

# Operating Systems

No. 5

ศรัณย์ อินทโกสุม

Sarun Intakosum

# **Interprocess Communication (IPC)**

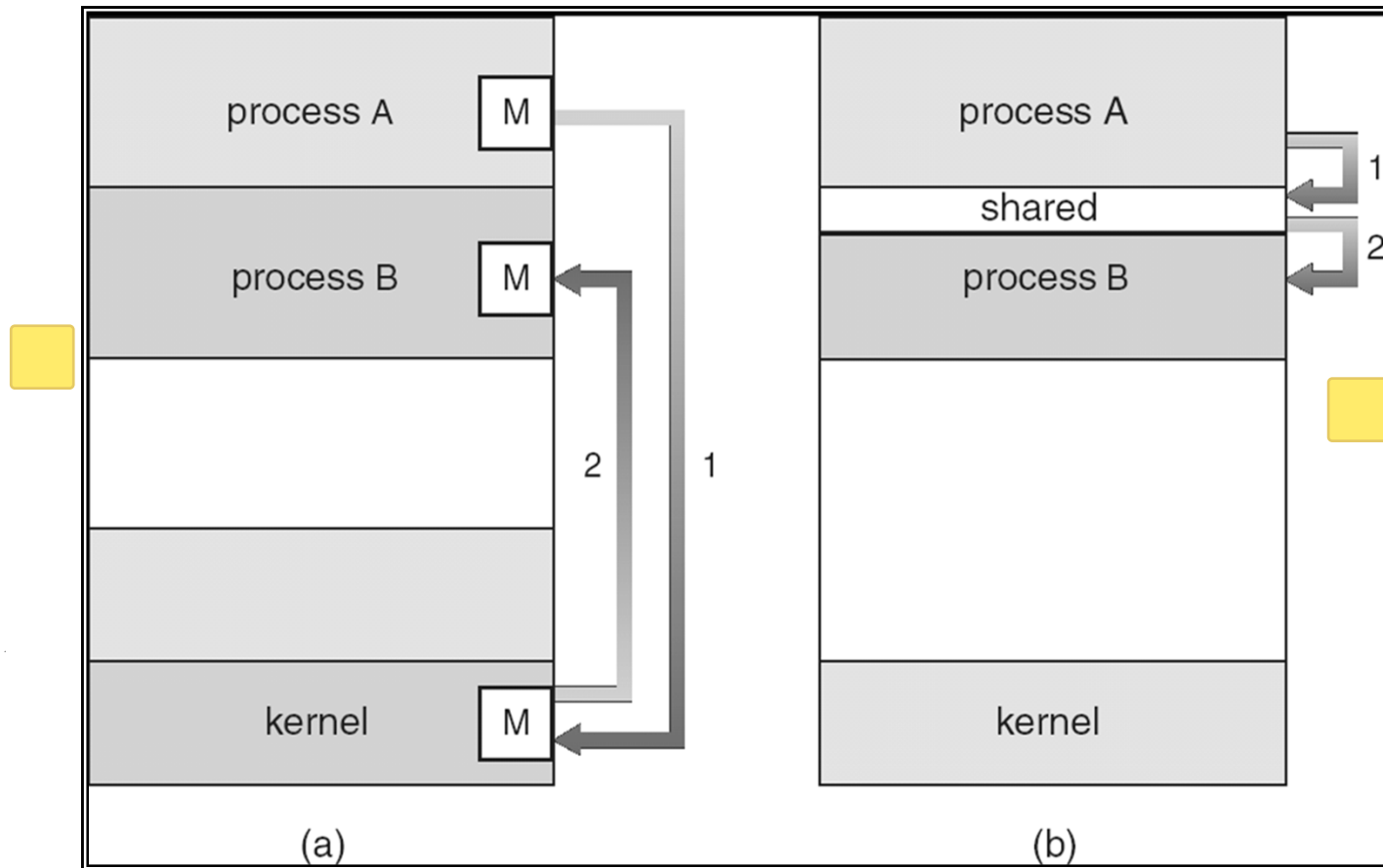
# Interprocess Communication

- ❑ Processes within a system may be **independent** or **cooperating**
- ❑ Cooperating process can affect or be affected by other processes, including sharing data
- ❑ Reasons for cooperating processes:
  - ❑ Information sharing
  - ❑ Computation speedup
  - ❑ Modularity
- ❑ Cooperating processes need **interprocess communication (IPC)**
- ❑ Three models of IPC
  - ❑ Signal
  - ❑ Shared memory
  - ❑ Message passing

# Signals

- Software interrupts that notify a process that an event has occurred
  - Do not allow processes to specify data to exchange with other processes
  - Processes may catch, use OS default, or mask a signal
    - Catching a signal involves specifying a routine that the OS calls when it delivers the signal
    - Use operating system's default action to handle the signal
    - Masking a signal instructs the OS to not deliver signals of that type until the process clears the signal mask

# Communications Models



# UNIX Inter Process Communications

- Signal
- Pipe
- Shared memory

# What Are Signals?

- Signals are generated when an event occurs that requires attention.
- Signals can be generated from the following sources:
  - Hardware
    - Example: division by zero, address protection
  - Other Processes
    - Example: a child process notifying its parent process that it has terminated.
  - User
    - Example: pressing a keyboard sequences that generate a quit, interrupt or stop signal.

## Signal (Contd.)


- Different systems define different signals. The available signals and their names are defined in the header file
- `signal.h`
- Note that `signal.h` is a header file of ANSI C. However, not all signals listed below are supported by the ANSI C standard.
- Under Unix, all signal names start with SIG. The following are some examples:



# Signal (Cont.)

- ❑ SIGFPE An arithmetic exception (e.g., division by zero, overflow and underflow) has occurred.
  - ❑ SIGINT An interrupt (e.g., Ctrl-C) has occurred.
  - ❑ SIGTERM Software termination.
  - ❑ SIGQUIT (non-ANSI C) Quit execution (e.g., Ctrl-\\).
  - ❑ SIGUSR1 and SIGUSR2 (e.g., non-ANSI C) User-defined signal 1 and 2.
  - ❑ SIGKILL (non-ANSI C) Kill the process (cannot be caught or ignored)
- 
- ❑ A program can take action when a signal is generated. The actual action taken, however, depends on the current signal handler. Some signals can be ignored, while some other cannot (e.g., SIGKILL). When you want to handle a particular signal, a signal handler must be installed.

# Handling Signals: Function signal()

- The most important function for handling signals is
- signal(). 
- signal() is a system call that returns the address of a function that takes an integer argument and has no return value.
  - This function is called a signal handler.

# signal() ( Contd.)

- signal() requires two arguments:
  - The first one is an integer int, which must be a signal name (e.g., SIGINT) or a predefined signal related constant.
  - The second one is a signal handler function that must be in the following format:
    - ▶ void (\*) (int)
    - ▶ Accepts an integer argument and returns nothing
- More precisely, if the system detects the signal specified in the first argument, the signal handler as specified in the second argument is called.

# Basic signal Handler (1)

```
#include <stdio.h>
#include <signal.h>
void INThandler(int);          /* Ctrl-C handler */
void main(void)
{
    signal(SIGINT, INThandler); /* install Ctrl-C handler */
    while (1)                  /* loop forever and wait */
        pause();               /* for Ctrl-C to come */
}
```

## Basic signal Handler (2)

```
void INThandler(int sig)
{
    char c;

    signal(sig, SIG_IGN);          /* disable Ctrl-C          */
    printf("OUCH, did you hit Ctrl-C?\n" /* print something */
        "Do you really want to quit? [y/n] ");
    c = getchar();                 /* read an input character */
    if (c == 'y' || c == 'Y')      /* if it is y or Y, then   */
        exit(0);                 /* exit. Otherwise,       */
    else
        signal(SIGINT, INThandler); /* reinstall the handler   */
}
```

## Basic signal Handler (3)

- It is important to remind you that in a signal handler you might want to disable all signals, including the one this handler is processing.
- This is because you perhaps do not want to be interrupted while the current signal is being processed.
- Of course, it is your choice.
- In some applications, you might want to disable some signals in a handler and allow some others to occur.

# Handling Multiple Signal Types

- ❑ Catching multiple types of signals is as easy as catching one type as discussed on the previous page. There are two choices:
- ❑ Install a signal handler for each signal. For example, to handle SIGINT and SIGQUIT, you can write a handler for each signal and install them separately as shown below:
- ❑ `signal(SIGINT, INThandler);`
- ❑ `signal(SIGQUIT, QUITHandler);`
- ❑ `void INThandler(int sig)`
- ❑ `{`
- ❑ `.....`
- ❑ `}`
- ❑ `void QUITHandler(int sig)`
- ❑ `{`
- ❑ `.....`
- ❑ `}`

# Handling Multiple Signal Types (Contd.)

- Or, you can write a single handler in which its only argument is used to determine what signal is sent in:
- `signal(SIGINT, SIGHandler);`
- `signal(SIGALRM, SIGHandler);`
- `void SIGHandler(int sig)`
- `{`
- `.....`
- `switch (sig) {`
- `case SIGINT : .....`
- `case SIGALRM : .....`
- `▶ default : .....`
- `}`
- `.....`
- `}`




# Multiple Signal Handlers (1)

```
#include <stdio.h>
#include <signal.h>
void SIGhandler(int);
void main(void)
{
    signal(SIGINT, SIGhandler);    /* install Ctrl-C handler */
    signal(SIGQUIT, SIGhandler);   /* install Ctrl-\ handler */
    while (1);                     /* loop forever */
}
```

# Multiple Signal Handlers (2)

```
void SIGhandler(int sig)
{
    char c;
    signal(SIGINT, SIG_IGN);          /* disable Ctrl-C      */
    signal(SIGQUIT, SIG_IGN);
    switch(sig) {
        case SIGINT:
            printf("Ha! Ha! You cannot interrupt me\n");
            break;
        case SIGQUIT:
            printf("Ha! Ha! you cannot quit me\n");
            break;
    }
    signal(SIGINT, SIGhandler);
    signal(SIGQUIT, SIGhandler);
}
```

# Sending a Signal: ANSI C Function `raise()`

- ❑ ANSI C has a library function `raise()` for sending a signal to the program that contains this `raise()` call. The following is the function prototype:
- ❑ `int raise(int sig)` 
- ❑ If this function is executed successfully, the signal specified in `sig` is generated. If the program has a signal handler, this signal will be processed by the that signal handler; otherwise, this signal will be handled by the default way. If the call is unsuccessful, `raise()` returns a non-zero value.
- ❑ Note that function `raise()` can only send a signal to the program that contains it. `raise()` cannot send a signal to other processes. To send a signal to other processes, the system call `kill()` should be used.

# raise() Example (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
long prev_fact, i;          /* global variables          */
void SIGhandler(int);       /* SIGUSR1 handler        */

void SIGhandler(int sig)
{
    printf("\nReceived a SIGUSR1. The answer is %ld! = %ld\n",
           i-1, prev_fact);
    exit(0);
}
```

## raise() Example (2)

```
void main(void)
{
    long fact;

    printf("Factorial Computation:\n\n");
    signal(SIGUSR1, SIGHandler); /* install SIGUSR1 handler */
    for (prev_fact = i = 1; ; i++, prev_fact = fact) {
        fact = prev_fact * i; /* computing factorial */
        if (fact < 0) /* if the results wraps around */
            raise(SIGUSR1); /* we have overflow, print it */
        else if (i % 3 == 0) /* otherwise, print the value */
            printf(" %ld! = %ld\n", i, fact);
    }
}
```

# **Sending a Signal to Another Process: System Call kill()**

- To send a signal to another process, we need to use the Unix system `kill()`. The following is the prototype of `kill()`:
- `int kill(pid_t pid, int sig)`
- System call `kill()` takes two arguments. The first, `pid`, is the process ID you want to send a signal to, and the second, `sig`, is the signal you want to send. Therefore, you have to find some way to know the process ID of the other party.
- If the call to `kill()` is successful, it returns 0; otherwise, the returned value is negative.
- Because of this capability, `kill()` can also be considered as a communication mechanism among processes with signals `SIGUSR1` and `SIGUSR2`.

# kill command

- As you might have learned, kill is also a command for you to kill a running process using a process ID. Moreover, we frequently use kill - 9 pid
- to kill a process that is so stubborn that a simple kill is not powerful enough to kill. So, what does -9 mean? Actually, this 9 is the signal number of SIGKILL for killing a process. If we can use the kill system call to kill a process (i.e., sending a SIGKILL to it), it should be possible to send any signal to a process with the kill command. This is true. The kill command has the following form:
  - kill **-signal** pid pid pid ... pid
  - The first argument, shown in boldface, is the signal name that represents the signal to be sent. This signal name does not include SIG. Thus, if you want to send signals SIGINT, SIGQUIT and SIGKILL, use INT, QUIT and KILL, respectively.

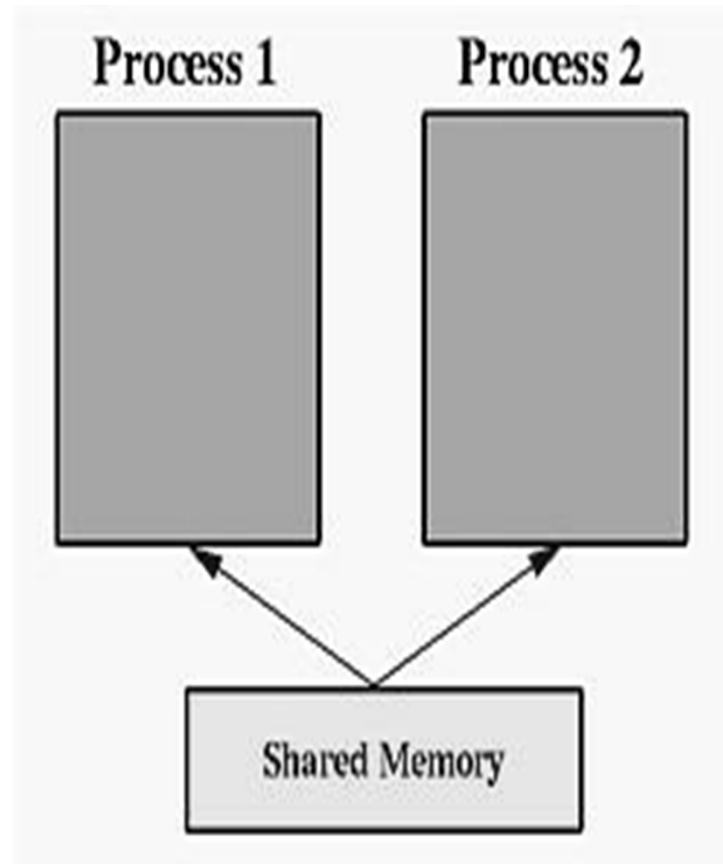
# kill command (Contd.)

- If the signal name is missing, the default SIGTERM is used.
- Following the signal name is a list of process IDs to which the signal will be sent. These process IDs are separated by white spaces. Thus, the following kill command sends signal SIGINT to process IDs 13579 and 36214.
- `kill -KILL 13579 36214`



# What is Shared Memory?

- A shared memory is an extra piece of memory that is attached to some address spaces for their owners to use. As a result, all of these processes share the same memory segment and have access to it. Consequently, race conditions may occur if memory accesses are not handled properly.
- The figure shows two processes and their address spaces. The yellow rectangle is a shared memory attached to both address spaces and both process 1 and process 2 can have access to this shared memory as if the shared memory is part of its own address space.



# What is Shared Memory? (Contd.)

- ❑ Shared memory is a feature supported by UNIX System V, including Linux, SunOS and Solaris.
- ❑ One process must explicitly ask for an area, using a key, to be shared by other processes. This process will be called the server.
- ❑ All other processes, the clients, that know the shared area can access it.
- ❑ However, there is no protection to a shared memory and any process that knows it can access it freely. To protect a shared memory from being accessed at the same time by several processes, a synchronization protocol must be setup.

# A general scheme of using shared memory (Server)

- For a server, it should be started before any client. The server should perform the following tasks:
  - Ask for a shared memory with a memory key and memorize the returned shared memory ID. This is performed by system call `shmget()`.
  - Attach this shared memory to the server's address space with system call `shmat()`.
  - Initialize the shared memory, if necessary.
  - Do something and wait for all clients' completion.
  - Detach the shared memory with system call `shmdt()`.
  - Remove the shared memory with system call `shmctl()`.

# A general scheme of using shared memory (client)

- For the client part, the procedure is almost the same:
  - Ask for a shared memory with the same memory key and memorize the returned shared memory ID.
  - Attach this shared memory to the client's address space.
  - Use the memory.
  - Detach all shared memory segments, if necessary.
  - Exit.

# Programming Shared Memory

- A shared memory segment is identified by a unique integer, the shared memory ID. The shared memory itself is described by a structure of type `shmid_ds` in header file `sys/shm.h`. The program should start with the following lines:
  - `#include <sys/types.h>`
  - `#include <sys/ipc.h>`
  - `#include <sys/shm.h>`

# Key

- Unix requires a key of type `key_t` defined in file `sys/types.h` for requesting resources such as shared memory segments, message queues and semaphores. A key is simply an integer of type `key_t`; however, you should not use `int` or `long`, since the length of a key is system dependent.
- There are three different ways of using keys, namely:
  - a specific integer value (e.g., 123456)
  - a key generated with function `ftok()`
  - a uniquely generated key using `IPC_PRIVATE` (i.e., a private key).

# Key (Contd.)

- The first way is the easiest one; however, its use may be very risky since a process can access your resource as long as it uses the same key value to request that resource. The following example assigns 1234 to a key:
- `key_t SomeKey;`
- `SomeKey = 1234;`

# ftok()

- The ftok() function has the following prototype:
- `key_t ftok( const char *path, /* a path string */`
- `int id /* an integer value */);`
- Function ftok() takes a character string that identifies a path and an integer (usually a character) value, and generates an integer of type `key_t` based on the first argument with the value of `id` in the most significant position. Thus, as long as processes use the same arguments to call ftok(), the returned key value will always be the same. The most commonly used value for the first argument is ".", the current directory.



## ftok() (Contd.)

- If all related processes are stored in the same directory, the following call to ftok() will generate the same key value:
  - `#include <sys/types.h>`
  - `#include <sys/ipc.h>`
  - `key_t SomeKey;`
  - `SomeKey = ftok(".", 'x');`

# IPC\_PRIVATE

- ❑ In this case, the system will generate a unique key and guarantee that no other process will have the same key.
- ❑ If a resource is requested with IPC\_PRIVATE in a place where a key is required, that process will receive a unique key for that resource.
- ❑ Since that resource is identified with a unique key unknown to the outsiders, other processes will not be able to share that resource and, as a result, the requesting process is guaranteed that it owns and accesses that resource exclusively.
- ❑ IPC\_PRIVATE is normally shared between related process such as parent/child, child/child.

# Asking for a Shared Memory Segment - shmget()

- It is defined as follows:
- `shm_id = shmget( key_t k, /* the key for the segment */`
- `int size, /* the size of the segment */`
- `int flag); /* create/use flag */`
- In the above definition, k is of type key\_t or IPC\_PRIVATE. size is the size of the requested shared memory. The purpose of flag is to specify the way that the shared memory will be used. For our purpose, only the following two values are important:
- `IPC_CREAT | 0666` for a server (i.e., creating and granting read and write access to the server)
- `0666` for any client (i.e., granting read and write access to the client)
- If shmget() can successfully get the shared memory, its function value is a non-negative integer, the shared memory ID; otherwise, the function value is negative.

# Example of using shmget()

- `#include <sys/types.h>`
- `#include <sys/ipc.h>`
- `#include <sys/shm.h>`
- `#include <stdio.h>`
- `int shm_id; /* shared memory ID */`
- `shm_id = shmget(IPC_PRIVATE, 4*sizeof(int), IPC_CREAT | 0666);`
- `if (shm_id < 0) {`
- `printf("shmget error\n");`
- `exit(1);`
- `}`
- If a client wants to use a shared memory created with `IPC_PRIVATE`, it must be a child process of the server, created after the parent has obtained the shared memory, so that the private key value can be passed to the child when it is created.

# Attaching a Shared Memory Segment to an Address Space - `shmat()`

- After a shared memory ID is returned, the next step is to attach it to the address space of a process. This is done with system call `shmat()`. The use of `shmat()` is as follows:
- `shm_ptr = shmat(int shmid, /* shared memory ID */`
- `char *ptr, /* a character pointer */`
- `int flag); /* access flag */`

## shmat() (Contd.)

- ❑ System call `shmat()` accepts a shared memory ID, `shm_id`, and attaches the indicated shared memory to the program's address space.
- ❑ The returned value is a pointer of type `(void *)` to the attached shared memory.
- ❑ Thus, casting is usually necessary. If this call is unsuccessful, the return value is `-1`.
- ❑ Normally, the second parameter is `NULL`. If the flag is `SHM_RDONLY`, this shared memory is attached as a read-only memory; otherwise, it is readable and writable.

# Detaching a Shared Memory Segment - `shmdt()`

- System call `shmdt()` is used to detach a shared memory. After a shared memory is detached, it cannot be used. However, it is still there and can be re-attached back to a process's address space, perhaps at a different address. To remove a shared memory, use `shmctl()`.
- The only argument of the call to `shmdt()` is the shared memory address returned by `shmat()`. Thus, the following code detaches the shared memory from a program:
  - `shmdt(shm_ptr);`
- where `shm_ptr` is the pointer to the shared memory. This pointer is returned by `shmat()` when the shared memory is attached. If the detach operation fails, the returned function value is non-zero.

# Removing a Shared Memory Segment – shmctl()

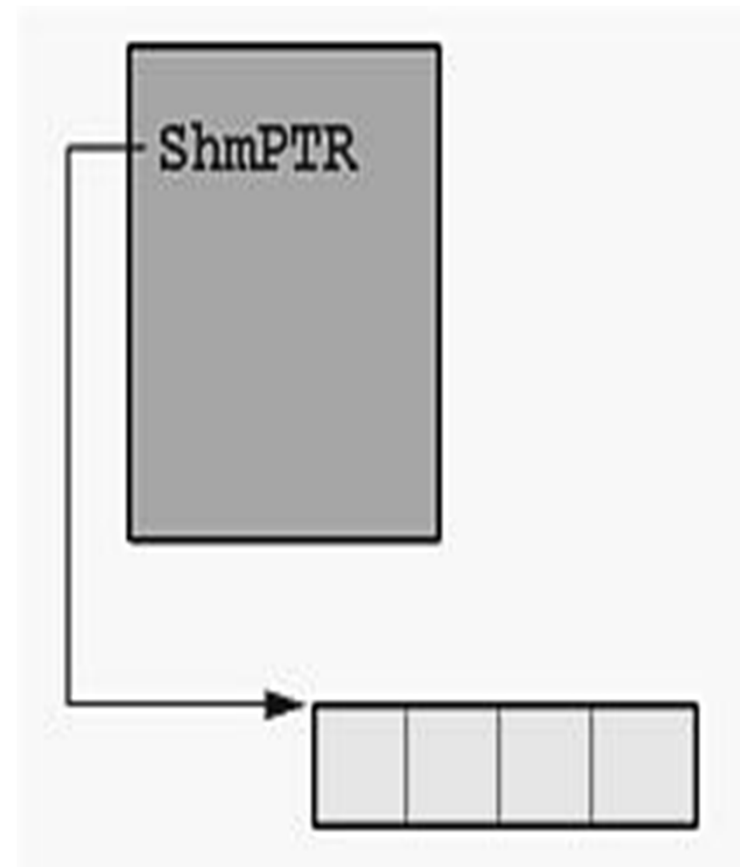
- To remove a shared memory segment, use the following code:
- `shmctl(shm_id, IPC_RMID, NULL);`
- where `shm_id` is the shared memory ID. `IPC_RMID` indicates this is a remove operation. Note that after the removal of a shared memory segment, if you want to use it again, you should use `shmget()` followed by `shmat()`.



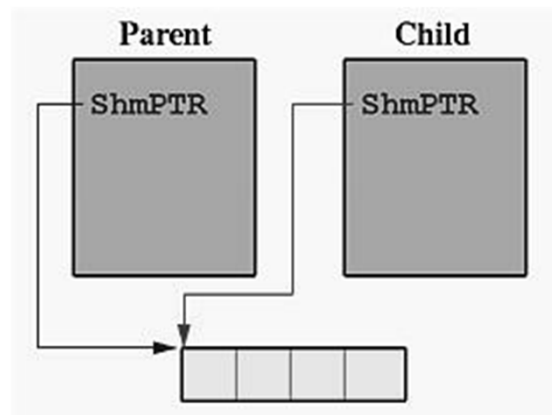
# Parent and Child Share Memory (1)

□ see shm-01.c

Before fork



After fork



# Parent and Child Share Memory (2)

```
/* shm-01.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
void ClientProcess(int []);
void main(int argc, char *argv[])
{
    int ShmID;
    int *ShmPTR;
    pid_t pid;
    int status;
    if (argc != 5) {
        printf("Use: %s #1 #2 #3 #4\n", argv[0]);
        exit(1);
    }
```

# Parent and Child Share Memory (3)

```
ShmID = shmget(IPC_PRIVATE, 4*sizeof(int), IPC_CREAT | 0666);
if (ShmID < 0) {
    printf("*** shmget error (server) ***\n");
    exit(1);
}
printf("Server has received a shared memory of four integers...\n");

ShmPTR = (int *) shmat(ShmID, NULL, 0);
if ((int) ShmPTR == -1) {
    printf("*** shmat error (server) ***\n");
    exit(1);
}
printf("Server has attached the shared memory...\n");
```

# Parent and Child Share Memory (4)

```
ShmPTR[0] = atoi(argv[1]);
ShmPTR[1] = atoi(argv[2]);
ShmPTR[2] = atoi(argv[3]);
ShmPTR[3] = atoi(argv[4]);
printf("Server has filled %d %d %d %d in shared memory...\n",
      ShmPTR[0], ShmPTR[1], ShmPTR[2], ShmPTR[3]);

printf("Server is about to fork a child process...\n");
pid = fork();
if (pid < 0) {
    printf("*** fork error (server) ***\n");
    exit(1);
}
else if (pid == 0) {
    ClientProcess(ShmPTR);
    exit(0);
}
```

# Parent and Child Share Memory (5)

```
wait(&status);
printf("Server has detected the completion of its child...\n");
shmdt((void *) ShmPTR);
printf("Server has detached its shared memory...\n");
shmctl(ShmID, IPC_RMID, NULL);
printf("Server has removed its shared memory...\n");
printf("Server exits...\n");
exit(0);
}
void ClientProcess(int SharedMem[])
{
    printf(" Client process started\n");
    printf(" Client found %d %d %d %d in shared memory\n",
           SharedMem[0], SharedMem[1], SharedMem[2],
           SharedMem[3]);
    printf(" Client is about to exit\n");
}
```

# Client-Server Shared Memory (1)

- see server.c, client.c
- Note that the above code assumes the server and client programs are in the current directory. In order for the client to run correctly, the server must be started first and the client can only be started after the server has successfully obtained the shared memory.

# Client-Server Shared Memory (2)

```
/*shm-02.h */  
#define NOT_READY -1  
#define FILLED    0  
#define TAKEN     1  
  
struct Memory {  
    int status;  
    int data[4];  
};
```

# Client-Server Shared Memory (3)

```
/*server.c*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "shm-02.h"
void main(int argc, char *argv[])
{
    key_t      ShmKEY;
    int        ShmID;
    struct Memory *ShmPTR;
    if (argc != 5) {
        printf("Use: %s #1 #2 #3 #4\n", argv[0]);
        exit(1);
    }
}
```



# Client-Server Shared Memory (4)

```
ShmKEY = ftok(".", 'x');
ShmID = shmget(ShmKEY, sizeof(struct Memory), IPC_CREAT | 0666);
if (ShmID < 0) {
    printf("*** shmget error (server) ***\n");
    exit(1);
}
printf("Server has received a shared memory of four integers...\n");
ShmPTR = (struct Memory *) shmat(ShmID, NULL, 0);
if ((int) ShmPTR == -1) {
    printf("*** shmat error (server) ***\n");
    exit(1);
}
printf("Server has attached the shared memory...\n");
ShmPTR->status = NOT_READY;
ShmPTR->data[0] = atoi(argv[1]);
ShmPTR->data[1] = atoi(argv[2]);
ShmPTR->data[2] = atoi(argv[3]);
ShmPTR->data[3] = atoi(argv[4]);
```

# Client-Server Shared Memory (5)

```
printf("Server has filled %d %d %d %d to shared memory...\n",  
ShmPTR->data[0], ShmPTR->data[1],  
ShmPTR->data[2], ShmPTR->data[3]);  
ShmPTR->status = FILLED;  
printf("Please start the client in another window...\n");  
  
while (ShmPTR->status != TAKEN)  
    sleep(1);  
  
printf("Server has detected the completion of its child...\n");  
shmdt((void *) ShmPTR);  
printf("Server has detached its shared memory...\n");  
shmctl(ShmID, IPC_RMID, NULL);  
printf("Server has removed its shared memory...\n");  
printf("Server exits...\n");  
exit(0);  
}
```

# Client-Server Shared Memory (6)

```
/* client.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "shm-02.h"
void main(void)
{
    key_t    ShmKEY;
    int      ShmID;
    struct Memory *ShmPTR;
    ShmKEY = ftok(".", 'x');
    ShmID = shmget(ShmKEY, sizeof(struct Memory), 0666);
    if (ShmID < 0) {
        printf("*** shmget error (client) ***\n");
        exit(1);
    }
}
```

# Client-Server Shared Memory (7)

```
printf(" Client has received a shared memory of four integers...\n");
ShmPTR = (struct Memory *) shmat(ShmID, NULL, 0);
if ((int) ShmPTR == -1) {
    printf("*** shmat error (client) ***\n");
    exit(1);
}
printf(" Client has attached the shared memory...\n");
while (ShmPTR->status != FILLED) ;
printf(" Client found the data is ready...\n");
printf(" Client found %d %d %d %d in shared memory...\n",
        ShmPTR->data[0], ShmPTR->data[1],
        ShmPTR->data[2], ShmPTR->data[3]);
ShmPTR->status = TAKEN;
printf(" Client has informed server data have been taken...\n");
shmdt((void *) ShmPTR);
printf(" Client has detached its shared memory...\n");
printf(" Client exits...\n");
exit(0);
}
```

# Message Passing

- Message-based interprocess communication
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Messages can be passed in one direction at a time
  - One process is the sender and the other is the receiver
- Message passing can be bidirectional
  - Each process can act as either a sender or a receiver
- Popular implementation is a pipe
  - A region of memory protected by the OS that serves as a buffer, allowing two or more processes to exchange data

# Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** has the sender send the message and continue
  - **Non-blocking receive** has the receiver receive a valid message or null

# Pipes

- A pipe is used for one-way communication of a stream of bytes. The command to create a pipe is `pipe`, which takes an array of two integers. It fills in the array with two file descriptors that can be used for low-level I/O.

# Creating a Pipe

- `int pfd[2];`
- `pipe(pfd);`



`pfd[1]`

`pfd[0]`

write to  
this end

read from  
this end



# I/O with a pipe

- These two file descriptors can be used for block I/O
  - `write(pfd[1], buf, size);`
  - `read(pfd[0], buf, SIZE);`

# Fork and a pipe

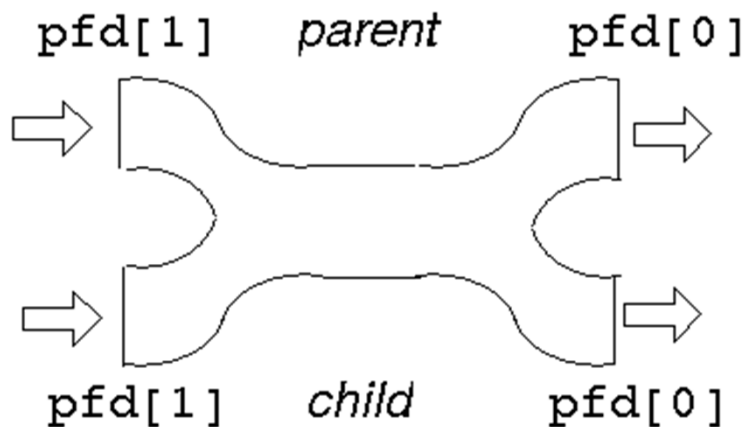
- A single process would not use a pipe. They are used when two processes wish to communicate in a one-way fashion. A process splits in two using ``fork''. A pipe opened before the fork becomes shared between the two processes.

# Fork and a pipe (Contd.)

Before fork



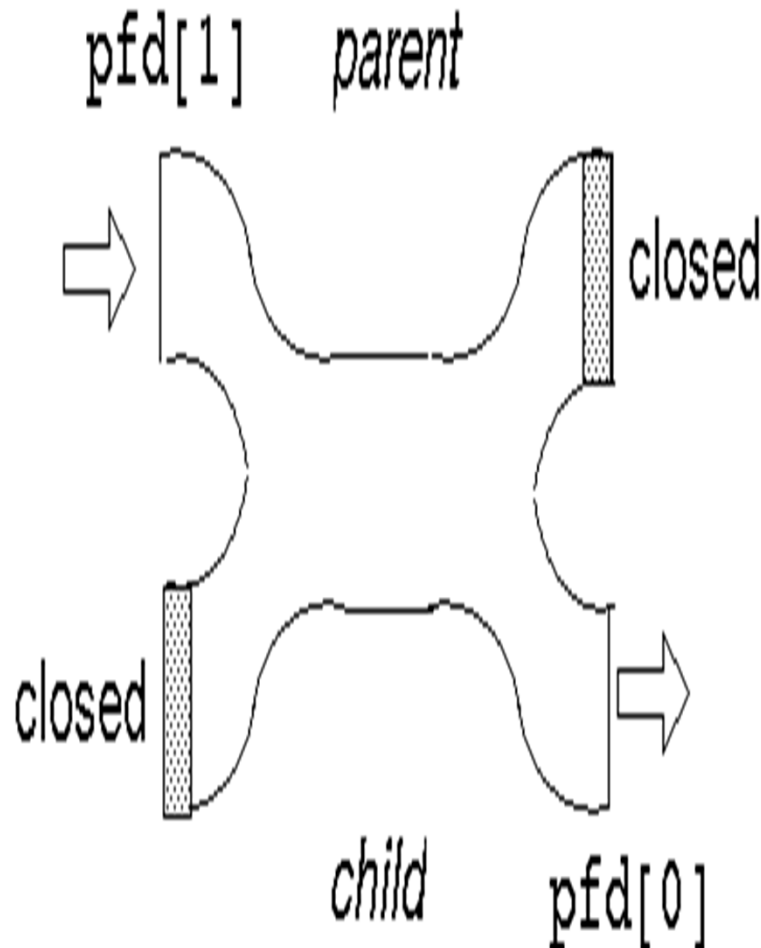
After fork



This gives two read ends and two write ends.

Either process can write into the pipe, and either can read from it. Which process will get what is not known.

## Fork and a pipe (Contd.)



- For predictable behaviour, one of the processes must close its read end, and the other must close its write end. Then it will become a simple pipeline again.
- Suppose the parent wants to write down a pipeline to a child. The parent closes its read end, and writes into the other end. The child closes its write end and reads from the other end.

# pipe() Example (1)

```
#include <stdio.h>
#define SIZE 1024
int main(int argc, char **argv)
{
    int pfd[2];
    int nread;
    int pid;
    char buf[SIZE];
    if (pipe(pfd) == -1)
    {
        perror("pipe failed");
        exit(1);
    }
    if ((pid = fork()) < 0)
    {
        perror("fork failed");
        exit(2);
    }
}
```

## pipe() Example (2)

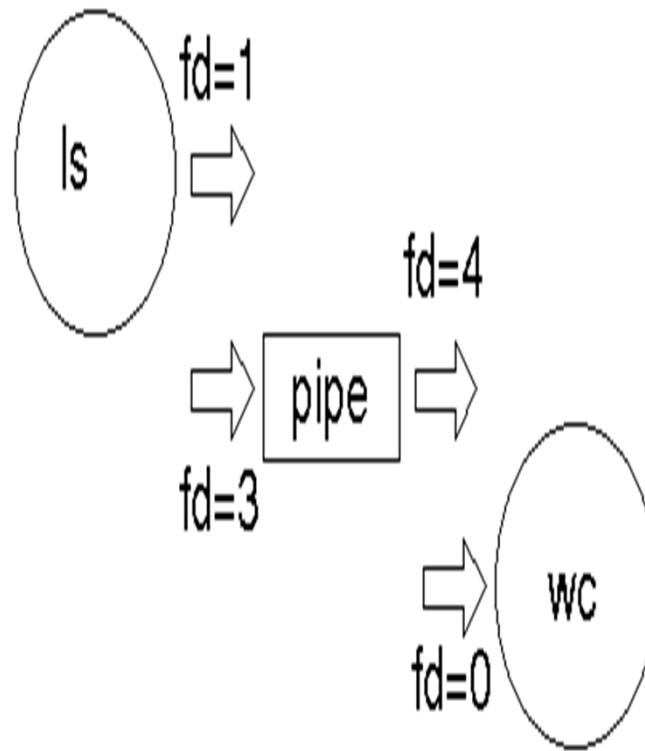
```
if (pid == 0)
{
    /* child */
    close(pfd[1]);
    while ((nread = read(pfd[0], buf, SIZE)) != 0)
        printf("child read %s\n", buf);
    close(pfd[0]);
} else {
    /* parent */
    close(pfd[0]);
    strcpy(buf, "hello...");
    /* include null terminator in write */
    write(pfd[1], buf,
        strlen(buf)+1);
    close(pfd[1]);
    wait();
}
exit(0);
}
```

# Dup

- A pipeline works because the two processes know the file descriptor of each end of the pipe. Each process has a stdin (0), a stdout (1) and a stderr (2). The file descriptors will depend on which other files have been opened, but could be 3 and 4, say.
- Suppose one of the processes replaces itself by an `exec`. The new process will have files for descriptors 0, 1, 2, 3 and 4 open. How will it know which are the ones belonging to the pipe? It can't.

# Example: why we need dup?

- To implement `ls | wc` the shell will have created a pipe and then forked. The parent will exec to be replaced by `ls`, and the child will exec to be replaced by `wc`. The write end of the pipe may be descriptor 3 and the read end may be descriptor 4. `ls` normally writes to 1 and `wc` normally reads from 0. How do these get matched up?

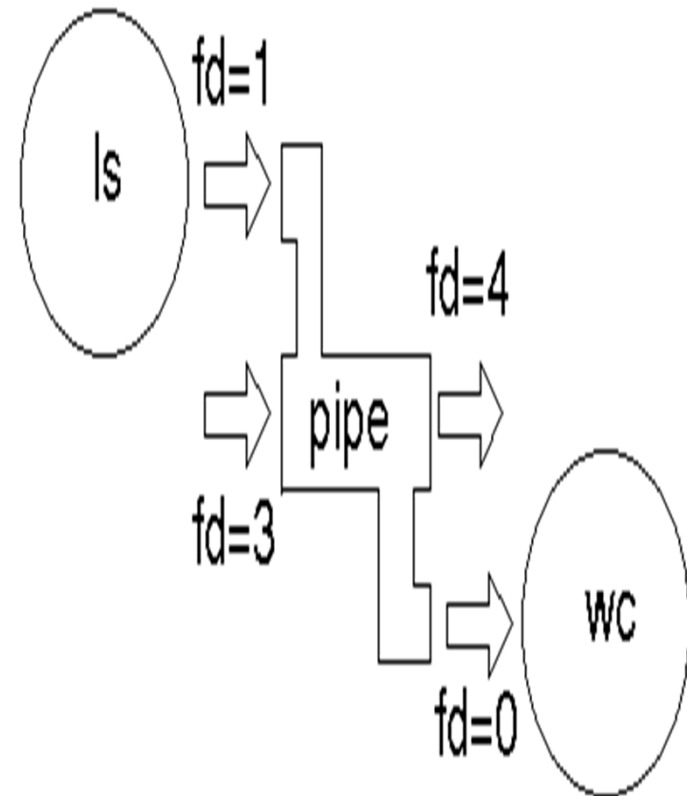




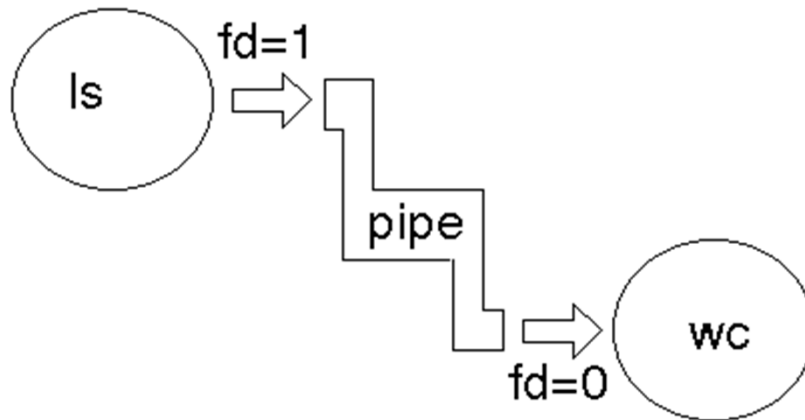
# dup2 function

- The ``dup2" function call takes an existing file descriptor, and another one that it ``would like to be". Here, fd=3 would also like to be 1, and fd=4 would like to be 0. So we dup2 fd=3 as 1, and dup2 fd=4 as 0. Then the old fd=3 and fd=4 can be closed as they are no longer needed.

Before Close



After close



# dup2() Example(1)

```
/* Prog: dup2.c */
#include <stdio.h>
int main(void)
{
    int pfd[2];
    int pid;

    if (pipe(pfd) == -1)
    {
        perror("pipe failed");
        exit(1);
    }
    if ((pid = fork()) < 0)
    {
        perror("fork failed");
        exit(2);
    }
}
```

## dup2() Example(2)

```
if (pid == 0)
{
    close(pfd[0]);
    dup2(pfd[1], 1);
    close(pfd[1]);
    execlp("ls", "ls", (char *) 0);
    perror("ls failed");
    exit(3);
} else {
    close(pfd[1]);
    dup2(pfd[0], 0);
    close(pfd[0]);
    execlp("wc", "wc", (char *) 0);
    perror("wc failed");
    exit(4);
}
exit(0);
}
```

# read/write from/to Pipe

- A process may want to both write to and read from a child. In this case it creates two pipes. One of these is used by the parent for writing and by the child for reading. The other is used by the child for writing and the parent for reading.

# Program to read and write Pipe (1)

```
#define SIZE 1024
int main(int argc, char **argv)
{
    int pfd1[2], int pfd2[2];
    int nread, status;
    pid_t pid;
    char buf[SIZE];
    if (pipe(pfd1) == -1)
    {
        perror("pipe failed");
        exit(1);
    }
    if (pipe(pfd2) == -1) {
        perror("pipe failed");
        exit(1);
    }
    if ((pid = fork()) < 0)
    { perror("fork failed");
      exit(2);
    }
}
```

# Program to read and write Pipe (2)

```
if (pid == 0)
{
    /* child */
    close(pfd1[1]);
    close(pfd2[0]);
    while ((nread = read(pfd1[0], buf, SIZE)) != 0) {
        printf("child read %s\n", buf);
    }
    close(pfd1[0]);
    strcpy(buf, "I am fine thank you");
    write(pfd2[1], buf, strlen(buf) + 1);
    close(pfd2[1]);
}
```

# Program to read and write Pipe (3)

```
else {
    /* parent */
    close(pfd1[0]);
    close(pfd2[1]);
    strcpy(buf, "How are you\0");
    /* include null terminator in write */
    write(pfd1[1], buf,
          strlen(buf)+1);
    close(pfd1[1]);
    while ((nread = read(pfd2[0], buf, SIZE)) != 0)
        printf("Parent read %s\n", buf);
    close(pfd2[0]);
}
exit(0);
}
```