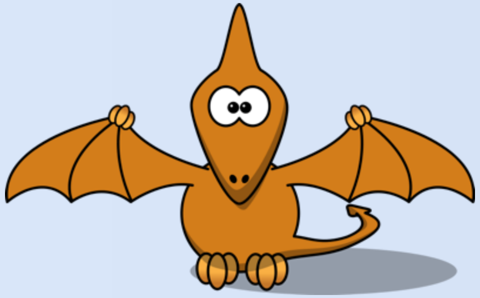
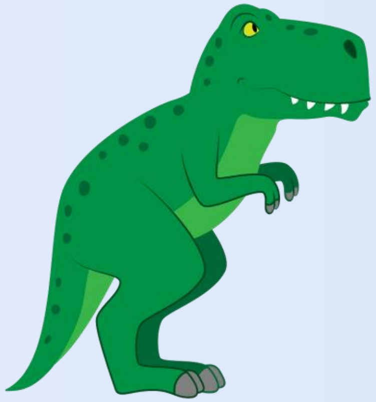


Chapter 3.

Processes 2.



Operating System Concepts (10th Ed.)

3.4 Interprocess Communication

- Processes executing concurrently may be
  - either *independent* processes or *cooperating* processes.
  - A process is ***independent***
    - if it *does not share data* with any other processes.
  - A process is ***cooperating***
    - if it can *affect* or *be affected* by the other processes.
    - Cleary, any processes that *shares data* with other processes is a cooperating process.

3.4 Interprocess Communication

- IPC: *Inter-Process Communication*
  - Cooperating processes require an IPC mechanism
    - that will allow them to exchange data
    - that is, ***send data*** to and ***receive data*** from each other.

- Two fundamental models of IPC:
  - ***shared memory***
  - ***message passing***

3.4 Interprocess Communication

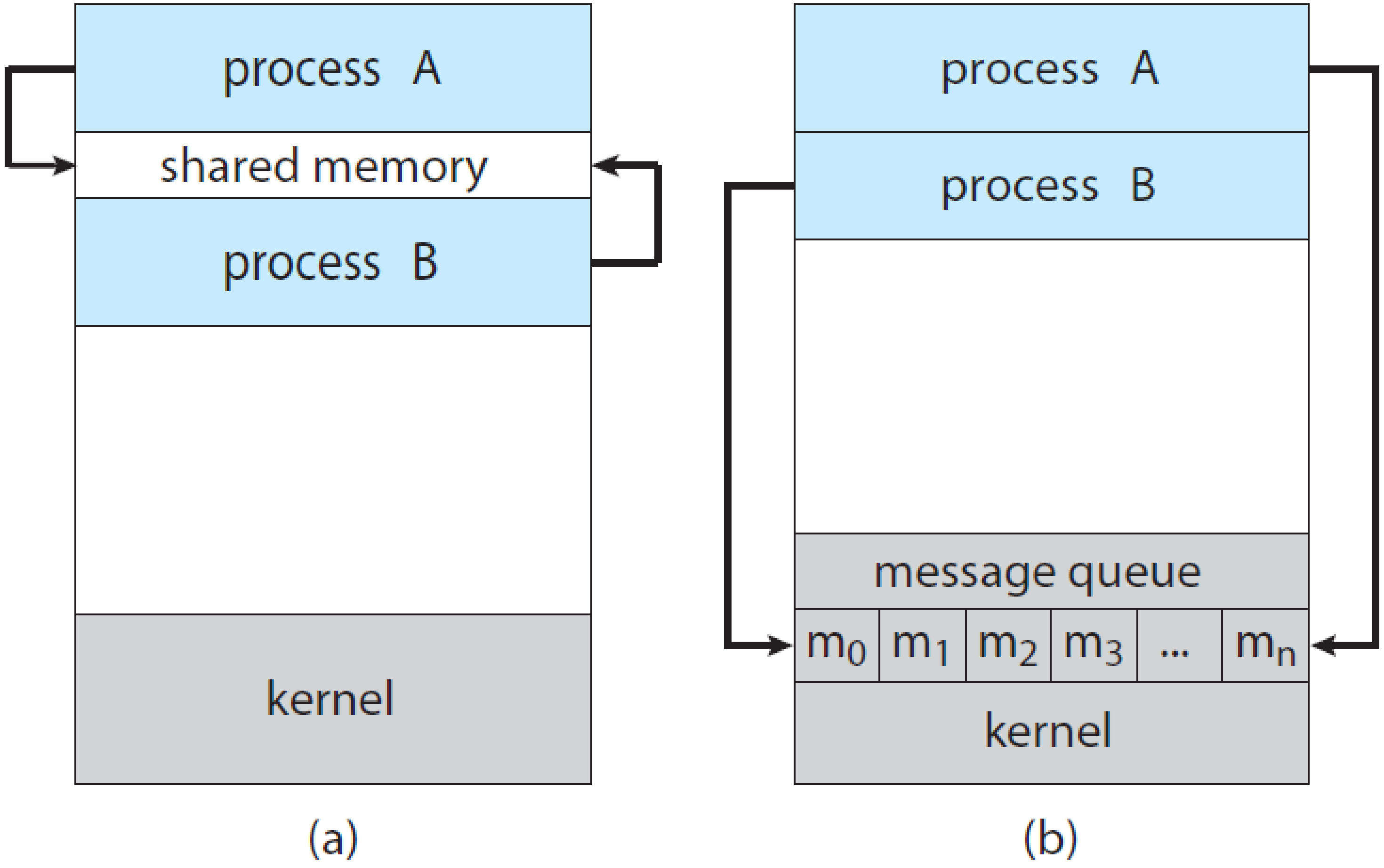


Figure 3.11 *Communication models. (a) Shared memory. and (b) Message Passing.*

3.5 IPC in Shared-Memory Systems

- Consider the *Producer-Consumer* Problem
  - to illustrate the concept of cooperating processes.
  - a common paradigm for cooperating processes.

- **Producer-Consumer Problem**:
  - A ***producer*** produces information that is consumed by a ***consumer***.
  - For example,
    - a *compiler* produces assembly code, and a *assembler* consumes it.
    - a *web server* produces an HTML file, and a *browser* consumes it.

3.5 IPC in Shared-Memory Systems

- A solution using **shared-memory**:
  - To allow producer and consumer to run *concurrently*.
  - Let a *buffer* of items be available,
    - a producer can *fill the buffer*, and
    - a consumer can *empty the buffer*.
  - A ***shared memory*** is a region of memory
    - that is shared by the producer and consumer processes.

3.5 IPC in Shared-Memory Systems

- Define a shared buffer
  - in a region of memory shared by the producer and consumer process.

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
} item;

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

3.5 IPC in Shared-Memory Systems

```
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;
}
```

Figure 3.12 *The producer process using shared memory.*

3.5 IPC in Shared-Memory Systems

```
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 */
}
```

Figure 3.13 *The consumer process using shared memory.*

3.6 IPC in Message-Passing Systems

- The scheme of using shared-memory
  - requires that these processes *share a region of memory* and that
  - the *code* for accessing and manipulating the shared memory
    - be written *explicitly* by the application programmer.

- **Message-Passing**:
  - O/S provides the means for cooperating processes
    - to communicate with each other via a *message-passing* facility.

3.6 IPC in Message-Passing Systems

- Two operations of the message-passing facility:
  - **send(message)**
  - **receive(message)**

```
message next_produced;

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

     send(next_produced);
}
```

Figure 3.14 *The producer process using message passing.*

3.6 IPC in Message-Passing Systems

```
message next_consumed;

while (true) {
     receive(next_consumed);

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

Figure 3.15 *The consumer process using message passing.*

3.6 IPC in Message-Passing Systems

- Communication Links:
  - if two processes P and Q want to communicate,
    - the must *send to* and *receive* messages *from* each other
  - This comm. link can be implemented in a variety of ways.
    - *direct* or *indirect* communication.
    - *synchronous* and *asynchronous* communication.
    - *automatic* or *explicit* buffering.

3.6 IPC in Message-Passing Systems

- Under ***direct*** communication,
  - each process that wants to communicate
    - must explicitly *name* the *recipient* or *sender* of the communication.
  - The primitives of this scheme:
    - send(P, message) – send a message to process P.
    - receive(Q, message) – receive a message from process Q.

3.6 IPC in Message-Passing Systems

- The properties of communication links in this scheme:
  - Links are established *automatically*
  - A link is associated with *exactly two processes*.
  - There exists *exactly one link* between each pair of processes.

3.6 IPC in Message-Passing Systems

- With ***indirect*** communication,
  - the messages are *sent to* and *received from* ***mailboxes***, or ***ports***.
  - A ***mailbox*** (also referred to as ***ports***)
    - can be viewed abstractly as an object
    - *into* which messages can be *placed* by processes, and
    - *from* which messages can be *removed*.
  - The primitives of this scheme:
    - send(*A*, message) – send a message to mailbox *A*.
    - receive(*A*, message) – receive a message from mailbox *A*.

3.6 IPC in Message-Passing Systems

- The properties of communication links in this scheme:
  - Links are 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*.
  - A number of *different links may exist,* between each pair of processes
    - with each link corresponding to one mailbox.

3.6 IPC in Message-Passing Systems

- OS provides a mechanism that allows a process to do:
  - *Create* a new mailbox.
  - *Send* and *Receive* messages through the mailbox.
  - *Delete* a mailbox.

3.6 IPC in Message-Passing Systems

- Different design options for implementation:
  - *blocking* or *non-blocking*: ***synchronous*** or ***asynchronous***

  - ***Blocking send***: the sender is blocked until the message is received.
  - ***Non-blocking send***: the sender is sends the message and continue.
  - ***Blocking receive***: the receiver blocks until a message is available.
  - ***Non-blocking receive***: the receiver retrieves either a valid message or a null message.

3.7 Examples of IPC Systems

- Examples of IPC Systems
  - Shared Memory: ***POSIX Shared Memory***
    - POSIX: Portable Operating System Interface (for uniX)
  - Message Passing: ***Pipes***
    - One of the earliest IPC mechanisms on UNIX systems.

3.7 Examples of IPC Systems

- POSIX shared memory
  - is organized using memory-mapped files,
    - which associate the region of shared memory with a file
  - First, *create* a shared-memory object:
    - `fd = shm_open(name, O_CREAT | ORDWR, 0666);`
  - Configure the *size* of the object in bytes:
    - `ftruncate(fd, 4096);`
  - Finally, establish a *memory-mapped file*:
    - `mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);`

3.7 Examples of IPC Systems


```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main()
{
    const int SIZE = 4096;      // the size of shared memory
    const char *name = "OS";    // the name of shared memory
    const char *message_0 = "Hello, ";
    const char *message_1 = "Shared Memory!\n";

    int shm_fd;     // the file descriptor of shared memory
    char *ptr;      // pointer to shared memory
```

3.7 Examples of IPC Systems

```
    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory */
    ftruncate(shm_fd, SIZE);

    /* map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory */
    sprintf(ptr, "%s", message_0);
    ptr += strlen(message_0);
    sprintf(ptr, "%s", message_1);
    ptr += strlen(message_1);

    return 0;
}
```

```
$gcc 3.16_shm_producer.c -lrt
```

Figure 3.16 *Producer process illustrating POSIX shared-memory API.*

3.7 Examples of IPC Systems

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main()
{
    const int SIZE = 4096;      // the size of shared memory
    const char *name = "OS";    // the name of shared memory

    int shm_fd;      // the file descriptor of shared memory
    char *ptr;       // pointer to shared memory
```

3.7 Examples of IPC Systems

```
    /* create the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s", (char *)ptr);

    /* remove the shared memory */
    shm_unlink(name);

    return 0;
}
```

```
$gcc 3.16_shm_consumer.c -lrt
```

Figure 3.17 *Consumer process illustrating POSIX shared-memory API.*

3.7 Examples of IPC Systems

- ***Pipes*** were
  - one of the first IPC mechanisms in early UNIX systems.
  - A ***pipe*** acts as a *conduit* allowing two processes to communicate.

- Four issues of pipe implementation:
  - Does the pipe allow *unidirectional* or *bidirectional* communication?
  - In the case of two-way comm., is it *half-duplex* or *full-duplex*?
  - Must a *relationship* exist between the communicating process?
    - such as parent-child
  - Can the pipes communicate *over a network*?

3.7 Examples of IPC Systems

- Two common types of pipes:
  - ***Ordinary pipes***:
    - *cannot be accessed from outside* the process that created it.
    - Typically, a *parent* process creates a pipe and uses it to communicate with a *child* process that it created.
  - ***Named pipes***:
    - can be accessed *without* a *parent-child* relationship

3.7 Examples of IPC Systems

- Ordinary Pipes
  - allow two processes to communicate in producer-consumer fashion.
    - the producer writes to one end of the pipe (*write end*)
    - the consumer reads from the other end (*read end*)
  - *unidirectional*: only *one-way* communication is possible.
  - *two-way* communication? use two pipes!

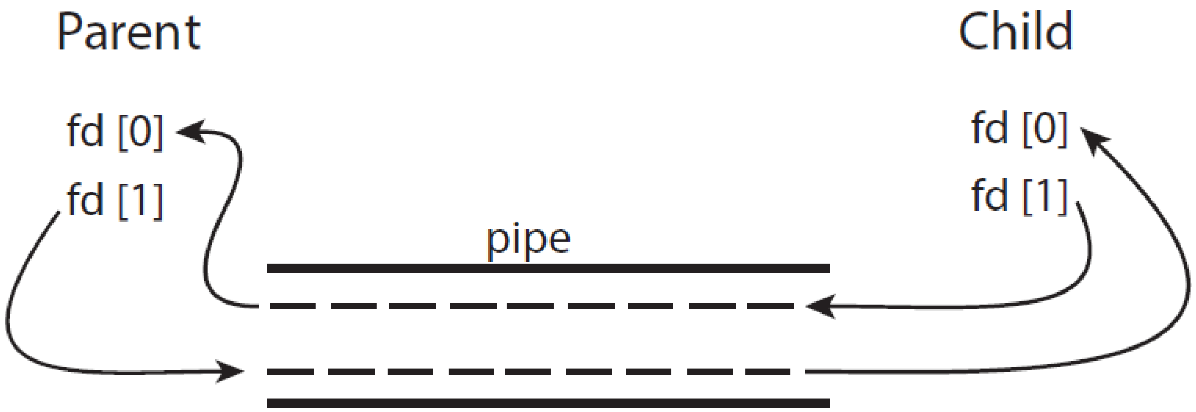


Figure 3.20 *File descriptors for an ordinary pipe.*

3.7 Examples of IPC Systems

- On UNIX systems,
  - ordinary pipes are constructed using the function:
    - `pipe(int fd[])`
    - `fd[0]`: the read end of the pipe
    - `fd[1]`: the write end

3.7 Examples of IPC Systems

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

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

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

    /* create the pipe */
    pipe(fd);
```

```c
    pid = fork();  // fork a new process

    if (pid > 0) { // parent process
        close(fd[READ_END]);
        /* write to the pipe */
        write(fd[WRITE_END], write_msg, strlen(write_msg) + 1);
        close(fd[WRITE_END]);
    }
    else if (pid == 0) { // child process
        close(fd[WRITE_END]);
        /* read to the pipe */
        read(fd[READ_END], read_msg, BUFFER_SIZE);
        printf("read %s\n", read_msg);
        close(fd[READ_END]);
    }

    return 0;
}
```

Figure 3.21 *Ordinary pipe in UNIX.*

3.8 Communication in Client-Server Systems

- Two other strategies in client-server systems
  - ***Sockets***
    - are defined as endpoints for communication.
  - ***RPCs*** (Remote Procedure Calls)
    - abstracts procedure calls between processes on networked systems.

3.8 Communication in Client-Server Systems

- A socket is
  - identified by an *IP address* concatenated with a *port* number.

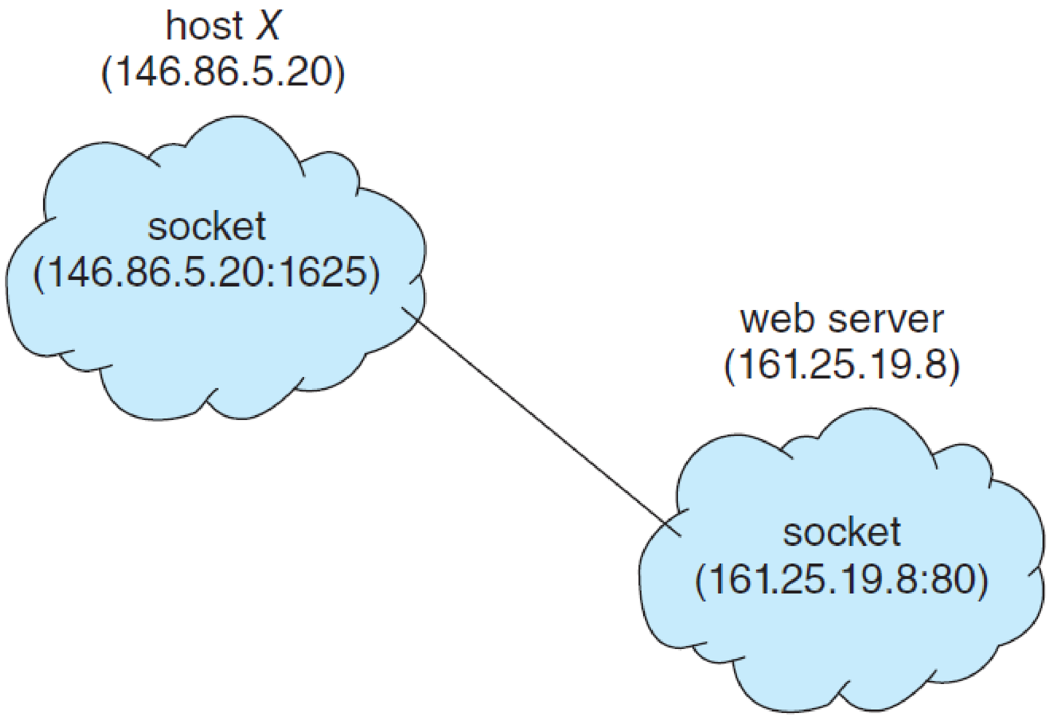


Figure 3.26 *Communication using sockets.*

3.8 Communication in Client-Server Systems

- Java provides
  - a much easier interface to sockets and
  - provides three different types of sockets
    - Socket class: ***connection-oriented*** (TCP)
    - DatagramSocket class: ***connectionless*** (UDP)
    - MulticastSocket class: multiple recipients

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

public class DateServer {

    public static void main(String[] args) throws Exception {
        ServerSocket server = new ServerSocket(6013);

        /* Now listen for connections */
        while (true) {
            Socket client = server.accept();
            PrintWriter pout = new PrintWriter(client.getOutputStream(), true);

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

            /* close the socket and resume listening for connections */
            client.close();
        }
    }
}
```

Figure 3.27 *Date server in Java.*

3.8 Communication in Client-Server Systems

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

public class DateClient {

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

        InputStream in = socket.getInputStream();
        BufferedReader br = new BufferedReader(new InputStreamReader(in));

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

        /* close the socket connections */
        socket.close();
    }
}
```

Figure 3.28 *Date client in Java.*

3.8 Communication in Client-Server Systems

- RPC (Remote Procedure Call)
  - one of the most common forms of *remote service*.
  - designed as a way to abstract the procedure-call mechanism
    - for use between systems with network connections.
  - A client invokes a procedure on a remote host
    - as it would invoke a procedure locally.

3.8 Communication in Client-Server Systems

- The RPC system
  - hides the details that allow communication to take place
    - by providing a ***stub*** on the client side.
  - The *stub* of client-side locates the server and
    - ***marshals*** the parameters.
  - The stub of server-side received this message,
    - unpacks the *marshalled* parameters, and
    - performs the procedure on the server.

Any Questions?