Inter process Communication

Processes executing concurrently in the OS may be either independent processes or cooperating processes.

- A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent.
- A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

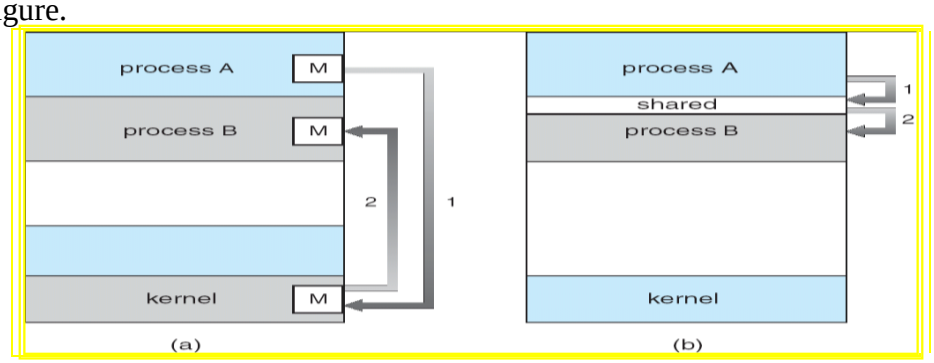
There are several reasons for providing an environment that allows process cooperation:

- **Information sharing**. Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
- **Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).
- **Modularity**. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
- **Convenience**. Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

Cooperating processes require an **inter process communication** (IPC) mechanism that will allow them to exchange data and information.

There are two fundamental models of inter process communication:

- **Shared Memory**. A region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
- **Message Passing**. Communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure.



Message Passing **Shared Memory**

Pros and cons of Inter process Communications

- Both of the models just discussed are common in OSs, and many systems implement both.

- Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement than is shared memory for inter computer communication.
- Shared memory allows maximum speed and convenience of communication, as it can be done at memory speeds when within a computer. Shared memory is faster than message passing, as message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention.
- In contrast, in shared-memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

1. Basic IPC

The IPC mechanisms can be classified into the following categories as given below:

a. Pipes
b. FIFOs
c. Shared memory
d. Mapped memory
e. Message queues
f. Sockets

1.1 Process of popen and pclose

Perhaps the simplest way of passing data between two programs is with the popen and pclose functions.

These have the following prototypes:

```
#include <stdio.h>
FILE *popen(const char *command, const char *open_mode);
int pclose(FILE *stream_to_close);
```

popen

The popen function allows a program to invoke another program as a new process and either pass data to it or receive data from it. The command string is the name of the program to run, together with any parameters. open_mode must be either "r" or "w".

If the open_mode is "r", output from the invoked program is made available to the invoking program and can be read from the file stream FILE * returned by popen, using the usual stdio library functions for reading (for example, fread). However, if open_mode is "w", the program can send data to the invoked command with calls to fwrite.

pclose

When the process started with popen has finished, you can close the file stream associated with it using pclose. The pclose call will return only when the process started with popen finishes. If it's still running when pclose is called, the pclose call will wait for the process to finish.

Assignment 12: Write a C program to invoke another program as a new process and receive data from the invoked program.

The following program use popen in a program to access information from ls. The ls -l command prints list of file with permission type. Open the pipe to ls -l, making it readable and setting rf to point to the output. At the end, the pipe pointed to by rf is closed.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{
FILE *rf;
char buffer[1024];
int res;
memset(buffer, '\0', sizeof(buffer));
rf = popen("ls -l", "r");
if (rf != NULL)
{
res = fread(buffer, sizeof(char), 1024, rf);
if (res > 0)
{
printf("Output was for the process %d:-\n%s\n",getpid(),buffer);
}
pclose(rf);
exit(EXIT_SUCCESS);
}
exit(EXIT_FAILURE);
}
```

The memset() function fills the first n bytes of the memory area pointed to by buffer with the constant byte 1024 the (buffer size).

The popen() function opens a process by creating a pipe, forking, and invoking the shell. Since a pipe is by definition unidirectional, the type argument may specify only reading or writing (“r” or “w”), not both; the resulting stream is correspondingly read-only or write-only.

The function fread() reads elements of data, each size bytes long, from the stream pointed to by rf, storing them at the location given by buffer.

Assignment 13: Write a C program to invoke another program as a new process and pass data to the invoked program.

In this program we are writing down a pipe.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{
FILE *wf;
char buffer[1024];
printf("Enter data\n");
scanf("%s",buffer);
wf = popen("od -c","w");
if (wf != NULL)
{
```

```
fwrite(buffer, sizeof(char), strlen(buffer), wf);
pclose(wf);
exit(EXIT_SUCCESS);
}
exit(EXIT_FAILURE);
}
```

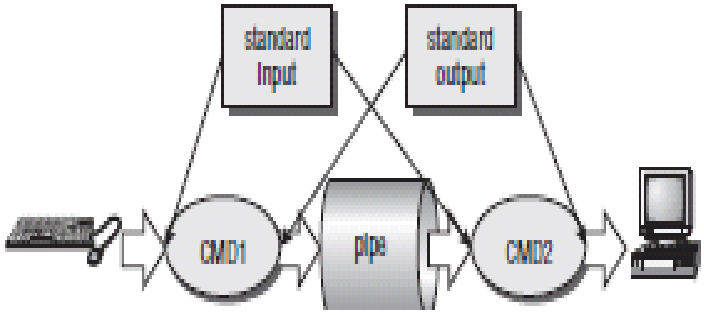
The function fwrite() writes elements of data, each size bytes long, to the stream pointed to by wf, obtaining them from the location given by buffer.
The program uses popen with the parameter "w" to start the od -c command, so that it can send data to that command. It then sends a string that the od -c command receives and processes; the od -c command then prints the result of the processing on its standard output.

1.2 Pipes

We use the term ***pipe*** to mean connecting a data flow from one process to another. Generally pipe is used to attach, the output of one process to the input of another. Most Linux users are already be familiar with the idea of a pipeline, linking shell commands together so that the output of one process is fed straight to the input of another. For shell commands, this is done using the pipe character to join the commands, such as
$cmd1 | cmd2
Or
$man ls | more
The shell arranges the standard input and output of the two commands, so that

- The standard input to cmd1 comes from the terminal keyboard.
- The standard output from cmd1 is fed to cmd2 as its standard input.
- The standard output from cmd2 is connected to the terminal screen.

What the shell has done, in effect, is reconnect the standard input and output streams so that data flows from the keyboard input through the two commands and is then output to the screen. See Figure for a visual representation of this process.



Pipes come in two varieties:
· **Unnamed.** Unnamed pipes can only be used by related processes (i.e. a process and one of its child processes, or two of its children). Unnamed pipes cease to exist after the processes are done using them.
· **Named.** Named pipes exist as directory entries, complete with permissions. This means that they are persistent and that unrelated processes can use them.
Most UNIX systems limit pipes to 5120K (typically ten 512K chunks). The unbuffered system call write ( ) is used to add data to a pipe. Write ( ) takes a file descriptor (which can refer to the pipe), a buffer containing the data to be written, and the size of the buffer

as parameters. The system assures that no interleaving will occur between writes, even if the pipeline fills temporarily. To get data from a pipe, the read ( ) system call is used. Read ( ) functions on pipes much the same as it functions on files. However, seeking is not supported and it will block until there is data to be read.

Implementation of unnamed pipe

The pipe ( ) system call is used to create unnamed pipes in UNIX. This call returns two pipes. Both support bidirectional communication (two pipes are returned for historical reasons: at one time pipes were unidirectional so two pipes were needed for bidirectional communication). In a full duplex environment (i.e. one that supports bidirectional pipes) each process reads from one pipe and writes to the other; in a half-duplex (i.e. unidirectional) setting, the first file descriptor is always used for reading and the second for writing.

Pipes are commonly used on the UNIX command line to send the output of one process to another process as input. When a pipe is used both processes run concurrently and there is no guarantee as to the sequence in which each process will be allowed to run. However, since the system manages the producer/consumer issue, both proceed per usual, and the system provides automatic blocking as required.

Using unnamed pipes in a UNIX environment normally involves several steps:

· Create the pipe(s) needed
· Generate the child process(es)
· Close/duplicate the file descriptors to associate the ends of the pipe
· Close the unused end(s) of the pipe(s)
· Perform the communication
· Close the remaining file descriptors
· Wait for the child process to terminate

The pipe function has the following prototype:

#include <unistd.h>
int pipe(int file_descriptor[2]);

Pipe is passed (a pointer to) an array of two integer file descriptors. It fills the array with two new file descriptors and returns a zero.

The two file descriptors returned are connected in a special way. Any data written to file_descriptor[1] can be read back from file_descriptor[0]. The data is processed in a *first in, first out* basis, usually abbreviated to *FIFO*. This means that if you write the bytes 1, 2, 3 to file_descriptor[1], reading from file_descriptor[0] will produce 1, 2, 3. This is different from a stack, which operates on a *last in, first out* basis, usually abbreviated to *LIFO*.

Assignment 14: Write a c Program(single process) to create unnamed pipe pass data through that pipe

See the following program how pipe work

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{
int res,pipes[2];
char data[1024];
char buffer[1024];
```

```
printf("enter data\n");
scanf("%s",&data);
memset(buffer, '\0', sizeof(buffer));
if(pipe(pipes) == 0) {
res=write(pipes[1], data, strlen(data));
printf("Wrote %d bytes\n",res);
res=read(pipes[0], buffer,1024);
printf("Read %d bytes: %s\n",res,buffer);
exit(EXIT_SUCCESS);
}
exit(EXIT_FAILURE);
}
```

The program creates a pipe using the two file descriptors in the array pipes[ ]. It then writes data into the pipe using the file descriptor pipes[1] and reads it back from pipes[0]. Notice that the pipe has some internal buffering that stores the data in between the calls to write and read.

pipe(pipes) create the pipe and on successful creation it return 0, that's why

`if(pipe(pipes) == 0) condition satisfied and execute the statement.`

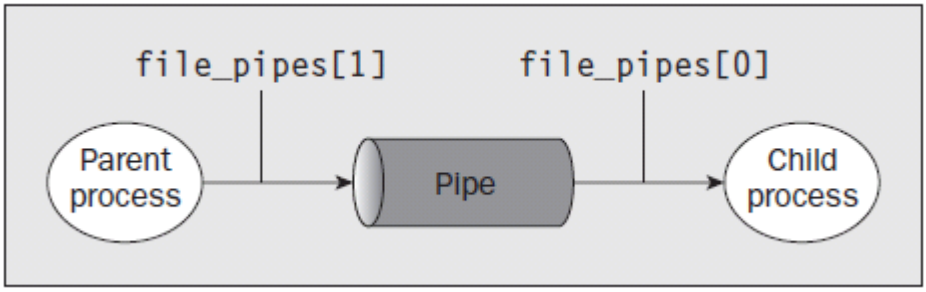
It's important to realize that these are file descriptors, not file streams, so you must use the lowerlevel read *and* write *system calls to access the data, rather than the stream library functions* fread *and* fwrite.

Assignment 15: Write a c Program which will create child process using fork ( ) and Parent process and Child process will pass data through unnamed pipe that pipe

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{
int res;
int pipes[2];
char msg[1024];
char buffer[BUFSIZ+1];
pid_t pid;
printf("Enter Data\n");
scanf("%s",&msg);
memset(buffer,'\0',sizeof(buffer));
if(pipe(pipes)==0)
{
pid=fork();
if (pid==-1)
{
fprintf(stderr, "Fork failure");
exit(EXIT_FAILURE);
}
if(pid==0)
{
res= read(pipes[0], buffer, BUFSIZ);
printf("processid %d Read %d bytes: %s from parent process
%d\n",getpid(),res, buffer,getppid());
exit(EXIT_SUCCESS);
```

```
}
else
{
res=write(pipes[1],msg,strlen(msg));
printf("Processid %d Wrote %d bytes\n",getpid(),res);
sleep(10);
}
}
exit(EXIT_SUCCESS);
}
```

First, the program creates a pipe with the pipe call. It then uses the fork call to create a new process. If the fork was successful, the parent writes data into the pipe, while the child reads data from the pipe. Both parent and child exit after a single write and read. If the parent exits before the child, you might see the shell prompt between the two outputs.
Although the program is superficially very similar to the first pipe example, we've taken a big step forward by being able to use separate processes for the reading and writing, as illustrated in Figure.



1.3 Named pipe

Named pipes can be created on the UNIX command line using mknod, but it is more interesting to look at how they can be used programatically. The mknod() system call, usable only by the superuser, takes a path, access permissions, and a device (typically unused) as parameters and creates a pipe referred to by the user-specified path. Often, mkfifo() will be provided as an additional call that can be used by all users but is only capable of making FIFO pipes.
FIFOs, often referred to as *named pipes*. A named pipe is a special type of file (remember that everything in Linux is a file!) that exists as a name in the file system but behaves like the unnamed pipes that you've met already.
You can create named pipes from the command line and from within a program.
$ **mkfifo *filename***
From inside a program, you can use following calls:
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *filename, mode_t mode);

Write a C Program to create named pipe
Creating a named FIFO

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

```
#include <sys/stat.h>
int main()
{
int res;
res=mkfifo("fifo2", 0755);
if (res == 0)
{
printf("FIFO created\n");
system("ls -l fifo2");
}
exit(EXIT_SUCCESS);
}
```

The program uses the mkfifo function to create a special file fifo2 and if path is not specified it is created in the current directory and on successful creation it return 0.
To check the created pipe fifo2 the following system call has been used;

```
system("ls -l fifo2");
```

Although you ask for a mode of 0777, this is altered by the user mask (umask) setting (in this case 022), just as in normal file creation, so the resulting file has mode 755.

Accessing a FIFO File

1. First, try reading the (empty) FIFO:

$ **cat < fifo2**

2. Now try writing to the FIFO. You will have to use a different terminal because the first command will now be hanging, waiting for some data to appear in the FIFO.

$ **echo "Hello World" > fifo2**

You will see the output appear from the cat command. If you don't send any data down the FIFO, the cat command will hang until you interrupt it, conventionally with Ctrl+C.

3. You can do both at once by putting the first command in the background:

$ **cat < fifo2 &**

[1] 1316

$ **echo "Hello World" > fifo2**

Hello World

[1]+ Done cat < fifo2

$

Because there was no data in the FIFO, the cat and echo programs both block, waiting for some data to arrive and some other process to read the data, respectively.
Looking at the third stage, the cat process is initially blocked in the background. When echo makes some data available, the cat command reads the data and prints it to the standard output.

Opening a FIFO with open

This is quite a sensible restriction because, normally, you use a FIFO only for passing data in a single direction, so there is no need for an O_RDWR mode. A process would read its own output back from a pipe if it were opened read/write.
If you do want to pass data in both directions between programs, it's much better to use either a pair of FIFOs or pipes, one for each direction, or (unusually) explicitly change the direction of the data flow by closing and reopening the FIFO.

The other difference between opening a FIFO and a regular file is the use of the open_flag (the second parameter to open) with the option O_NONBLOCK. Using this open mode not only changes how the open call is processed, but also changes how read and write requests are processed on the returned file descriptor.
There are four legal combinations of O_RDONLY, O_WRONLY, and the O_NONBLOCK flag. We'll consider each in turn.
open(const char *path, O_RDONLY);
In this case, the open call will block; it will not return until a process opens the same FIFO for writing.
open(const char *path, O_RDONLY | O_NONBLOCK);
The open call will now succeed and return immediately, even if the FIFO has not been opened for writing by any process.
open(const char *path, O_WRONLY);
In this case, the open call will block until a process opens the same FIFO for reading.
Open (const char *path, O_WRONLY | O_NONBLOCK);
This will always return immediately, but if no process has the FIFO open for reading, open will return an error, –1, and the FIFO won't be opened. If a process does have the FIFO open for reading, the file descriptor returned can be used for writing to the FIFO.

Assignment16: Write a program which will write on pipe and another program to read data from that pipe

Program for writing data (Producer)

```
#include<unistd.h>
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<fcntl.h>
#include<sys/types.h>
#include<sys/stat.h>
int main()
{
int pipe_fd;
int res;
int s=0;
char buffer[1024];
printf("Enter data\n");
fgets(buffer,1024,stdin);
res = mkfifo("fifo3", 0777);

printf("Process %d opening FIFO O_WRONLY\n", getpid());
pipe_fd=open("fifo3",O_WRONLY);
printf("Process %d result %d\n", getpid(), pipe_fd);
res=write(pipe_fd, buffer, sizeof(buffer));
(void)close(pipe_fd);

unlink("fifo3");
printf("Process %d finished\n", getpid());
exit(EXIT_SUCCESS);
}
```

The following fget() function will take input data from the keyboard and store it in buffer.

```
fgets(buffer,1024,stdin);
```

The following statement mkfifo()system call will create pipe named fifo3 with permission 777 and successful creation of pipe will return 0.

```
res = mkfifo("fifo3", 0777);
```

The next open ( ) system call open the created pipe fifo3 with write only and block option and the O_WRONLY is defined in <fcntl.h> header file. The open ( ) system return the value pointing to file descriptor pipe_fd.

```
pipe_fd=open("fifo3",O_WRONLY);
```

The next write system call write ( ) will write data from buffer to file descriptor pipe_fd with buffer size and return the no of character written.

```
res = write(pipe_fd, buffer, sizeof(buffer));
```

Finally the following close( ) system call close the file descriptor pipe_fd and unlink the created fifo3.

```
(void)close(pipe_fd);
unlink("fifo3");
```

But when you run this program the named pipe fifo3 will simply hang as long as the other program read the data of the fifo3.

Program for reading data (consumer)

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
int main()
{
int fd;
int r,l;
char buffer[1024];
printf("Process %d opening FIFO O_RDONLY\n", getpid());
fd=open("fifo3",O_RDONLY);
printf("Process %d result %d\n",getpid(),fd);
r=read(fd, buffer, sizeof(buffer));
l=strlen(buffer)-1;
(void)close(fd);
unlink("fifo3");
printf("Process %d finished, with reading %s of %d bytes\n",
getpid(),buffer,l);
exit(EXIT_SUCCESS);
}
```

Here the open ( ) system call open the created pipe fifo3 with read only and block option and the O_RDONLY is defined in <fcntl.h> header file. The open ( ) system return the value pointing to file descriptor pipe_fd.

```
fd=open("fifo3",O_RDONLY);
```

The next read system call read ( ) will read data from file descriptor fd with buffer size to buffer and return the no of character read.

```
res = write(fd, buffer, sizeof(buffer));
```

The following strlen( )function will count the no of character read in buffer and store in variable l.

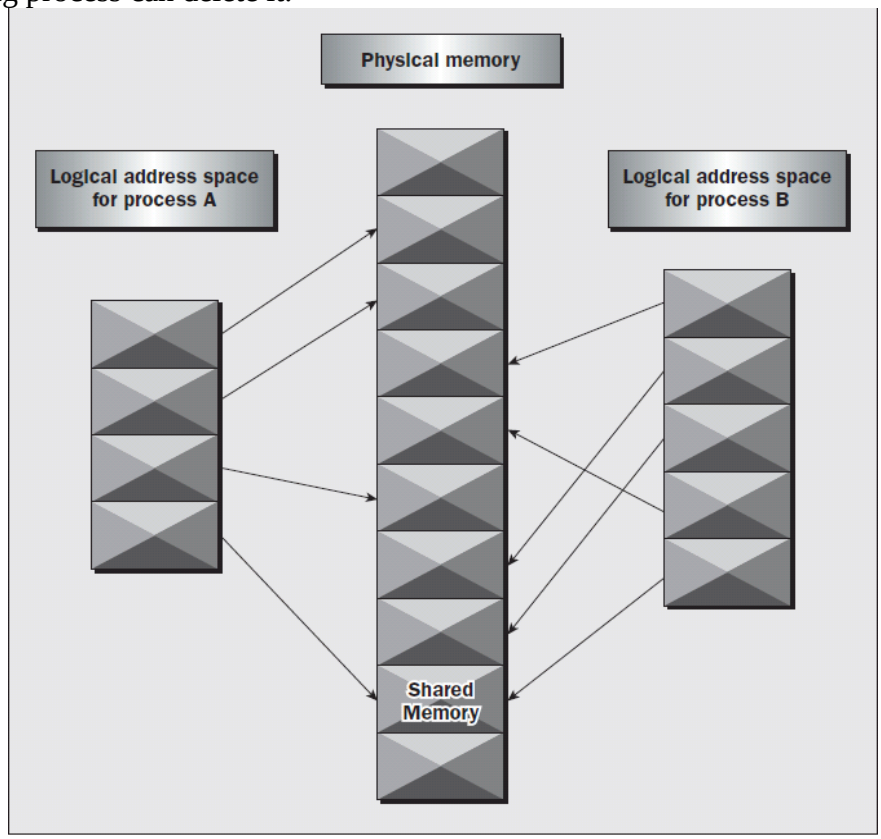
```
l=strlen(buffer)-1;
```

1.4 Shared-Memory Systems

- Inter process communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment.
- Other processes that wish to communicate using this shared-memory segment must attach it to their address space.
- Recall that, normally, the OS tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas.
- The form of the data and the location are determined by these processes and are not under the OS's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.
- To illustrate the concept of cooperating processes, let's consider the **producer-consumer problem**, which is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process.
- One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.
- Two types of buffers can be used.
    1. The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.
    2. The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Shared memory is unique in that it allows random access of data, whereas most other IPC mechanisms mandate sequential access to data.
Typically, one process creates a shared segment, and needs to set access permissions for the segment.
It is then mapped into the process's address space after the process attaches to it.
Usually the creating process initializes the memory, but after this other processes that have been given access permission to the segment can use it freely. When another process does use it, it is append into its address space.
Oftentimes semaphores are used by the original process to be sure that only one other process is allowed to access the memory at any one time, although a readers/writers lock

may be a more suitable solution. When no process needs the memory segment anymore, the creating process can delete it.



Creating a Shared Memory Segment

When programming for UNIX, a new shared memory segment can be created using the shmget() call. Using shmget() also generates the required data structures.

int shmget(key_t key, size_t size, int shmflg);

Program provides key, which effectively names the shared memory segment, and the shmget function returns a shared memory identifier that is used in subsequent shared memory functions. There's a special key value, IPC_PRIVATE, that creates shared memory private to the process.

You wouldn't normally use this value, and you may find the private shared memory is not actually private on some Linux systems.

The second parameter, size, specifies the amount of memory required in bytes.

The third parameter, shmflg, consists of nine permission flags that are used in the same way as the mode flags for creating files. A special bit defined by IPC_CREAT must be bitwise ORed with the permissions to create a new shared memory segment. It's not an error to have the IPC_CREAT flag set and pass the key of an existing shared memory segment. The IPC_CREAT flag is silently ignored if it is not required.

The permission flags are very useful with shared memory because they allow a process to create shared memory that can be written by processes owned by the creator of the shared memory, but only read by processes that other users have created. You can use this to

provide efficient read-only access to data by placing it in shared memory without the risk of its being changed by other users.
If the shared memory is successfully created, shmget returns a nonnegative integer, the shared memory identifier. On failure, it returns –1.

Enable access to shared memory

When you first create a shared memory segment, it's not accessible by any process. To enable access to the shared memory, you must attach it to the address space of a process. You do this with the shmat function:

void *shmat(int shm_id, const void *shm_addr, int shmflg);

The first parameter, shm_id, is the shared memory identifier returned from shmget.
The second parameter, shm_addr, is the address at which the shared memory is to be attached to the current process. This should almost always be a null pointer, which allows the system to choose the address at which the memory appears.
The third parameter, shmflg, is a set of bitwise flags. The two possible values are SHM_RND, which, in conjunction with shm_addr, controls the address at which the shared memory is attached, and SHM_RDONLY, which makes the attached memory read-only. It's very rare to need to control the address at which shared memory is attached; you should normally allow the system to choose an address for you, because doing otherwise will make the application highly hardware-dependent.
If the shmat call is successful, it returns a pointer to the first byte of shared memory. On failure –1 is returned.
The shared memory will have read or write access depending on the owner (the creator of the shared memory), the permissions, and the owner of the current process. Permissions on shared memory are similar to the permissions on files.
An exception to this rule arises if shmflg & SHM_RDONLY is true. Then the shared memory won't be writable, even if permissions would have allowed write access.

Detaching the shared memory

The shmdt function detaches the shared memory from the current process. It takes a pointer to the address returned by shmat. On success, it returns 0, on error –1. Note that detaching the shared memory doesn't delete it; it just makes that memory unavailable to the current process.

Control functions for shared memory

The control functions for shared memory are (thankfully) somewhat simpler than the more complex ones for semaphores:

int shmctl(int shm_id, int command, struct shmid_ds *buf);

The shmid_ds structure has at least the following members:

```
struct shmid_ds {
uid_t shm_perm.uid;
uid_t shm_perm.gid;
mode_t shm_perm.mode;
}
```

The first parameter, shm_id, is the identifier returned from shmget.
The second parameter, command, is the action to take. It can take three values, shown in the following table.

Command	Description
IPC_STAT	Sets the data in the shmid_ds structure to reflect the values associated with the shared memory.
IPC_SET	Sets the values associated with the shared memory to those provided in the shmid_ds data structure, if the process has permission to do so.
IPC_RMID	Deletes the shared memory segment.

The third parameter, buf, is a pointer to the structure containing the modes and permissions for the shared memory.
Now we will write one program to write data on shared memory and read data from shared memory.

Assignment 17: Write a consumer program which will create shared memory to communicate with another producer program using the same shared memory and Producer program will write to shared memory and the consumer program will read data from that share memory.
First create a common header file to describe the shared memory you want to pass around. Call this shm_com.h:

```
struct sm {
int wr;
char st[2048];
};
```

This defines a structure to use in both the consumer and producer programs. You use an int flag wr to tell the consumer when data has been written to the rest of the structure and arbitrarily decide that you need to transfer up to 2k of text by declaring char st[2048].

Now you write the consumer program and include the header file “shm_com.h”

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/shm.h>
#include "shm_com.h"
#include<sys/types.h>
int main()
{
int running = 1;
void *shared_memory = (void *)0;
struct sm *sh;
int shmid,1;
shmid = shmget((key_t)34, sizeof(struct sm), 0666 | IPC_CREAT);
printf("%d",shmid);
shared_memory = shmat(shmid, (void *)0, 0);
```

```
printf("Memory attached at %X\n", (int)shared_memory);
sh= (struct sm *)shared_memory;
sh->wr = 0;
while(running)
{
if (sh->wr)
{
printf("You wrote: %s in processid %d\n ", sh->st,getpid());
sleep(1);
sh->wr = 0;
if(strncmp(sh->st, "end", 3) == 0)
{
running = 0;
}
}
}
shmctl(shmid,IPC_RMID,0);
exit(EXIT_SUCCESS);
}
```

The following statement will create the share memory and return an integer type identifier of the shared memory to variable shmid.

```
shmid = shmget((key_t)34, sizeof(struct sm), 0666 | IPC_CREAT);
```

The program provides key, which effectively names the shared memory segment. There's a special key value, IPC_PRIVATE, that creates shared memory private to the process.
The second parameter, size, specifies the amount of memory required in bytes i.e to the size of the structure named sm.
The third parameter, special bit defined by IPC_CREAT must be bitwise ORed with the permissions to create a new shared memory segment. The permission flags are very useful with shared memory because they allow a process to create shared memory that can be written by processes owned by the creator of the shared memory, but only read by processes that other users have created. You can use this to provide efficient read-only access to data by placing it in shared memory without the risk of its being changed by other users. If the shared memory is successfully created, shmget returns a nonnegative integer, the shared memory identifier. On failure, it returns –1.

Now the following shmat system call will attach the created shared memory identified by shmid to the executing process's address space.

```
shared_memory = shmat(shmid, (void *)0, 0);
```

The second parameter, (void *)0, is the address at which the shared memory is to be attached to the current process. This should almost always be a null pointer, which allows the system to choose the address at which the memory appears.
The third parameter, 0, is a set of bitwise flags and controls the address at which the shared memory will be attached.

```
sh= (struct sm *)shared_memory;
```

The shared memory address is bind to the structure size and address size is return to sh which is pointer to a structure sm.
Sh->wr=0 ;

This statement will be used to set the condition that unless and until any thing is written on the shared memory by Producer program the consumer program will have noting to read and that's why the program will to read.

```
while(running)
{
if (sh->wr)
{
printf("You wrote: %s in processid %d\n ", sh->st,getpid());
sleep(1);
sh->wr = 0;
if(strncmp(sh->st, "end", 3) == 0)
{
running = 0;
}
}
}
```

This section of code is used for infinite loop to read data in the shared memory as long as the "end" string come.

Finally the following statement will delete the shared memory and exit successfully from the program.

```
shmctl(shmid,IPC_RMID,0)
```

Now in another terminal you write the Producer program and also here you include the same "shm_com.h" header file to use the same structure.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/shm.h>
#include<sys/types.h>
#include "shm_com.h"
int main()
{
int running = 1;
void *shared_memory = (void *)0;
struct sm *sh;
char buffer[2048];
int shmid;
shmid = shmget((key_t)34, sizeof(struct sm), 0666 | IPC_CREAT);
printf("%d",shmid);
shared_memory = shmat(shmid, (void *)0, 0);
printf("Memory attached at %X\n", (int)shared_memory);
sh = (struct sm *)shared_memory;
while(running)
{
while(sh->wr == 1)
{
sleep(1);
printf("waiting for client...\n");
}
printf("Enter some text from process id %d: ",getpid());
fgets(buffer, 2048, stdin);
strncpy(sh->st, buffer, 2048);
sh->wr = 1;
if (strncmp(buffer, "end", 3) == 0)
```

```
{
running = 0;
}
}
exit(EXIT_SUCCESS);
}
```

The following statement will create the share memory and return an integer type identifier of the shared memory to variable shmid and the same shared memory is created with same key value so it will return the same identifier like the producer consumer.

```
shmid = shmget((key_t)34, sizeof(struct sm), 0666 | IPC_CREAT);
```

Then the following statements shared memory will be attached to the consumer process's address space and that address space will be b
ind to sh which is pointer to a structure sm.

```
shared_memory = shmat(shmid, (void *)0, 0);
printf("Memory attached at %X\n", (int)shared_memory);
sh = (struct sm *)shared_memory;
```

The following section of loop will run infinitely as long as the "end string is written on the Producer process. If sh->wr==1 then only it will write.

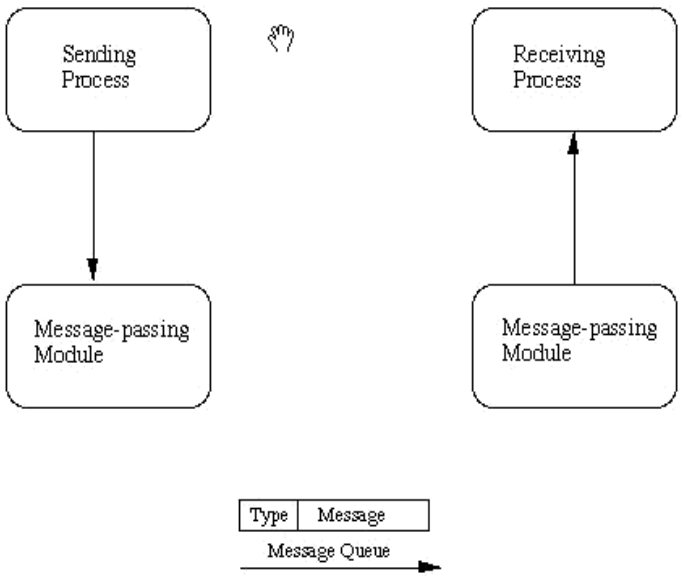
```
while(running)
{
while(sh->wr == 1)
{
sleep(1);
printf("waiting for client...\n");
}
printf("Enter some text from process id %d: ",getpid());
fgets(buffer, 2048, stdin);
strncpy(sh->st, buffer, 2048);
sh->wr = 1;
if (strncmp(buffer, "end", 3) == 0)
{
running = 0;
}
}
```

Now you can run the two separate program in two separate terminal and communicate.

Message queue

The basic idea of a ***message queue*** is a simple one.

Two (or more) processes can exchange information via access to a common system message queue. The ***sending*** process places via some (OS) message-passing module a message onto a queue which can be read by another process (Figure). Each message is given an identification or type so that processes can select the appropriate message. Process must share a common key in order to gain access to the queue in the first place.



Basic Message Passing IPC messaging lets processes send and receive messages, and queue messages for processing in an arbitrary order. Unlike the file byte-stream data flow of pipes, each IPC message has an explicit length. Messages can be assigned a specific type. Because of this, a server process can direct message traffic between clients on its queue by using the client process PID as the message type. For single-message transactions, multiple server processes can work in parallel on transactions sent to a shared message queue.

Before a process can send or receive a message, the queue must be initialized (through the msgget function see below) Operations to send and receive messages are performed by the msgsnd() and msgrcv() functions, respectively.

When a message is sent, its text is copied to the message queue. The msgsnd() and msgrcv() functions can be performed as either blocking or non-blocking operations. Non-blocking operations allow for asynchronous message transfer -- the process is not suspended as a result of sending or receiving a message. In blocking or synchronous message passing the sending process cannot continue until the message has been transferred or has even been acknowledged by a receiver. IPC signal and other mechanisms can be employed to implement such transfer. A blocked message operation remains suspended until one of the following three conditions occurs:

- The call succeeds.
- The process receives a signal.
- The queue is removed.

Message queues provide a reasonably easy and efficient way of passing data between two unrelated processes. They have the advantage over named pipes that the message queue exists independently of both the sending and receiving processes, which removes some of the difficulties that occur in synchronizing the opening and closing of named pipes.

Message queues provide a way of sending a block of data from one process to another. Additionally, each block of data is considered to have a type, and a receiving process may receive blocks of data having different type values independently. The good news is

that you can almost totally avoid the synchronization and blocking problems of named pipes by sending messages. Even better, you can "look ahead" for messages that are urgent in some way. The bad news is that, just like pipes, there's a maximum size limit imposed on each block of data and also a limit on the maximum total size of all blocks on all queues throughout the system.

msgget

You create and access a message queue using the msgget function:

int msgget(key_t key, int msgflg);

The program must provide a key value that, as with other IPC facilities, names a particular message queue.

The special value IPC_PRIVATE creates a private queue, which in theory is accessible only by the current process. A special bit defined by IPC_CREAT must be bitwise ORed with the permissions to create a new message queue. It's not an error to set the IPC_CREAT flag and give the key of an existing message queue. The IPC_CREAT flag is silently ignored if the message queue already exists.

The msgget function returns a positive number, the queue identifier, on success or –1 on failure.

msgsnd

The msgsnd function allows you to add a message to a message queue:

int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);

The structure of the message is constrained in two ways. First, it must be smaller than the system limit, and second, it must start with a long int, which will be used as a message type in the receive function.

When you're using messages, it's best to define your message structure something like this:

```
struct my_message {
long int message_type;
/* The data you wish to transfer */
}
```

Because the message_type is used in message reception, you can't simply ignore it. You must declare your data structure to include it, and it's also wise to initialize it so that it contains a known value.

The first parameter, msqid, is the message queue identifier returned from a msgget function.

The second parameter, msg_ptr, is a pointer to the message to be sent, which must start with a long int type as described previously.

The third parameter, msg_sz, is the size of the message pointed to by msg_ptr. This size must not include the long int message type.

The fourth parameter, msgflg, controls what happens if either the current message queue is full or the systemwide limit on queued messages has been reached. If msgflg has the IPC_NOWAIT flag set, the function will return immediately without sending the message and the return value will be –1.

If the msgflg has the IPC_NOWAIT flag clear, the sending process will be suspended, waiting for space to become available in the queue.
On success, the function returns 0, on failure –1. If the call is successful, a copy of the message data has been taken and placed on the message queue.

msgrcv
The msgrcv function retrieves messages from a message queue:
int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);
The first parameter, msqid, is the message queue identifier returned from a msgget function.

The second parameter, msg_ptr, is a pointer to the message to be received, which must start with a long int type as described previously in the msgsnd function.

The third parameter, msg_sz, is the size of the message pointed to by msg_ptr, not including the long int message type.

The fourth parameter, msgtype, is a long int, which allows a simple form of reception priority to be implemented. If msgtype has the value 0, the first available message in the queue is retrieved. If it's greater than zero, the first message with the same message type is retrieved. If it's less than zero, the first message that has a type the same as or less than the absolute value of msgtype is retrieved.

The fifth parameter, msgflg, controls what happens when no message of the appropriate type is waiting to be received. If the IPC_NOWAIT flag in msgflg is set, the call will return immediately with a return value of –1. If the IPC_NOWAIT flag of msgflg is clear, the process will be suspended, waiting for an appropriate type of message to arrive.
On success, msgrcv returns the number of bytes placed in the receive buffer, the message is copied into the user-allocated buffer pointed to by msg_ptr, and the data is deleted from the message queue. It returns –1 on error.

msgctl
The final message queue function is msgctl, which is very similar to that of the control function for shared memory:
int msgctl(int msqid, int command, struct msqid_ds *buf);
The msqid_ds structure has at least the following members:
struct msqid_ds {
uid_t msg_perm.uid;
uid_t msg_perm.gid
mode_t msg_perm.mode;
}
The first parameter, msqid, is the identifier returned from msgget.
The second parameter, command, is the action to take. It can take three values, described in the following.

Command	Description
IPC_STAT	Sets the data in the msqid_ds structure to reflect the values associated with the message queue.
IPC_SET	If the process has permission to do so, this sets the values associated with the message queue to those provided in the msqid_ds data structure.
IPC_RMID	Deletes the message queue.

Assignment 19: Write a C program using message queue to send and receive data between two unrelated processes.

When you're using messages, it's best to define your message structure something like this:
You create a header file writing this structure in and include that header file both in sender and receiver program. Let the header file name is "mq_com.h"

```
struct{
long int ty;
char text[1024];
}data;
```

Now you write the receiver program

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/msg.h>
#include "mq_com.h"
int main()
{
int running = 1;
int msgid,l,m;
long int rc= 0;
msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
while(running)
{
l=msgrcv(msgid, (void *)&data, 1024,rc, 0);
printf("You wrote: %s", data.text);
if (strncmp(data.text, "end", 3) == 0)
{
running = 0;
}
}
m=msgctl(msgid, IPC_RMID, 0);
exit(EXIT_SUCCESS);
}
```

In the statement
long int rc=0
initialize the message type that the receiver will receive from the message queue. If msgtype has the value 0, the first available message in the queue is retrieved.

Then the message queue has been created by the following statement

```
msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
```

The 1st parameter will give private key value that names a particular message queue.
The 2nd parameter, a special bit defined by IPC_CREAT must be bitwise ORed with the permissions to create a new message queue.
The msgget function returns a positive number, the queue identifier, stored in msgid, on success or –1 on failure.

Now within infinite while loop you receive the message using msgrcv function

```
l=msgrcv(msgid, (void *)&data, 1024,rc, 0);
```

The 1st parameter msgid is the identifier to th Message queue from where the data will be retrieved.
The 2nd parameter is address of name of the structure.
3rd parameter is the size of the message queue.
4th parameter is the type of the message queue.
And finally we pass 0 as the 5th parameter for IPC_NOWAIT flag of msgflg is clear, the process will be suspended, waiting for an appropriate type of message to arrive.

After that you print whatever you write and use a terminator for the infinite loop.

After receiving the data the message queue will be deleted by following statement

```
m=msgctl(msgid, IPC_RMID, 0);
```

Now if you run this program the receiver program will simply wait (hang on) for data to be sent to the identified message queue by another program.

Now in another terminal you write the sender program

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/msg.h>
#include "mq_com.h"
#define MAX 512
int main()
{
int running = 1;
int msgid,l;
char buffer[1024];
msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
while(running) {
printf("Enter some text:\n" );
```

```
fgets(buffer, 1024, stdin);
data.ty = 1;
strcpy(data.text, buffer);
l=msgsnd(msgid, (void *)&data, MAX, 0);
if (strncmp(buffer, "end", 3) == 0)
{
running = 0;
}
}
exit(EXIT_SUCCESS);
}
```

The same way you can create the message queue with same identifier same as the sender program using msgget function.
And within the infinite while loop you enter the data and copy it to the message queue and use a terminator for the infinite loop.