Chapter 3: Processes

1

Chapter Topics

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Communication in Client-Server Systems

2

Operating System (OS)

- An OS is a software that manages the computer hardware.
- An OS can be viewed as:
  - a resource allocator, where resources are CPU time, memory space, file-storage space, I/O devices, etc.
  - a control program that manages the execution of user programs to prevent errors and improper use of the computer.
- OS kernel handles at least process management, memory management, and inter-process communication.
- A user process requests a service from OS by via a system call.

3

Examples of Windows and UNIX System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

4

Transition from User to Kernel Mode

- A user process triggers a transition from user mode to kernel mode by executing a system call.

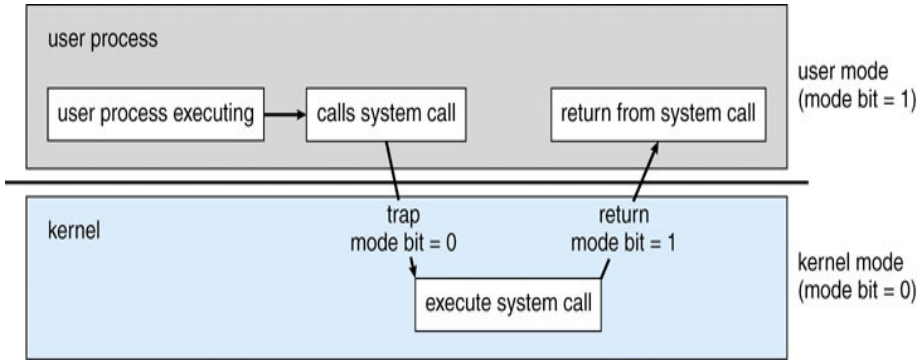


Figure 1.13 Transition from user to kernel mode

5

Operating System (OS) (cont'd)

- Multiprogramming: When a process has to wait for some event to occur (e.g., disk I/O), the CPU is switched to another process.
- Time-sharing: Each process can use the CPU up to certain time limit, called CPU time quantum (aka CPU time slice).
  - CPU time quantum is usually set to 10-100 msec.

6

Process

- Process: A program in execution.
  - OS processes execute system codes and user processes execute user codes.
- Somewhat interchangeable with *task*.
- A program becomes a process when the program (i.e., executable file) is loaded into memory.

7

Logical vs. Physical Address Space

- An address (of an instruction or a data) generated by the CPU is commonly referred to as a logical address (thus device independent), whereas an address seen by the memory unit is commonly referred as a physical address (meaning a main memory address).
  - A logical address is also called a virtual address.
- A set of all logical addresses generated by a process is a logical address space of the process.
- The set of all physical addresses corresponding to these logical addresses of a process is a physical address space of the process (i.e., main memory space).

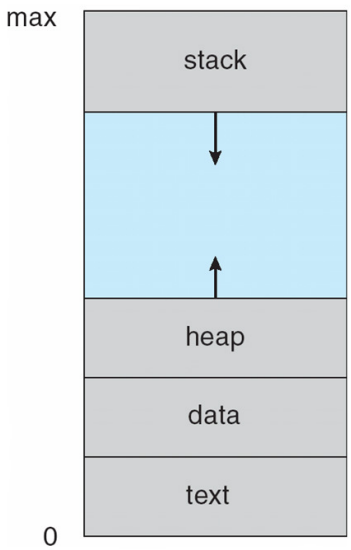
8

Logical Address Space of a Process

- The logical address space of a process includes:
  - The program code (also known as the text section).
  - A data section which contains global and static variables.
  - The stack which contains temporary data (such as function parameters, return addresses, and local variables).
  - A process may also include a heap, which is memory that is dynamically allocated during its run time.

9

Logical Address Space of a Process (cont'd)



10

Process State

- As a process executes, it changes *state.*
  - **new**: The process is being created.
  - **running**: Instructions are being executed.
  - **waiting** (aka **blocked**): The process is waiting for some event to occur (e.g., completion of an I/O).
  - **ready**: The process is waiting (in the ready queue) to be assigned a processor by the CPU scheduler.
  - **terminated**: The process has finished execution.

11

Process (cont'd)

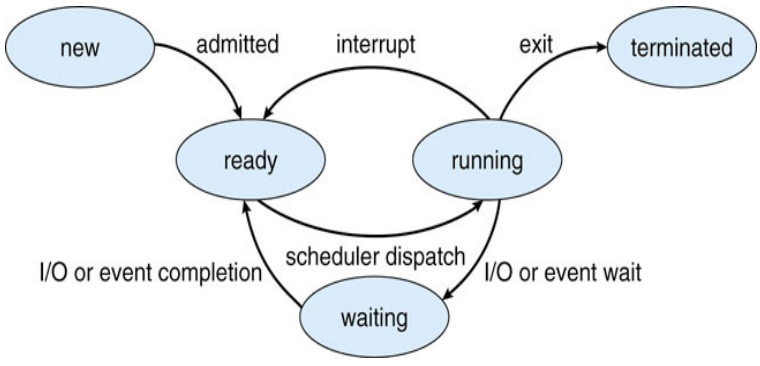


Figure 3.2 Diagram of Process State

12

Process Control Block (PCB)

- The process state is new, running, waiting, ready, or terminated.
- CPU registers include accumulators, index registers, stack pointers, and general-purpose registers.
- Memory management information includes the values of the base and limit registers, and the page table (or segment table).

Figure 3.3 Process Control Block (PCB)

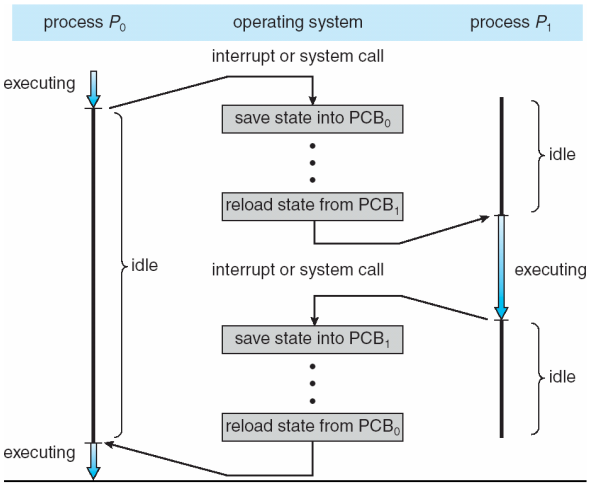
13

Context Switch

- The current ***execution state*** (aka ***context***) of a process is represented by the value of the program counter and the contents of the processor's registers.
- When the CPU is switched from one process to another process, the system must save the state of the currently running process and load the saved state of the process scheduled to run. This task is known as a context switch.
- The context of a process is saved in its PCB for the process to be resumed later.
- Context-switch time, which may takes a few milliseconds, is pure overhead because the system doesn't do any useful work during the switching.
  - Sun UltraSPARC processors provide multiple sets of registers, so a context switch (between them) simply requires changing the pointer to the current register set.

14

CPU Switch from one Process to Another Process



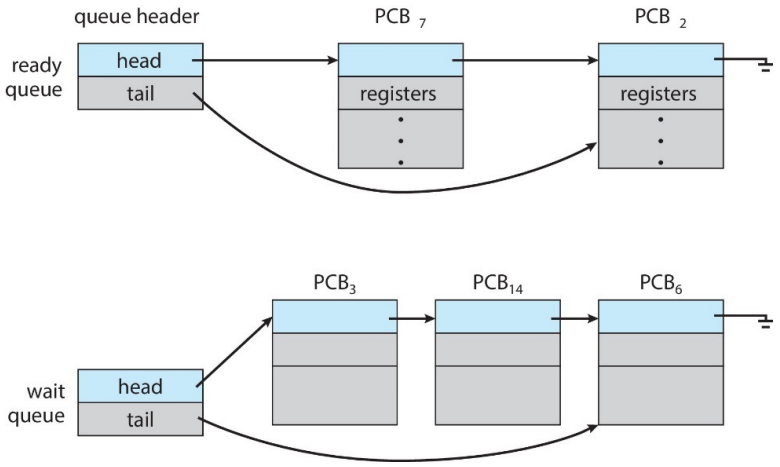
15

Process Scheduling Queues

- **Job queue:** The list of all the processes in the system. As processes enter the system, they are put into a job queue.
- **Ready queue:** The list of all processes residing in main memory, ready and waiting to execute.
  - The CPU scheduler selects a process (to be executed on the CPU) from the ready queue,
- **Wait queue:** The list of processes waiting for certain event to occur— such as completion of I/O .
- A processes migrates between the various queues during its lifetime.

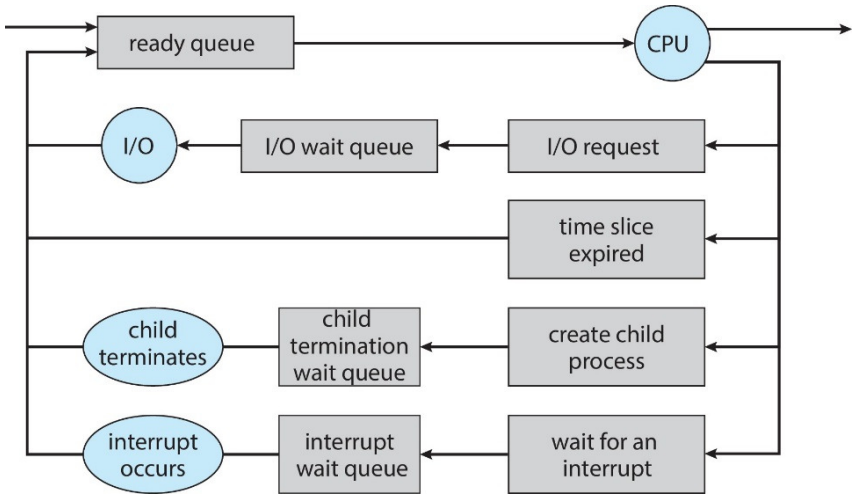
16

Ready Queue and Wait Queues



17

Queueing Diagram Representation of Process Scheduling



18

Schedulers

- The **Long-term scheduler** (or **job scheduler**): selects which processes should be brought into the ready queue.
- The **Short-term scheduler** (or **CPU scheduler**): selects which process should be executed next and allocates the CPU.
- The long-term scheduler executes much less frequently than the short-term scheduler.
  - The long-term scheduler controls the **degree of multiprogramming**, which means the number of processes in main memory (i.e., sharing the main memory space). So, the long-term scheduler may be invoked only when a process leaves the system.

19

Schedulers (cont'd)

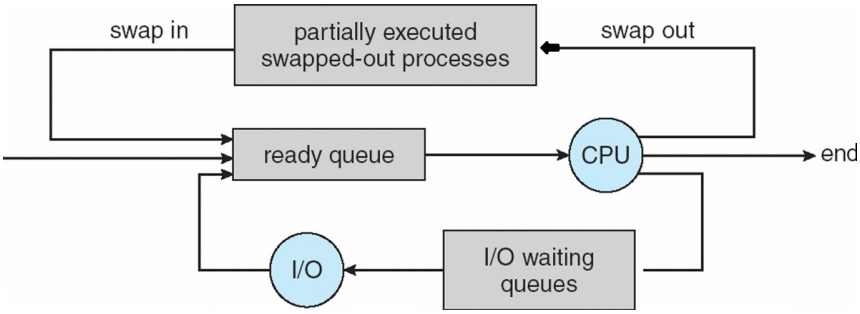
- Most processes can be described as either:

1. An **I/O-bound process** spends more time doing I/O than computations ⇒ many short CPU bursts.
   - CPU burst is the time a process utilizes the CPU before its next I/O request.
2. A **CPU-bound (or compute bound) process** spends more time doing computations ⇒ few very long CPU bursts.

- The long-term scheduler should select a good mix of the CPU-bound processes and the I/O-bound processes (if possible), in order to utilize the CPU and I/O devices in a balance manner.

20

Addition of Medium-term Scheduling



- The Medium-term scheduler swaps out (i.e., removes) some partially executed processes from the main memory into the swap space (created and managed on the disk), then later swaps them in, in order to control the degree of multiprogramming and/or to allocate more main memory space to other processes.

21

Process Creation

- A process may create a new process, via executing create-process system call.
  - The creating process is called a **parent process**, and the new process is called a **child process**, which in turn may create its own child processes.
- Possible resource sharing:
  - The parent process and its child process share all resources (e.g., code section, files, pipes).
  - A child process shares a subset of its parent's resources.
  - A child process and its parent do not share any resource.
- Possible execution:
  - The parent continues to execute concurrently with its children, or
  - The parent waits until some or all of its children have terminated.

22

Process Creation (cont'd)

- Address space of the child process:

1. The child process is a duplicate (aka clone) of the parent process as it has the same program and (copies of) data of the parent when it is created.
2. The child process can load a new program into its address space. Then it becomes a totally different process (not a clone of the parent process anymore).

23

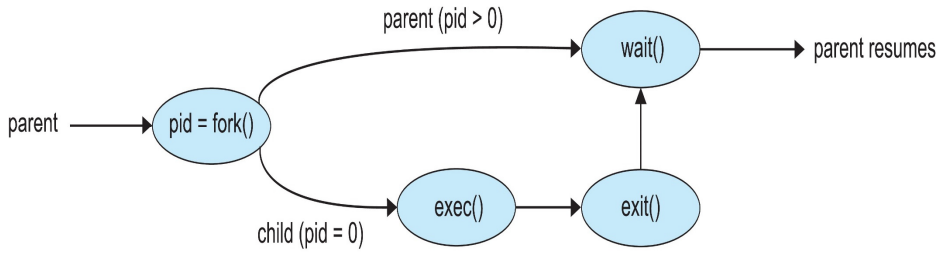
Process Creation in Unix

- In Unix, each process is identified by its process identifier (pid), which is a unique positive integer value.
- A new process is created by a **`fork()`** system call. The child process has a copy of the address space of the parent process.
- Both parent and child processes continue execution at the instruction after the **`fork()`** system call, with one difference:
  - The return value of the **`fork()`** system call is zero for the new child process.
  - The return value of the **`fork()`** system call is the process id of the new child process for the parent process.
- Usually an **`exec()`** system call (e.g., **`execlp()`**) is used after the **`fork()`** system call by one of the two processes to replace the process's address space with a new program specified, and then starts its execution.

24

Process Creation in Unix (cont'd)

- The parent process can issue **`wait()`** to wait for its child process terminates, instead of running concurrently with the child process.

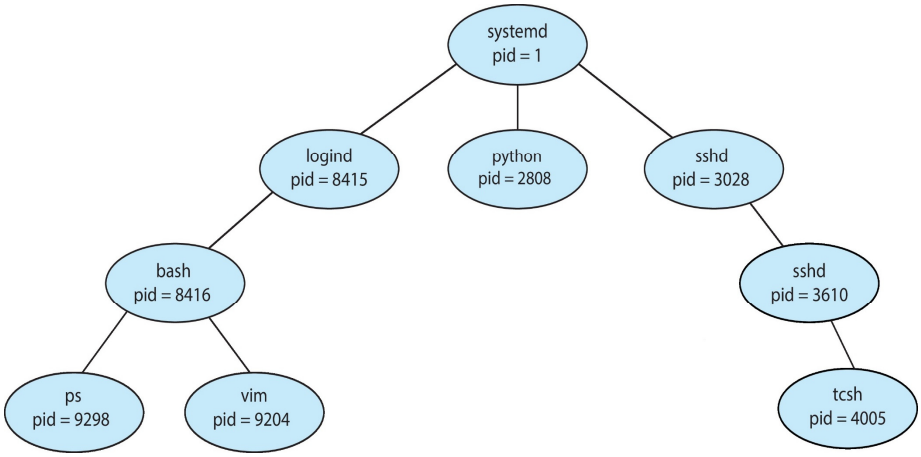


25

Figure 3.10 Creating a separate process using the UNIX fork() system call

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

int main()
{
pid_t pid;

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

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

   return 0;
}
```

A Tree of Processes on a Linux System



27

Process Termination

- A process terminates normally when it executes the last statement (or **`exit()`** system call).
- Parent may terminate the execution of a child process via system call (e.g., **`kill()`** in Unix, **`TerminateProcess()`** in Win32).
- Some operating systems do not allow a child to continue if its parent terminates:
  - All children are terminated, which is called **cascading termination.**
  - In UNIX, if the parent terminates, all its children have assigned the **`init`** process, which is the root parent process, as their new parent.

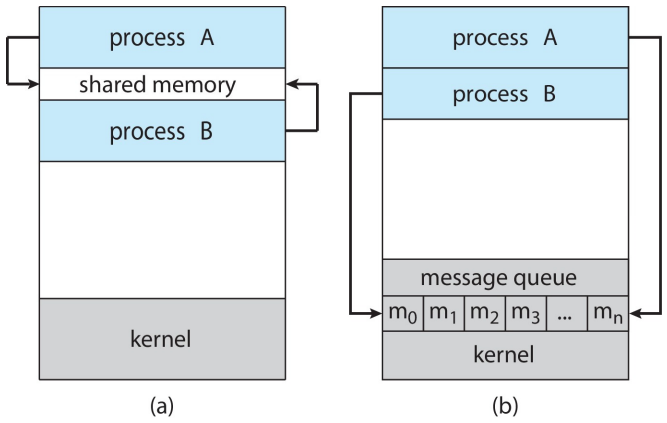
28

Interprocess Communication (IPC)

- Processes are concurrent if they exist at the same time.
  - E.g., multiple bank transactions processed at the same time.
- Independent processes cannot affect or be affected by the other processes executed in the system, as they don't share data.
  - The state of an independent process is not dependent on other processes, and its execution is deterministic.
- Cooperating processes can affect or be affected by other processes in the system as they share data.
  - Concurrent access to shared data may result in data inconsistency.
- Cooperating processes need interprocess communication (IPC) to share data and information.
- Two fundamental models of IPC are:
  1. Shared memory
  2. Message passing

29

Two Communications Models



Shared Memory **Message Passing**

30

Producer-Consumer Problem (aka Bounded Buffer Problem)

- A producer process produces data items that are consumed by a consumer process.
- To allow the producer and consumer processes to run concurrently, a (logical ring) buffer of items in memory can be defined and shared by the producer and consumer processes.
- The producer and consumer must be synchronized:
  - The consumer has to wait if the buffer is empty.
  - The producer has to wait if the buffer is full.
- We can use a counter that keeps track of the number of items in the buffer.
  - Initially, the counter is set to 0.
  - It is incremented by the producer after it places a new item in the buffer and is decremented by the consumer after it consumes an item in the buffer.

31

Producer-Consumer Problem (cont'd)

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0; // points to the next free buffer element
int out = 0; // points to the first full buffer element
int count = 0; // counts the number of data items in
  the buffer
```

32

Producer

```
item nextProduced;
while (true) {
        /*  produce an item and put in
nextProduced  */
    while (count == BUFFER_SIZE);
       // do nothing
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

33

Consumer

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

34

Race Condition

- count++ could be implemented as:
  register1 = count
  register1 = register1 + 1
  count = register1
- count-- could be implemented as:
  register2 = count
  register2 = register2 – 1
  count = register2
- Consider this execution interleaving with "count = 5" initially:
  S0: producer execute register1 = count {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = count {register2 = 5}
  S3: consumer execute register2 = register2 – 1 {register2 = 4}
  S4: producer execute count = register1 {count = 6 }
  S5: consumer execute count = register2 {count = 4}

35

Race Condition (cont'd)

- **Race condition**: The situation where several processes access and modify shared data concurrently, so that the outcome of the execution depends on the particular order in which the access takes place.
- To prevent race conditions, cooperating processes must be **synchronized on shared data**:
  – At least, cooperating processes should be designed such that, only one process (at a time) can access and change the shared data. This requirement is called **mutual exclusion**.

36

Interprocess Communication – Message Passing

- Message passing is a mechanism for processes to communicate and to synchronize their actions without sharing the same address space, so it is particularly useful in a distributed system.
  - A message can be received only after it is sent, so it can be used to synchronize the actions of two processes.
- A message-passing system provides at least two operations:
  - **send**(*message*)
  - **receive**(*message*)
- Message size is either fixed or variable.
- If two processes *P* and *Q* wish to communicate, a *communication link* between them must be established.

37

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 a communication link:
  - A link is established automatically.
  - A link is associated with exactly one pair of communicating processes.
  - Between each pair of processes, there exists exactly one link (during communication).
  - The link may be unidirectional, but is usually bidirectional.
  - A link is a logical connection here, rather than a physical communication channel, so we don't care about how a physical link can be implemented (for example, using a shared-memory, shared file, LAN, WAN, etc.)

38

Indirect Communication

- Messages are sent to and received from mailboxes.
  - A mailbox (aka *message queue, message slot*) can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
  - Each mailbox has a unique id.
  - Processes can communicate indirectly only if they share a mailbox.
- Send and receive primitives are defined as:
  - **send**(*A, message*): send a message to mailbox *A.*
  - **receive**(*A, message*): receive a message from mailbox *A.*
- Properties of a communication link:
  - A link is established only if processes have a shared mailbox.
  - A link (i.e., mailbox) may be associated with more than two processes.
  - Between two processes, there may be multiple communication links, each of which corresponds to a mailbox.

39

Indirect Communication (cont'd)

- A mailbox may be owned either by a process or by the OS.
  1. If a mailbox is owned by a process
     - The owner can only receive messages through the mailbox.
     - The user can only send messages to the mailbox.
     - If the owner process terminates, the mailbox disappears.
  2. A mailbox owned by OS can has an existence of its own, not attached to any particular process.
- The OS must provide a mechanism that allows a process to do the following:
  - create a new mailbox.
  - send and receive messages through a mailbox.
  - delete a mailbox.

40

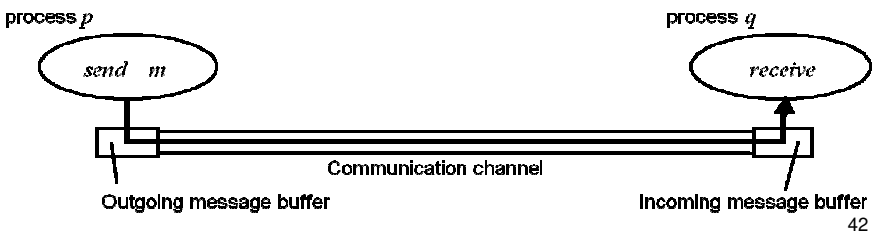
Blocking and Nonblocking Message Passing

- Message passing may be either blocking (aka *synchronous*) or nonblocking (aka *asynchronous*):
  - **Blocking send:** the sending process is blocked until the message is received by the receiving process (or by the mailbox).
  - **Nonblocking send:** the sending process sends the message and proceeds (i.e., resumes its operation).
  - **Blocking receive:** the receiver is blocked until a message arrives.
  - **Nonblocking receive:** The receiving process can proceeds with its program after issuing a *receive* operation if there is no message available, but later it must be notified of the arrival of the message in its buffer.

41

Buffering of Messages

- Messages exchanged by communicating processes reside in a temporary queue associate with a link. The capacity of a queue can be implemented as:
  - **Zero capacity:** no message can be held in the queue. The sender must be blocked until the message it sent is received.
  - **Bounded capacity:** at most *n* messages can be held in the queue. The sender must be blocked if the queue is full.
  - **Unbounded capacity:** potentially infinite number of messages can be held in the queue. So, the sender is never blocked.



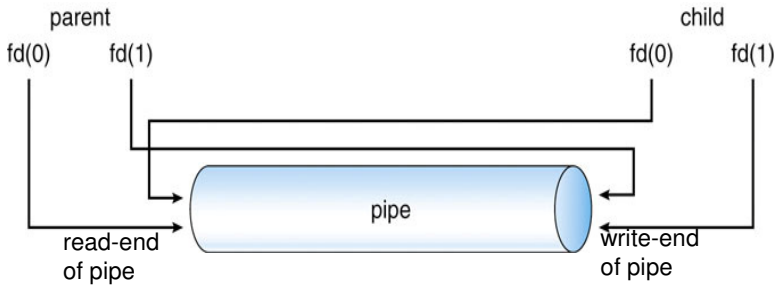
42

Ordinary Pipe

- An *ordinary pipe* (aka *anonymous pipe*) acts as a unidirectional conduit (i.e., FIFO queue) and allows a parent process and its child process to communicate in producer-consumer fashion:
  - One process writes to one end of the pipe (the write-end), and the other process reads from the other end (the read-end) of the pipe.
    - So, each pipe is used for one-way communication (i.e., data transfer).
  - In UNIX, an ordinary pipe is created by using **`pipe(int fd[2])`** system call. The pipe is accessed through the file descriptors **`fd[0]`** and **`fd[1]`:**
    - **`fd[0]`** corresponds to the read-end, and **`fd[1]`** corresponds to the write-end.
  - UNIX treats a pipe as a special type of file; thus, pipes can be accessed using ordinary **`read()`** and **`write()`** system calls.

43

File Descriptors for an Ordinary Pipe



- If the parent process executes **`pipe()`** system call, and then creates a child process, the child process inherits **`fd[0]`** and **`fd[1]`**, such that both processes share the pipe.

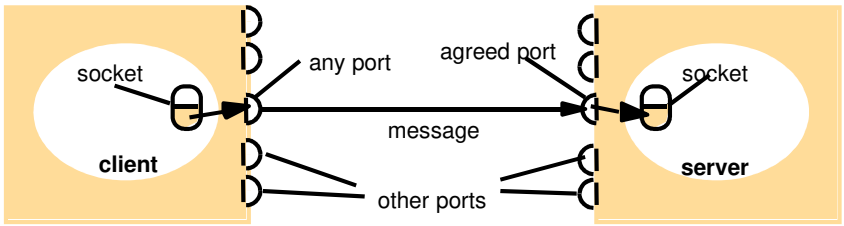
44

Communications in Client-Server Systems

- Sockets
  - The socket is simply a software construct that represents one endpoint of a communication between processes.
  - To send or receive a message, a process must create a socket bound to an IP address and a port number.
  - A program can read from a socket and write to a socket as if reading from a file or writing to a file.
- Remote Procedure Call (RPC)
  - Allows a program to invoke a procedure on a remote host.
- Remote Method Invocation (in Java)
  - Allows an object in one process to invoke the methods of an object in another process.

45

Sockets and Ports



- In the Internet protocol, a message is addressed to a (Internet address, port number) pair. A port is a message destination within a computer, and specified as a 16-bit integer.
- Severs generally publicize their port numbers for use by clients.
  - All port numbers below 1024 are considered well-known. For example, port 23 for telnet server, FTP server listens to port 21, and Web (or HTTP) server listens to port 80.

46

Sockets and Ports (cont'd)

- To send or receive a message, a process must create a socket bound to an IP address and a port number:
  - A server binds its socket to a specific server port.
  - A client binds its socket to any free local port.
- Messages sent to a particular (Internet address, port number) can be received only by a process whose socket is associated with that (Internet address, port number).
- However, multiple processes may send messages to the same port on the same computer.
- Socket-based communications enable applications to view networking as if it were file I/O — a program can read from a socket and write to a socket as if reading from a file and writing to a file.
- Each socket is associated with a particular Transport Layer Protocol — either UDP (*User Datagram Protocol*) or TCP (*Transmission Control Protocol*) — so it can be called a *datagram socket* or a *stream socket*.

47

Java API for Internet Addresses

- Java provides a class, *InetAddress*, that represents Internet addresses.
- An instance of *InetAddress* (i.e., *InetAddress* object) that contains an IP address can be created by calling a static method *getByName()* of *InetAddress*, giving a DNS hostname as an argument, as:

```
InetAddress  IP_address_of_computer=
  InetAddres.getByName(DBmachine.cs.wright.edu)
```

48

UDP Datagram Communication

- A datagram sent by UDP is transmitted from a sending process to a receiving process without acknowledgement and retries.
- The receiving process needs to specify an array of bytes of particular size in which to store a received message. Most cases, the underlying IP packet size is limited to 8 KB.
- **Sockets normally provide non-blocking *sends* and blocking *receives*:** The send operation returns when it has handed the message to the UDP or TCP protocol. On arrival, the message is placed in a queue for the socket that is bound to the destination port.
- The receiver process can set a timeout on its socket, just in case the message is lost.
  - When the timeout is set, receive method will block for the time specified then throw an InterruptedIOException.

49

Java API for UDP Datagrams

- For datagram communication, Java provides two classes: *DatagramPacket* and *DatagramSocket*.
- To send a message, a process can use a constructor of *DatagramPacket* class that makes an instance of *DatagramPacket* by specifying an array of bytes comprising the message, the length of the message, the IP address and the port number of the destination socket:

  DatagramPacket request = new DatagramPacket(message, length_of_message, server_IP_address, server_port_number);
- To receive a message, a process can use another constructor of *DatagramPacket* by specifying an array of bytes (to store the received message) and its length:

  DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
- A received message is put into the *DatagramPacket* together with the IP address and the port number of the sending socket.
  - The message can be retrieved from the *DatagramPacket* by using the method *getData()*.
  - The methods *getAddress()* and *getPort()* retrieve the IP address and port number (of the sending socket), respectively.

50

Java API for UDP Datagrams (cont'd)

- *DatagramSocket* provides a constructor that takes a port number as argument, and also provides a no-argument constructor that allows the system to choose a free local port:

  aSocket = new DatagramSocket(6789);

  aSocket = new DatagramSocket();
- *DatagramSocket* provides methods including the following:
  - *send()*: transmits a *DatagramPacket*.
  - *receive()*: waits for a *DatagramPacket.*
  - *setSoTimeout()*: sets a timeout in msec for the *receive()* method.
  - *connect()*: for a sever to connect to a client with a particular remote IP address and a port number. Packets are sent to or received from only that remote IP address and port number.
    - Attempts to send packets to a different host or a port will throw an *IllegalArgumentExeption*.
    - Packets received from a different host or a different port will be discarded without an exception or other notification.
    - By default a datagram socket of a server is not *connected* to only a specific client.

51

UDP client sends a message to the server and gets a reply

```
import java.net.*;
import java.io.*;
public class UDPClient{
   public static void main(String args[]){
            // args give message content and server hostname
            DatagramSocket aSocket = null;
             try {
                        aSocket = new DatagramSocket();
                        byte[] m = args[0].getBytes();
                        InetAddress aHost = InetAddress.getByName(args[1]);
                        int serverPort = 6789;
                        DatagramPacket request = new DatagramPacket(m,  args[0].length(), aHost, serverPort);
                        aSocket.send(request);
                        byte[] buffer = new byte[1000];
                        DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
                        aSocket.receive(reply);
                        System.out.println("Reply: " + new String(reply.getData()));
              }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
              }catch (IOException e){System.out.println("IO: " + e.getMessage());}
            }finally {if(aSocket != null) aSocket.close();}
  }
}
```

52

UDP server repeatedly receives a request and sends it back to the client

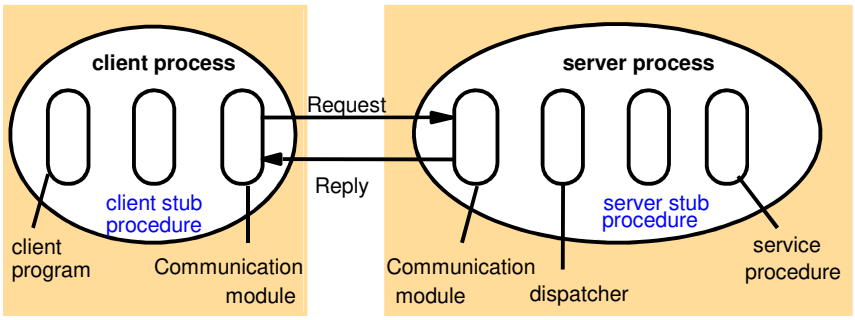
```
import java.net.*;
import java.io.*;
public class UDPServer{
        public static void main(String args[]){
        DatagramSocket aSocket = null;
          try{
                aSocket = new DatagramSocket(6789);
                byte[] buffer = new byte[1000];
                while(true){
                  DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                  aSocket.receive(request);
                  DatagramPacket reply = new DatagramPacket(request.getData(),
                        request.getLength(), request.getAddress(), request.getPort());
                  aSocket.send(reply);
                }
           }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
           }catch (IOException e) {System.out.println("IO: " + e.getMessage());}
        }finally {if(aSocket != null) aSocket.close();}
  }
}
```


TCP Stream Communication in Java

- For TCP streams, Java API provides two classes: *ServerSocket* and *Socket*.
- The client creates a stream socket (a *Socket*) bound to any local port and then makes a connection request to a server at its server port.
- The server creates a listening socket (a *ServerSocket*) bound to a server port and waits for clients to request connections. The listening socket maintains a queue of incoming connection requests.
- When the server accepts a connection, a new stream socket is created for the server to communicate with the client, meanwhile the listening socket is retained to receive the connection requests from other clients.
- On its *Socket*, a process can create a *DataInputStream* and a *DataOutputStream* that implement methods for reading and writing primitive data types.
  - A process can send data to the other one by writing to its *DataOutputStream*, and the other process obtains the data by reading from its *DataInputStream*.

Remote Procedure Call

- Remote Procedure Call (RPC) allows client programs to call the procedures in server programs running in separate processes and generally in different computers from the client.



55

Role of Client and Server Stub Procedures in RPC

- The client includes one *stub procedure* for each remote procedure. The *client stub procedure* is the client-side proxy for the actual procedure on the server and behaves like the local procedure to the client, in order to make the remote call transparent to the client.
- Instead of executing the call, the client-side stub locates the server and marshals the procedure identifier and the arguments into a request message, then sends it to the server via the communication module.
  - *Marshaling* is the process of taking a collection of data items and assembling them into a suitable form (e.g., XDR) in a message.
  - *External Data Representation* (XDR) is a standard for the machine-independent representation of the data structures and primitive values.
- The *dispatcher* selects one of the server stub procedures according to the procedure identifier in the request message.
- The *server stub procedure* unmarshals the arguments in the request message, calls the corresponding service procedure which implements the requested procedure, and then marshals the return values (received from the service procedure) into the reply message.
  - *Unmarshaling* produces an equivalent collection of data items by disassembling a message at the destination.

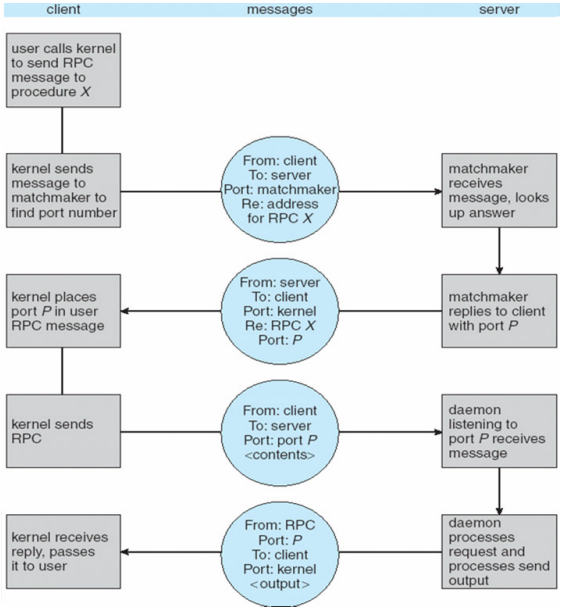
56

RPC Using a *Matchmaker* (aka *Port Mapper*)

- A question is how does a client know the port number for a RPC on the server machine?
- OS may provide a *matchmaker* daemon on a fixed RPC port (e.g., 111) on the server machine. The matchmaker maintains a list of the port numbers of the RPCs available on the server machine.
- A client's message containing the name of the RPC is sent to the matchmaker, requesting the port number of the RPC it needs to execute.
- The requested port number is returned by the matchmaker to the client-side, and the client's RPC request is sent to that port number.

57

RPC Using a Matchmaker (cont'd)



58