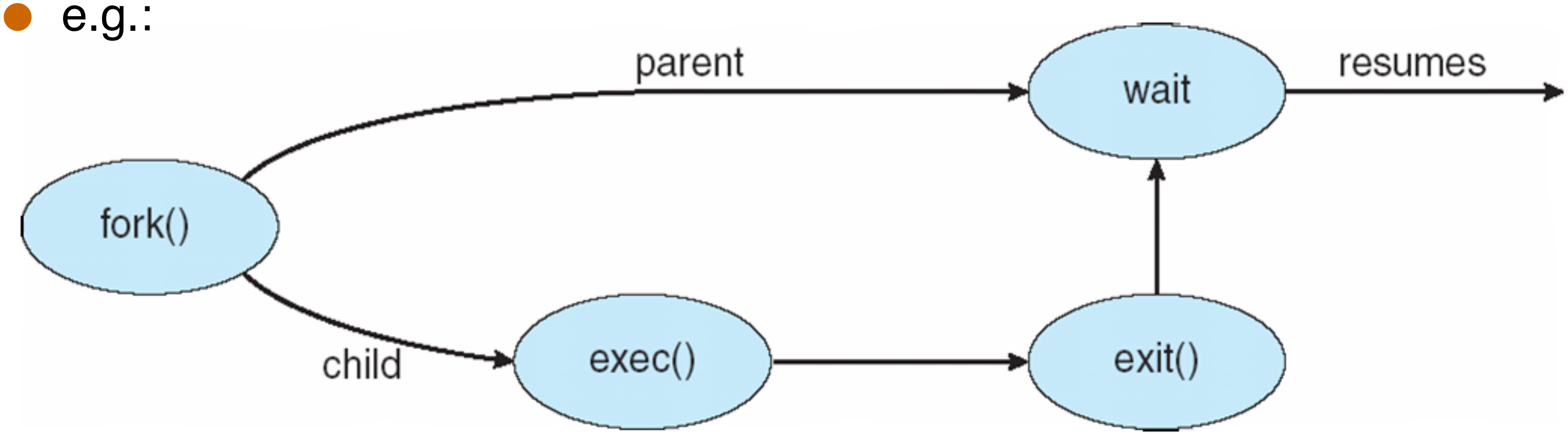
Process Creation (cont)

- Generally, a process is identified and managed via a unique **process identifier** (**pid**)
  - is an integer number
  - e.g. used in Windows and UNIX
- A process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its tasks.
- When a process creates a sub-process, then
  - resource sharing possibilities:
    - Parent and children may share all resources
    - Children may share subset of parent's resources
    - Parent and child may share no resources
  - execution possibilities:
    - Parent and children may execute concurrently
    - Parent may wait until children terminate

Process Creation (cont)

- (Continue):
  - address space possibilities:
    - Child duplicate of parent
    - Child has a program loaded into it

- UNIX examples
  - **fork** system call creates new process
  - **exec** system call is used after a **fork** to replace the process' memory space with a new program
  - e.g.:



C Program Forking Separate Process

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main(void)
{
    pid_t  pid;
   /* fork another 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;
}
```

Fig 3.10 of the textbook: An example of creating a separate process using the UNIX fork() system call

Process Termination

- A process normally terminates when it finished executing its last statement and asks the operating system to delete it (**`exit()`** system call)
  - All process' resources are deallocated by the OS
  - The process may return a status value to its parent (via **`wait()`** system call)

- Parent may terminate the execution of children processes (**`abort`**), Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - All children terminated - **cascading termination**

- Waiting for termination, returning the pid: **`pid = wait(&status);`**
  - If no parent waiting, then terminated process is a **zombie**
  - If parent terminated, processes are **orphans**

Lecture Outline

- Processes and Scheduling
  - What is a process ?
  - How are processes created ?

- **Inter-process Communication (IPC)**
  - Examples of IPC Systems
  - Synchronization and IPCs

- Threads and multi-threading

Interprocess Communication

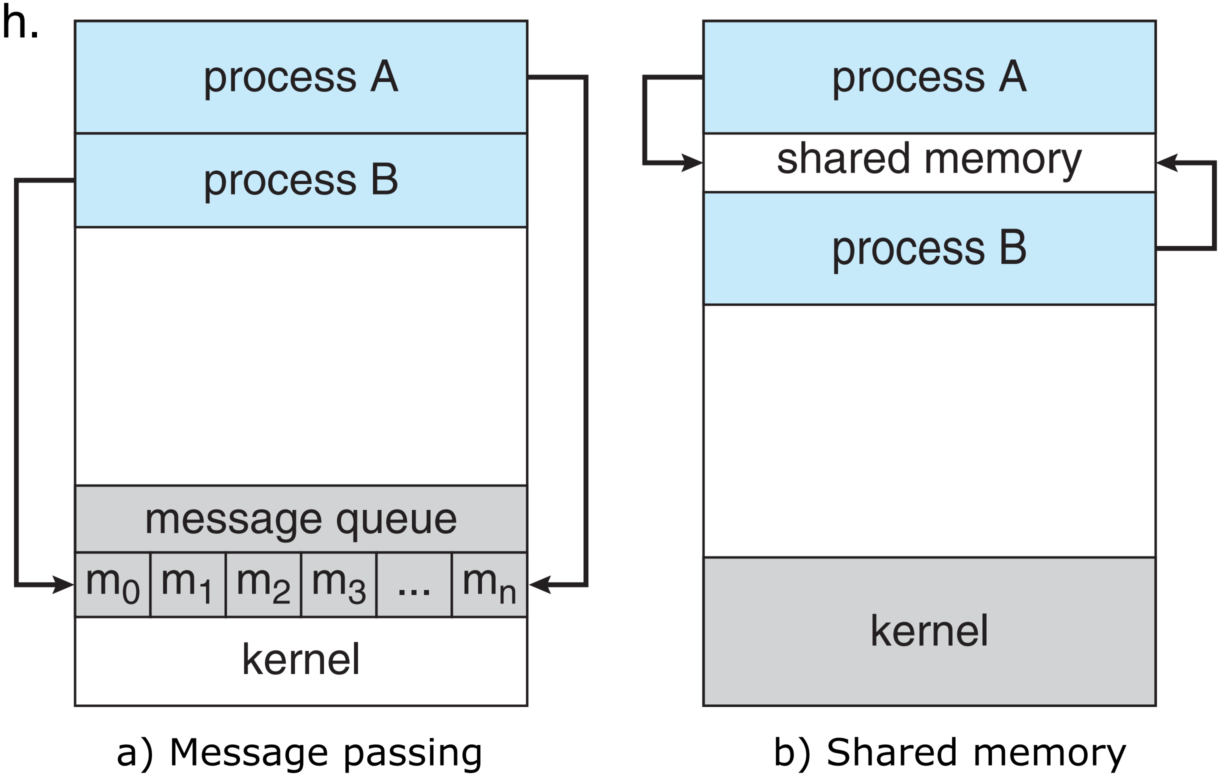
- Processes executing concurrently in the OS may be either:
  - **Independent** process cannot affect or be affected by the execution of another process, or
  - **Cooperating** process can affect or be affected by the execution of another process

- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience

- Cooperating processes need **interprocess communication** (**IPC**) that would allow them to exchange data and information

Communications Models

- Two models of IPC
  - Shared memory: a region of memory is used that is shared by the cooperating processes
  - Message passing: Communication takes place by means of messages exchanged between cooperating processes
- Both of these two models are common in the OSs, and many systems implement both.



Shared Memory Systems

- Recall that, normally, the OS tries to prevent one process from accessing another process's memory.
  - Shared-memory requires that two or more processes agree to remove this restriction
  - Usually the shared-memory resides in the address space of the process creating the shared-memory segment
  - Then they can exchange information by reading and writing data in the shared areas (not under the OS's control)
- The processes are responsible for ensuring that they are not writing to the same location simultaneously.
  - To illustrate this concept, we consider the producer-consumer problem.

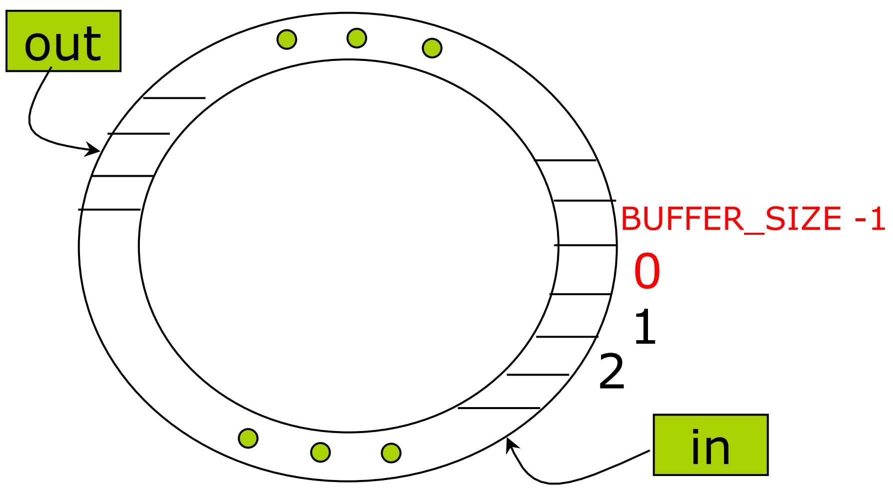
Producer-Consumer Paradigm

- A producer process produces information that is consumed by a consumer process.
  - e.g. a compiler produces assembly code, which is consumed by the assembler.
  - could be considered as a metaphor for client-server paradigm too.
- One solution to the producer-consumer problem uses shared memory
  - We must have available a buffer of items that can be filled by the producer and emptied by the consumer.
  - The producer and consumer must be synchronized
- There are two types of buffers:
  - *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

- The following variables reside in a region of memory shared by the producer and consumer processes.

```
#define BUFFER_SIZE 10
typedef struct {
/* . . . */
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```



- The above shared buffer is implemented as a <u>circular array</u> with <u>two logical pointers</u>: **in** and **`out`**.
- *Bounded-buffer:* The above is one correct solution, but it can only use **`BUFFER_SIZE-1`** elements

Bounded-Buffer (cont)

Producer

```
item nextProduced;
while (true)
{
    /* Produce an item in nextProduced*/
   while (((in + 1) % BUFFER_SIZE)  == out)
      ; /* do nothing -- no free buffers */

    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Consumer

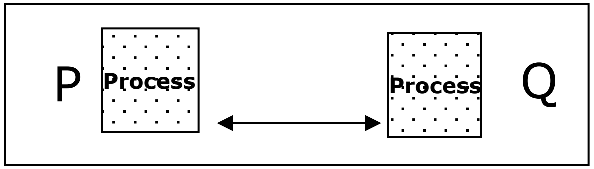
```
item nextConsumed;
while (true)
{
    while (in == out)
         ; /* do nothing -- nothing to consume */

     /* remove an item from the buffer */
     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     /* consume the item in nextConsumed */
     return nextConsumed;
}
```

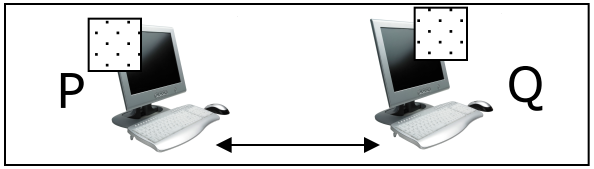
Message Passing Systems

- Message passing is the other method that provides a mechanism for processes <u>to communicate and to synchronize</u> their actions
  - processes communicate with each other without resorting to shared variables
  - A particularly useful and practical method in distributed environment (e.g. chat programs)

- A message passing facility provides at least two operations:
  - **send(*message*)** and **receive(*message*)**
  - The message size can be fixed or variable

- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive





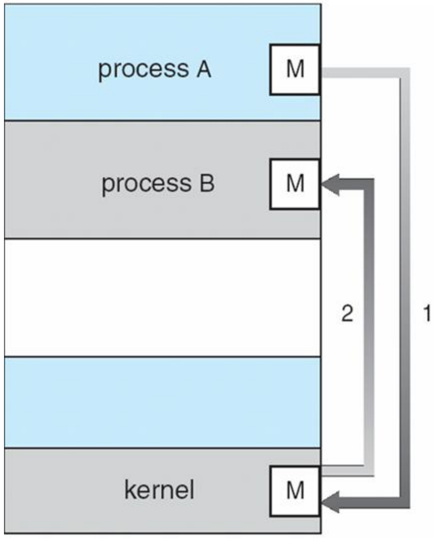
Synchronization

- Communication between processes takes place through calls to **`send()`** and **`receive()`** primitives
- There are different design options for implementing each primitive:
  - Message passing may be either blocking or non-blocking
- Blocking is considered <u>*synchronous*</u>
  - **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 <u>*asynchronous*</u>
  - **Non-blocking send** has the sender send the message and continue
  - **Non-blocking receive** has the receiver receive a valid message or a null
- Different combination of **`send()`** and **`receive()`** are possible. When Both send and receive are blocking, we have a <u>rendezvous</u> between them.

Buffering

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.

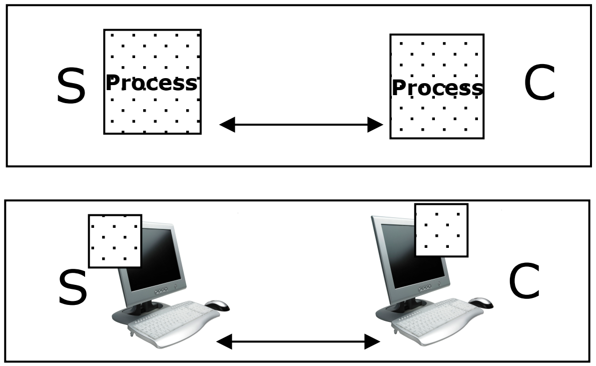
- Such queues can be implemented in one of three ways
  - Zero capacity (queue max length is zero, i.e. 0 messages)
    - Sender must wait for the receiver
  - Bounded capacity (finite length of *n* messages)
    - Sender must block if the link is full, otherwise it can continue without waiting
  - Unbounded capacity (infinite length)
    - Sender never waits



Communications in Client-Server Systems

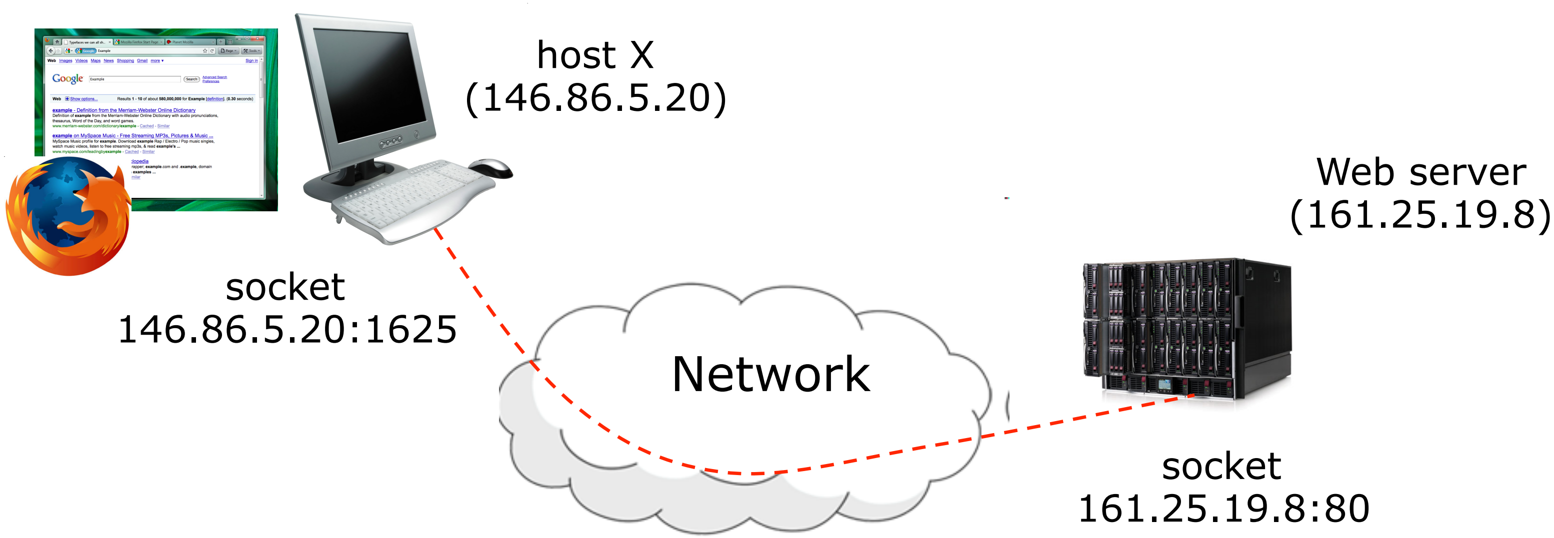
- So far, it was described how processes can communicate using:
  - shared memory and
  - message passing

- There are <u>three other strategies</u> for communication in client-server systems:
  - Sockets
  - Remote Procedure Calls (RPC)
  - Pipes



Sockets

- A socket is defined as an *endpoint for communication*
  - It is identified by an IP address concatenated with a port number.
    - e.g. the socket **146.86.5.20:1625** refers
      - to port **1625** on
      - host **146.86.5.20**
- A pair of processes communicating over network employ a pair of sockets
- Connection-oriented (TCP) Vs. connectionless (UDP) sockets

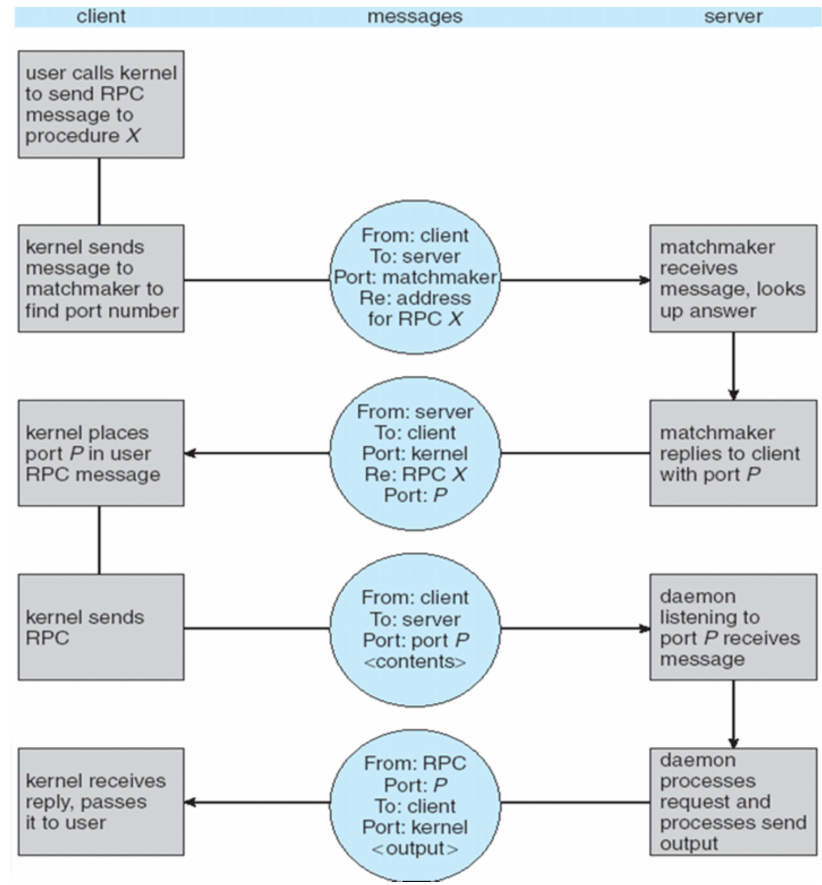


Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - We must use a message-based communication to provide remote service
  - The messages are well structured (in contrast to the IPC)
  - Each message is addressed to an RPC daemon listening to a port on a remote system, and contains
    - the identifier of the function to execute and
    - the parameters to pass to that function
- The semantics of RPCs allow a client to invoke a procedure on a remote host as it would invoke a procedure locally
  - **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

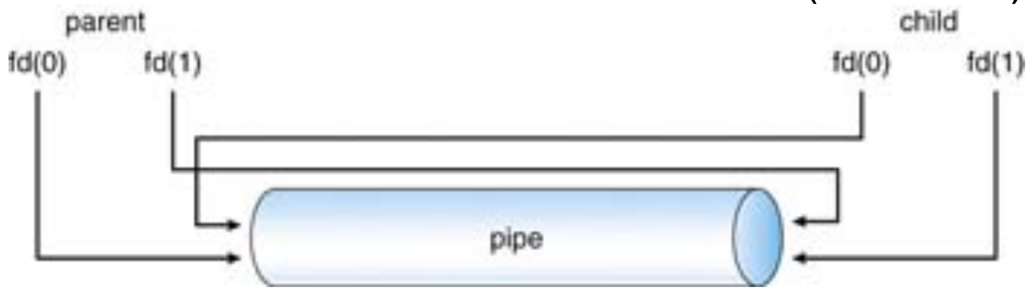
Remote Procedure Calls (cont)

- An important issue that must be dealt with concerns differences in data representation of the client and server machines.
  - Big-endian: some systems store the most significant byte first.
  - Little-endian: some other systems store the least significant byte first.
- Execution of RPC:



Pipes

- A pipe acts as a conduit providing one of the simpler ways for processes to communicate
  - Ordinary pipes allow two processes to communicate in standard producer-consumer fashion
    - The producer writes to one end of the pipe (*write-end*) and the consumer reads from the other end (*read-end*)



`pipe(int fd[])`

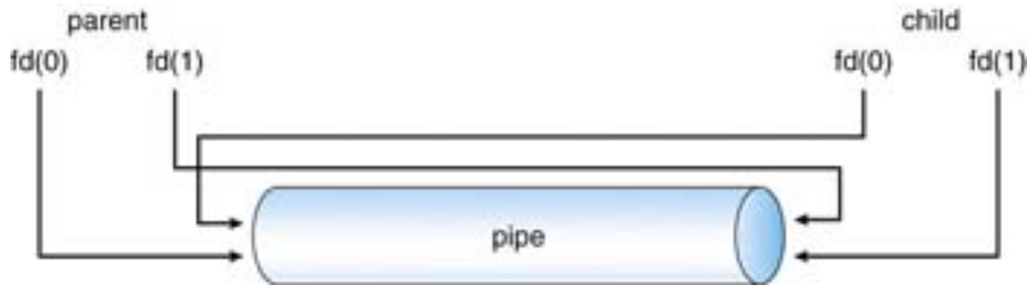
- Pipes are used quite often in the UNIX command-line environment in which the output of one command serves as input to the second
  - A pipe can be constructed on the CLI using the | character
    - e.g.:

```
ls | less
```

  - For the DOS shell, the equivalent command for the above is

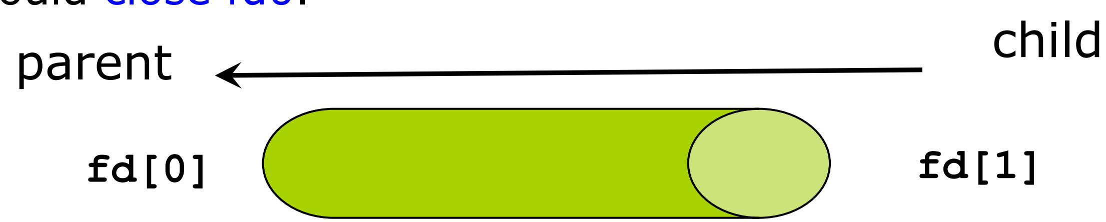
```
dir | more
```

Ordinary Pipes

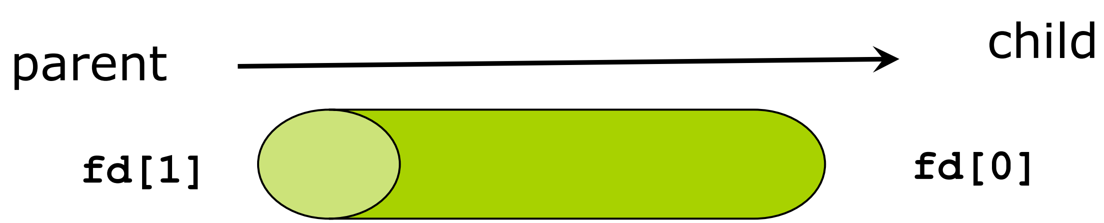



```
pipe(int fd[])
fd[0] is the read-end of the pipe
fd[1] is the write-end of the pipe
```

If the parent wants to receive data from the child, it should close fd1, and the child should close fd0.



If the parent wants to send data to the child, it should close fd0, and the child should close fd1.



Lecture Outline

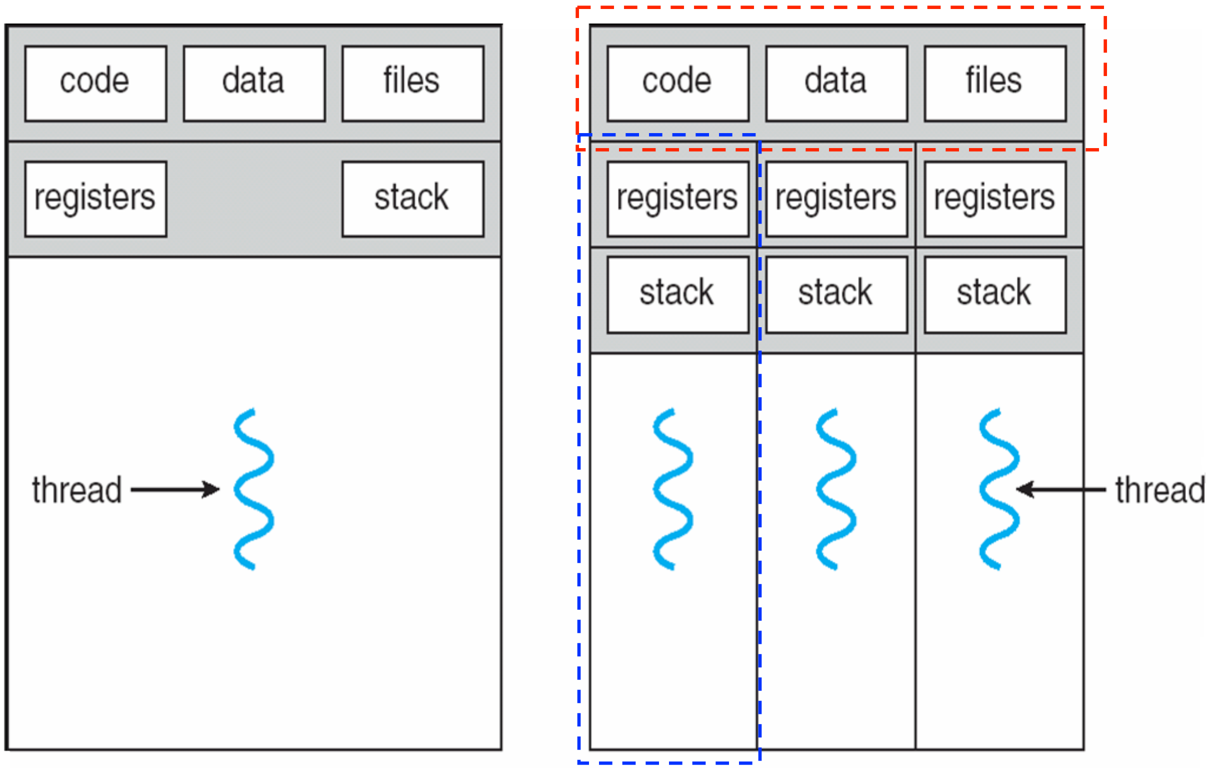
- Processes and Scheduling
  - What is a process ?
  - How are processes created ?

- Inter-process Communication (IPC)
  - Examples of IPC Systems
  - Synchronization and IPCs

- **Threads and multi-threading**

Threads

- A thread is a basic unit of CPU utilization
  - It comprises a thread ID, a program counter, a register set, and a stack
  - It shares with other threads belonging to the same process its code section, data section and other OS resources (e.g. open files)



single-threaded process   multithreaded process

Single and Multithreaded Processes

- The process model described so far assumed that a process was an executing program with a single thread of control

- Many software packages that run on modern desktop PCs are multithreaded
  - An application typically is implemented as a separate process with several threads of control
  - If a process has multiple threads of control, it can perform more than one task at a time
    - e.g. a multithreaded web server

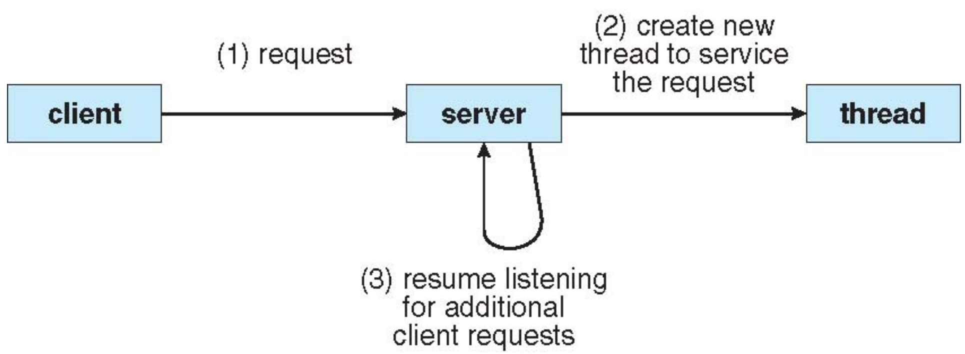


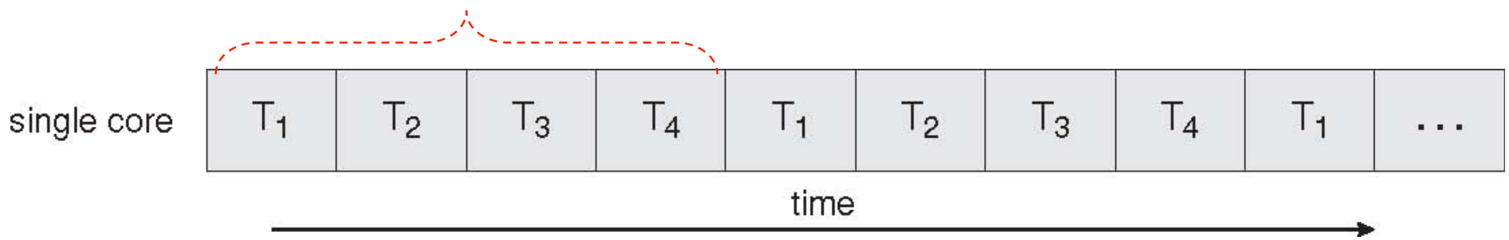
Fig 4.2

Why use Multi-threading ?

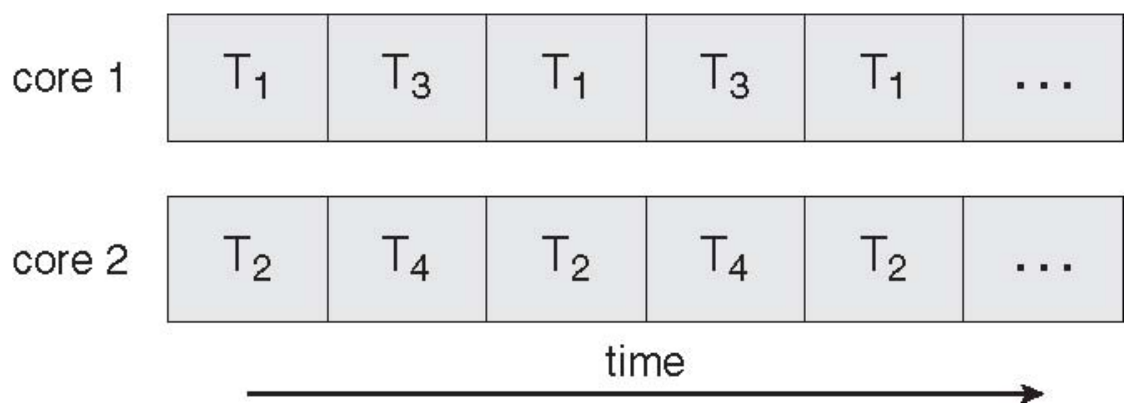
- Q: why not use process-creation method?
  - Process creation is time consuming and resource intensive
  - If the new process will perform similar tasks, it is generally more efficient to use one process that contains multiple threads instead.
- **Benefits** of multithreaded programming
  - Responsiveness: 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

Why use multi-threading (contd) ?

- Multithreaded programming provides a mechanism for more efficient use of multiple core and improved concurrency.
  - On a system with a <u>single computing core</u>, concurrency merely means that the execution of the threads will be interleaved over time.



  - On a system with <u>multiple cores</u>, concurrency means that the threads can run in parallel.



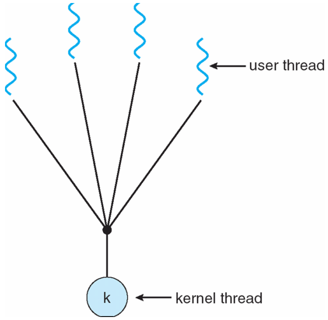
User and Kernel Threads

- Support for threads may be provided either
  - at the user level, for user threads, or by the kernel, for kernel threads
    - User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the OS
- For user threads, thread management is done by a user-level thread library
  - Three primary thread libraries:
    - POSIX Pthreads
    - Win32 threads
    - Java threads
- Virtually all contemporary OSs support kernel threads
  - *Examples*: Windows, Solaris, Linux, Mac OS X

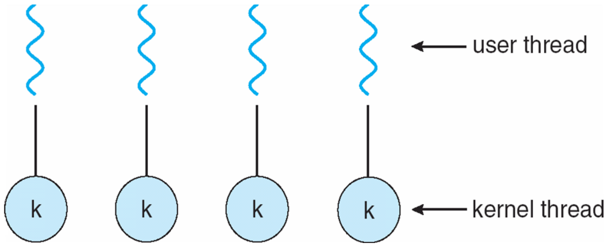
Multithreading Models

- Ultimately, a relationship must exist between user threads and kernel threads
- Three common ways of establishing such a relationship are:

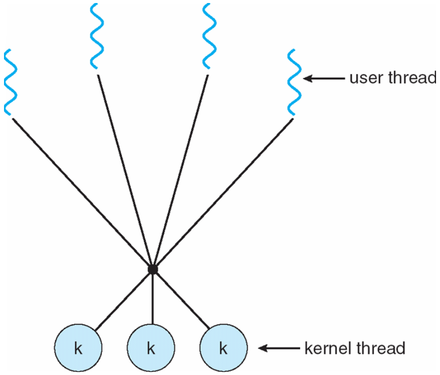
  - Many-to-One



  - One-to-One



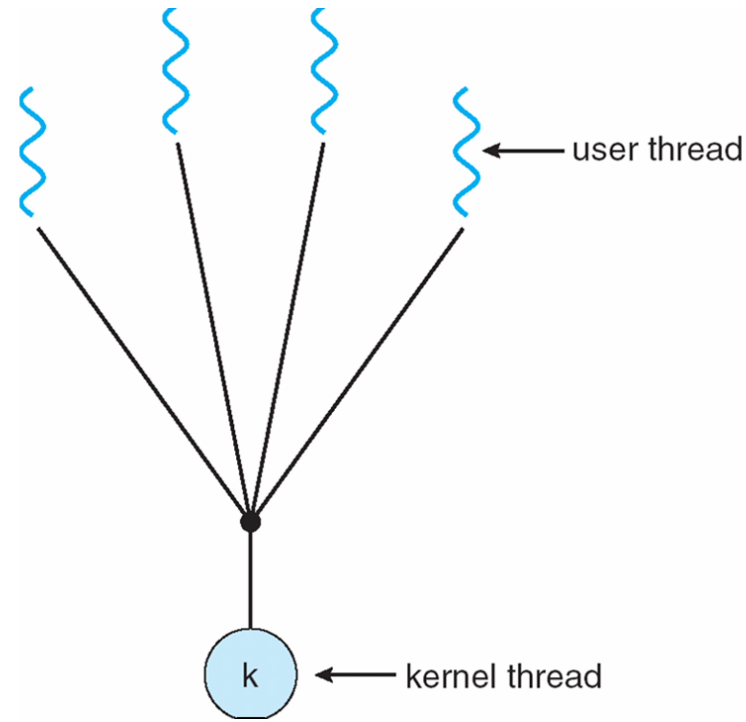
  - Many-to-Many



Many-to-One Model

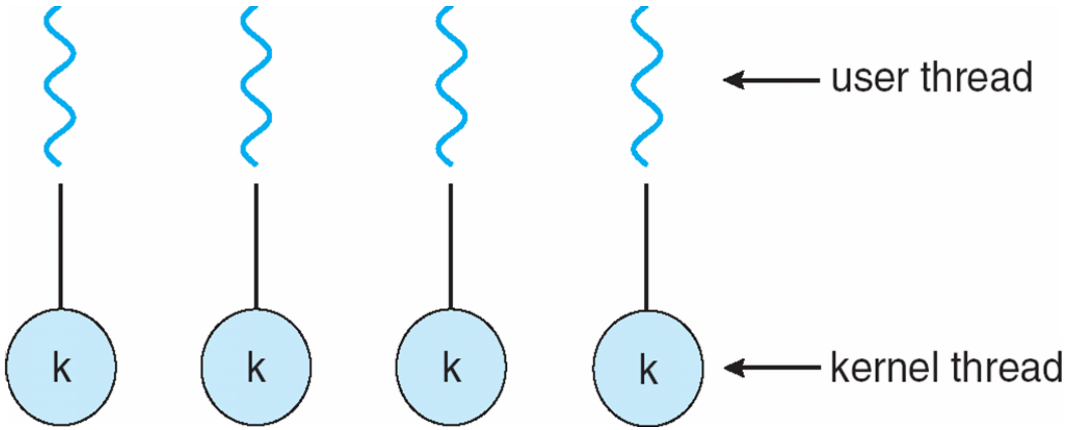
- In this model, many user-level threads are mapped to a single kernel thread
  - It is efficient as thread management is done by the thread library in user space
  - Though the entire process will block if a thread makes a blocking system call

- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



One-to-One Model

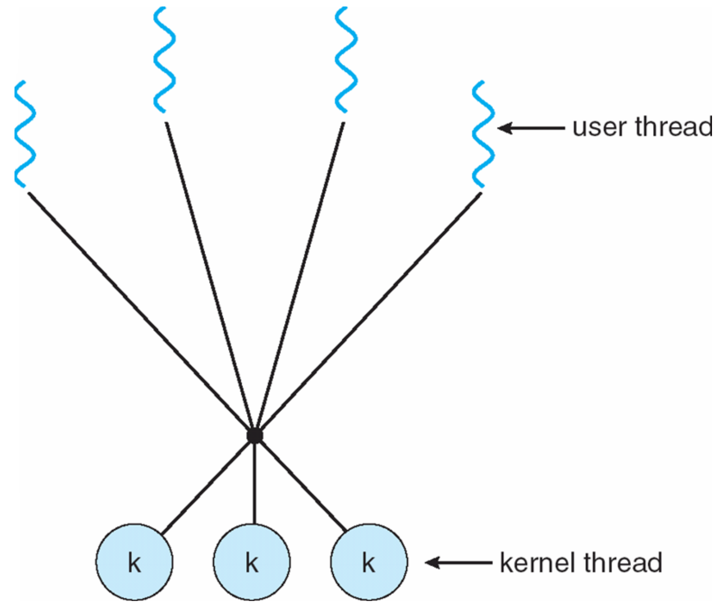
- In this model, each user-level thread is mapped to a kernel thread
  - It provides more concurrency
  - The drawback is the overhead of creating the corresponding kernel threads
    - This can burden the performance of an application



- Examples
  - Windows
  - Linux
  - Solaris 9 and later

Many-to-Many Model

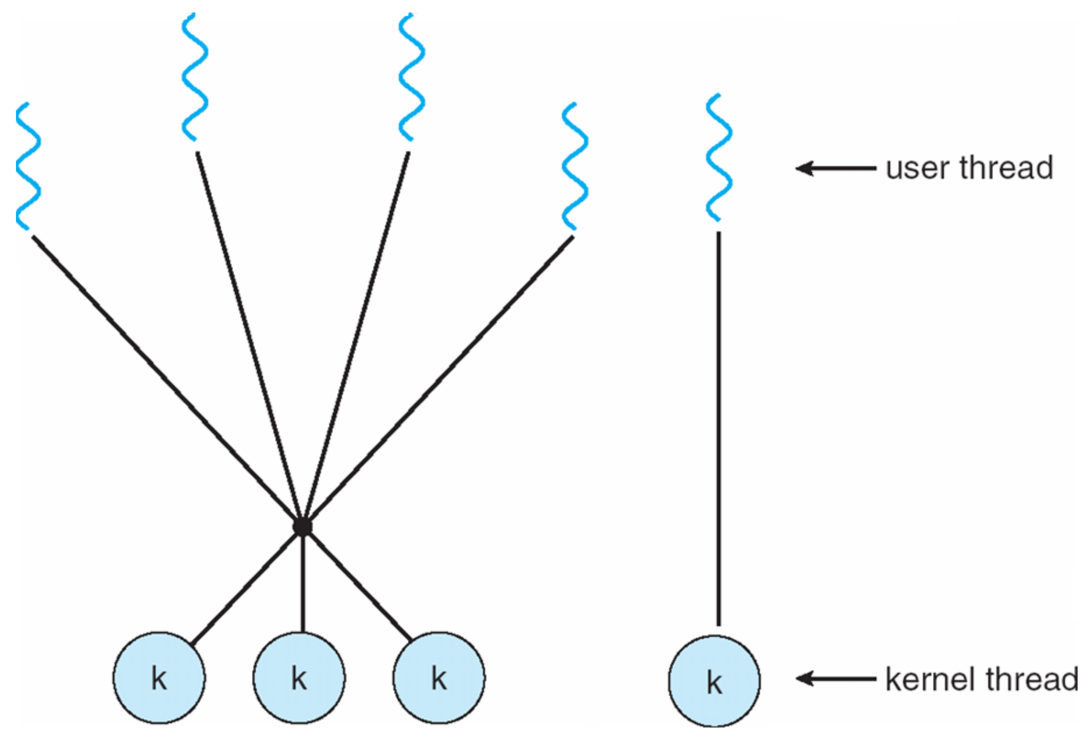
- This model multiplexes many user level threads to be mapped to a smaller or equal number of kernel threads
  - Allows the operating system to create a sufficient number of kernel threads
- The many-to-many model has neither of the shortcomings of the previous two models:
  - developer can create as many user threads as necessary (concurrency),
  - the corresponding kernel threads can run in parallel on a multiprocessor, and
  - when a thread performs a blocking system call, the kernel can schedule another thread for execution.



- Examples
  - Solaris prior to version 9
  - Windows NT/2000 with the *Thread Fiber* package

Two-level Model

- One popular variation on the many-to-many model is the two-level model
  - It is similar to the many-to-many model, except that it allows a user thread to be bound to a kernel thread



- Examples
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

Thread Libraries

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

- Two primary ways of implementation:
  - Library entirely in user space
    - All code and data structures for the library exist in user space
    - Invoking a function results in a local function call (not a system call)
  - Kernel-level library supported directly by the OS
    - Code and data structures for the library exist in kernel space
    - Invoking a function in the API for the library typically results in a system call
    - e.g. Win32 thread library is kernel-level library

Libraries: Pthreads

- Pthreads refers to a POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
  - This is a specification for thread behaviour, not an implementation
    - Implementation is up to development of the library
  - Common in UNIX operating systems (Solaris, Linux, Mac OS X)
  - POSIX Pthreads may be provided as either a user- or kernel-level library

- See figure 4.9 of the textbook for a Pthreads example.

- See figure 4.11 of the textbook for a Windows multithreaded example.

Libraries: Java Threads

- Java Thread API allows threads to be created and managed directly in Java programs.
  - Threads are the fundamental model of program execution in a Java program
  - Java threads are managed by the JVM
  - Since JVM is usually running on top of a host system, it is typically implemented using the thread model provided by the underlying OS
    - e.g. on Windows systems, Java threads are typically implemented using the Win32 API

Threading Issues

- Semantics of **fork()** and **exec()** system calls can change in a multithreaded program
  - Does **fork()** duplicate only the calling thread or all threads?
    - Depends on the application
    - if exec() is called right after, duplicating all threads is unnecessary
    - Otherwise, the separate process should duplicate all threads

- Blocking System Calls
  - Should all threads block or only the current thread ?

- How should signals be delivered ?

Signal Handling

- A Signal is used in UNIX systems to notify a process that a particular event has occurred.

- All signals (whether synchronous or asynchronous) follow the same pattern:
  1. A signal is generated by the occurrence of a particular event
  2. The generated signal is delivered to a process
  3. Once delivered, the signal must be handled by a signal handler

- A signal handler is used to process signals

- A signal may be handled by:
  - A default signal handler (every signal has one)
  - A user-defined signal handler

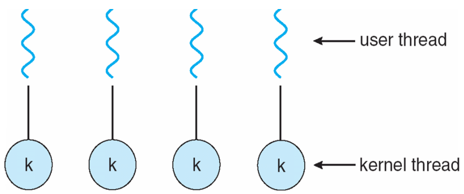
Signal Handling 2

- Every signal has default handler that kernel runs when handling signal
  - User-defined signal handler can override default
  - For single-threaded, signal delivered to process

- In a multithreaded program, the question is where the signal should be delivered? In general, the following options exist:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

Example 1: Windows Threads

- A Windows application runs as a separate process, and each process may contain one or more threads.
- Windows implements the one-to-one mapping



- Each thread contains
  - a thread id
  - a register set
  - a separate user stack (when running in user mode) and a kernel stack (when running in kernel mode)
  - a private storage area (used by DLLS, ..)
- The register set, stacks, and private storage area are known as the context of the threads
- page 155: Win32 thread example

Example 2: Linux Threads

- Linux uses the term *tasks* rather than *threads* or *processes*
- Linux provides the **`fork()`** system call with its traditional functionality
- Thread creation is done through **`clone()`** system call
  - Varying level of sharing is possible
    - **`clone()`** allows a child task to share the address space of the parent task (process)
  - When invoked, it is passed a set of flags

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.

- Several Linux distributions include the NPTL (Native POSIX Thread Library)

Lecture Outline

- Processes and Scheduling
  - What is a process ?
  - How are processes scheduled ?
- Inter-process Communication (IPC)
  - Examples of IPC Systems
  - Synchronization and IPCs
- Threads and multi-threading