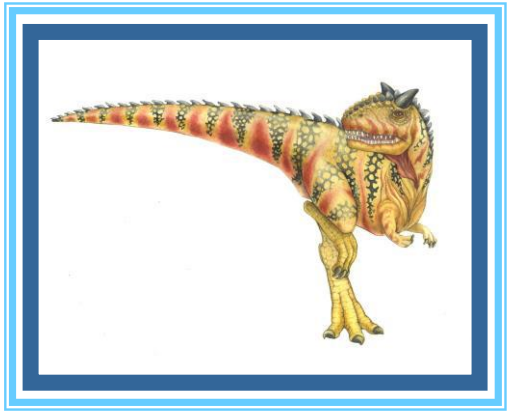
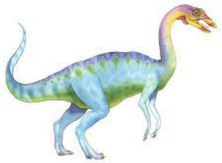
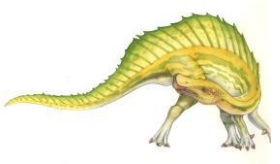
4: IPC

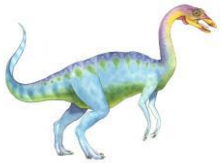




4: IPC

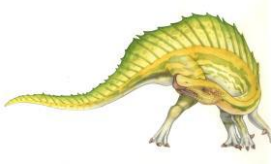
- Interprocess Communication(IPC)
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems

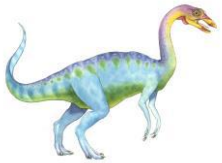




Objectives

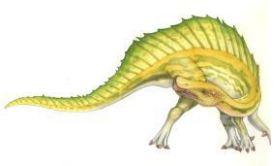
- Describe and contrast interprocess communication using shared memory and message passing.
- Design programs that uses pipes and POSIX shared memory to perform interprocess communication.

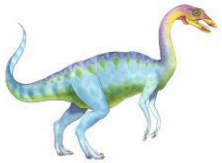




Motivations for Program Reuse

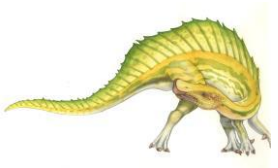
- What would you do if you need to go through a file containing 10,000 names to count the number of names containing the “sam”?

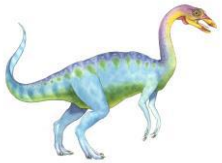




Interprocess Communication

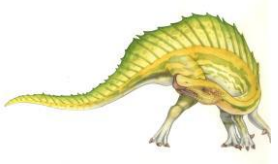
- Processes within a system may be ***independent*** or ***cooperating***
- ***Independent*** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by other processes, including sharing data
- Advantages/Reasons of process cooperation
  - Information sharing: to share the already processed data
  - Computation speed-up: to focus on its own task
  - Modularity: e.g., database in one process which can be reused
  - Convenience
- But:
  - Added complexity
  - Deadlocks possible
  - Starvation possible

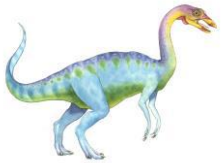




Communications Models

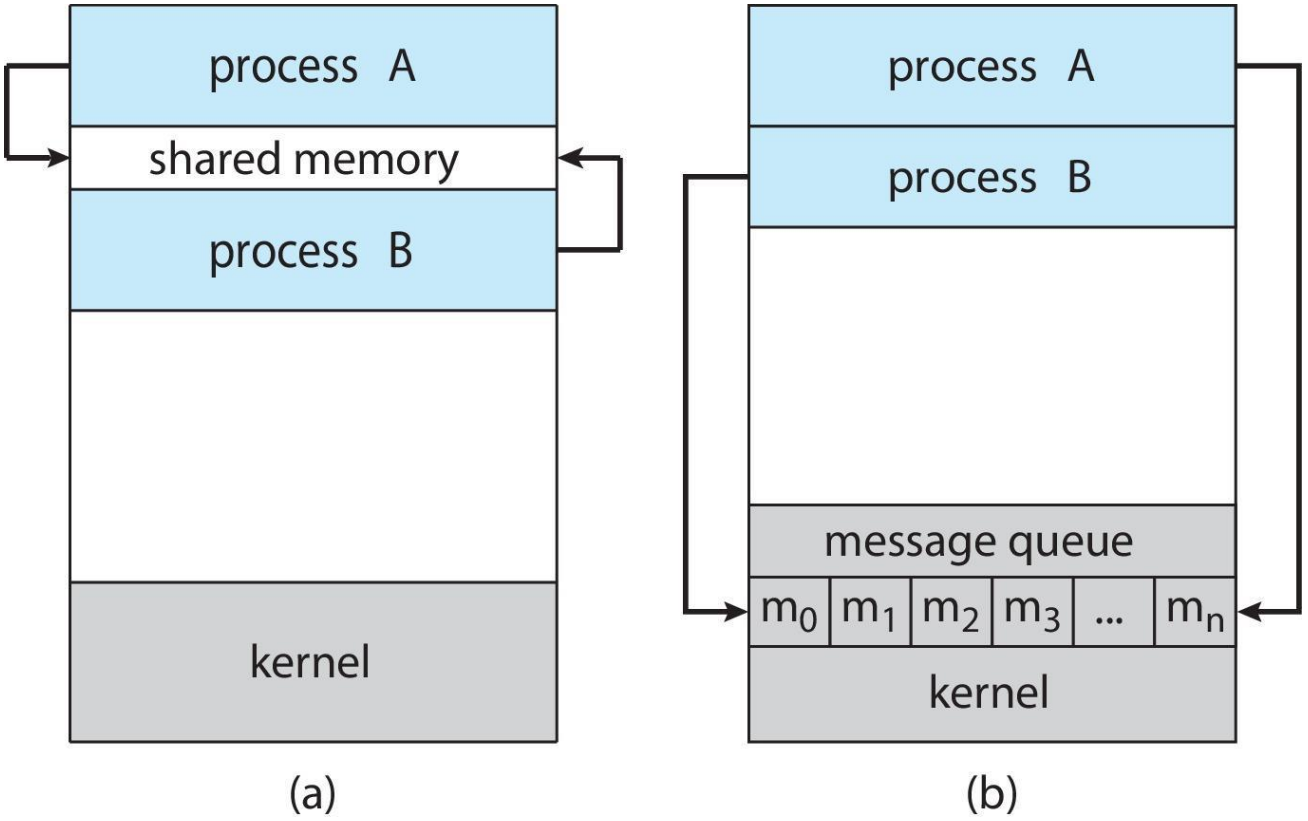
- Cooperating processes need **interprocess communication** (**IPC**)
- Interprocess communication:
  - Two (or more) different processes
  - Send and receive control information or data
  - Communicating through the **kernel** or **other shared resources**
  - Two sides of the interaction:
    - sender/receiver, client/server, caller/callee
  - The content: "**message**"
    - The *message* **size** is either **fixed or variable**





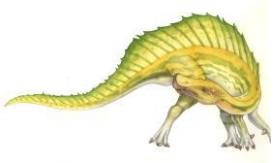
Communications Models

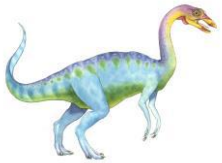
- Two models of IPC
  - **Shared memory**
  - **Message passing**



(a) Shared memory.

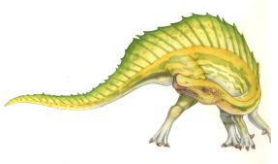
(b) Message passing.

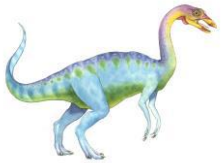




Interprocess Communication – Shared Memory

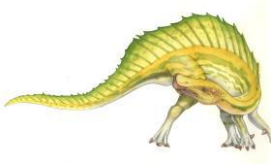
- The system kernel maps an area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to **synchronize their actions** when they access shared memory.
  - Sender doesn't delete unread data
  - Receiver doesn't read other data
- **Synchronization** is discussed in great details later

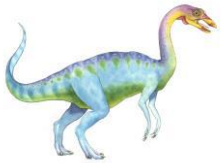




Producer-Consumer Problem

- 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
    - Producer never has to wait because there is always extra space available for new information; only consumer might have to wait if no information is available to read
  - **bounded-buffer** assumes that there is a fixed buffer size
    - Producer might have to wait if there is no space available to store new information; consumer might have to wait if no information is available to read

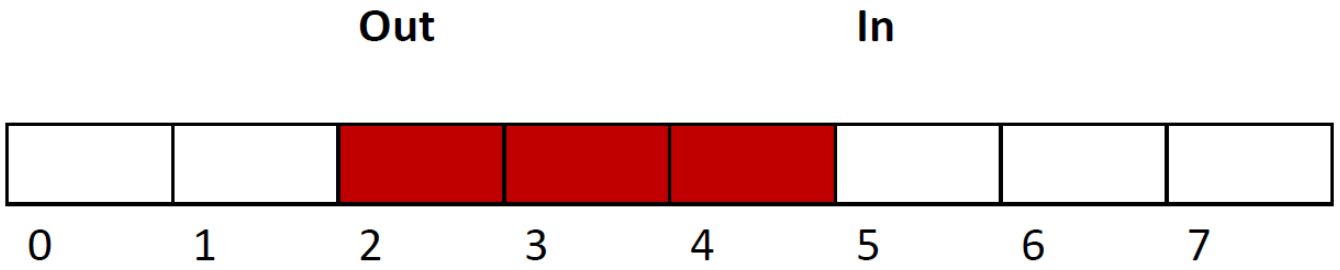


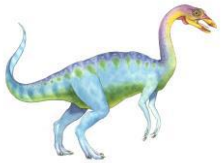


Bounded-Buffer – Shared-Memory Solution

- Shared data

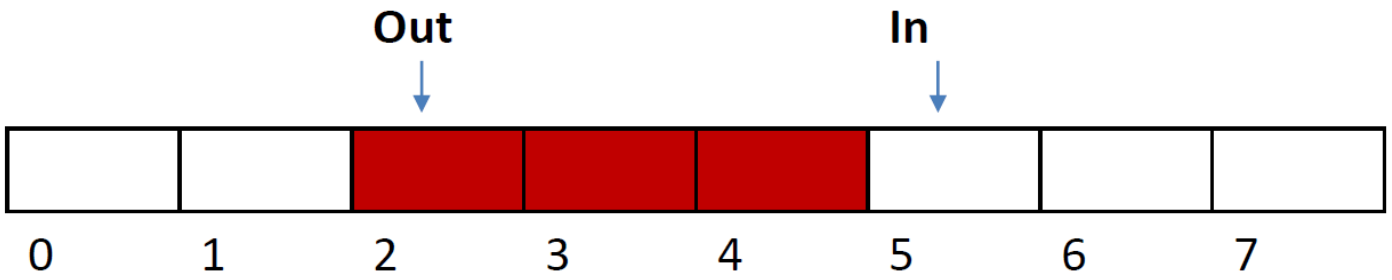
```
#define BUFFER_SIZE 8
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE]; // shared data
int in = 0; // sharing status: produced
int out = 0; //sharing status: consumed
```

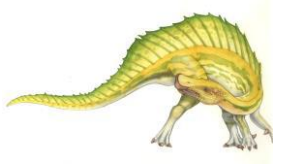




Producer Process – Shared Memory

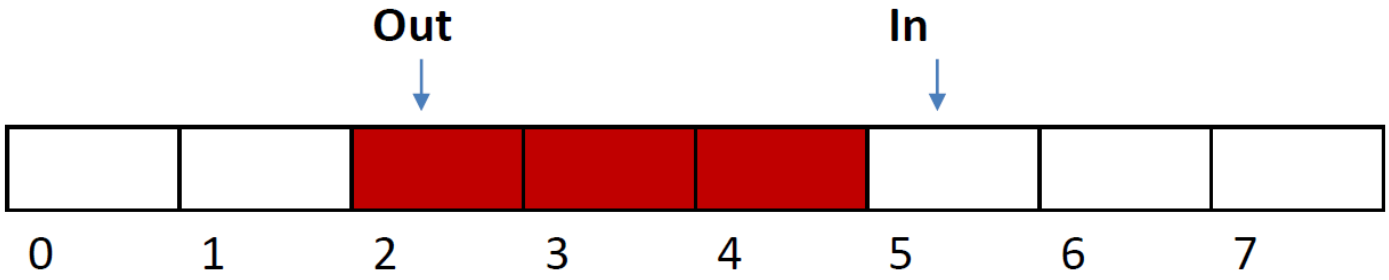
```
item next_produced;

while (true) {
   /* produce an item */
   while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing (busy wait) no free buffer */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
}
```

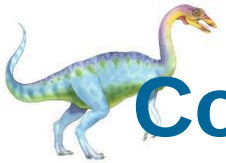




Consumer Process – Shared Memory

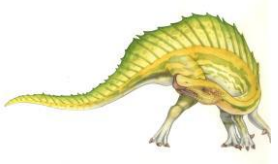
```
item next_consumed;

while (true) {
        while (in == out)
                ; /* do nothing, new item to consume */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item from the buffer */


}
```

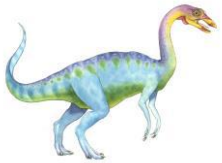




Consumer Process – Shared Memory

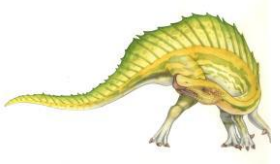
- Solution is correct, but can **only use `BUFFER_SIZE-1` elements**
- Problems
  - Polling can be wasteful
  - Fixed checking time (ready or not), longer delay time

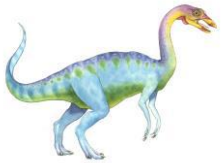




Message Passing

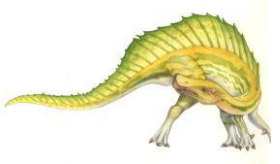
- Mechanism for processes to communicate and to synchronize their actions
- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via **send()/receive()**
- Implementation issues:
  - How are **links established**?
  - 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**?

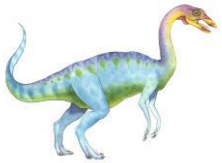




Message Passing

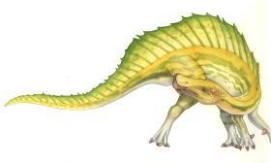
- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - **Direct** (process to process) or **indirect** (mail box)
    - **Synchronous** (blocking) or **asynchronous** (non-blocking)
    - Automatic or explicit buffering

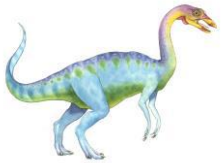




Direct Communication

- 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**
  - Links are 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**



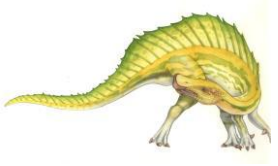


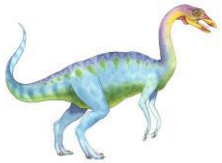
Indirect Communication

- 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
  - Link established only if processes **share a common mailbox**
  - A link may be associated with **many processes**
  - Each pair of processes may share **several communication links**
  - Link may be **unidirectional or bi-directional**
- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- **Primitives** are defined as:

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

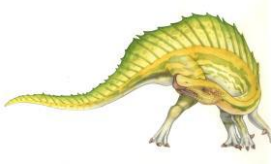
  `receive`(*A, message*) – receive a message from mailbox A

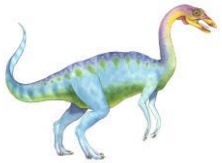




Indirect Communication

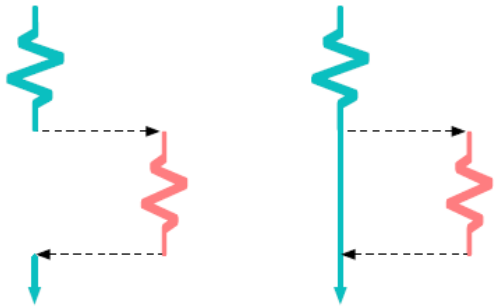
- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - **Who gets the message**?
- Possible Solutions
  - Allow a link to be associated with **at most two processes**
  - Allow only **one process** at a time to execute a **receive** operation
  - Allow the system to **select arbitrarily the receiver**. Sender is notified who the receiver was.

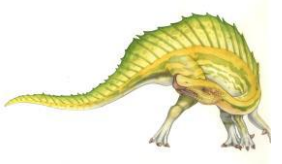


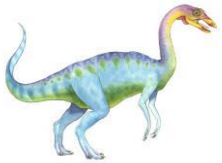


Synchronization

- Message passing may be either **blocking or non-blocking**
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continues without waiting for the message to be received
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

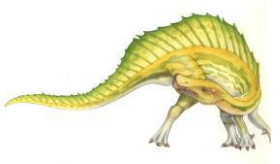


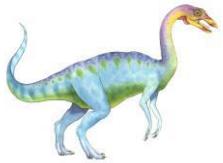




Buffering

- Queue of messages attached to the link, in **kernel memory**
- Implemented in one of 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 full
  3. Unbounded capacity – infinite length
     Sender never waits

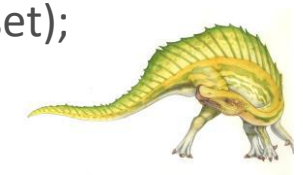


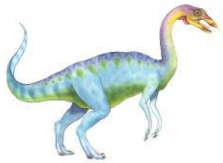


Examples of IPC Systems - POSIX

- POSIX Shared Memory -- Producer
  - Producer process first **creates shared memory** segment
    ```
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    ```
  - Also used to open an existing segment to share it
  - **Set the size of the file object**
    ```
    ftruncate(shm_fd, 4096);
    ```
    int ftruncate(int fd, off_t length)
  - **map the shared memory** segment in the address space of the process
    ```
    ptr = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
                shm_fd, 0);
    ```
  - Now the process could write to the shared memory
    ```
    sprintf(ptr, "%s", "Writing to shared memory");
    ```

void* mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);





Examples of IPC Systems - POSIX

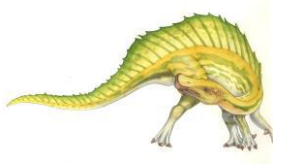
- POSIX Shared Memory - Consumer
  - consumer process opens shared memory object name

    shm_fd = shm_open (name, O_RDONLY, 0666);
    - **Returns file descriptor (int) which identifies the file**
  - map the shared memory object in the address space of the process
    - **ptr = mmap(0,SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);**
  - Now the process can read from the shared memory object

    printf("%s", (char *)ptr);
  - remove the shared memory object once finished

    shm_unlink(name);

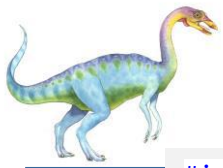
```
int shm_open( const char *name, int oflag, mode_t mode);
void* mmap(void* start,size_t length,int prot,int flags,int fd,off_t offset);
int shm_unlink( const char *name);
```



IPC POSIX Producer

shm_producer.c
compiling command:
gcc –o producer shm_producer.c -lrt

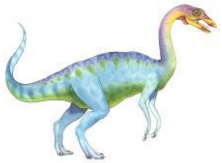
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>
int main() {
        const int SIZE = 4096;          //the size of shared memory object
        const char *name = "OS";        //name of the shared memory object
        const char *message0= "Studying Operating Systems ";//string written to shared memory
        const char *message1= "Is Fun!\n";

        int shm_fd;          //shared memory file descriptor
        void *ptr;           //pointer to shared memory object

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

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

        /* now map the shared memory segment in the address space of the process */
        ptr = mmap(0,SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
        if (ptr == MAP_FAILED) {
              printf("Map failed\n");
              return -1;
        }
        /* Now write to the shared memory region.
         * Note we must increment the value of ptr after each write. */
        sprintf(ptr,"%s",message0);
        ptr += strlen(message0);
        sprintf(ptr,"%s",message1);
        ptr += strlen(message1);
        return 0;
}
```



IPC POSIX Consumer

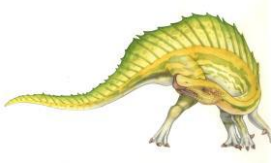
shm_consumer.c
compiling command:
gcc –o consumer shm_consumer.c -lrt

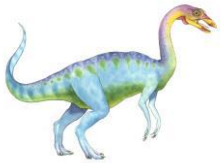
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
int main() {
        const char *name = "OS";          //name of the shared memory object
        const int SIZE = 4096;            //size of shared memory object
        int shm_fd;                       //shared memory file descriptor
        void *ptr;                        //pointer to shared memory object

        /* open the shared memory segment */
        shm_fd = shm_open(name, O_RDONLY, 0666);
        if (shm_fd == -1) {
                printf("shared memory failed\n");
                exit(-1);
        }

        /* now map the shared memory segment in the address space of the process */
        ptr = mmap(0,SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
        if (ptr == MAP_FAILED) {
                printf("Map failed\n");
                exit(-1);
        }

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

        /* remove the shared memory segment */
        if (shm_unlink(name) == -1) {
                printf("Error removing %s\n", name);
                exit(-1);
        }
        return 0;
}
```



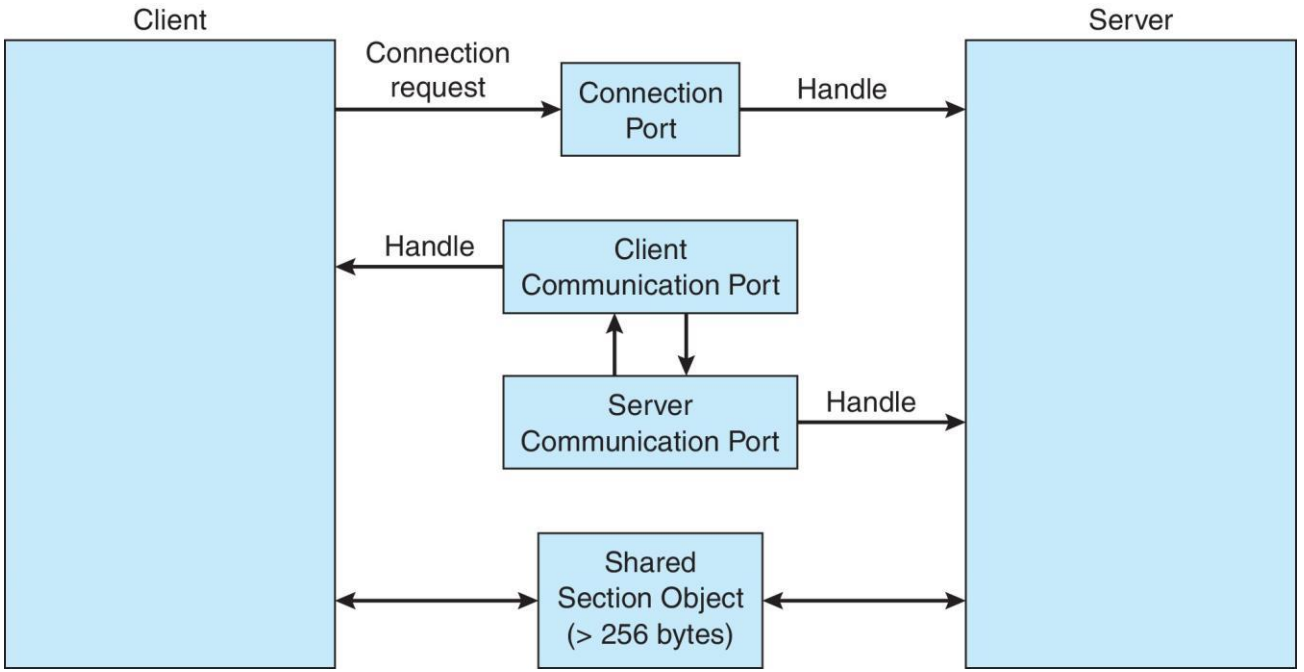
Examples of IPC Systems – Windows

- Message-passing centric via **advanced local procedure call (ALPC)** facility
  - Only works between processes on the same system
  - Uses **ports** (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - The client **opens** a handle to the subsystem's **connection port** object.
    - The client sends a connection request.
    - The server creates two private **communication ports** and returns the handle to one of them to the client.
    - The client and server **use the corresponding port handle to send messages** or callbacks and to listen for replies.



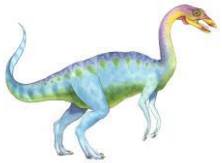


ALPC in Windows



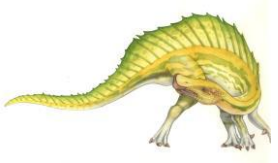
1. For small messages (up to 256 bytes), the port's message queue is used as intermediate storage, and the messages are copied from one process to the other.
2. Larger messages must be passed through a **section object**, which is a region of shared memory associated with the channel.
3. When the amount of data is too large to fit into a section object, an API is available that allows server processes to read and write directly into the address space of a client.

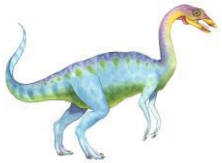
The ALPC facility in Windows is not part of the Windows API and hence is not visible to the application programmer.



Pipes

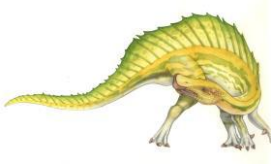
- Acts as a conduit allowing two processes to communicate on the same computer
- Issues:
  1. Is communication unidirectional or bidirectional?
  2. In two-way communication, is it half or full-duplex(data can travel in both directions at the same time)?
  3. Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  4. Can the pipes be used over a network?
- **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.

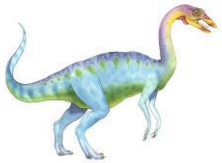




Pipes

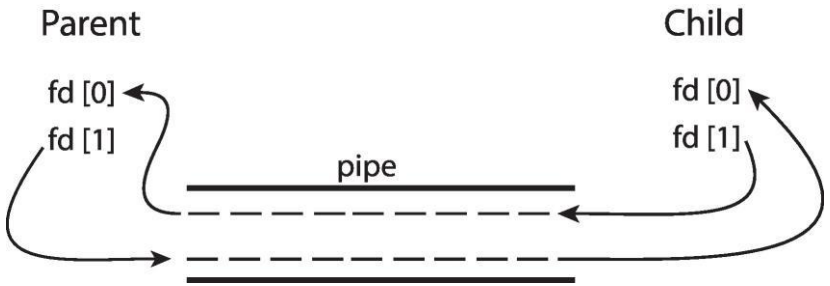
- A stream of communication between two processes
- You can think of it as a virtual file stream shared between two processes
  - A process can read and/or write to a pipe
  - The pipe function gets two descriptors (integer labels)
  - Read descriptor – read from the pipe
  - Write descriptor – write to the pipe
  - Both processes must know the descriptors
  - read and write are used with the pipe



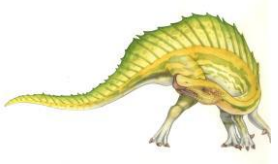


Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer **writes to one end** (the **write-end** of the pipe)
- Consumer **reads from the other end** (the **read-end** of the pipe)
- Ordinary pipes are therefore **unidirectional**
  - create two separate pipes if bidirectional communication is necessary
- Require **parent-child relationship** between communicating processes

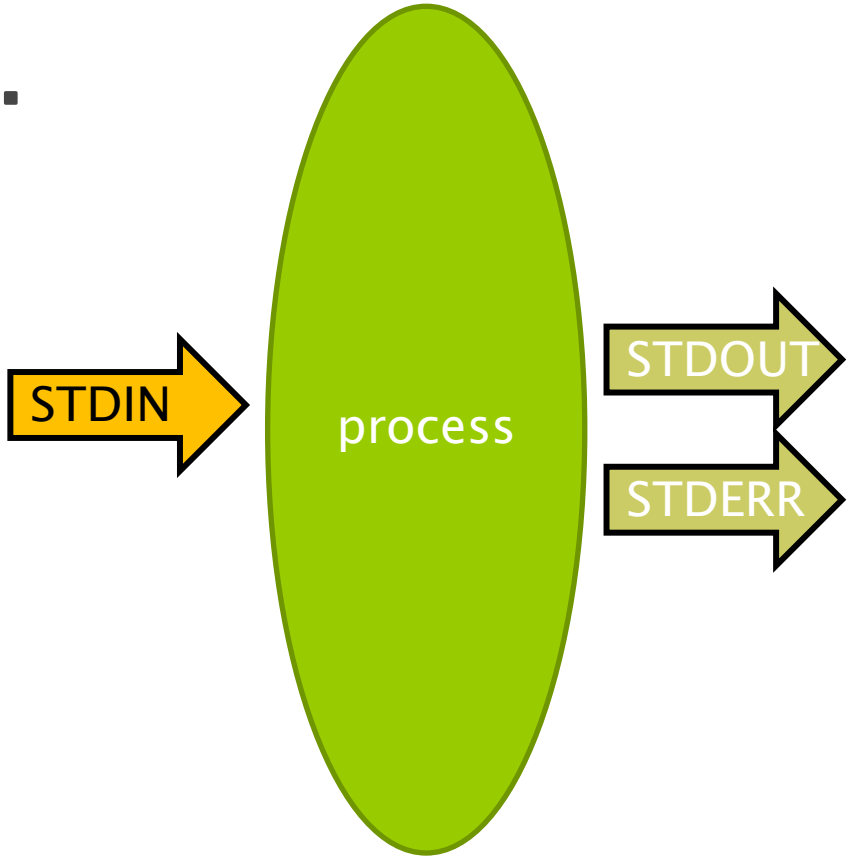


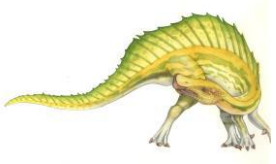
- Windows calls these **anonymous pipes**



How does the “grep … | wc” work?

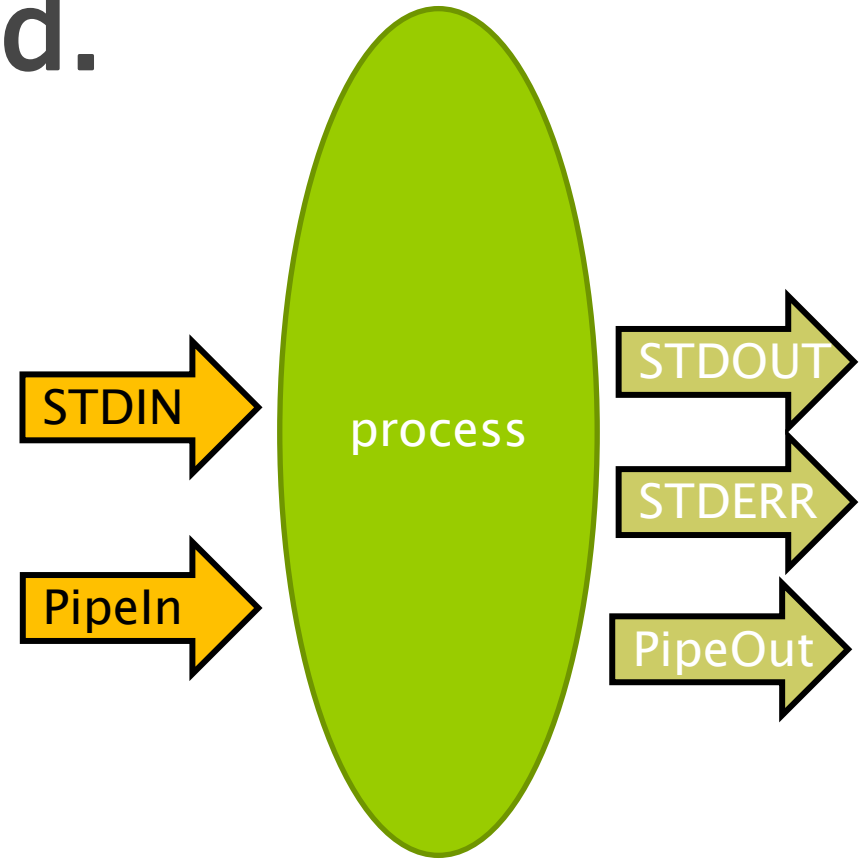
Every process has at least 3 Files…

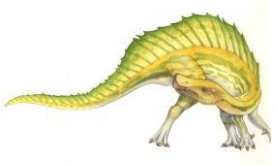


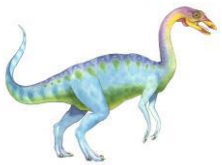


How does the “grep … | wc” work?

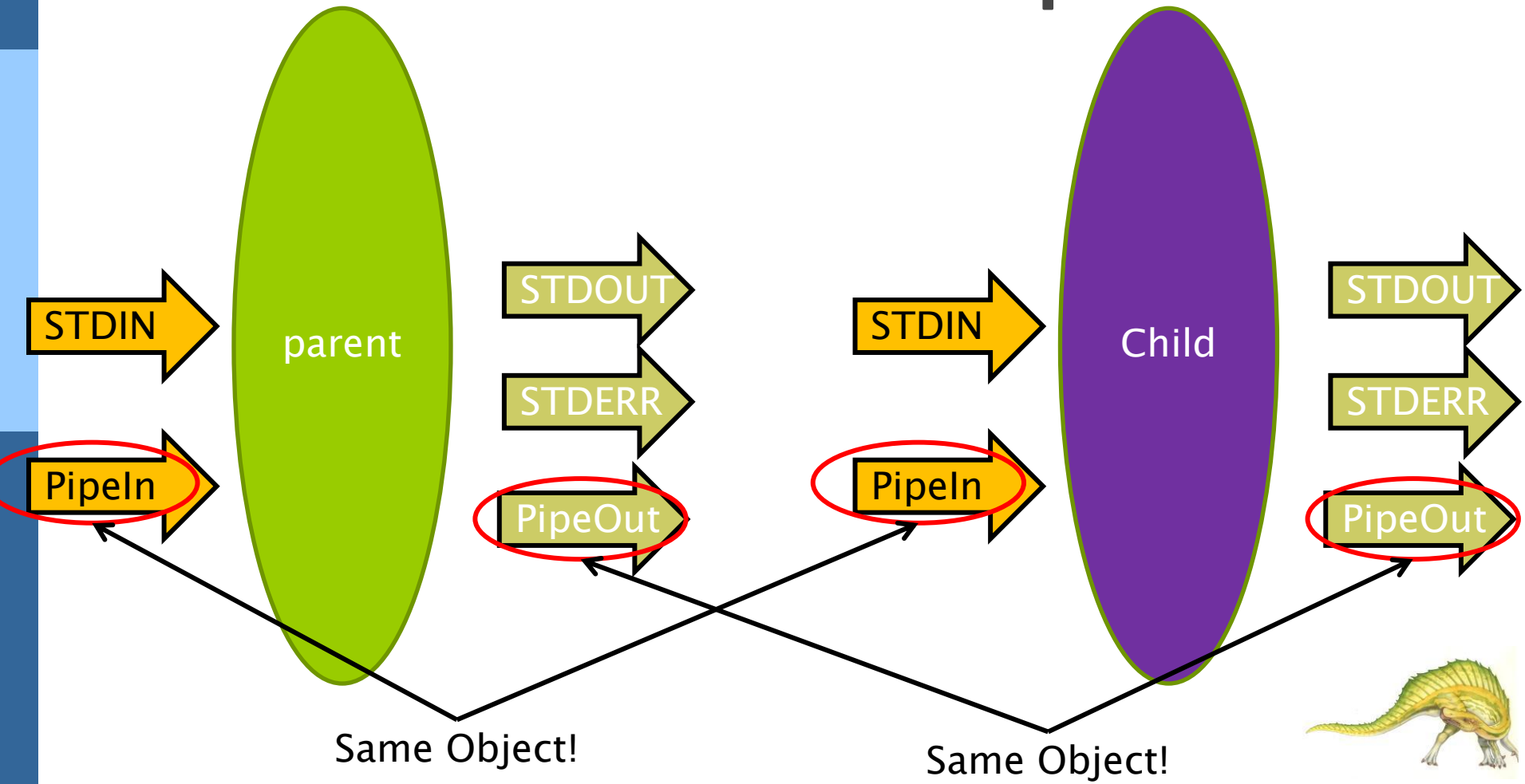
If the process creates a pipe, it would have 2 more files opened.

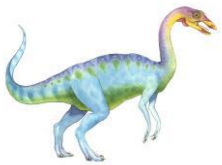




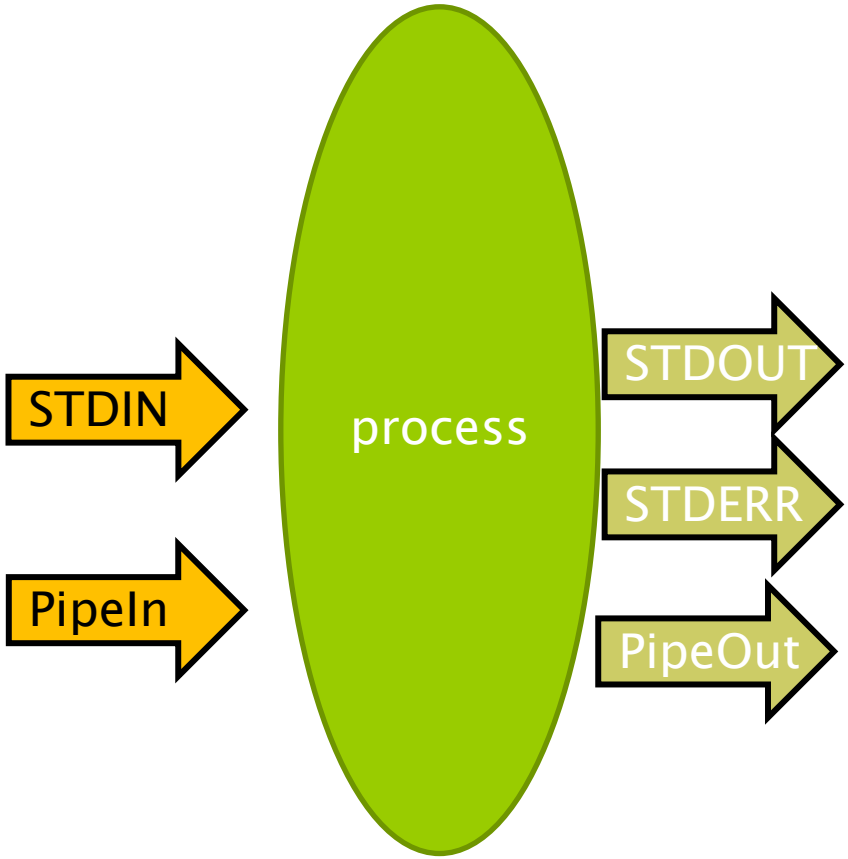


Spawn a child process, it would have the same files opened.

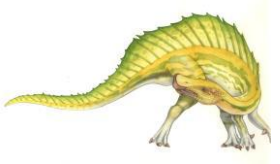


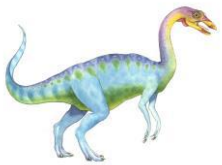


File redirection

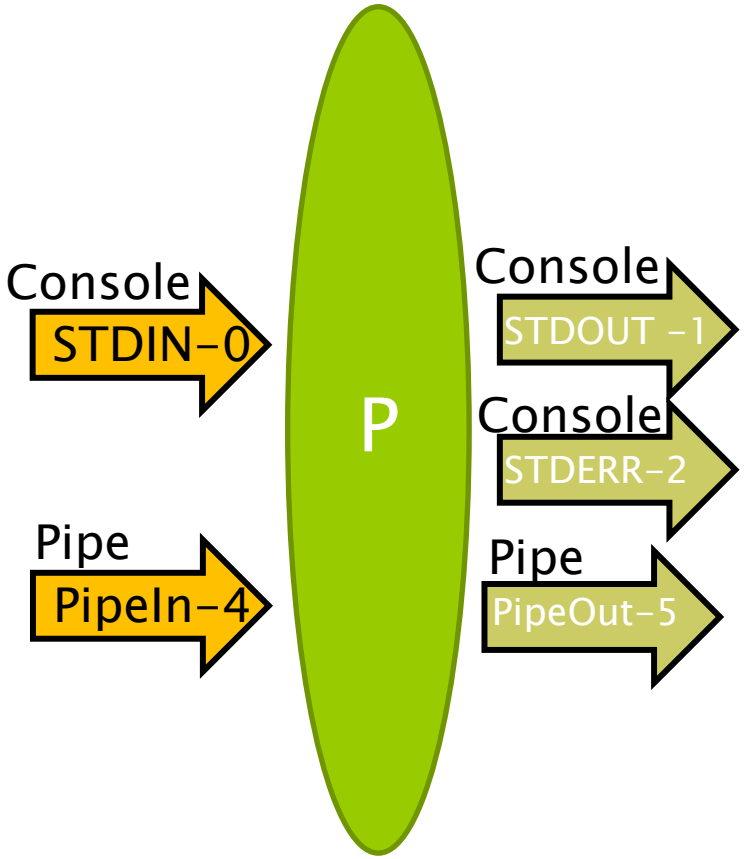


- File Descriptors
  - Just a Number
- In Linux/Unix,
  - 0 for stdin
  - 1 for stdout
  - 2 for stderr
- Any number for the PipeIn or PipeOut

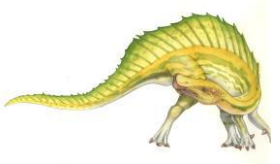


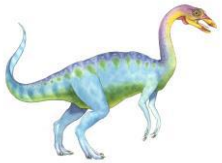


File redirection



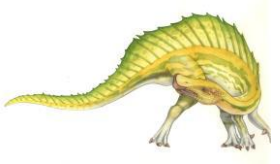
- Aim: get 1 to be the same object as pipeout.
- Use `dup` system call.

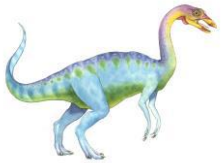




Unix File Descriptors

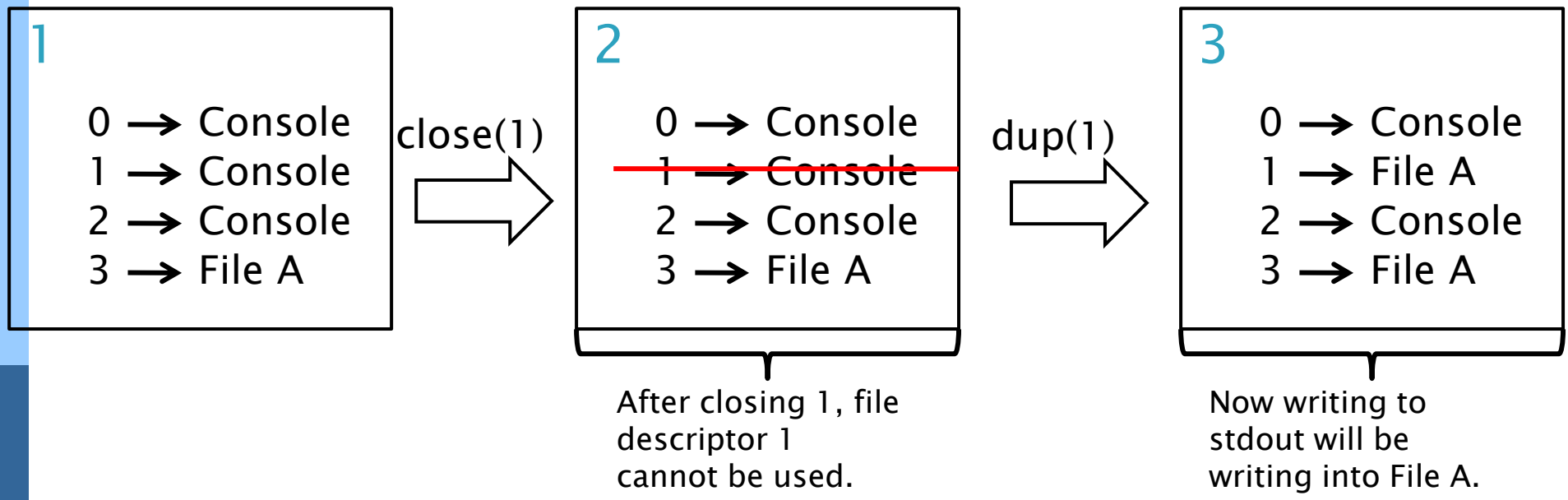
- Unix/Linux files are numbered
  - 0, 1, 2 … etc
  - 0 is stdin, 1 is stdout, 2 is stderr
  - By default,
    - scanf and std::cin gets input from stdin.
    - printf and std::cout will print to stdout.
    - The default “file” opened for stdin, stdout and stderr will be the console. (where you can observe the printing)
  - However, using redirection, we can actually get 0, 1 or 2 to be “files” other than the console.

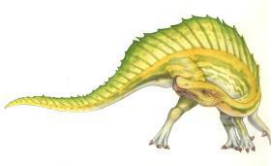


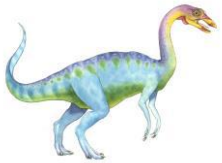


Using dup

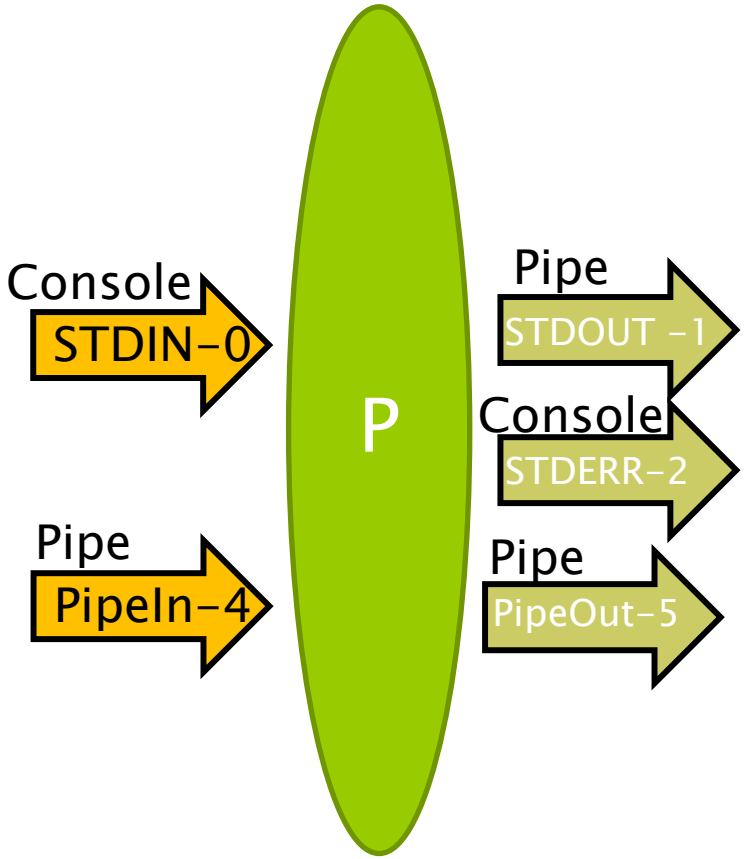
- **close()** system call closes a file. (Renders the file descriptor invalid)
- The **dup()** system call duplicates the given file descriptor using the lowest available file descriptor number.

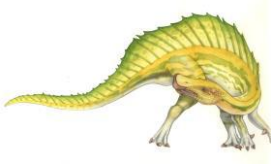


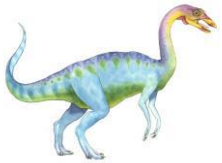




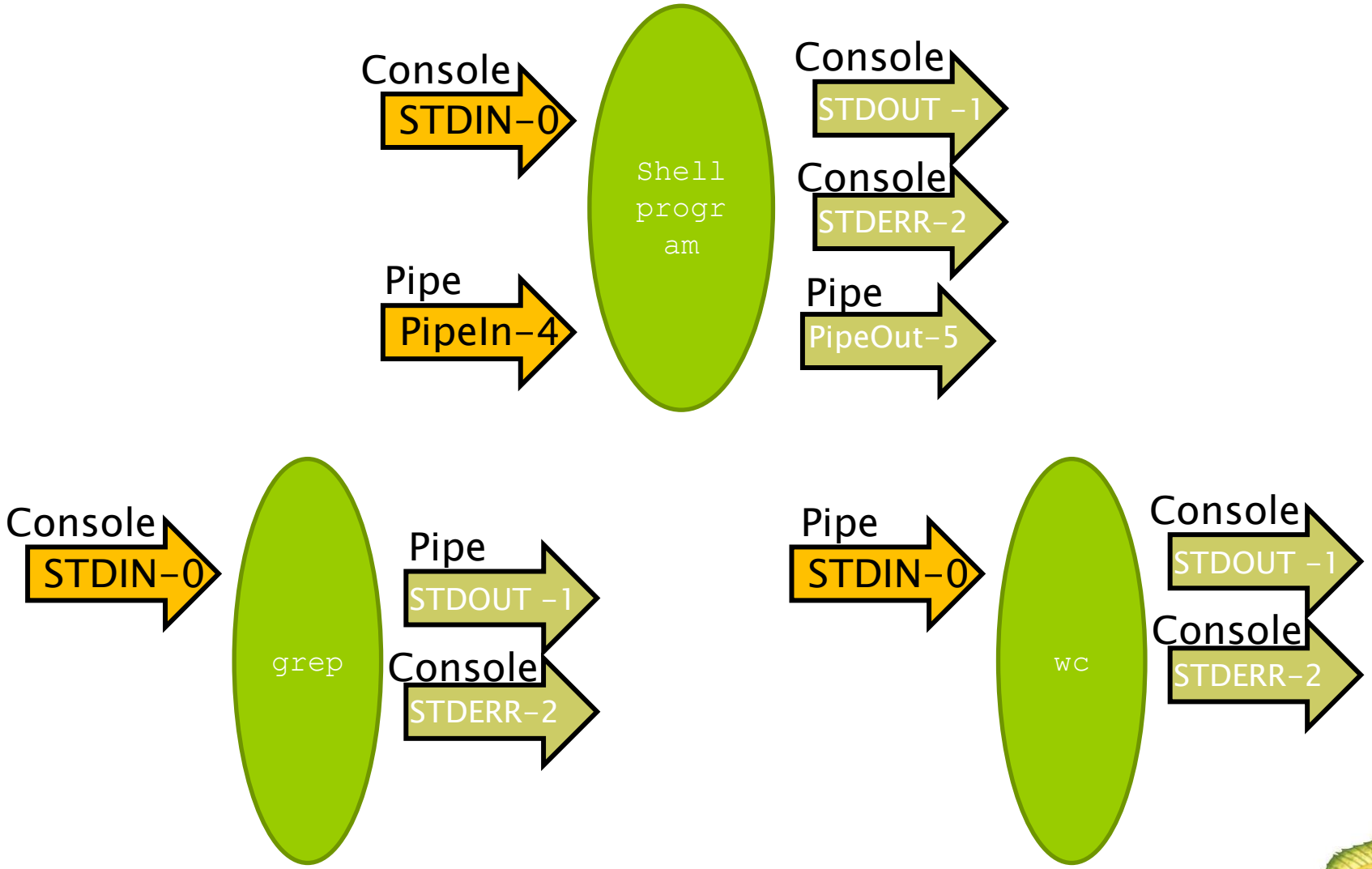
What does dup do?

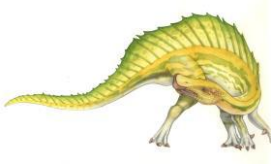


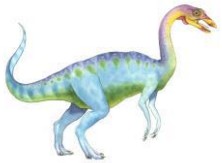




“grep abc | wc”


Console
STDIN-0
Pipe
PipeIn-4
Shell
progr
am
Console
STDOUT -1
Console
STDERR-2
Pipe
PipeOut-5
Console
STDIN-0
grep
Pipe
STDOUT -1
Console
STDERR-2
Pipe
STDIN-0
wc
Console
STDOUT -1
Console
STDERR-2

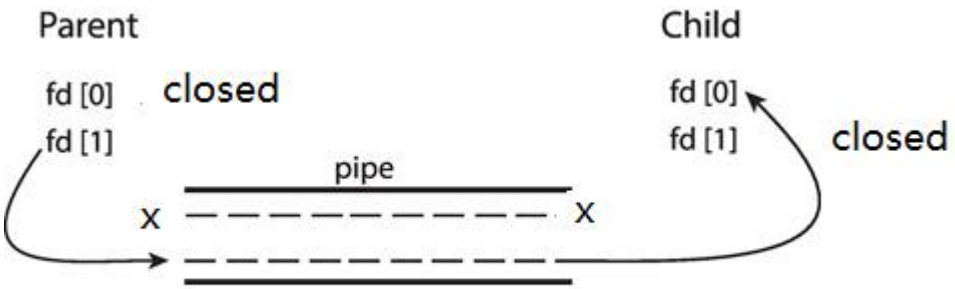




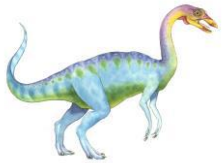
Ordinary Pipes

Ordinary pipe in UNIX pipe.c

gcc –o pipe pipe.c



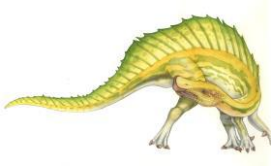

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

#define BUFFER_SIZE 80
#define READ_END 0
#define WRITE_END 1

int main(void)
{
        char write_msg[BUFFER_SIZE] = "Greetings
        from parent process";
        char read_msg[BUFFER_SIZE];
        int fd[2];
        pid_t pid;
        int status;
        /* create the pipe */
        if (pipe(fd) == -1) {
                fprintf(stderr,"Pipe failed");
                return 1;
        }
        /* fork a child process */
        pid = fork();
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                return 1;
        }
        if (pid > 0) { /* parent process */
                /* close the unused end of the pipe */
                close(fd[READ_END]); //optional
                /* write to the pipe */
                write(fd[WRITE_END], write_msg,
                strlen(write_msg)+1);
                /* close the write end of the pipe */
                close(fd[WRITE_END]);
                pid = wait(&status);
                printf("Parent: child(%d) exit(%d)\n",pid,
                        status);
        }
        else { /* child process */
                /* close the unused end of the pipe */
                close(fd[WRITE_END]); //optional
                /* read from the pipe */
                read(fd[READ_END], read_msg, BUFFER_SIZE);
                printf("Child Read: %s",read_msg);
                /* close the read end of the pipe */
                close(fd[READ_END]);
        }
    return 0;
}
```

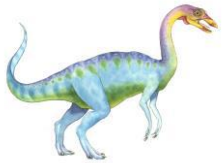


Ordinary Pipes

Ordinary pipe in Windows win_pipe_parent.c

```
//Windows anonymous pipe -- parent process
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#define BUFFER_SIZE 25
int main(VOID) {
        HANDLE ReadHandle, WriteHandle;
        STARTUPINFO si;
        PROCESS_INFORMATION pi;
        char message[BUFFER_SIZE] = "Greetings";
        DWORD written;

        /* set up security attributes allowing pipes to be inherited */
        SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),NULL,TRUE};

        /* allocate memory */
        ZeroMemory(&pi, sizeof(pi));

        /* create the pipe */
        if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
                fprintf(stderr, "Create Pipe Failed");
                return 1;
        }

        /* establish the START_INFO structure for the child process */
        GetStartupInfo(&si);
        si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

        /* redirect standard input to the read end of the pipe */
        si.hStdInput = ReadHandle;
        si.dwFlags = STARTF_USESTDHANDLES;
```

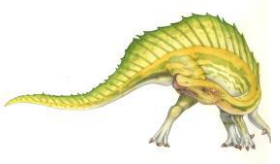


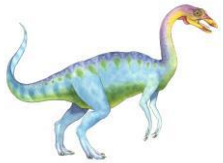


Ordinary Pipes

Ordinary pipe in Windows cont. win_pipe_parent.c

```
        /* don't allow the child to inherit the write end of pipe */
        SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);

        /* create the child process */
        CreateProcess(NULL, "child.exe", NULL, NULL,
        TRUE, /* inherit handles */
        0, NULL, NULL, &si, &pi);

        /* close the unused end of the pipe */
        CloseHandle(ReadHandle);

        /* the parent writes to the pipe */
        if (!WriteFile(WriteHandle, message,BUFFER_SIZE,&written,NULL))
                fprintf(stderr, "Error writing to pipe.");

        /* close the write end of the pipe */
        CloseHandle(WriteHandle);

        /* wait for the child to exit */
        WaitForSingleObject(pi.hProcess, INFINITE);
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
        //Sleep(10000);
        return 0;
}
```

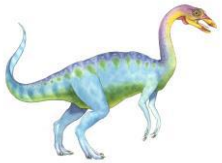




Ordinary Pipes

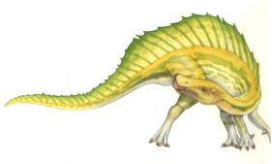
Ordinary pipe in Windows cont. win_pipe_child.c

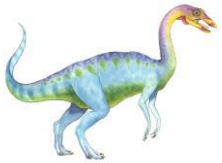
```
//Windows anonymous pipe -- child process
//executable filename: child.exe

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

#define BUFFER_SIZE 25

int main(VOID) {
        HANDLE ReadHandle;
        CHAR buffer[BUFFER_SIZE];
        DWORD read;

        /* get the read handle of the pipe */
        ReadHandle = GetStdHandle(STD_INPUT_HANDLE);

        /* the child reads from the pipe */
        if (ReadFile(ReadHandle, buffer, BUFFER_SIZE, &read, NULL))
                printf("child read message[%s]",buffer);
        else
                fprintf(stderr, "Error reading from pipe");
        return 0;
}
```



Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication can be **bidirectional**
- **No parent-child relationship** is necessary between the communicating processes
- **Several (>=2) processes** can use the named pipe for communication
- Provided on both UNIX and Windows systems





Named Pipes Example1

writer.c

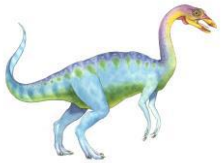
gcc –o writer writer.c
./writer &

```
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    int fd;
    char *myfifo = "/tmp/myfifo";
    char msg[]= "Hi,this is named pipe";

    /* create the FIFO (named pipe) */
    mkfifo(myfifo, 0666);

    /* write msg to the FIFO */
    fd = open(myfifo, O_WRONLY);
    write(fd, msg, sizeof(msg));

    close(fd);

    /* remove the FIFO */
    unlink(myfifo);
    return 0;
}
```

reader.c

gcc –o reader reader.c
./reader

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>

#define MAX_BUF 1024
int main() {
    int fd;
    char *myfifo = "/tmp/myfifo";
    char buf[MAX_BUF];

    // open, read, & display the msg from the FIFO
    fd = open(myfifo, O_RDONLY);
    read(fd, buf, MAX_BUF);
    printf("Received: %s\n", buf);

    close(fd);

    return 0;
}
```



Named Pipes Example2

fifo_server.c

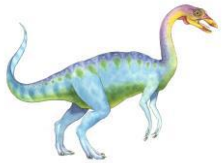
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>

#define FIFO_FILE       "MYFIFO2"

int main(void) {
        FILE *fp;
        char readbuf[80];

        /* Create the FIFO if it does not exist */

        umask(0);  // set file mode creation mask
        mknod(FIFO_FILE, S_IFIFO | 0666, 0);
                        //create a FIFO-special file
        while (1) {
                fp = fopen(FIFO_FILE, "r");
                fgets(readbuf, 80, fp);
                printf("Received string: %s\n", readbuf);
                fclose(fp);
                if (strncmp(readbuf, "exit", 4) == 0)
                    break;
        }

        return(0);
}
```

fifo_client.c

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

#define FIFO_FILE       "MYFIFO2"

int main(int argc, char *argv[]) {
        FILE *fp;

        if (argc != 2) {
                printf("USAGE: fifo_client [string]\n");
                exit(1);
        }

        if ((fp = fopen(FIFO_FILE, "w")) == NULL) {
                perror("fopen");
                exit(1);
        }

        fputs(argv[1], fp);

        fclose(fp);
        return(0);
}
```

int mknod(const char *_pathname_, mode_t _mode_, dev_t _dev_);

The system call **mknod**() creates a filesystem node (file, device special file or named pipe) named *pathname*, with attributes specified by *mode* and *dev*.



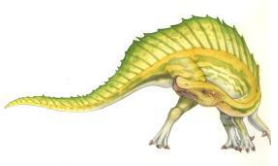
Named Pipes

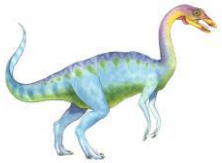
mode_t umask(mode_t *mask*);

umask is used to set permission of the file or directory

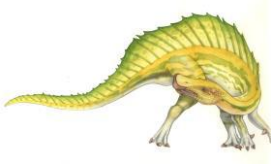
Octal digit in **umask** command	Permissions the mask will prohibit from being set during file creation
0	any permission may be set (read, write, execute)
1	setting of execute permission is prohibited (read and write)
2	setting of write permission is prohibited (read and execute)
3	setting of write and execute permission is prohibited (read only)
4	setting of read permission is prohibited (write and execute)
5	setting of read and execute permission is prohibited (write only)
6	setting of read and write permission is prohibited (execute only)
7	all permissions are prohibited from being set (no permissions)

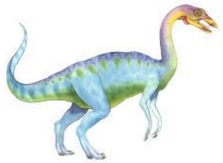
Octal digit in **umask** command	Binary in the mask	Negation of mask	Logical AND with "rwx" request[5]
0	000	111	rwx
1	001	110	rw-
2	010	101	r-x
3	011	100	r--
4	100	011	-wx
5	101	010	-w-
6	110	001	--x
7	111	000	---





Named Pipes	Anonymous Pipes
Sharable btw any processes (subject to permissions)	Sharable btw processes of the same ancestry
Full Duplex in Windows Half Duplex in Linux	Unidirectional
`CreateNamedPipes` (Windows) `mkfifo` (Unix/Linux)	`CreatePipes` (Windows) `pipe` (Linux)
Obtain the pipe by the name of the pipe	Obtain the pipe from parent





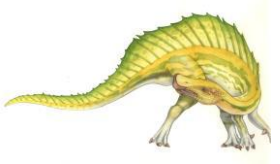
Message Queue

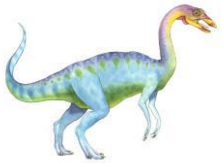
- “Typed” Message Passing
- Message Queue: organizes messages in a linked list
  - Supports asynchronous communication (non-blocking)
- Format of the message: type + data
  - Type: represented by an integer, the specific meaning is determined by the user
- Message queues are indirect messaging
  - Establish connections by sharing a queue

System V example

```
key = ftok("./msgque", 11);
msgid = msgget(key, 0666
| IPC_CREAT);
message.mesg_type = 1;
msgsnd(msgid, &message,
sizeof(message), 0);
```

```
key = ftok(”./msgque", 11);
msgid = msgget(key, 0666 |
IPC_CREAT);
msgrcv(msgid, &message,
sizeof(message), 1, 0);
msgctl(msgid, IPC_RMID,
NULL);
```

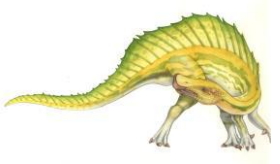


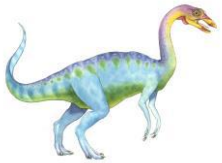


Message Queue

- Organization of message queues
  - Basically, follow the FIFO (First-In-First-Out) principle
  - Write to message queue: append to the end of the queue
  - Read from the message queue: popped from the head of the queue

- Allow query by type: Recv(A, type, message)
  - type is 0: returns the first message (FIFO)
  - oterwise, query based on the type number
    - If type > 0, return the first message of that type

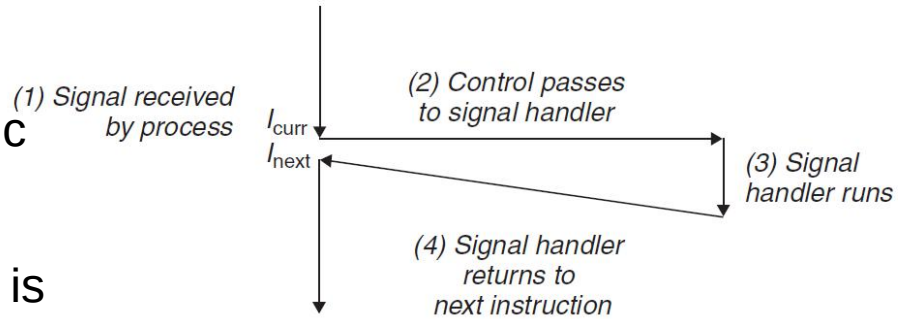


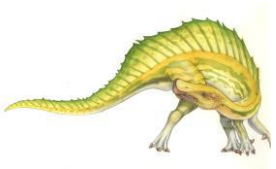


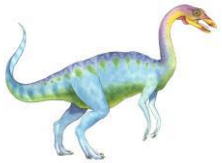
Signal

- A signal is a small message that notifies a process that an event of some type has occurred in the system.
  - linux> */bin/kill -9 -15213*
  - kill(15213, SIGKILL);
- Whenever there is a transition from kernel-mode to user-mode execution
  - e.g., on return from a syscall c
  - scheduling onto the CPU
- The kernel checks whether there is a pending unblocked signal for which the process has established a signal handler.
- If there is, start signal handling steps

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```



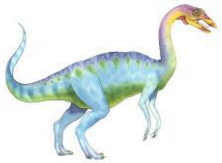




Signal

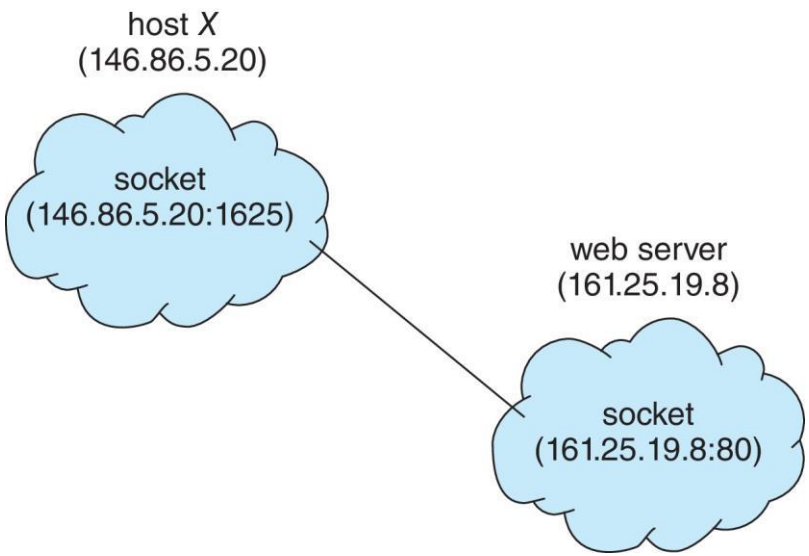
Number	Name	Default action	Corresponding event
1	SIGHUP	Terminate	Terminal line hangup
2	SIGINT	Terminate	Interrupt from keyboard
3	SIGQUIT	Terminate	Quit from keyboard
4	SIGILL	Terminate	Illegal instruction
5	SIGTRAP	Terminate and dump core [a]	Trace trap
6	SIGABRT	Terminate and dump core [a]	Abort signal from `abort` function
7	SIGBUS	Terminate	Bus error
8	SIGFPE	Terminate and dump core [a]	Floating-point exception
9	SIGKILL	Terminate [b]	Kill program
10	SIGUSR1	Terminate	User-defined signal 1
11	SIGSEGV	Terminate and dump core [a]	Invalid memory reference (seg fault)
12	SIGUSR2	Terminate	User-defined signal 2
13	SIGPIPE	Terminate	Wrote to a pipe with no reader
14	SIGALRM	Terminate	Timer signal from `alarm` function
15	SIGTERM	Terminate	Software termination signal
16	SIGSTKFLT	Terminate	Stack fault on coprocessor
17	SIGCHLD	Ignore	A child process has stopped or terminated
18	SIGCONT	Ignore	Continue process if stopped
19	SIGSTOP	Stop until next SIGCONT [b]	Stop signal not from terminal
20	SIGTSTP	Stop until next SIGCONT	Stop signal from terminal
21	SIGTTIN	Stop until next SIGCONT	Background process read from terminal
22	SIGTTOU	Stop until next SIGCONT	Background process wrote to terminal
23	SIGURG	Ignore	Urgent condition on socket
24	SIGXCPU	Terminate	CPU time limit exceeded
25	SIGXFSZ	Terminate	File size limit exceeded
26	SIGVTALRM	Terminate	Virtual timer expired
27	SIGPROF	Terminate	Profiling timer expired
28	SIGWINCH	Ignore	Window size changed
29	SIGIO	Terminate	I/O now possible on a descriptor
30	SIGPWR	Terminate	Power failure

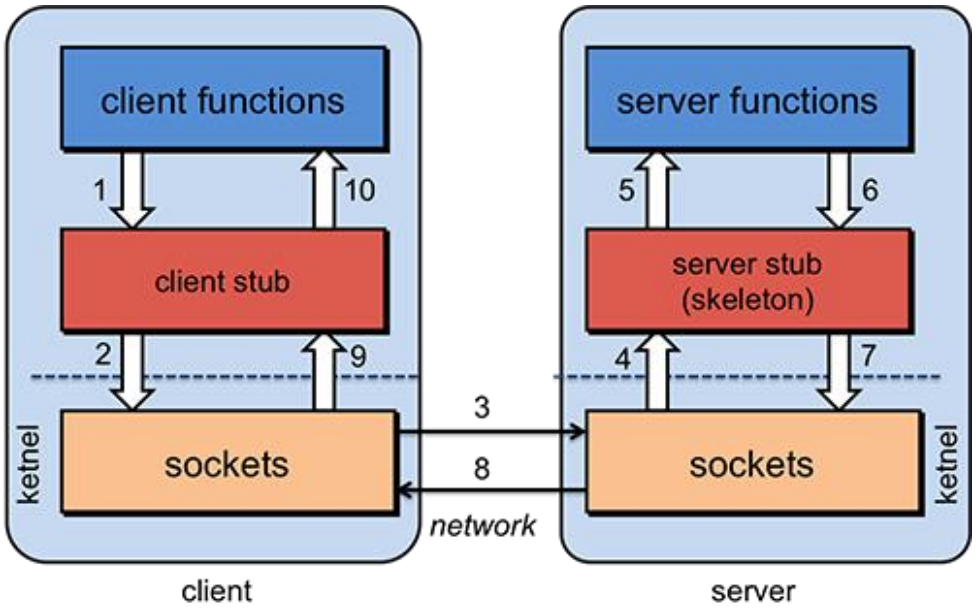
Figure 8.26 **Linux signals.** *Notes:* (a) Years ago, main memory was implemented with a technology known as *core memory*. "Dumping core" is a historical term that means writing an image of the code and data memory segments to disk. (b) This signal can be neither caught nor ignored. (Source: `man 7 signal`. Data from the Linux Foundation.)

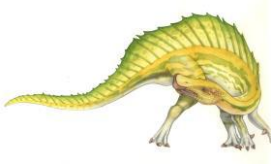


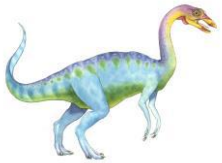
Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls (RPC)









Summary

What is a process?
Process states
- new/ready/running/waiting/terminated

How does OS represent a process? PCB
- task-struct

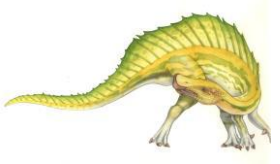
Scheduling queues
- ready queue, wait queue

Interprocess communication methods:
- shared memory/message passing

Process creation and termination
- fork(), exit(), abort(), wait(), execlp(), execvp()
- zombie and orphans

Understand process tree
Communication methods:
- pipes/shared memory/massage queue/sockets/rpc/

Two kinds of pipes:
- Ordinary / Named pipes



End of Chapter 4