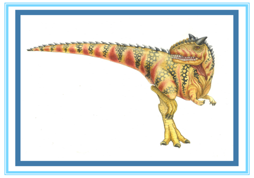
Chapter 3: Process Concept



Chapter 3: Process Concept

- Process Concept
- Process Scheduling
- Operations on Processes
- Inter-Process Communication (IPC)
- Communication in Client-Server Systems

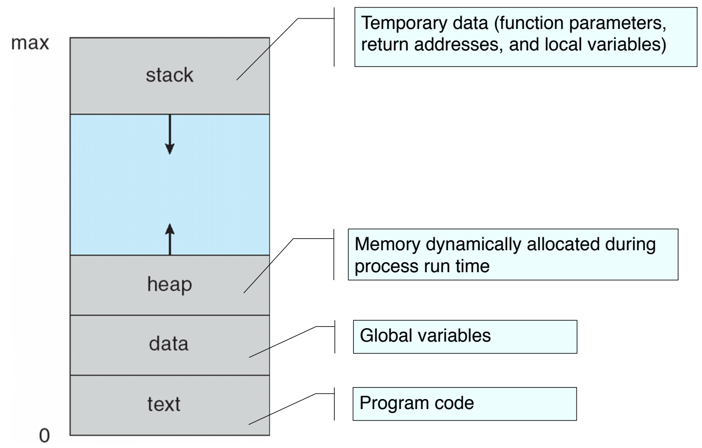
Objectives

- To introduce the notion of a process -- a program in execution, which forms the basis of all computations
- To describe the various operations and features of processes, including scheduling, creation and termination, and communication
- To explore interprocess communication using shared memory and message passing
- To describe communication in client-server systems

Process Concept

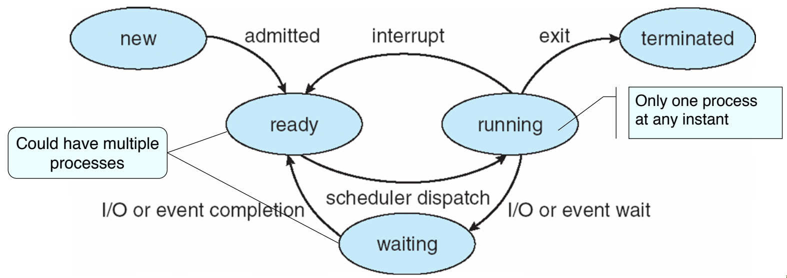
- An operating system executes a variety of programs:
  - A batch system executes **jobs** and a time-shared systems has **user programs** or **tasks**
- Textbook uses the terms ***job*** and ***process*** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
- A process include:
  - The program code, also called **text section**
  - Current activity, represented by the **program counter**, processor registers
  - **Stack** containing temporary data (such as function parameters, return addresses, local variables)
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time
- Program is a **passive** entity such as a file containing a list of instructions stored on disk (**executable file**),
- A process is an **active** entity, with a program counter specifying the next instruction to execute and a set of associated resources.
- A program becomes a process when an executable file is loaded into memory
  - Execution of program started via (1) GUI mouse clicks, (2) command line entry of its name
- One program can be used by several processes
  - Consider multiple users executing the same program

Process in Memory



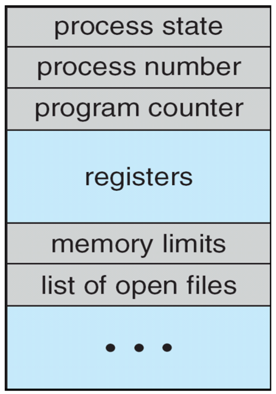
Process States and Diagram

- As a process executes, it changes state, which is defined by the current activity
  - **New:** The process is being created
  - **Running:** Instructions are being executed
  - **Waiting:** The process is waiting for some event to occur (such as I/O completion)
  - **Ready:** The process is waiting to be assigned to a processor (CPU)
  - **Terminated:** The process has finished execution

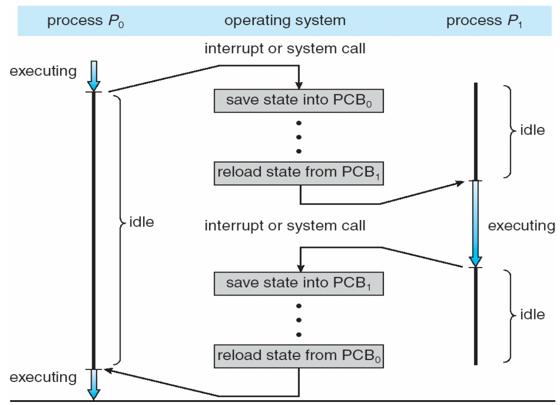


Process Control Block (PCB)

- Each process is represented by a **process control block** in the operating system (also called task control block)
- **Process state** – running, waiting, ready, halted, and so on
- **Program counter** – location of instruction to next execute
- **CPU registers** – contents of all process-centric registers
- **CPU scheduling information** - priorities, scheduling queue pointers
- **Memory-management information** – memory allocated to the process
- **Accounting information** – CPU used, clock time elapsed since start, time limits
- **I/O status information** – I/O devices allocated to process, list of open files



CPU Switch From Process to Process



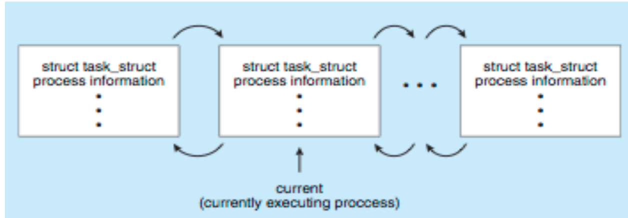
Threads

- So far, process has a single **thread** of execution
  - This single thread of control allows a process to perform only one task at a time.
  - If this is the case, a word-processor program cannot simultaneously type in characters and run the spell checker at the same time.
  - Most modern OS allows a process to have multiple threads of execution, thus to perform more than one task at a time.
  - This can best take advantage of the multicore systems, where multiple threads of one process can run in parallel.
- PCB has to be expanded to include information for each thread
  - Multiple locations can execute at once
  - Multiple program counters, one for each thread
- Chapter 4 discusses the details on **Thread**

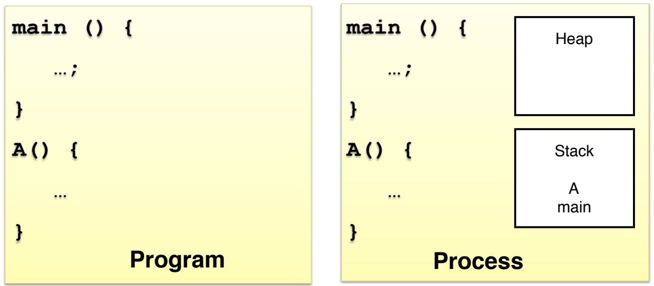
Process Representation in Linux

- Represented by the C structure `task_struct`

```
pid t pid; /* process identifier */
long state; /* state of the process */
unsigned int time slice /* scheduling information */
struct task struct *parent; /* this process's parent */
struct list head children; /* this process's children */
struct files struct *files; /* list of open files */
struct mm struct *mm; /* address space of this process */
```



Process =? Program

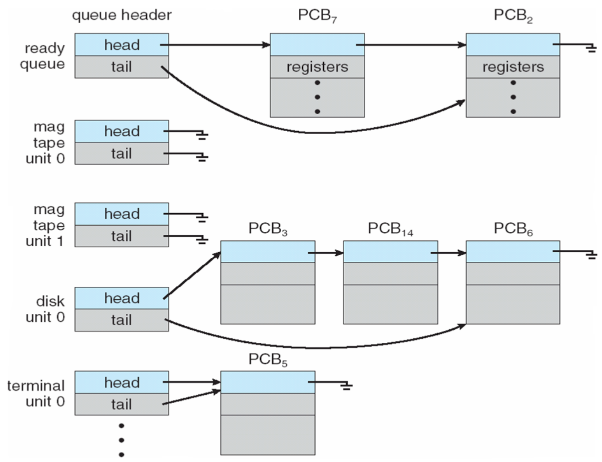


- A process is more than just a program:
  - A program is just part of the process state
- A process is "less" than a program:
  - A program can be invoked or called by more than one process
- A program is static (line of codes stored) and a process has a "life" and is always in some "state"

Process Scheduling

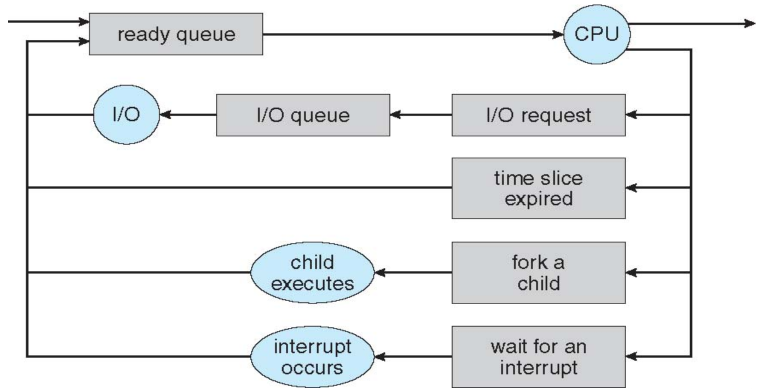
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

- Queuing diagram represents queues, resources, flows



Schedulers

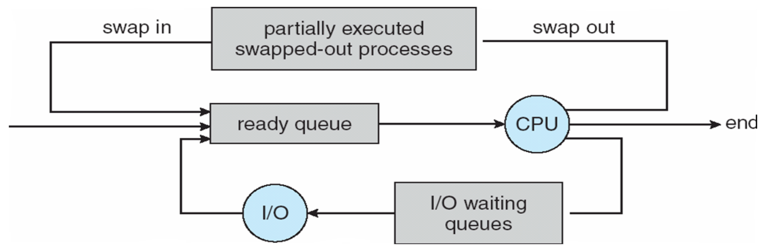
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU

- Short-term scheduler is invoked very frequently (milliseconds) ⌧ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⌧ (may be slow)
- The long-term scheduler controls the **degree of multiprogramming**

- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to be decreased
  - Remove a process from the memory, store on disk, bring back later in from disk to continue execution: **swapping**



Multitasking in Mobile Systems

- Some systems / early systems allow only one process to run, others suspended
- Apple, beginning with iOS4, provides a limited form of multitasking for user applications
  - A single foreground application run concurrently with multiple background applications
  - Single **foreground** process- currently on display and controlled via user interface
  - Multiple **background** processes– remain in memory, running, but not on the display
  - Limited applications can run in background include single, finite-length task, receiving notification of events, specific long-running tasks like audio playback
  - Constrained by battery life and memory usage
- Android runs foreground and background, with fewer limits
  - There is no constraint on the types of applications that can run in background
  - Background process uses a **service** to perform tasks, in which the service is a separate application component that runs on behalf of the background process
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use, thus efficient

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 overhead; the system does no useful work while switching
  - The more complex the OS and the PCB -> longer the context switch
  - Typical speed is a few milliseconds
- Context-switch times are highly dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU (such as the SUN UltraSPARC) -> multiple contexts loaded at once

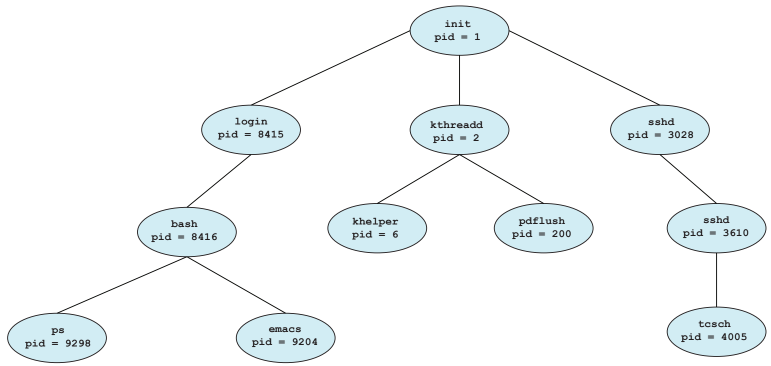
Operations on Processes

- The processes in most systems can execute concurrently, and they may be created and deleted dynamically.
- System must provide mechanisms for process creation, termination, and so on as detailed next

Process Creation

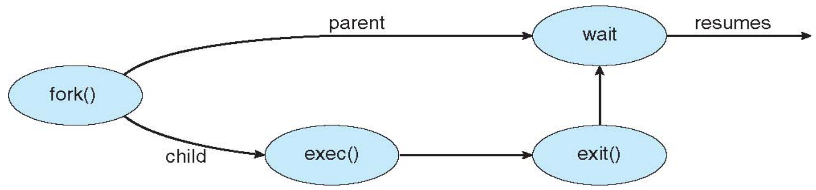
- A **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset 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



Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - `fork()` system call creates new process, which duplicates the address space of the parent
  - `exec()` system call used after a `fork()` to replace the process' memory space with a new program



What does it take to Create a Process?

- Must construct new PCB
  - Inexpensive
- Must set up new *page tables* for address space
  - More expensive
- Copy data from parent process? (Unix fork())
  - Semantics of Unix fork() are that the child process gets a complete copy of the parent memory and I/O state
  - Originally *very* expensive
- Copy I/O state (file handles, etc)
  - Medium expense

fork(): create a new process

Parent & Child:

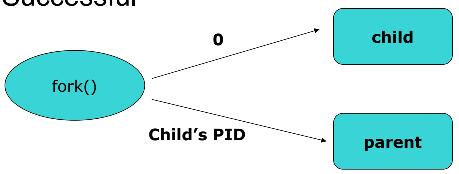
- Duplicated
  - Address space
  - Global & local variables
  - Current working directory
  - Root directory
  - Process resources
  - Resource limits
  - etc…
- Different
  - PID
  - Running time
  - Running state
  - Return values from fork()

Return values of fork()

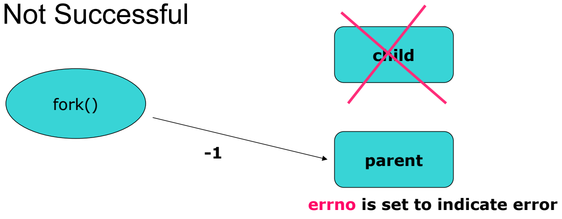
- The return value of the function is which discriminates the two processes of execution.
- Upon successful completion, fork() return 0 to the child process and return the process ID of the child process to the parent process.
- Otherwise, (pid_t)-1 is returned to the parent process, no child process is created, and errno is set to indicate the error.

Return values of fork()

- Successful



- Not Successful



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);
   }
}
```


Creating a Separate Process via Windows API

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

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

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

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

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

Fork() and CreateProcess()

- fork() has the child process inheriting the address space of its parent, while CreateProcess() requires loading a specified program into the address space of the child process at process creation
- fork() is passed no parameters, CreateProcess() expects no fewer than 10 parameters. In the example above, application `mspaint.exe` is loaded

Process Termination

- Process executes last statement and asks the operating system to delete it (**`exit()`**)
  - Output data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**`abort()`**)
  - Child has exceeded its usage of some of the resources that has been allocated
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating systems do not allow child to continue if its parent terminates
      - All children terminated - **cascading termination**

- A parent process may wait for the termination of a child process by using wait() system call, returning the pid, so the parent process can tell which of its children has terminated.

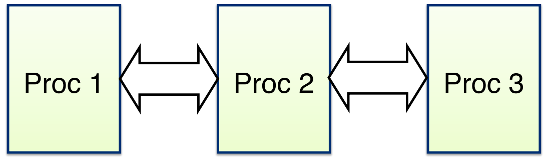
```
pid t pid; int status;
pid = wait(&status);
```

- If no parent waiting, then terminated process is a **zombie.** Once the parent calls wait(), the process identifier of the zombie process and its entry in the process table are released
- If parent terminated without calling wait(), the child processes are **orphans.** Linux and Unix assign the init process as the new process to orphan processes.

Interprocess Communication

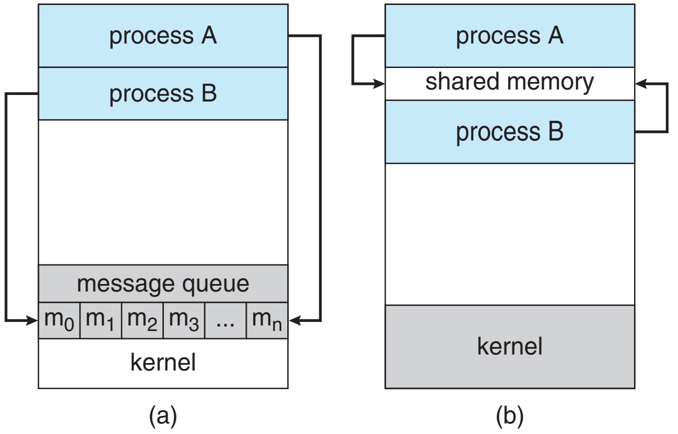
- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing, for instance a shared file
  - Computation speedup: subtasks of a task execute in parallel on multicore
  - Modularity: system functions are divided into multiple processes or threads
  - Convenience: users may work on multiple tasks in parallel
- Cooperating processes need an **interprocess communication** (**IPC**) mechanism that allow them to exchange data and information
- Two models of IPC, both common in operating systems
  - **Shared memory**
  - **Message passing**

Multiple Processes Collaboration



- Need communication mechanisms:
  - Separate address spaces different processes
  - Shared-Memory Mapping
    - Accomplished by mapping addresses to shared-memory regions
    - System calls such as **read**() and **write**() through memory
    - This suffers from cache coherency issues in multicores (with multiple cache)
  - Message Passing
    - **send**() and **receive**() messages
    - Can work across network
    - Better performance in multicore systems.

Communications Models



Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

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

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

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

Bounded-Buffer – Producer

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

Bounded Buffer – Consumer

```
item next consumed;

while (true) {
     while (in == out)
          ; /* do nothing */
     next consumed = buffer[out];
     out = (out + 1) % BUFFER SIZE;

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


Interprocess Communication – Message Passing

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

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - A link is 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
  - A link is established only if processes share a common mailbox
  - A link may be associated with more than two processes
  - Each pair of processes may share several communication links, i.e., mailboxes
  - Link may be unidirectional or bi-directional

Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

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

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

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Different methods can be chosen:
  - Allow a link to be associated with at most two processes
  - Allow at most one process at a time to execute a receive() operation
  - Allow the system to select arbitrarily the receiver. The system may define an algorithm for selecting which process will receive the message (for example, round-robin), Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
  - **Blocking send:** The sending process is blocked until the message is received by the receiving process or by the 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
  - **Non-blocking receive**: The receiver retrieves a valid message or null

}

Synchronization (Cont.)

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

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

message next consumed;
while (true) {
   receive(next consumed);

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


Buffering

- Queue of messages attached to the link (director indirect); implemented in one of three ways
  1. Zero capacity – 0 messages
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

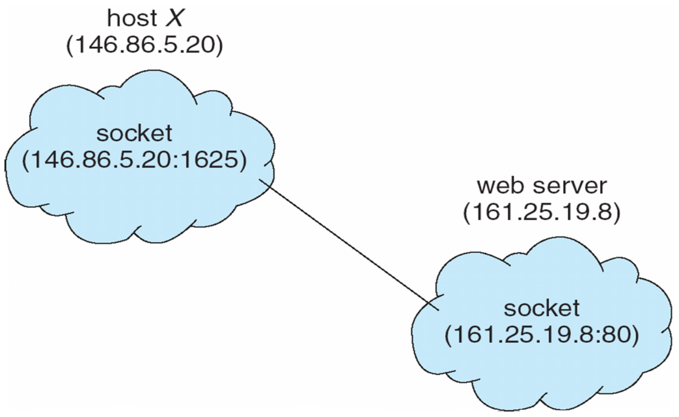
Communications in Client-Server Systems

- Sockets
- Pipes

Sockets

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

Socket Communication

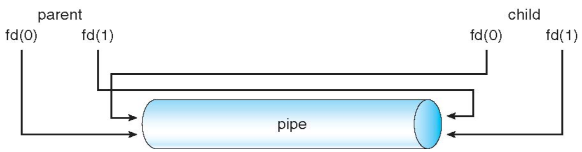


Pipes

- Acts as a conduit allowing two processes to communicate
- Pipes were one of the first IPC mechanisms in early UNIX systems
- **Four Issues** must be considered:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (such as ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network or must reside on the same machine?

Ordinary Pipes

- Ordinary pipes allow communication in standard producer-consumer style- pipe(int fd[])
- 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]
- Ordinary pipes are therefore unidirectional, UNIX treats a pipe as a special type of file.
- Require parent-child relationship between communicating processes on the same machine
- Ordinary pipe ceases to exist after the processes have finished communicating and terminated



- Windows calls these **anonymous pipes**
- See Unix and Windows code samples in textbook

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
- Name pipes continue to exist after communicating processes have finished.

End of Chapter 3