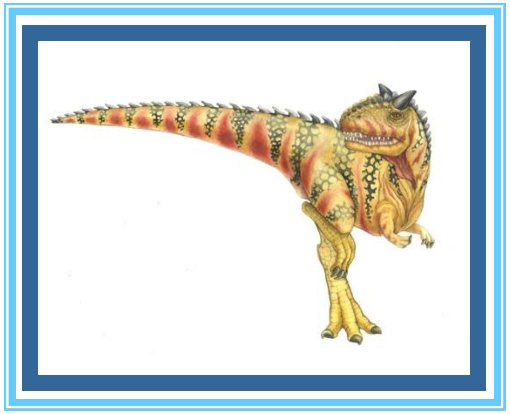
Chapter 3: Processes

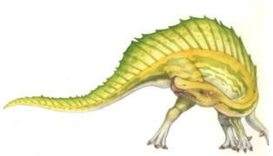


Chapter 3: Processes

- Process Concept
- Basic Scheduling Concept
- Operations on Processes – Process Creation and Termination
- Interprocess Communications
- Communication in Client-Server Systems – Socket and RPC

Objectives

- Identify the components of a process and illustrate how they are represented and scheduled in operating systems.
- Describe how processes are **created** and **terminated** in an operating system, using the appropriate *system calls* that perform these operations.
- Describe and contrast inter-process communications using **shared memory** and **message passing** methods



Four Fundamental OS Concepts

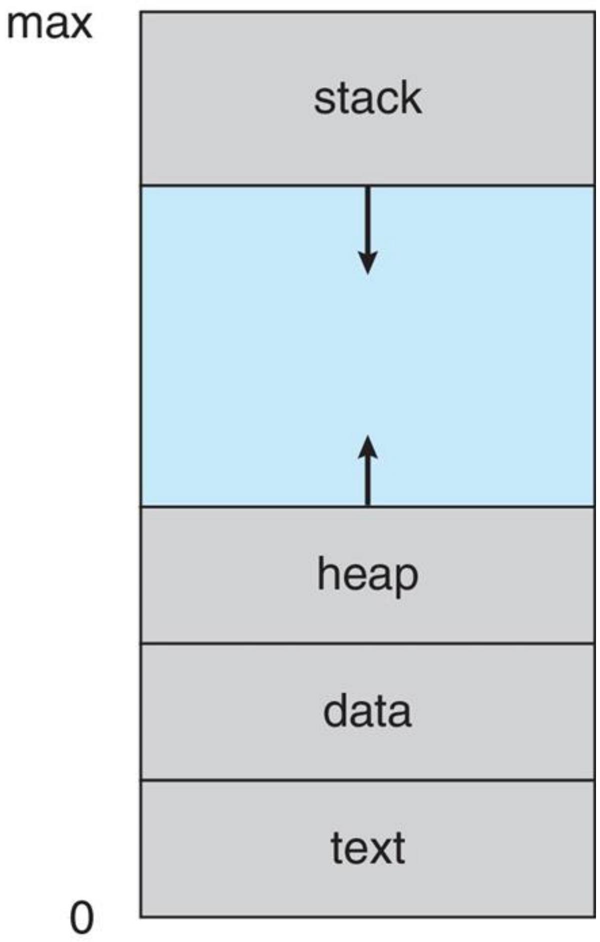
- Process
  - An instance of an executing program is a process - consisting of an address space and one or more threads of control
- Thread (more in Chapter 4)
  - A single unique execution context; fully describes program state – captured by program counter, registers, execution flags, and stack
- Address Space (with address translation – to be discussed in Chapter 9)
  - Programs execute in an address space that is distinct from the memory space of the physical machine – each program starts with address 0 (logical or virtual address to be discussed in Chapters 9-10)
- Dual mode operation – Basic Protection
  - User programs and OS (kernel) programs run in different modes
  - Only the operating system is allowed to access certain resources
  - The OS and the hardware are protected from user programs
  - User programs are protected and isolated from one another

Process Concept

- OS executes a variety of programs, each runs as a process
- **Process** captures a program execution; process executions progress in a **sequential** manner - *von Neumann* architecture
- The term "process" contains multiple "parts"
  - The program code, also called text section
  - Current activities or state - program counter, processor registers
  - Stack - temporary data storage when invoking functions such as function parameters, return addresses, and local variables
  - Data section containing global variables
  - Heap containing memory dynamically allocated during run time
- A program is a passive entity stored on disk (executable file), while a process is active entity, with a program counter specifying the next instruction to fetch and execute, as well as a set of associated resources
- A process has a life cycle – from program execution to termination

Process Concept (Cont.)

- A program "becomes" a process when an executable file is loaded into main memory and gets ready to run
  - Double-click a program icon or input a command on *Shell*
- Program executes in an address space (right side), different from actual main memory location – logical address space (discussed in Chapters 9-10)
- One program (code) can be executed by multiple processes
- A process can itself be an execution environment for other programs, e.g., a Java program is executed within the Java virtual machine (JVM)

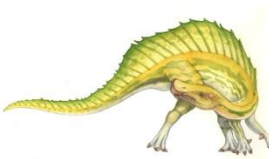


Address Space

Loading Program into Memory

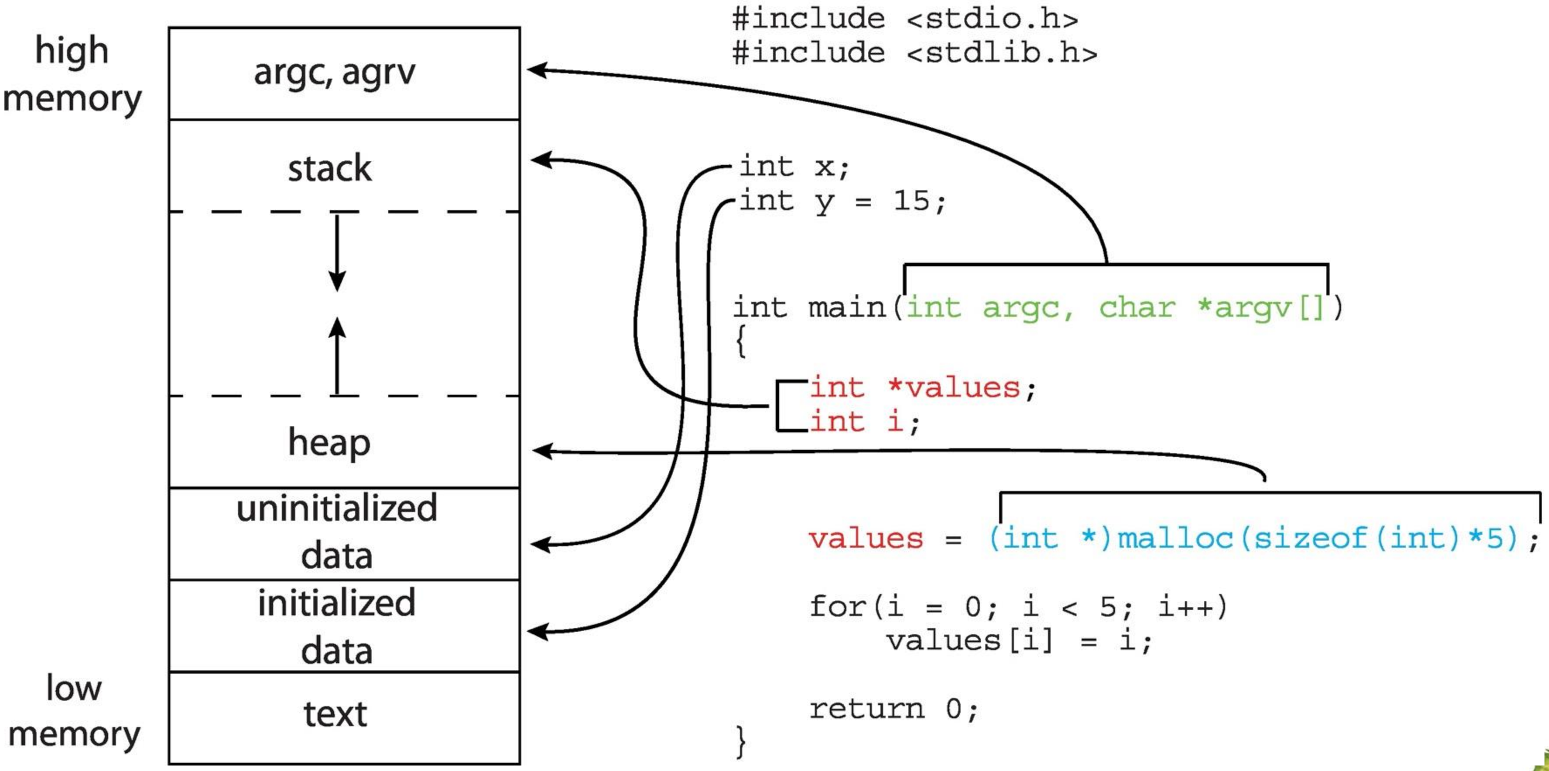
A program must be loaded inside memory before execution – a program icon is double clicked on a personal computer, or a job is submitted to a remote mainframe

1. Load its code and any static data (e.g., initialized variables) into memory, or into the address space of the process.
2. Some memory must be allocated for the program's runtime stack - C programs use the stack for local variables, function parameters, and return addresses. Fill in the parameters to the `main()` function, i.e., `argc` and the `argv` array
3. OS may allocate memory for the program's heap - programs request memory space by calling `malloc()` and free it explicitly by calling `free()`
4. OS will also do some other initialization tasks, esp. related to input/output (I/O). For example, in UNIX systems, each process by default has three open file descriptors, for standard input, output, and error.



Memory Layout of a C Program

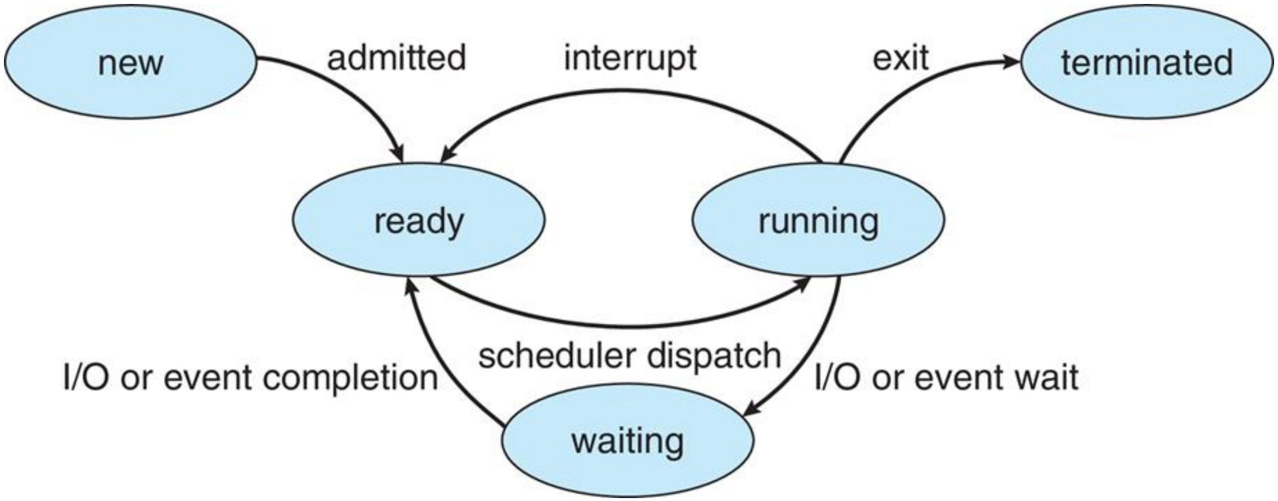
- The global data section is divided into (1) initialized data and (2) uninitialized data
- A separate section provided for argc and argv parameters passed to main() function



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

int x;
int y = 15;

int main(int argc, char *argv[])
{
      int *values;
      int i;

      values = (int *)malloc(sizeof(int)*5);

      for(i = 0; i < 5; i++)
          values[i] = i;

      return 0;
}
```

Process State

- In traditional Unix systems, each process consists of one thread, thus the process state is defined in part by the current activity of that process
- As a process executes, it changes its **state** from one to another
  - New: The process is being created – allocating resources
  - Ready: The process is waiting to be assigned to a CPU core
  - Running: Instructions are being fetched and executed on a CPU
  - Waiting: The process is waiting for some event(s) to occur
  - Terminated: The process finishes execution – deallocating resources

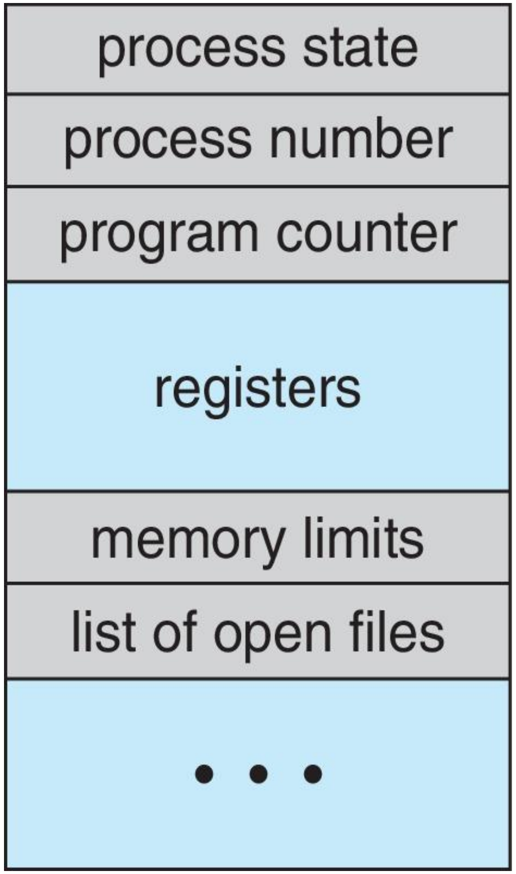


Process Control Block (PCB)

Information associated with each process

(also called **task control block**)

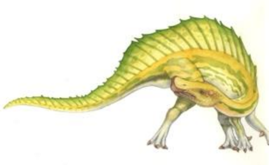
- Process state – running, waiting, etc.
- Program counter – location of instruction to be fetched next to execute
- CPU registers –accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.
- CPU scheduling information – priorities, pointers to scheduling queue, scheduling parameters
- Memory-management information – memory allocated to the process (complicated data structure – paging and segmentation tables)
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, for instance, a list of open files



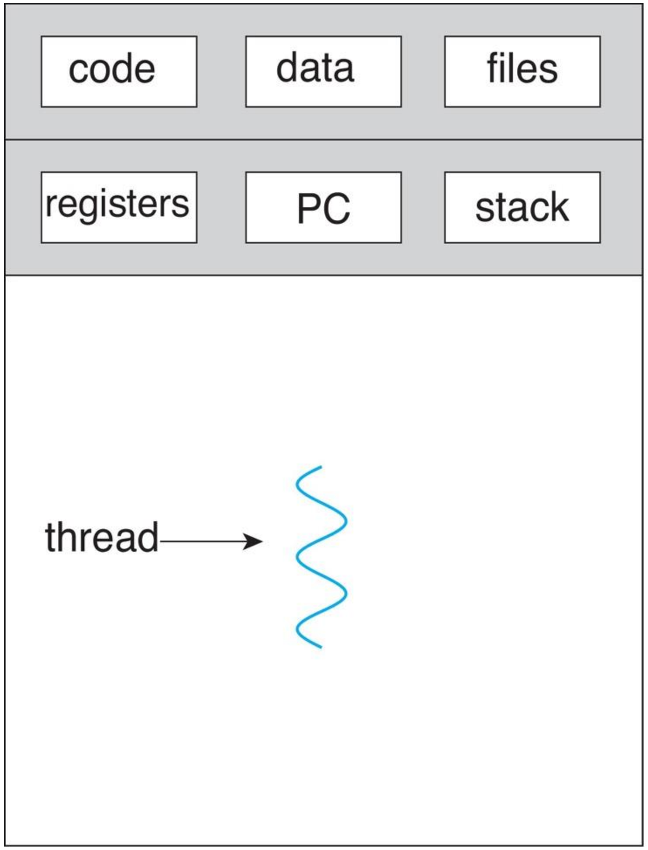
Threads

- So far, we assume that a process has a single thread of execution
  - It executes a program from the first line of codes to the last line of code, *sequentially*, in a serialized manner. The single execution context fully describes the program state - the current activity of the thread
  - A thread is represented by program counter, registers, execution flags, and stack – part of a process
  - A thread is executing on a processor or CPU core when it is resident in processor (CPU) registers - hardware registers
- Most modern operating systems allow a process to have multiple threads of execution and thus can perform more than one task at a time, in parallel
  - A web browser (a process) might have one thread displaying images or text while another thread retrieving data from the network
  - On multicore systems, multiple threads of a process can run in parallel
- A process can consist of one or multiple thread(s) and they run within the same address space (i.e., share code, data and files), each thread has its own unique **state**
- Multithreads in a process provide concurrency, "active" components

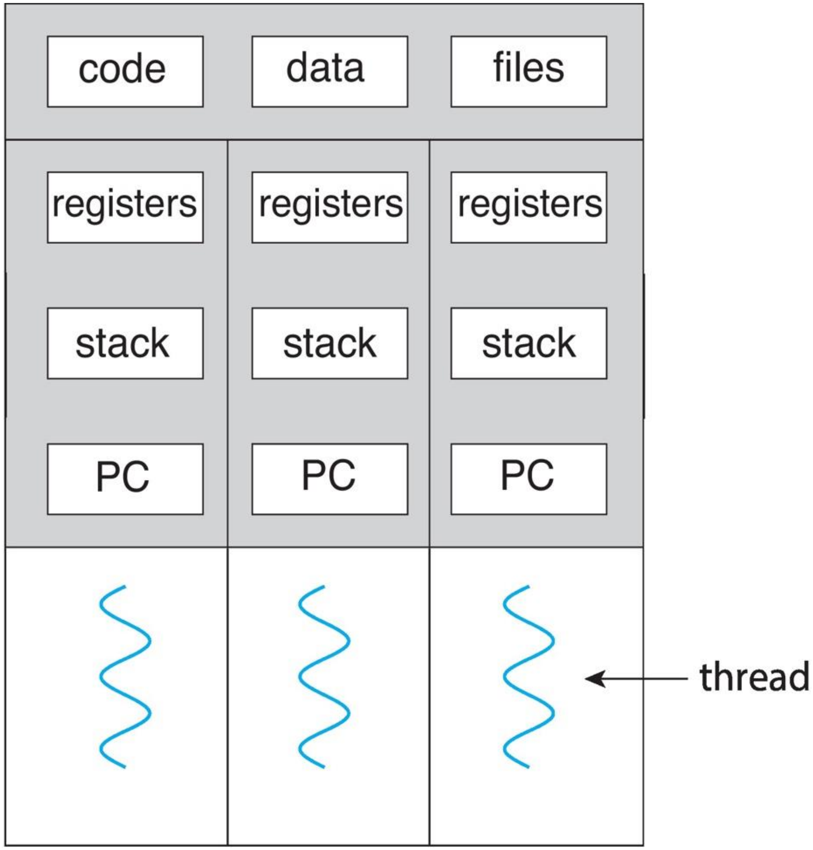
More details to come in Chapter 4…



Single and Multi-threaded Processes



single-threaded process

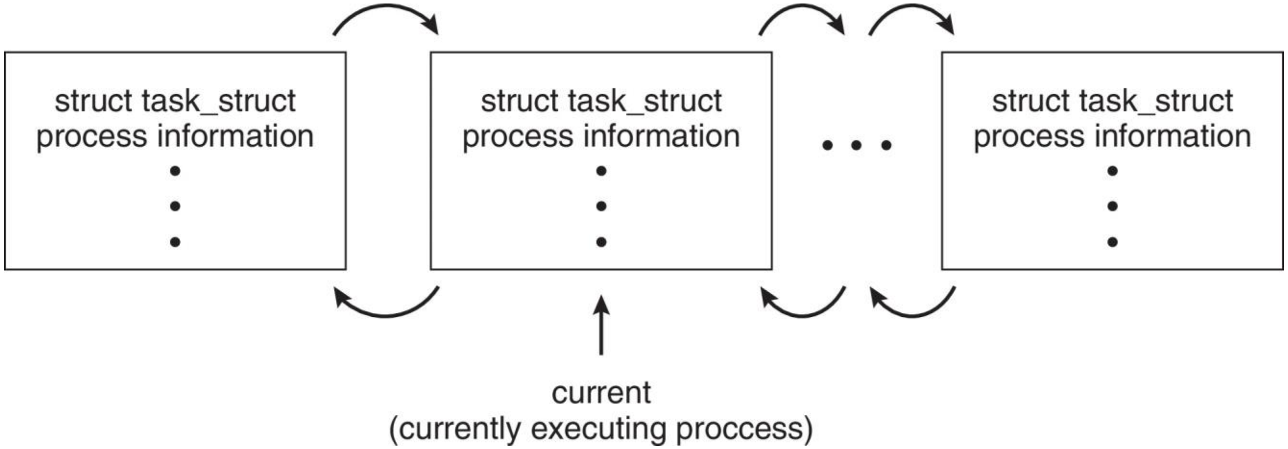


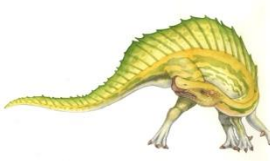
multithreaded process

Process Representation in Linux

- The Linux kernel stores the list of processes in a *circular doubly linked list* called the **task list**. Each element in the task list is a process descriptor (i.e., PCB) of the type struct **`task_struct`**

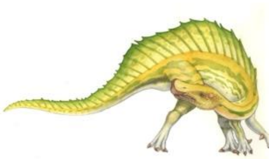
```
pid t_pid;                     /* process identifier */
long state;                    /* state of the process */
unsigned int time_slice        /* scheduling information */
struct task_struct *parent;/* this process’s parent */
struct list_head children; /* this process’s children */
struct files_struct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





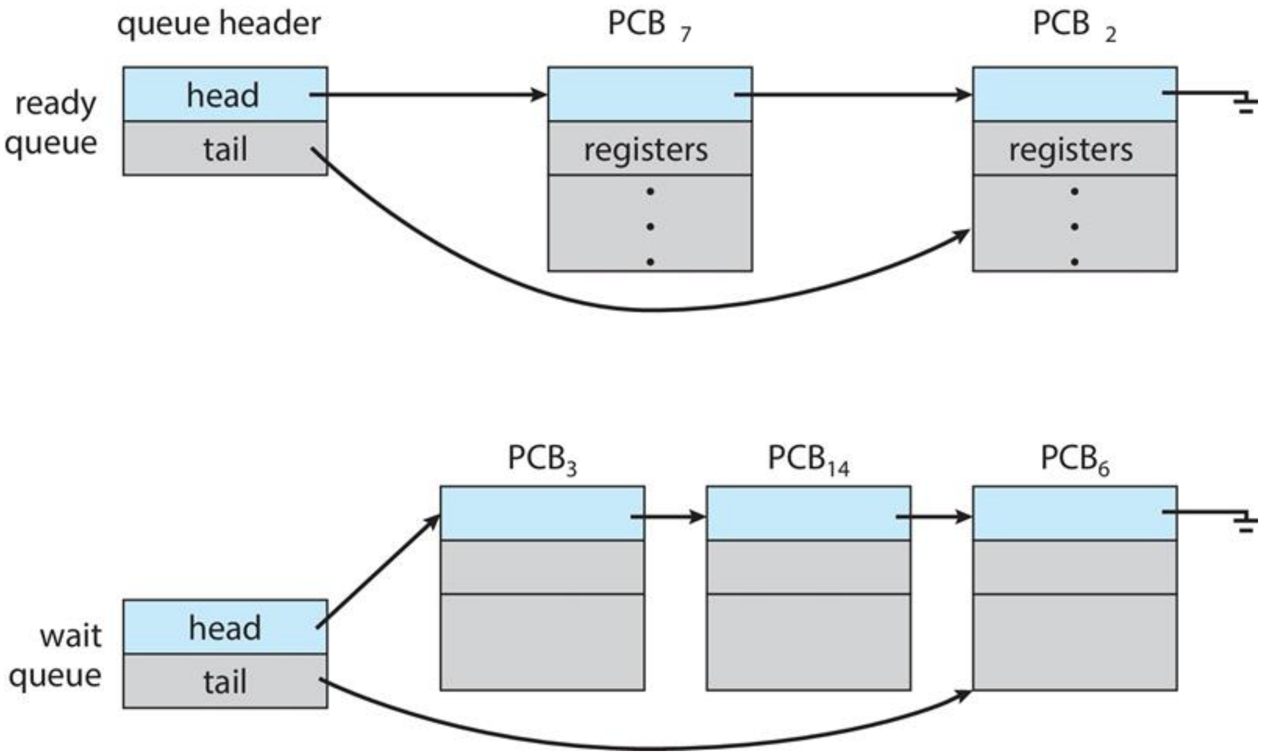
Process Scheduling

- All modern operating systems are multiprogramming - the primary objective is to try to have some processes always running on the CPU for maximizing CPU utilization – recall CPU usually runs much faster than other devices
- **Process scheduler** – an OS mechanism selecting a process (from available processes inside main memory) for execution on one CPU core
  - Scheduling criteria: CPU utilization, job response time, fairness, real-time guarantee, latency optimization – to be discussed in Chapter 5
- For a system with only a single CPU, there will never be more than one process or thread running at a time
- The number of processes currently residing in memory is known as the **degree of multiprogramming** – determining the resource consumption
- Processes (useful for mainframes) can be “roughly” described as either:
  - I/O-bound process – spends more of its time doing I/O than it spends doing computations; many short CPU bursts or CPU cycles
  - CPU-bound process – in contrast, generates I/O requests infrequently, using more of its time doing computations; few very long CPU bursts



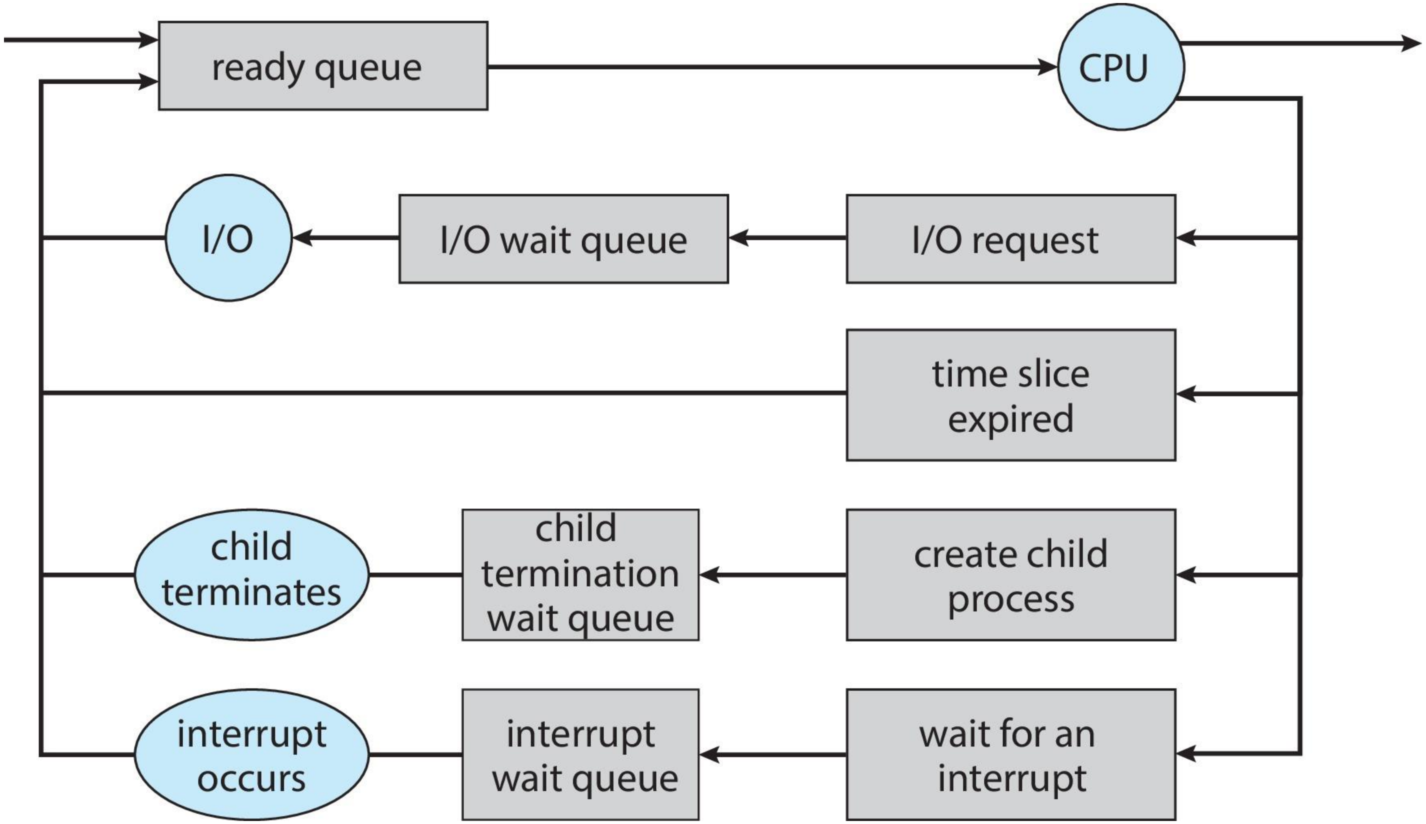
Process Scheduling (Cont.)

- The OS maintains different scheduling queues of processes
  - Ready queue – set of processes in main memory, ready and waiting to execute
  - Wait queues – set of processes waiting for an event such as completion of I/O
  - Processes essentially migrate among various queues during their lifetime



Representation of Process Scheduling

- A common representation of process scheduling is a queueing diagram. The types of queues present a ready queue and a set of wait queues.

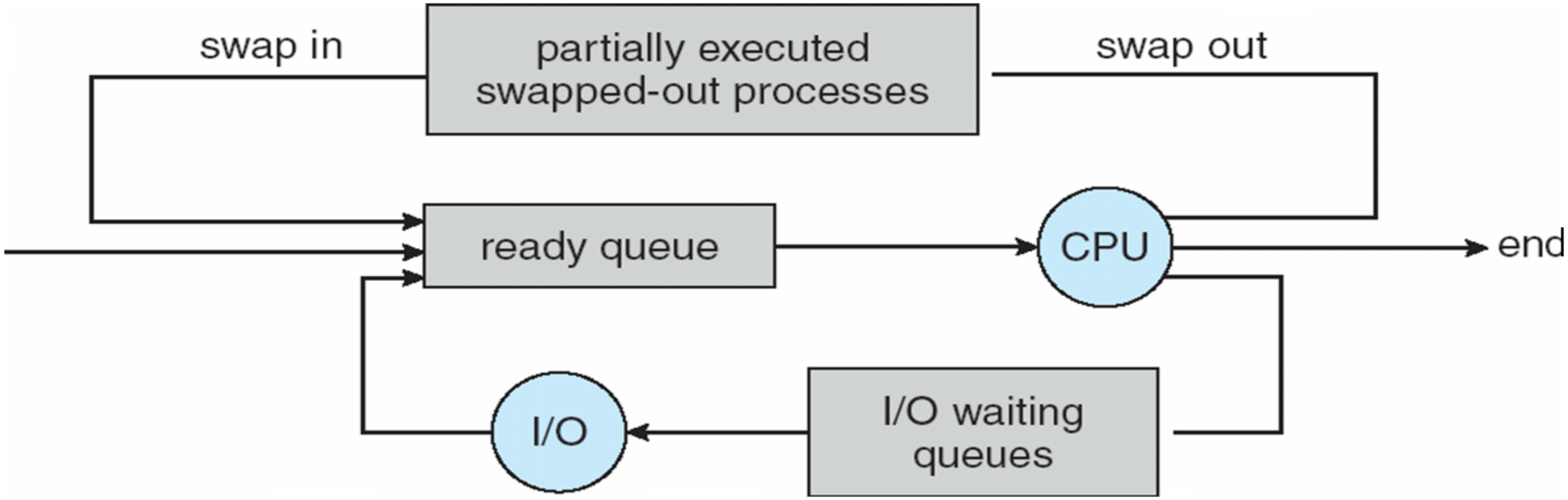


Schedulers

- Long-term scheduler (or job scheduler) – selects which processes should be brought into memory (after resource allocation such as memory and files) – used by mainframe and minicomputers, not personal computers
  - It consists of a job queue that hosts jobs submitted to mainframe computers (shared by multiple users), yet brought into the memory
- Short-term scheduler (or CPU scheduler) – selects a process from the ready queue to be executed next and allocates a CPU core to run it
- The short-term scheduler is invoked very frequently (milliseconds) $\Rightarrow$ (must be fast), while long-term scheduler is invoked very infrequently (seconds, or minutes) $\Rightarrow$ (may be slow)
- The long-term scheduler dictates the degree of multiprogramming – the number of processes that can concurrently be running in a computer.
- Long-term scheduler for a mainframe computer system strives for good ***process mix*** – mixed I/O-bound and CPU-bound processes

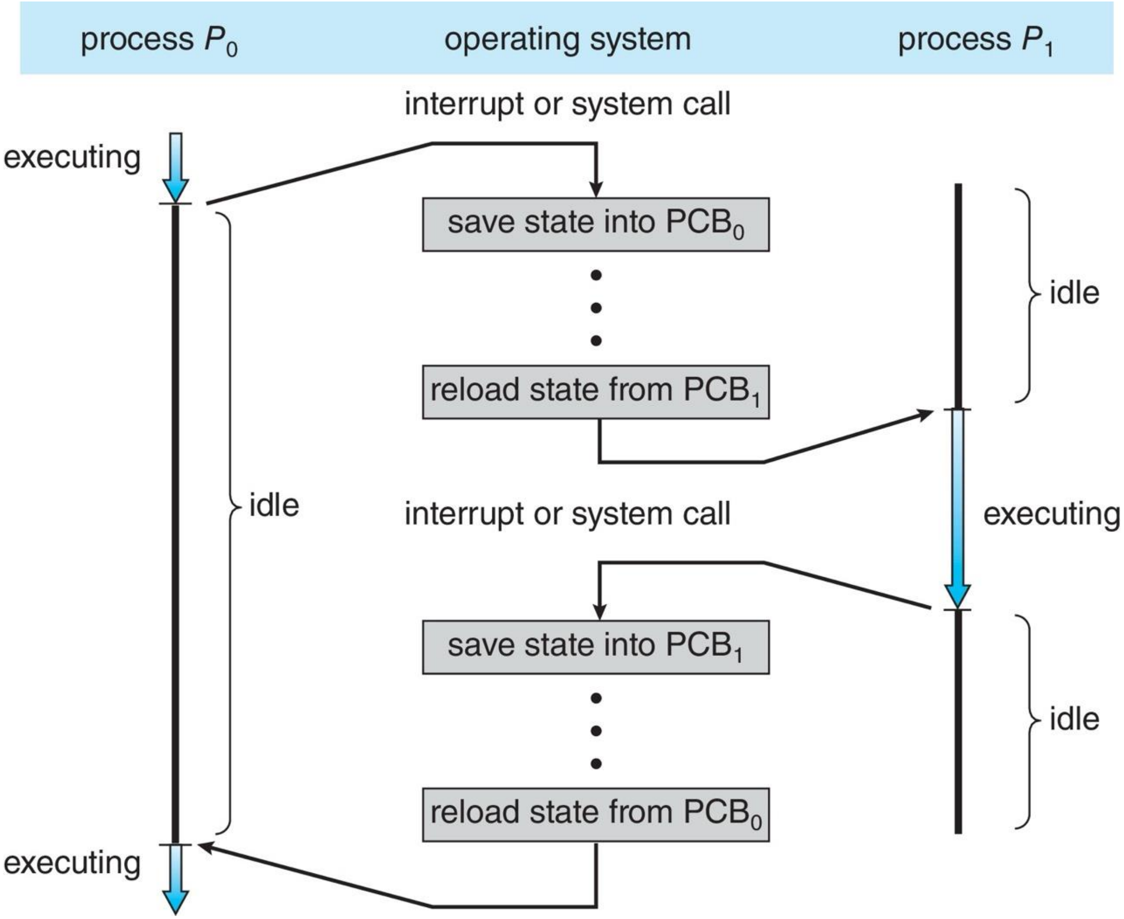
Medium Term Scheduling

- Medium-term scheduler can be added if degree of multiple programming needs to be reduced - to free up memory space for remaining processes
- It removes a process from memory, store that on disk (secondary storage) temoprarily, bring back later from disk to memory for continuing execution when appropriate – this technique is referred as **swapping** – swap space on the secondary memory



CPU Switch From Process to Process

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



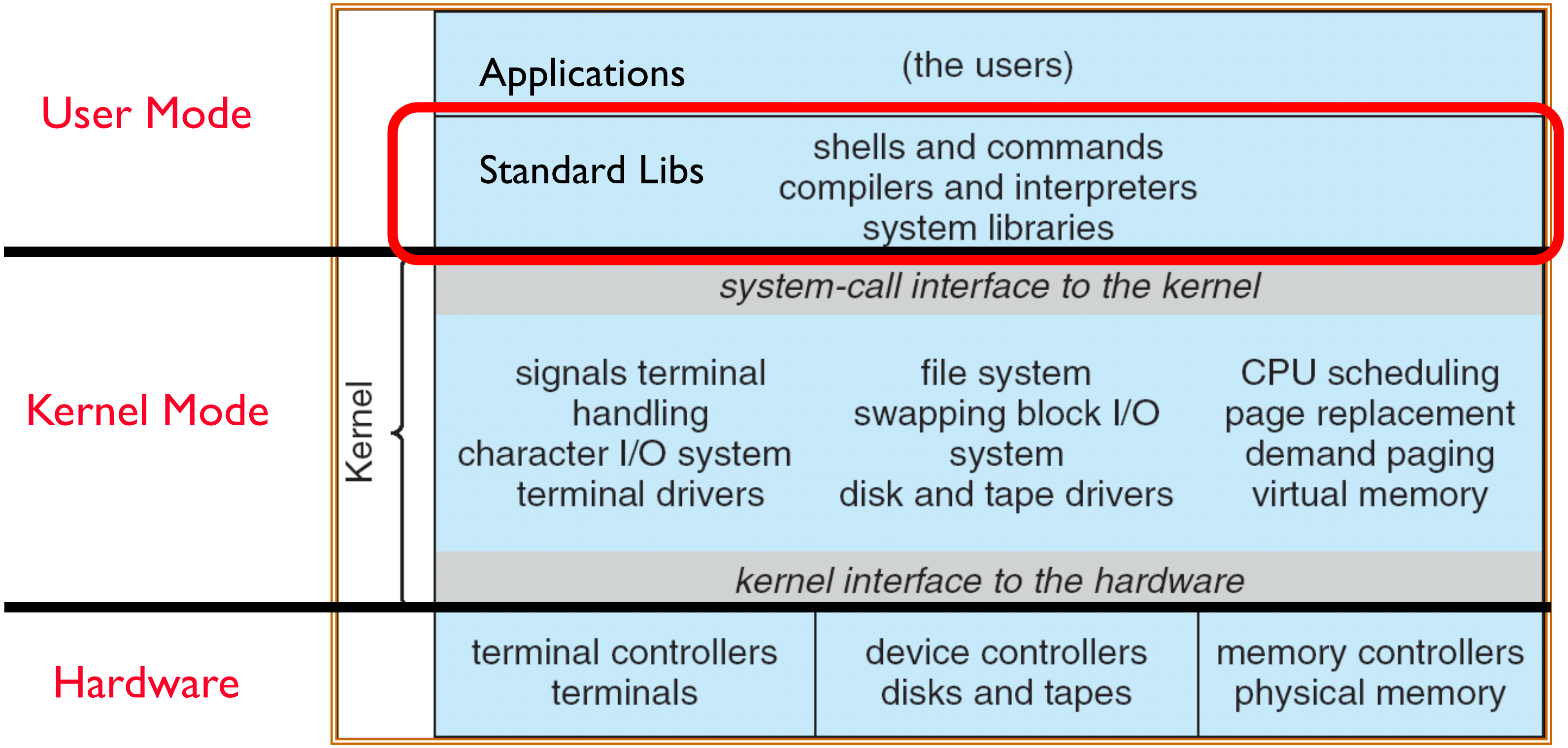
Context Switch

- When CPU switches to another process, OS must save the state of the old process and load the saved state from a new process via a context switch
- The context of a process represented in the PCB or the thread
  - Typically, this includes registers, program counter, and stack pointer
- Context-switch time is an overhead, during which the system does no useful work while switching
  - The more complex the OS and the PCB, the longer the context switch
- Switching speed varies from machine to machine, depending on the memory speed, number of registers copied, and the existence of special instructions such as a single instruction to load or store all registers
- Context-switch times are highly dependent on the underlying hardware
  - For instance, some processors provide multiple sets of registers (e.g., SUN UltraSPARC). A context switch simply requires changing the pointer to a different register set – in this case, multiple contexts can be loaded at once

Dual-mode Operation

Dual-mode operation allows OS to **protect** itself and other system components

- User mode and kernel mode - to distinguish between the execution of operating-system (kernel) code and user-defined code



User Mode

Applications (the users)

Standard Libs

shells and commands
compilers and interpreters
system libraries

Kernel Mode

Kernel

system-call interface to the kernel

signals terminal handling
character I/O system
terminal drivers

file system
swapping block I/O system
disk and tape drivers

CPU scheduling
page replacement
demand paging
virtual memory

kernel interface to the hardware

Hardware

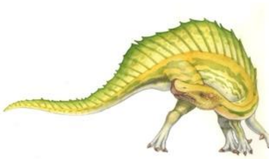
terminal controllers
terminals

device controllers
disks and tapes

memory controllers
physical memory

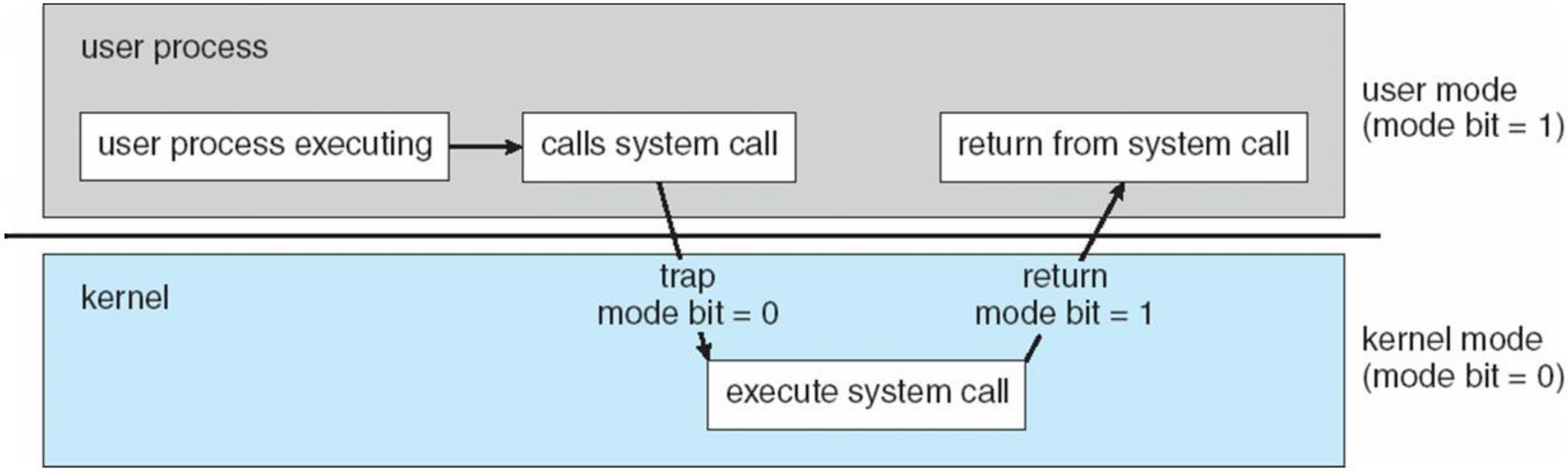
Dual-mode Operation (Cont.)

- The **dual-mode operation** provides the basic means of protecting the operating system from errant users and errant users from one another
- A mode bit provided by hardware - it provides ability to distinguish when the system is running user code (mode bit=1) or kernel code (mode bit=0)
- Some instructions designated as privileged, which can only be executed in the kernel mode, e.g., I/O control, timer management, and interrupt management
- If an attempt is made to execute a privileged instruction in user mode, the hardware treats it as illegal and traps it to the operating system
- The concept of dual modes can be extended beyond two modes.
    - For example, Intel processors have four separate protection rings, where ring 0 is kernel mode and ring 3 is user mode. Though rings 1 and 2 could be used for various operating-system services, in practice they are rarely used
    - ARMv8 systems have seven modes. CPUs that support virtualization frequently have a separate mode to indicate when the virtual machine manager (VMM) is in control of the system. In this mode, the VMM has more privileges than user processes but fewer than the kernel (to be discussed in Chapter 18)



Dual-mode Operation (Cont.)

- The initial control resides in the operating system - kernel mode. When control is given to a user application, the mode is set to user mode. Eventually, control is switched back to the operating system via an interrupt, a trap, or a system call
  - Can we make sure that a user program always returns to OS - timer
- When a user program requests a service from the operating system (via a system call), the system must transition from user mode to kernel mode to fulfil the request, return from system call resets it to user mode



Three Types of Mode Transfer

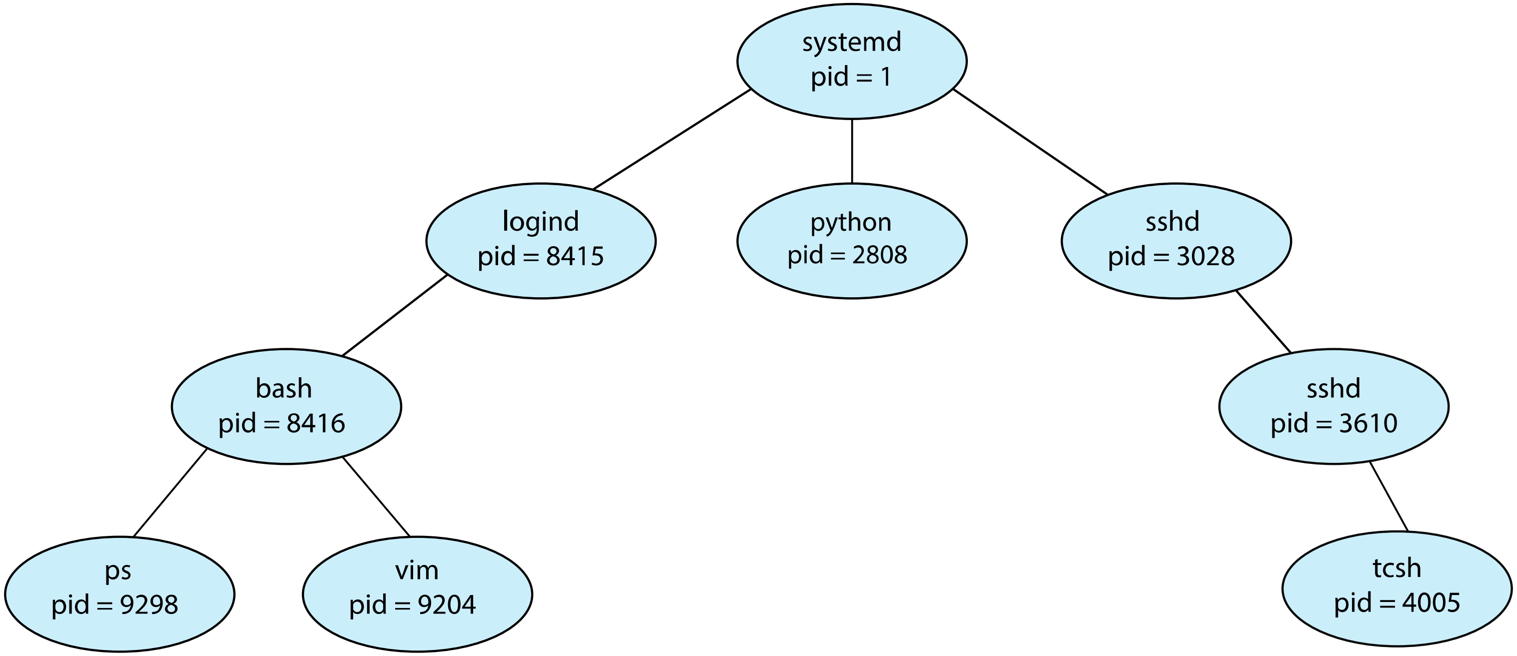
- System call
  - A user process requests a service from operating system, e.g., `exit()`,`write()`, `read()`,
  - Like a function call, but “outside” the user process
- Interrupt
  - External asynchronous events trigger context switch, e.g., timer, I/O
  - Independent of user process
- Trap or Exception
  - Internal synchronous events in process trigger context switch
  - E.g., protection violation (segmentation fault), algorithmic error (divided by zero), etc.

Operations on Processes

- The processes in most computer systems execute concurrently, and they are created and deleted dynamically. Thus, the operating systems must provide mechanisms for:
  - Process creation – to run a new program
  - Process termination – program execution completes or be terminated
- In general, a **parent** process can create **children** processes, which, in turn, can create other child processes, forming a **tree** of processes
  - Commonly, the desktop itself can be a parent process, double click an icon (mouse or touch screen), or input a valid command on a Shell, it starts executing a new process, i.e., a child process of the desktop process (i.e., a parent process)

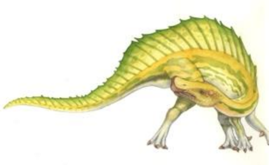
A Tree of Processes on Linux System

- A tree of processes on a typical Linux system



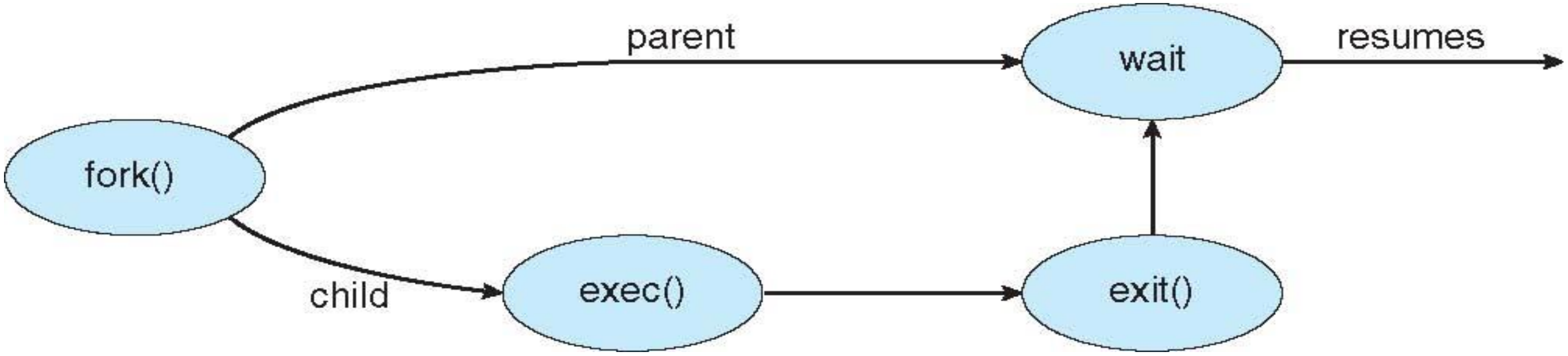
Process Creation

- Most operating systems (e.g., UNIX, Linux, Windows) identify a process according to a unique process identifier or pid, typically an integer
- When creating a child process, the child process needs certain resources - CPU time, memory, files, I/O devices to accomplish its task. The parent process may also pass along initialization data (input) to the child process
- During a process creation, there can be **three choices** in term of how resources can be shared between a parent process and its child process
  - Parent and children almost share all resources - fork()
  - Children share only a subset of parent's resources – clone() (in Chapter 4)
  - Parent and children share no resources
- There are typically **two options** for execution after process creation
  - The parent continues to execute concurrently with its children
  - The parent may wait until some or all of its children have finished execution.
  - For example - **`Unix shell`** process waits for **`a.out`** process to complete or **`shell`** process and **`a.out&`** process concurrently execute



Process Creation (Cont.)

- Consider **two choices** of address space for a new child process
  - A child process duplicates of parent (it has same program and data) – Unix/Linux
  - The child process has a new program loaded into it (different program and data)
- UNIX examples
  - **`fork()`** creates a new process, which duplicates entire address space of the parent. Both processes continue execution at the next instruction after **`fork()`**
  - **`exec()`** system call can be used after a **`fork()`** to replace the child process's address space with a new program - load a new program
  - Parent process can call **`wait()`** system call to wait for a child to finish

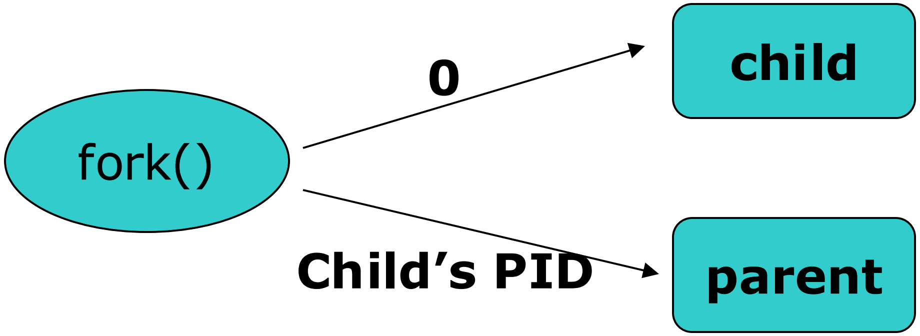


Process Creation (Cont.)

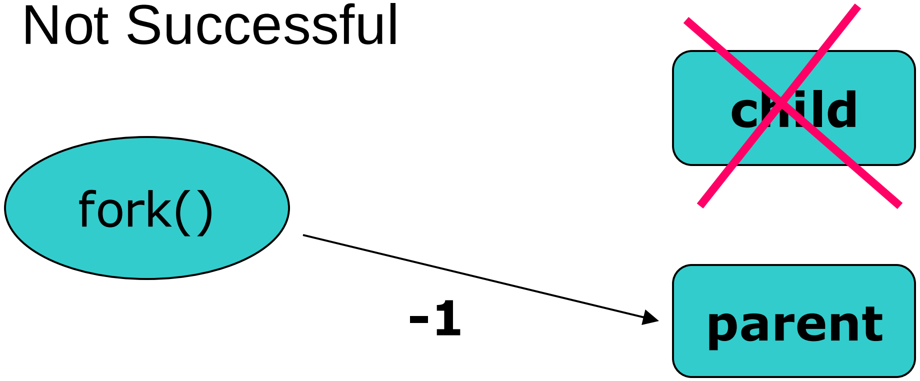
- `fork()` system call creates a copy of current process with a new pid
- Parent resumes execution after `fork()` and the child also starts execution with the same program counter, where the call to `fork()` returns
- The return value from `fork()` – two different integer values, one for parent, one for child; in another word, each process receives exactly one return value from `fork()`, respectively.
- The return value can be:
  - When > 0: running in (original) parent process
    - The return value (positive integer) is pid of the newly created child
  - When = 0: running in newly created child process
  - When < 0: (When = -1) running in original process
    - Error! Must handle somehow – child process not created successfully
- All state of original (parent) process duplicated in the child process
  - Memory (program and data), file descriptors, and etc…
- Because the child is a **copy** or **duplication** of the parent, each process has its own copy of the program and all the data – time consuming

Return values of fork()

- Successful



- Not Successful



errno is set to indicate error

Steps in UNIX fork

Steps to implement UNIX fork

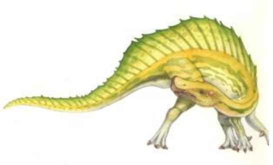
- Create and initialize the process control block (PCB) in the kernel
- Create a new address space – allocating memory
- Initialize the address space with a copy of the entire contents of the address space of the parent – time consuming
- Inherit the execution context of the parent (e.g., any open files)
- Inform the CPU scheduler that the newly created child process is ready to run

Parent vs. Child Processes

After **`fork()`** Parent and Child:

- **Duplicated**
  - Address space
  - Global & local variables
  - Current working directory
  - Root directory
  - Process resources
  - Resource limits
  - Program Counter - PC
  - …

- **Different**
  - PID
  - Return value from **`fork()`**
  - Running time
  - Running state



fork() example

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

#define BUFSIZE 1024
int main(int argc, char *argv[])
{
  char buf[BUFSIZE];
  size_t readlen, writelen, slen;
  pid_t cpid, mypid;
  pid_t pid = getpid();         /* get current processes PID */
  printf("Parent pid: %d\n", pid);
  cpid = fork();
  if (cpid > 0) {                          /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
  }  else if (cpid == 0) {         /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
  } else {
    perror("Fork failed");
    exit(1);
  }
  exit(0);
}
```

System Calls

- **`exec()`, `execlp()`**…
  - system call to change the program being run by the current process
  - creates a new process image from a regular executable file
- **`wait()`** – system call to wait for a (child) process to finish or on general wait for an event
- **`exit()`** – system call to terminate the current process, and free all its resources
- **`signal()`** – system call to send a notification to another process
- More details in UNIX man pages and in tutorial

C Program Forking Separate Process

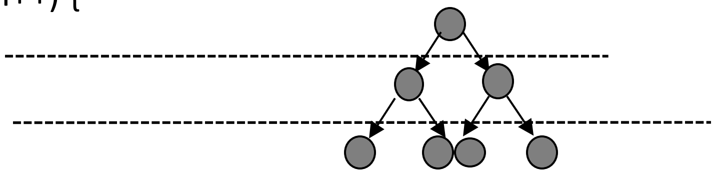
```
int main()
{
  pid_t  pid;
      /* fork another process */
      pid = fork();
      if (pid < 0) { /* error occurred */
                 fprintf(stderr, "Fork Failed");
                 exit(-1);
      }
      else if (pid == 0) { /* child process */
                 execlp("/bin/ls", "ls", NULL);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

Implementing a Shell

```
char *prog, **args;
int child_pid;

// Read and parse the input a line at a time
while (readAndParseCmdLine(&prog, &args)) {
   child_pid = fork();      // create a child process
    if (child_pid == 0) {
       exec(prog, args);       // I'm the child process.  Run program
      // NOT REACHED
    } else {
       wait(child_pid);       // I'm the parent, wait for child
       return 0;
    }
}
```

C Program Forking Separate Process

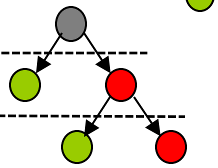
```
int main()
{
          for (int i = 0; i < 10; i++) {
          fork();
                fork();
          }
          return 0;
}
```



- Illustration:
  - Totally there are 10 iterations.
  - Inside the loop
    - 2 fork() function calls create 4 processes
  - Finally, there are 4^{10} processes.

C Program Forking Separate Process

```
int main()
{
      for (int i = 0; i < 10; i++)
            if (fork())
                        fork();
      return 0;
}
```



🔴 - Parent process after fork()
(return value > 0)

🟢 - Child process after fork()
(return value = 0)

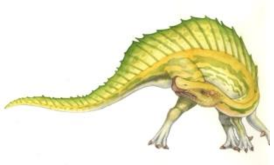
- Illustration:
  - Totally there are 10 iterations.
  - One process creates 2 child processes after each iteration.
  - Finally, there are 3^{10} processes.

Final Notes on fork()

- Process creation in Unix is unique. Most operating systems creates a process in a new address space, read in an *executable file*, and begin executing it. Unix takes an unusual approach of separating these steps into two distinct system calls: **`fork()`** and **`exec()`**
- In Linux, **`fork()`** is implemented via the use of copy-on-write pages. Copy-on-write or COW (to be discussed in Chapter 10) is a technique to delay or prevent copying of data. Rather than copy the address space, the parent and the child can **share** a single copy, i.e., the child process "points to" the parent process address space without duplication
  - This delays the copying in the address space until the content is changed or written to. In the case that the pages are never written — for example, if **`exec()`** is called immediately after **`fork()`** — they never need to be copied
  - This greatly simplifies and speeds up the process creation
- Linux also implements **`fork()`** via the **`clone()`** system call, which is considered to be more general. **`clone()`** system call uses a series of flags that allows to specify which set of resources, if any, the parent and child processes should share (to be discussed in Chapter 4)

Process Termination

- After a process executes the last statement and it asks the operating system to delete it using the **`exit()`** system call.
  - The process may return a status value (typically an integer) to its waiting parent process (via **`wait()`** system call from the parent process)
  - Most of the resources including memory (program, data, heap and stack), open files, and I/O buffers – are de-allocated and reclaimed by operating system
- Parent may terminate the execution of children processes using the **`abort()`** in Unix and Linux, **`TerminateProcess()`** in Windows
- Notice that a parent needs to know the identity or PID of its child during termination. Interestingly, when creating a child process, the identity of the newly created child process is passed to the parent with **`fork()`**
- A variety of reasons for a parent to terminate the execution of a child
  - The child has exceeded allocated resources (parent can inspect the child state)
  - Task assigned to child is no longer required
  - The parent is exiting, and some operating systems do not allow a child to continue if its parent terminates



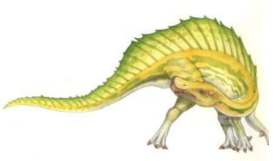
Process Termination (Cont.)

- Some OSes do not allow child to exist if its parent has terminated. If a process terminates, all its children must also be terminated. This is referred to as cascading termination, initiated by the operating system
- Under normal termination, **`exit()`** will be called either directly or indirectly, as the C runtime library includes a call to **`exit()`** by default
- The parent process may wait for termination of a child process by using the **`wait()`** system call
- The **`wait()`** is passed a parameter that allows the parent to obtain the exit status of the child, as well as the process identifier of the terminated child so that the parent can tell which of its children has terminated

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

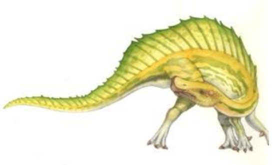
Process Termination (Cont.)

- When a process terminates, its resources are deallocated by the OS.
- However, there are a few clean up that needs to be done by the parent and OS. For instance, its entry in the process table (a kernel structure that keeps track of all processes in a system) remains until the parent calls **`wait()`**, as the process table or process list entry contains the process's exit status
- A process that has terminated, but whose parent has not yet called **`wait()`**, is known a zombie process – its corresponding entry in the process table, process ID, and PCB still exist. All other resources allocated are released. So, it is not runnable as it does not have an address space (program and data have all been deleted)
- All processes transition into this zombie state before termination, but generally they exist as zombies only very briefly
  - Once the parent calls **`wait()`**, the process identifier of the zombie process and its corresponding entry in the process table and PCB are released
- Such a design enables the parent, which in turn, informs the OS that a child process has terminated, so to trigger the final cleaning up



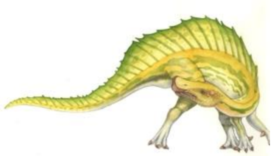
Process Termination (Cont.)

- If a parent terminated without invoking **wait()**, child process becomes an orphan – this process may still be runnable (if cascading is not used) or becomes a zombie process. Some mechanism must exist to reparent such orphan child processes to a new process. Otherwise, parentless terminated processes would forever remain zombies, wasting system memory.
- The **init** or **systemd** is assigned as a parent that periodically invokes **wait()**, allowing exit status of any orphaned process to be collected and releasing orphan's process identifier and the corresponding entry in the process table



Interprocess Communication

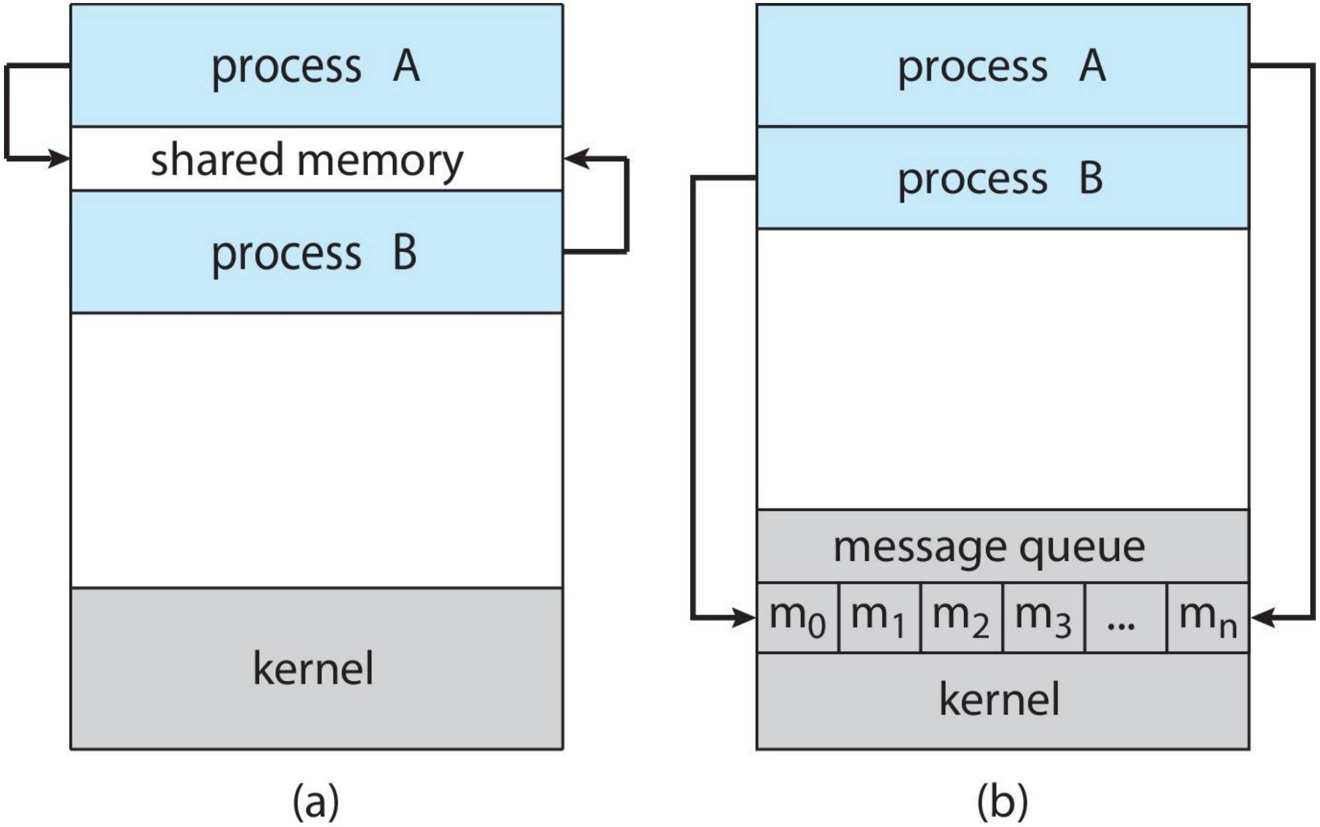
- Processes within a system may be independent or cooperating
- Cooperating processes can affect or be affected by other process, for instance two or more processes share data
- Reasons for cooperating processes or for a multi-threaded process:
  - Information sharing - several applications may be interested in the same piece of information (for instance, copying and pasting)
  - Computation speedup – executing parallel tasks or subtasks, for instance machine learning tasks
  - Modularity – functions divided into separate processes or threads
- Cooperating processes require an interprocess communication (IPC) - a mechanism that will allow them to exchange data
- There are generally two models of IPC
  - Shared memory - a region of memory is shared, and cooperating processes can exchange information by reading and writing data to the shared region
  - Message passing – explicit communication takes place by means of messages exchanged between cooperating processes



Communication Models

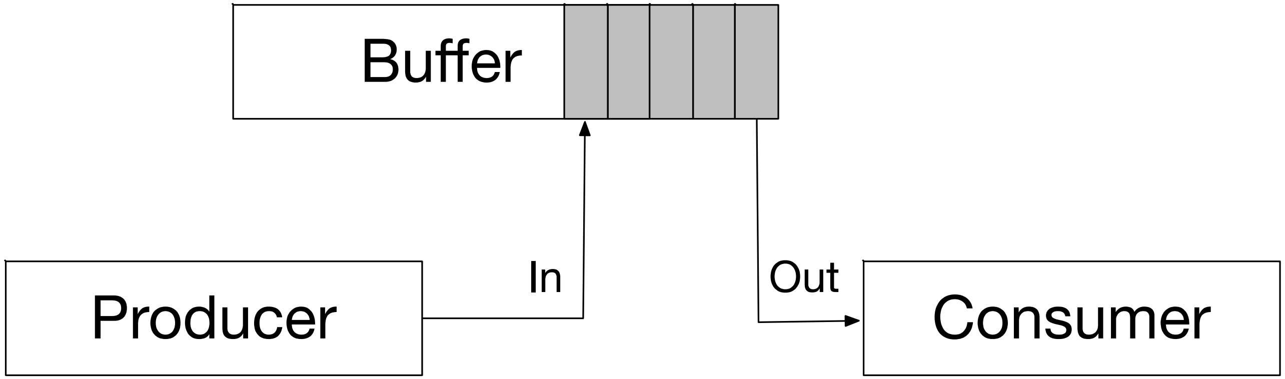
(a) Shared memory

(b) Message passing



Producer-Consumer Problem

- One paradigm for cooperating processes – **Producer** and **Consumer** represents a common type of cooperation between processes (We will discuss other type of cooperation in Chapters 6 and 7)
- An example: a producer process produces information that is consumed by a consumer process
  - unbounded-buffer places no practical limit on the size of the buffer
  - bounded-buffer assumes that there is a fixed buffer size



Bounded-Buffer – Shared-Memory Solution

- Shared data

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

- The above solution is simple nd correct, but only allows at most **`BUFFER_SIZE-1`** items in the buffer at the same time.
- Different implementations can use all **`BUFFER_SIZE`** items – easily fixed

Producer Process – Shared Memory

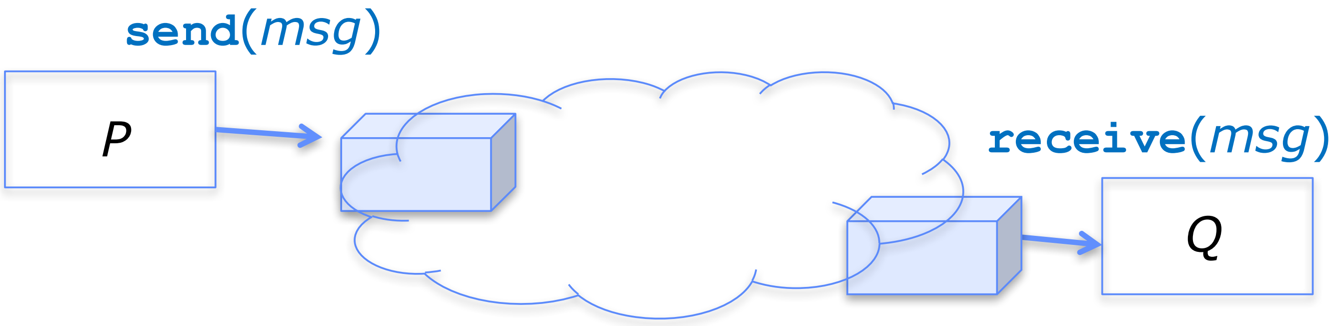
```
item next_produced;
while (true) {
   /* produce an item in next produced */
   while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE; /* move ptr
   forward */
}
```

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 */
}
```

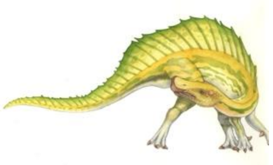
Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions without sharing the same address space
- It is particularly useful in a distributed environment
- IPC facility provides two operations:
  - **send**(*message*)
  - **receive**(*message*)
- The message size could be either fixed or variable



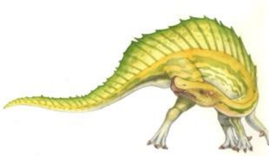
Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a communication link between them
  - Exchange messages via primitives **`send()`** and **`receive()`**
- There exist different methods in implementation of a link and **`send()`** and **`receive()`** operations
  - Direct or indirect communication
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering
- **Naming**: processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication



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 the communication link
  - A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
- The disadvantage in direction communication is the limited modularity of the resulting process definitions. For instance, changing the identifier of a process may necessitate examining all other cooperating processes – with the new identifier



Indirect Communication

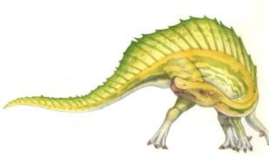
- Messages are sent to and received from mailboxes
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - A link is established between a pair of processes only if both members of the pair have a shared mailbox.
  - A link may be associated with more than two processes
  - Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox

Indirect Communication (Cont.)

- The operating system then must provide a mechanism for
  - Create a new mailbox
  - Send and receive messages through the mailbox
  - Destroy a mailbox
- Primitives are defined as:

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

  `receive``(`*`A, message`*`)` – receive a message from mailbox A
- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A, P_1, sends; P_2 and P_3 receive?
- 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, also known as synchronous and non-synchronous

- **Blocking** is considered **synchronous**
  - **Blocking send:** The sending process is blocked until the message is received by a receiving process or by a mailbox
  - **Blocking receive:** The receiver blocks until a message is available

- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send**: The sending process sends the message and resumes its operation without waiting for the message reception
  - **Non-blocking receive**: The receiver retrieves either a valid message or null

Synchronization (Cont.)

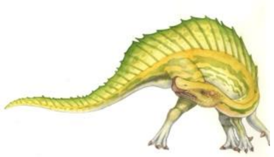
- Different combinations of **send()** and **receive()** possible. When both **send()** and **receive()** are blocking, we have a **rendezvous** between the sender and the receiver
- Producer-consumer problem solution becomes trivial

```
message next_produced;

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

   send(next_produced);

}

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

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

Buffering

- Whether communication is **direct** or **indirect**, messages exchanged by communicating processes reside in a temporary queue. Such queues can be implemented in 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 is full
3. Unbounded capacity – infinite length
   Sender never waits

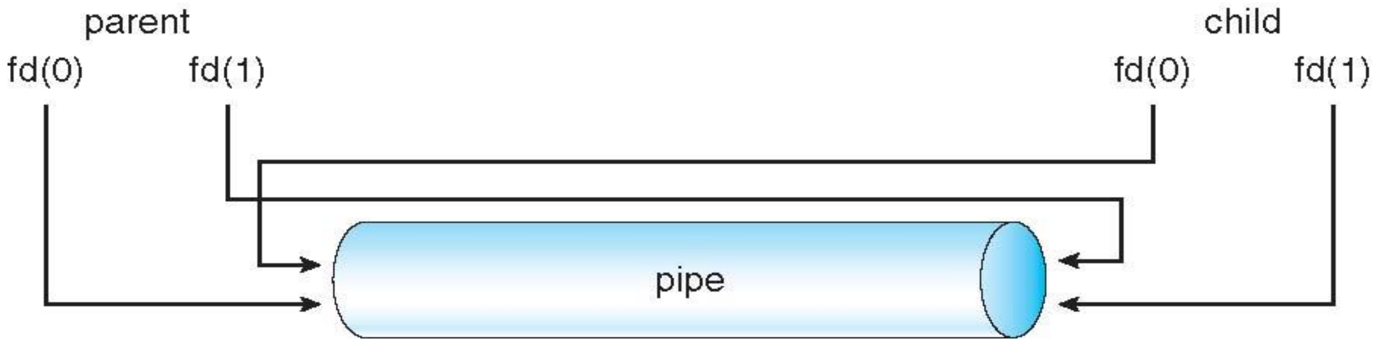


Pipes

- **Pipes** - one of the first IPC mechanisms in early UNIX systems. There are four issues to be considered when implementing a **`pipe()`**
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication (i.e., bidirectional), is it half duplex (data can travel only one way at a time) or full-duplex (data can travel in both directions at the same time)?
  - Must there exist a relationship (i.e., parent-child) between the communicating processes or any two processes?
  - Can the pipes be used over a network, or only on the same machine?
- **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 creates via **`fork()`**.
- **Named Pipes** – can be accessed without a parent-child relationship.

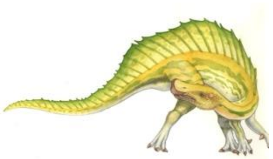
Ordinary Pipes

- Ordinary pipes communicate in a standard *Producer-Consumer* fashion - **`pipe(int fd[])`** in UNIX – unidirectional, allowing one-way communication
  - Producer writes to one end (the **write-end** of the pipe) **`fd[1]`**
  - Consumer reads from the other end (the **read-end** of the pipe) **`fd[0]`**
- This requires a parent-child relationship between two communicating processes running on the same machine
- An ordinary pipe cannot be accessed from outside the process that creates it
- An ordinary pipe ceases to exist after processes have finished communicating and terminated



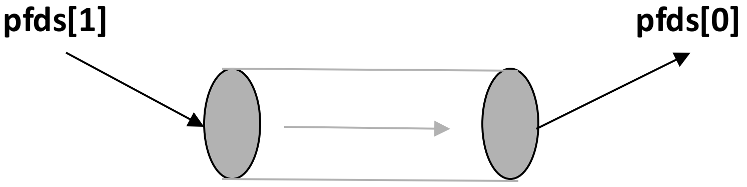
Ordinary Pipes (Cont.)

- UNIX treats a pipe as a special type of file, standard **`read()`** and **`write()`**, and the child **inherits** the pipe from its parent process
- In UNIX, the parent process uses **`pipe()`** creates a pipe and then uses **`fork()`** to create a child process. The child process inherits everything from the parent including the ordinary pipe (a special file). In this case, the parent and the child can communicate with each other using this pipe
- Noticing, if the parent needs to write to the pipe and the child reads from it, the parent must explicitly close the read-end of the pipe, and the child has to explicitly close the write-end of the pipe
- Ordinary pipes on Windows systems termed anonymous pipes, and behave similarly, access through **`ReadFile()`** and **`WriteFile()`** functions



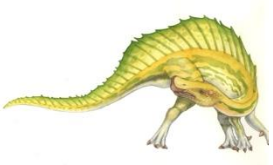
Pipe Example (Parent => Child)

```
int main()
{
  int pfds[2];
  char buf[30];
  pipe(pfds); /* Create a message pipe*/
  pid_t pid = fork(); /* 0 (child), non-zero (parent) */
  if ( pid != 0 ) {
    printf("PARENT: writing to pipe\n");
    close(pfds[0]);
    write(pfds[1], "test", 5);
    wait(NULL); /* Wait until the child returns*/
    printf("PARENT: exiting\n");
  } else {
    printf("CHILD: reading from pipe\n");
    close(pfds[1]);
    read(pfds[0], buf, 5);
    printf(“CHILD: read \"%s\"\n", buf);
  }
  return 0;
}
```



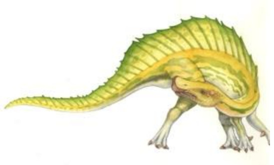
Named Pipes

- Named pipes are more powerful than ordinary pipes - communication is bidirectional, no parent-child relationship is required, and several processes can use it for communication, typically with several writers
- Name pipes continue to exist after the communicating processes have finished, until they are explicitly deleted
- Named pipes are referred to as FIFOs in UNIX systems, created using **`mkfifo()`** system call. They appear as typical files in the file system
  - FIFOs only supports half-duplex transmission. If data must travel in both directions simultaneously, two FIFOs are typically used. Additionally, the communicating processes must reside on the same machine
- Named pipes on Windows provide a richer communication mechanism
  - Full-duplex communication is allowed, and the communicating processes may reside on either the same or different machines
  - Named pipes are created with the **`CreateNamedPipe()`** function



Communications in Client-Server Systems

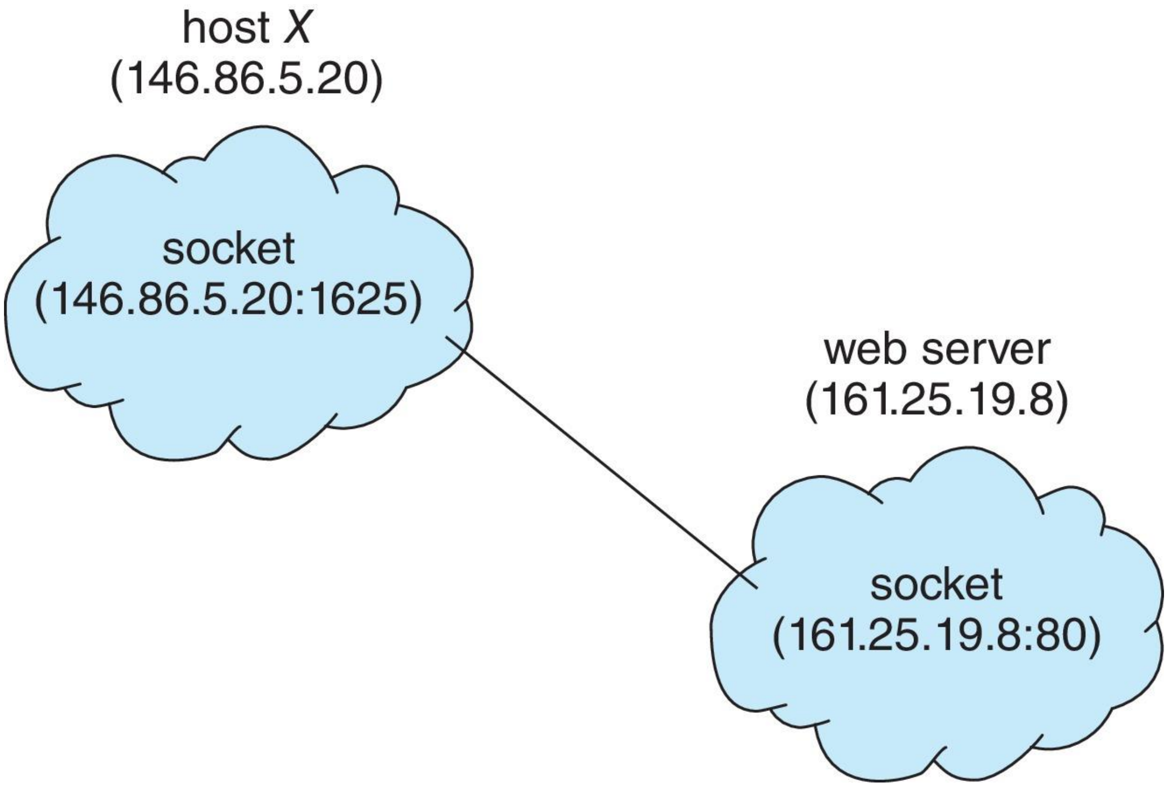
- **Sockets**
  - A socket is defined as an endpoint for communication. A pair of processes communicating over a network employs a pair of socket
  - A socket is uniquely identified by an IP address and a port number specifying the process or application on either client or server side
  - The server (always running) waits for incoming client requests by listening to a specified port (well knowns). Once a request is received, the server can choose to accept the request from the client to establish a socket connection for communication
- **Remote Procedure Calls** or **RPC**
  - One of the most common forms of remote services - the procedure-call mechanism for use between systems with network connections
  - A message-based communication scheme to provide remote services
  - The RPC scheme is commonly used for implementing distributed file systems by providing a set of RPC daemons and clients



Client-Server Communication - Sockets

- Servers (such as SSH, FTP, and HTTP) listen to well-known ports (an SSH server listens to port 22; an FTP server listens to port 21; and a web, or HTTP, server listens to port 80). All ports below 1024 are well known and are reserved for standard services - server port numbers
- Sever host address (IP address) or name (DNS translate that into an IP address) and server port number are well known. Servers are always ready waiting clients to contact
- Only clients can initiate the communication or send request to a server
- When a client initiates a request for a connection, it knows its own IP address, selects a client port number (greater than 1024), not current in use
- IP address – globally unique
    - 128.32.244.172 (IPv4 32-bit)
    - Fe80::4ad7:5ff:fecf:2607 (IPv6 128-bit)
- The 4-tuple (source IP address, source port number, destination IP address, destination port number) uniquely determines a pair of communication

Socket Communication



Communication between a pair of sockets

End of Chapter 3