CEF 427: ADVANCED OPERATING SYSTEMS

Chapter 2: Elements of IPC

I. Related System Calls (System V)

Following table lists the various System calls along with their description.

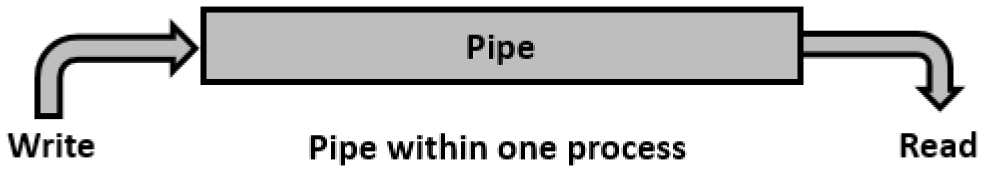
Category	System Call	Description
General	open ()	This system call either opens an already existing file or creates and opens a new file.
General	creat ()	Creates and opens a new file.
General	read ()	Reads the contents of the file into the required buffer.
General	write ()	Writes the contents of buffer into the file.
General	close ()	Closes the file descriptor.
General	stat ()	Provides information on the file.
Pipes	pipe ()	Creates pipe for communication which returns two file descriptors for reading and writing.
Named Pipes or Fifo	mknod ()	Creates a memory device file or special file to create FIFOs
Named Pipes or Fifo	mkfifo ()	Creates a new FIFO
Shared Memory	shmget ()	Creates a new shared memory segment or gets the identifier of the existing segment.
Shared Memory	shmat ()	Attaches the shared memory segment and makes the segment a part of the virtual memory of the calling process.
Shared Memory	shmdt ()	Detaches the shared memory segment.
Shared Memory	shmctl ()	Performs control operations for the shared memory. Few of the generic control operations for the shared memory are removing the shared memory segment (IPC_RMID), receiving the information of the shared memory (IPC_STAT) and updating new values of the existing shared memory (IPC_SET).
Message Queues	msgget ()	Creates a new message queue or accesses an already existing message queue and gets the handle or identifier to perform operations with regard to message queue, such as sending message/s to queue and receiving message/s from the queue.

Message Queues	msgsnd ()	Sends a message to the required message queue with the required identification number.
Message Queues	msgrcv ()	Receives a message from the message queue. By default, this is infinite wait operation, means the call will be blocked until it receives a message.
Message Queues	msgctl ()	Performs control operations for the message queue. Few of the generic control operations for the message queue are removing the message queue (IPC_RMID), receiving the information of the message queue (IPC_STAT) and updating new values of the existing message queue (IPC_SET).
Semaphores	semget ()	Creates a new semaphore or gets the identifier of the existing semaphore. Semaphores are used to perform synchronization between various IPCs working on the same object.
Semaphores	semop ()	Performs semaphore operations on semaphore values. The basic semaphore operations are either acquiring or releasing the lock on the semaphore.
Semaphores	semctl ()	Performs control operations for the semaphore. Few of the generic control operations for the semaphore are removing the semaphore (IPC_RMID), receiving the information of the semaphore (IPC_STAT) and updating new values of the existing semaphore (IPC_SET).
Signals	signal ()	Setting the disposition of the signal (signal number) and the signal handler. In other terms, registering the routine, which gets executed when that signal is raised.
Signals	sigaction ()	Same as signal(), setting the disposition of the signal i.e., performing certain action as per the registered signal handler after the receipt of the registered signal. This system call supports finer control over signal() such as blocking certain signals, restoring signal action to the default state after calling the signal handler, providing information such as consumed time of the user and the system, process id of sending process, etc.
Memory Mapping	mmap ()	Mapping files into the memory. Once mapped into the memory, accessing files is as easy as accessing data using addresses and also in this way, the call is not expensive as system calls.
Memory Mapping	munmap ()	Un-mapping the mapped files from the memory

II. Pipes

Pipe is a communication medium between two or more related or interrelated processes. It can be either within one process or a communication between the child and the parent processes. Communication can also be multi-level such as communication between the parent, the child and the grand-child, etc. Communication is achieved by one process writing into the pipe and other reading from the pipe. To achieve the pipe system call, create two files, one to write into the file and another to read from the file.

Pipe mechanism can be viewed with a real-time scenario such as filling water with the pipe into some container, say a bucket, and someone retrieving it, say with a mug. The filling process is nothing but writing into the pipe and the reading process is nothing but retrieving from the pipe. This implies that one output (water) is input for the other (bucket).



#include<unistd.h>

```
int pipe(int pipedes[2]);
```

This system call would create a pipe for one-way communication i.e., it creates two descriptors, first one is connected to read from the pipe and other one is connected to write into the pipe.

Descriptor pipedes[0] is for reading and pipedes[1] is for writing. Whatever is written into pipedes[1] can be read from pipedes[0].

This call would return zero on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

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

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Even though the basic operations for file are read and write, it is essential to open the file before performing the operations and closing the file after completion of the required operations. Usually, by default, 3 descriptors opened for every process, which are used for input (standard input – stdin), output (standard output – stdout) and error (standard error – stderr) having file descriptors 0, 1 and 2 respectively.

This system call would return a file descriptor used for further file operations of read/write/seek (lseek). Usually file descriptors start from 3 and increase by one number as the number of files open.

The arguments passed to open system call are pathname (relative or absolute path), flags mentioning the purpose of opening file (say, opening for read, O_RDONLY, to write, O_WRONLY, to read and write, O_RDWR, to append to the existing file O_APPEND, to create file, if not exists with O_CREAT and so on) and the required mode providing permissions of read/write/execute for user or owner/group/others. Mode can be mentioned with symbols.

Read – 4, Write – 2 and Execute – 1.

For example: Octal value (starts with 0), 0764 implies owner has read, write and execute permissions, group has read and write permissions, other has read permissions. This can also be represented as S_IRWXU | S_IRGRP | S_IWGRP | S_IROTH, which implies or operation of 0700|0040|0020|0004 → 0764.

This system call, on success, returns the new file descriptor id and -1 in case of error. The cause of error can be identified with errno variable or perror() function.

```
#include<unistd.h>

int close(int fd)
```

The above system call closing already opened file descriptor. This implies the file is no longer in use and resources associated can be reused by any other process. This system call returns zero on success and -1 in case of error. The cause of error can be identified with errno variable or perror() function.

```
#include<unistd.h>

ssize_t read(int fd, void *buf, size_t count)
```

The above system call is to read from the specified file with arguments of file descriptor fd, proper buffer with allocated memory (either static or dynamic) and the size of buffer.

The file descriptor id is to identify the respective file, which is returned after calling open() or pipe() system call. The file needs to be opened before reading from the file. It automatically opens in case of calling pipe() system call.

This call would return the number of bytes read (or zero in case of encountering the end of the file) on success and -1 in case of failure. The return bytes can be smaller than the number of bytes requested, just in case no data is available or file is closed. Proper error number is set in case of failure.

To know the cause of failure, check with errno variable or perror() function.

#include<unistd.h>

```
ssize_t write(int fd, void *buf, size_t count)
```

The above system call is to write to the specified file with arguments of the file descriptor fd, a proper buffer with allocated memory (either static or dynamic) and the size of buffer.

The file descriptor id is to identify the respective file, which is returned after calling open() or pipe() system call.

The file needs to be opened before writing to the file. It automatically opens in case of calling pipe() system call.

This call would return the number of bytes written (or zero in case nothing is written) on success and -1 in case of failure. Proper error number is set in case of failure.

To know the cause of failure, check with errno variable or perror() function.

Example Programs

Following are some example programs.

Example program 1 − Program to write and read two messages using pipe.

Algorithm

Step 1 − Create a pipe.

Step 2 − Send a message to the pipe.

Step 3 − Retrieve the message from the pipe and write it to the standard output.

Step 4 − Send another message to the pipe.

Step 5 − Retrieve the message from the pipe and write it to the standard output.

Note − Retrieving messages can also be done after sending all messages.

Source Code: simplepipe.c

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

int main() {
   int pipefds[2];
   int returnstatus;
   char writemessages[2][20]={"Hi", "Hello"};
   char readmessage[20];
   returnstatus = pipe(pipefds);

   if (returnstatus == -1) {
      printf("Unable to create pipe\n");
      return 1;
   }

   printf("Writing to pipe - Message 1 is %s\n", writemessages[0]);
   write(pipefds[1], writemessages[0], sizeof(writemessages[0]));
   read(pipefds[0], readmessage, sizeof(readmessage));
   printf("Reading from pipe – Message 1 is %s\n", readmessage);
   printf("Writing to pipe - Message 2 is %s\n", writemessages[0]);
   write(pipefds[1], writemessages[1], sizeof(writemessages[0]));
   read(pipefds[0], readmessage, sizeof(readmessage));
   printf("Reading from pipe – Message 2 is %s\n", readmessage);
   return 0;
}
```

Note − Ideally, return status needs to be checked for every system call. To simplify the process, checks are not done for all the calls.

Execution Steps

Compilation

```
gcc -o simplepipe simplepipe.c
```

Execution/Output

```
Writing to pipe - Message 1 is Hi
Reading from pipe – Message 1 is Hi
Writing to pipe - Message 2 is Hi
Reading from pipe – Message 2 is Hell
```

Example program 2 − Program to write and read two messages through the pipe using the parent and the child processes.

Algorithm

Step 1 − Create a pipe.

Step 2 − Create a child process.

Step 3 − Parent process writes to the pipe.

Step 4 − Child process retrieves the message from the pipe and writes it to the standard output.

Step 5 − Repeat step 3 and step 4 once again.

Source Code: pipewithprocesses.c

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

int main() {
   int pipefds[2];
   int returnstatus;
   int pid;
   char writemessages[2][20]={"Hi", "Hello"};
   char readmessage[20];
   returnstatus = pipe(pipefds);
   if (returnstatus == -1) {
      printf("Unable to create pipe\n");
      return 1;
   }
```

Execution/Output

```
Writing to pipe - Message 1 is Hi
Reading from pipe – Message 1 is Hi
Writing to pipe - Message 2 is Hi
Reading from pipe – Message 2 is Hell
```

Example program 2 − Program to write and read two messages through the pipe using the parent and the child processes.

Algorithm

Step 1 − Create a pipe.

Step 2 − Create a child process.

Step 3 − Parent process writes to the pipe.

Step 4 − Child process retrieves the message from the pipe and writes it to the standard output.

Step 5 − Repeat step 3 and step 4 once again.

Source Code: pipewithprocesses.c

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

int main() {
   int pipefds[2];
   int returnstatus;
   int pid;
   char writemessages[2][20]={"Hi", "Hello"};
   char readmessage[20];
   returnstatus = pipe(pipefds);
   if (returnstatus == -1) {
      printf("Unable to create pipe\n");
      return 1;
   }
```

Following are the steps to achieve two-way communication −

Step 1 − Create two pipes. First one is for the parent to write and child to read, say as pipe1. Second one is for the child to write and parent to read, say as pipe2.

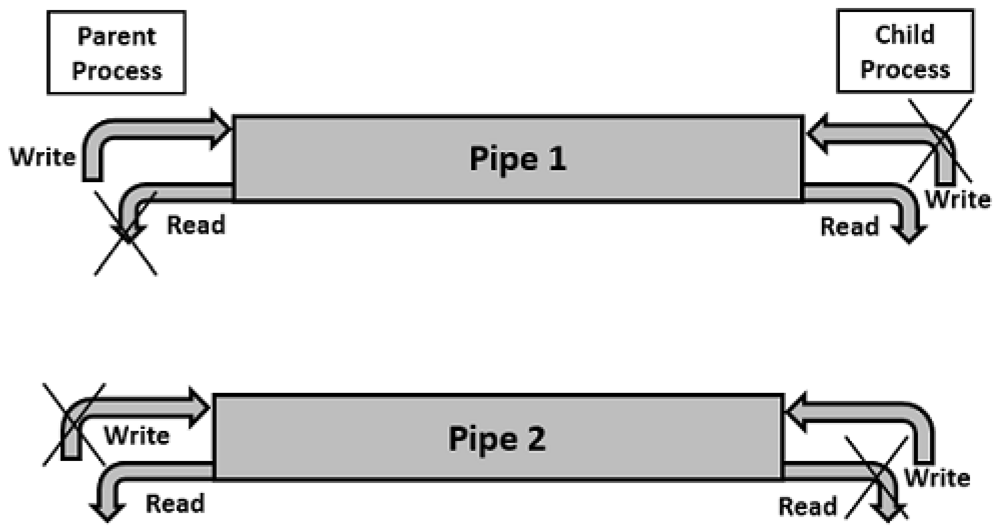
Step 2 − Create a child process.

Step 3 − Close unwanted ends as only one end is needed for each communication.

Step 4 − Close unwanted ends in the parent process, read end of pipe1 and write end of pipe2.

Step 5 − Close the unwanted ends in the child process, write end of pipe1 and read end of pipe2.

Step 6 − Perform the communication as required.



Sample Programs

Sample program 1 − Achieving two-way communication using pipes.

Algorithm

Step 1 − Create pipe1 for the parent process to write and the child process to read.

Step 2 − Create pipe2 for the child process to write and the parent process to read.

Step 3 − Close the unwanted ends of the pipe from the parent and child side.

Step 4 − Parent process to write a message and child process to read and display on the screen.

Step 5 − Child process to write a message and parent process to read and display on the screen.

Source Code: twowayspipe.c

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

int main() {
   int pipefds1[2], pipefds2[2];
   int returnstatus1, returnstatus2;
   int pid;
   char pipe1writemessage[20] = "Hi";
   char pipe2writemessage[20] = "Hello";
   char readmessage[20];
   returnstatus1 = pipe(pipefds1);

   if (returnstatus1 == -1) {
      printf("Unable to create pipe 1 \n");
```

```
    return 1;
  }
  returnstatus2 = pipe(pipefds2);

  if (returnstatus2 == -1) {
    printf("Unable to create pipe 2 \n");
    return 1;
  }

 pid = fork();

  if (pid != 0) // Parent process {
    close(pipefds1[0]); // Close the unwanted pipe1 read side
    close(pipefds2[1]); // Close the unwanted pipe2 write side
    printf("In Parent: Writing to pipe 1 – Message is %s\n", pipe1writemessage);
    write(pipefds1[1], pipe1writemessage, sizeof(pipe1writemessage));
    read(pipefds2[0], readmessage, sizeof(readmessage));
    printf("In Parent: Reading from pipe 2 – Message is %s\n", readmessage);
  } else { //child process
    close(pipefds1[1]); // Close the unwanted pipe1 write side
    close(pipefds2[0]); // Close the unwanted pipe2 read side
    read(pipefds1[0], readmessage, sizeof(readmessage));
    printf("In Child: Reading from pipe 1 – Message is %s\n", readmessage);
    printf("In Child: Writing to pipe 2 – Message is %s\n", pipe2writemessage);
    write(pipefds2[1], pipe2writemessage, sizeof(pipe2writemessage));
  }
  return 0;
}
```

Execution Steps

Compilation

```
gcc twowayspipe.c –o twowayspipe
```

Execution

```
In Parent: Writing to pipe 1 – Message is Hi
In Child: Reading from pipe 1 – Message is Hi
In Child: Writing to pipe 2 – Message is Hello
In Parent: Reading from pipe 2 – Message is Hello
```

III. Named pipes

Pipes were meant for communication between related processes. Can we use pipes for unrelated process communication, say, we want to execute client program from one terminal and the server program from another terminal? The answer is No. Then how can we achieve unrelated processes communication, the simple answer is Named Pipes. Even though this works for related processes, it gives no meaning to use the named pipes for related process communication.

We used one pipe for one-way communication and two pipes for bi-directional communication. Does the same condition apply for Named Pipes. The answer is no, we can use single named pipe that can be used for two-way communication (communication between the server and the client, plus the client and the server at the same time) as Named Pipe supports bi-directional communication.

Another name for named pipe is **FIFO (First-In-First-Out)**. Let us see the system call (mknod()) to create a named pipe, which is a kind of a special file.

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

int mknod(const char *pathname, mode_t mode, dev_t dev);
```

This system call would create a special file or file system node such as ordinary file, device file, or FIFO. The arguments to the system call are pathname, mode and dev. The pathname along with the attributes of mode and device information. The pathname is relative, if the directory is not specified it would be created in the current directory. The mode specified is the mode of file which specifies the file type such as the type of file and the file mode as mentioned in the following tables. The dev field is to specify device information such as major and minor device numbers.

File Type	Description	File Type	Description
S_IFBLK	block special	S_IFREG	Regular file
S_IFCHR	character special	S_IFDIR	Directory
S_IFIFO	FIFO special	S_IFLNK	Symbolic Link

File Mode	Description	File Mode	Description
S_IRWXU	Read, write, execute/search by owner	S_IWGRP	Write permission, group
S_IRUSR	Read permission, owner	S_IXGRP	Execute/search permission, group
S_IWUSR	Write permission, owner	S_IRWXO	Read, write, execute/search by others
S_IXUSR	Execute/search permission, owner	S_IROTH	Read permission, others

S_IRWXG	Read, write, execute/search by group	S_IWOTH	Write permission, others
S_IRGRP	Read permission, group	S_IXOTH	Execute/search permission, others

File mode can also be represented in octal notation such as 0XYZ, where X represents owner, Y represents group, and Z represents others. The value of X, Y or Z can range from 0 to 7. The values for read, write and execute are 4, 2, 1 respectively. If needed in combination of read, write and execute, then add the values accordingly.

Say, if we mention, 0640, then this means read and write (4 + 2 = 6) for owner, read (4) for group and no permissions (0) for others.

This call would return zero on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

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

int mkfifo(const char *pathname, mode_t mode)
```

This library function creates a FIFO special file, which is used for named pipe. The arguments to this function is file name and mode. The file name can be either absolute path or relative path. If full path name (or absolute path) is not given, the file would be created in the current folder of the executing process. The file mode information is as described in mknod() system call.

This call would return zero on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

Let us consider a program of running the server on one terminal and running the client on another terminal. The program would only perform one-way communication. The client accepts the user input and sends the message to the server, the server prints the message on the output. The process is continued until the user enters the string “end”.

Let us understand this with an example −

Step 1 − Create two processes, one is fifoserver and another one is fifoclient.

Step 2 − Server process performs the following −

- Creates a named pipe (using system call mknod()) with name “MYFIFO”, if not created.
- Opens the named pipe for read only purposes.
- Here, created FIFO with permissions of read and write for Owner. Read for Group and no permissions for Others.

Waits infinitely for message from the Client.

- If the message received from the client is not “end”, prints the message. If the message is “end”, closes the fifo and ends the process.

Step 3 − Client process performs the following −

- Opens the named pipe for write only purposes.
- Accepts the string from the user.
- Checks, if the user enters “end” or other than “end”. Either way, it sends a message to the server. However, if the string is “end”, this closes the FIFO and also ends the process.
- Repeats infinitely until the user enters string “end”.

Now let’s take a look at the FIFO server file.

```
/* Filename: fifoserver.c */
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define FIFO_FILE "MYFIFO"
int main() {
   int fd;
   char readbuf[80];
   char end[10];
   int to_end;
   int read_bytes;

   /* Create the FIFO if it does not exist */
   mknod(FIFO_FILE, S_IFIFO|0640, 0);
   strcpy(end, "end");
   while(1) {
      fd = open(FIFO_FILE, O_RDONLY);
      read_bytes = read(fd, readbuf, sizeof(readbuf));
      readbuf[read_bytes] = '\0';

      printf("Received string: \"%s\" and length is %d\n", readbuf, (int)strlen(readbuf));
      to_end = strcmp(readbuf, end);
      if (to_end == 0) {
         close(fd);
         break;
      }
   }
   return 0;
}
```

Compilation and Execution Steps

```
Received string: "this is string 1" and length is 16
Received string: "fifo test" and length is 9
Received string: "fifo client and server" and length is 22
Received string: "end" and length is 3
```

Now, let’s take a look at the FIFO client sample code.

```c
/* Filename: fifoclient.c */
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define FIFO_FILE "MYFIFO"
int main() {
   int fd;
   int end_process;
   int stringlen;
   char readbuf[80];
   char end_str[5];
   printf("FIFO_CLIENT: Send messages, infinitely, to end enter \"end\"\n");
   fd = open(FIFO_FILE, O_CREAT|O_WRONLY);

  strcpy(end_str, "end");

   while (1) {
      printf("Enter string: ");
      fgets(readbuf, sizeof(readbuf), stdin);
      stringlen = strlen(readbuf);
      readbuf[stringlen - 1] = '\0';
      end_process = strcmp(readbuf, end_str);

      //printf("end_process is %d\n", end_process);
      if (end_process != 0) {
         write(fd, readbuf, strlen(readbuf));
         printf("Sent string: \"%s\" and string length is %d\n", readbuf, (int)strlen(readbuf));
      } else {
         write(fd, readbuf, strlen(readbuf));
         printf("Sent string: \"%s\" and string length is %d\n", readbuf, (int)strlen(readbuf));
         close(fd);
         break;
      }
   }
   return 0;
}
```

Let’s take a at the arriving output.

Compilation and Execution Steps

```
FIFO_CLIENT: Send messages, infinitely, to end enter "end"
Enter string: this is string 1
Sent string: "this is string 1" and string length is 16
Enter string: fifo test
Sent string: "fifo test" and string length is 9
Enter string: fifo client and server
Sent string: "fifo client and server" and string length is 22
Enter string: end
Sent string: "end" and string length is 3
```

Two-way Communication Using Named Pipes

The communication between pipes are meant to be unidirectional. Pipes were restricted to one-way communication in general and need at least two pipes for two-way communication. Pipes are meant for inter-related processes only. Pipes can’t be used for unrelated processes communication, say, if we want to execute one process from one terminal and another process from another terminal, it is not possible with pipes. Do we have any simple way of communicating between two processes, say unrelated processes in a simple way? The answer is YES. Named pipe is meant for communication between two or more unrelated processes and can also have bi-directional communication.

Already, we have seen the one-directional communication between named pipes, i.e., the messages from the client to the server. Now, let us take a look at the bi-directional communication i.e., the client sending message to the server and the server receiving the message and sending back another message to the client using the same named pipe.

Following is an example −

Step 1 − Create two processes, one is fifoserver_twoway and another one is fifoclient_twoway.

Step 2 − Server process performs the following −

- Creates a named pipe (using library function mkfifo()) with name “fifo_twoway” in /tmp directory, if not created.
- Opens the named pipe for read and write purposes.
- Here, created FIFO with permissions of read and write for Owner. Read for Group and no permissions for Others.
- Waits infinitely for a message from the client.
- If the message received from the client is not “end”, prints the message and reverses the string. The reversed string is sent back to the client. If the message is “end”, closes the fifo and ends the process.

Step 3 − Client process performs the following −

- Opens the named pipe for read and write purposes.
- Accepts string from the user.
- Checks, if the user enters “end” or other than “end”. Either way, it sends a message to the server. However, if the string is “end”, this closes the FIFO and also ends the process.
- If the message is sent as not “end”, it waits for the message (reversed string) from the client and prints the reversed string.
- Repeats infinitely until the user enters the string “end”.

Now, let’s take a look at FIFO server sample code.

```
/* Filename: fifoserver_twoway.c */
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define FIFO_FILE "/tmp/fifo_twoway"
void reverse_string(char *);
int main() {
   int fd;
   char readbuf[80];
   char end[10];
   int to_end;
   int read_bytes;

   /* Create the FIFO if it does not exist */
   mkfifo(FIFO_FILE, S_IFIFO|0640);
   strcpy(end, "end");
   fd = open(FIFO_FILE, O_RDWR);
   while(1) {
      read_bytes = read(fd, readbuf, sizeof(readbuf));
      readbuf[read_bytes] = '\0';
      printf("FIFOSERVER: Received string: \"%s\" and length is %d\n", readbuf,
(int)strlen(readbuf));
      to_end = strcmp(readbuf, end);

 if (to_end == 0) {
         close(fd);
         break;
      }
      reverse_string(readbuf);
```

```
    printf("FIFOSERVER: Sending Reversed String: \"%s\" and length is %d\n", readbuf, (int)
strlen(readbuf));
    write(fd, readbuf, strlen(readbuf));
    /*
    sleep - This is to make sure other process reads this, otherwise this
    process would retrieve the message
    */
    sleep(2);
  }
  return 0;
}

void reverse_string(char *str) {
  int last, limit, first;
  char temp;
  last = strlen(str) - 1;
  limit = last/2;
  first = 0;

  while (first < last) {
    temp = str[first];
    str[first] = str[last];
    str[last] = temp;
    first++;
    last--;
  }
  return;
}
```

Compilation and Execution Steps

```
FIFOSERVER: Received string: "LINUX IPCs" and length is 10
FIFOSERVER: Sending Reversed String: "sCPI XUNIL" and length is 10
FIFOSERVER: Received string: "Inter Process Communication" and length is 27
FIFOSERVER: Sending Reversed String: "noitacinummoC ssecorP retnI" and length is 27
FIFOSERVER: Received string: "end" and length is 3
```

Now, let’s take a look at FIFO client sample code.

```
/* Filename: fifoclient_twoway.c */
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define FIFO_FILE "/tmp/fifo_twoway"
int main() {
  int fd;
  int end_process;
  int stringlen;
  int read_bytes;
  char readbuf[80];
  char end_str[5];
  printf("FIFO_CLIENT: Send messages, infinitely, to end enter \"end\"\n");
  fd = open(FIFO_FILE, O_CREAT|O_RDWR);
  strcpy(end_str, "end");

  while (1) {
    printf("Enter string: ");
    fgets(readbuf, sizeof(readbuf), stdin);
    stringlen = strlen(readbuf);
    readbuf[stringlen - 1] = '\0';
    end_process = strcmp(readbuf, end_str);


    //printf("end_process is %d\n", end_process);
    if (end_process != 0) {
      write(fd, readbuf, strlen(readbuf));
      printf("FIFOCLIENT: Sent string: \"%s\" and string length is %d\n", readbuf,
(int)strlen(readbuf));
      read_bytes = read(fd, readbuf, sizeof(readbuf));
      readbuf[read_bytes] = '\0';
      printf("FIFOCLIENT: Received string: \"%s\" and length is %d\n", readbuf,
(int)strlen(readbuf));
    } else {
      write(fd, readbuf, strlen(readbuf));
      printf("FIFOCLIENT: Sent string: \"%s\" and string length is %d\n", readbuf,
(int)strlen(readbuf));
      close(fd);
      break;
    }
  }
  return 0;
}
```

Compilation and Execution Steps

```
FIFO_CLIENT: Send messages, infinitely, to end enter "end"
Enter string: LINUX IPCs
FIFOCLIENT: Sent string: "LINUX IPCs" and string length is 10
FIFOCLIENT: Received string: "sCPI XUNIL" and length is 10
Enter string: Inter Process Communication
FIFOCLIENT: Sent string: "Inter Process Communication" and string length is 27
FIFOCLIENT: Received string: "noitacinummoC ssecorP retnI" and length is 27
Enter string: end
FIFOCLIENT: Sent string: "end" and string length is 3
```

IV. Shared memory

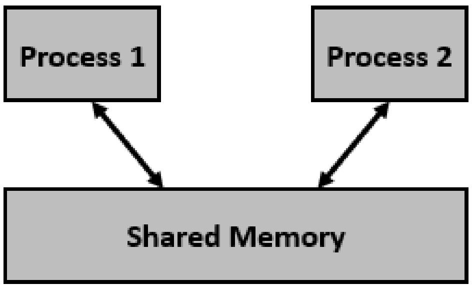
Shared memory is a memory shared between two or more processes. However, why do we need to share memory or some other means of communication?

To reiterate, each process has its own address space, if any process wants to communicate with some information from its own address space to other processes, then it is only possible with IPC (inter process communication) techniques. As we are already aware, communication can be between related or unrelated processes.

Usually, inter-related process communication is performed using Pipes or Named Pipes. Unrelated processes (say one process running in one terminal and another process in another terminal) communication can be performed using Named Pipes or through popular IPC techniques of Shared Memory and Message Queues.

We have seen the IPC techniques of Pipes and Named pipes and now it is time to know the remaining IPC techniques viz., Shared Memory, Message Queues, Semaphores, Signals, and Memory Mapping.

In this section, we will know all about shared memory.



We know that to communicate between two or more processes, we use shared memory but before using the shared memory what needs to be done with the system calls, let us see this −

- Create the shared memory segment or use an already created shared memory segment (shmget())
- Attach the process to the already created shared memory segment (shmat())
- Detach the process from the already attached shared memory segment (shmdt())
- Control operations on the shared memory segment (shmctl())

Let us look at a few details of the system calls related to shared memory.

```
#include <sys/ipc.h>
#include <sys/shm.h>
```

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

The above system call creates or allocates a System V shared memory segment. The arguments that need to be passed are as follows

The first argument, key, recognizes the shared memory segment. The key can be either an arbitrary value or one that can be derived from the library function ftok(). The key can also be IPC_PRIVATE, means, running processes as server and client (parent and child relationship) i.e., inter-related process communiation. If the client wants to use shared memory with this key, then it must be a child process of the server. Also, the child process needs to be created after the parent has obtained a shared memory**.**

The **second argument, size,** is the size of the shared memory segment rounded to multiple of PAGE_SIZE.

The **third argument, shmflg,** specifies the required shared memory flag/s such as IPC_CREAT (creating new segment) or IPC_EXCL (Used with IPC_CREAT to create new segment and the call fails, if the segment already exists). Need to pass the permissions as well.

Note − Refer earlier sections for details on permissions.

This call would return a valid shared memory identifier (used for further calls of shared memory) on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

```
#include <sys/types.h>
#include <sys/shm.h>

void * shmat(int shmid, const void *shmaddr, int shmflg)
```

The above system call performs shared memory operation for System V shared memory segment i.e., attaching a shared memory segment to the address space of the calling process. The arguments that need to be passed are as follows −

The first argument, shmid, is the identifier of the shared memory segment. This id is the shared memory identifier, which is the return value of shmget() system call.

The second argument, shmaddr, is to specify the attaching address. If shmaddr is NULL, the system by default chooses the suitable address to attach the segment. If shmaddr is not NULL and SHM_RND is specified in shmflg, the attach is equal to the address of the nearest multiple of SHMLBA (Lower Boundary Address). Otherwise, shmaddr must be a page aligned address at which the shared memory attachment occurs/starts.

The third argument, shmflg, specifies the required shared memory flag/s such as SHM_RND (rounding off address to SHMLBA) or SHM_EXEC (allows the contents of segment to be executed) or SHM_RDONLY (attaches the segment for read-only purpose, by default it is read-write) or SHM_REMAP (replaces the existing mapping in the range specified by shmaddr and continuing till the end of segment).

This call would return the address of attached shared memory segment on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

```
#include <sys/types.h>
#include <sys/shm.h>

int shmdt(const void *shmaddr)
```

The above system call performs shared memory operation for System V shared memory segment of detaching the shared memory segment from the address space of the calling process. The argument that needs to be passed is −

The argument, shmaddr, is the address of shared memory segment to be detached. The to-be-detached segment must be the address returned by the shmat() system call.

This call would return 0 on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

The above system call performs control operation for a System V shared memory segment. The following arguments needs to be passed −

The first argument, shmid, is the identifier of the shared memory segment. This id is the shared memory identifier, which is the return value of shmget() system call.

The second argument, cmd, is the command to perform the required control operation on the shared memory segment.

Valid values for cmd are

IPC_STAT − Copies the information of the current values of each member of struct shmid_ds to the passed structure pointed by buf. This command requires read permission to the shared memory segment**.**

- **IPC_SET** − Sets the user ID, group ID of the owner, permissions, etc. pointed to by structure buf.
- **IPC_RMID** − Marks the segment to be destroyed. The segment is destroyed only after the last process has detached it.
- **IPC_INFO** − Returns the information about the shared memory limits and parameters in the structure pointed by buf.

- **SHM_INFO** − Returns a shm_info structure containing information about the consumed system resources by the shared memory.

The third argument, buf, is a pointer to the shared memory structure named struct shmid_ds. The values of this structure would be used for either set or get as per cmd.

This call returns the value depending upon the passed command. Upon success of IPC_INFO and SHM_INFO or SHM_STAT returns the index or identifier of the shared memory segment or 0 for other operations and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

Let us consider the following sample program.

- Create two processes, one is for writing into the shared memory (shm_write.c) and another is for reading from the shared memory (shm_read.c)
- The program performs writing into the shared memory by write process (shm_write.c) and reading from the shared memory by reading process (shm_read.c)
- In the shared memory, the writing process, creates a shared memory of size 1K (and flags) and attaches the shared memory
- The write process writes 5 times the Alphabets from ‘A’ to ‘E’ each of 1023 bytes into the shared memory. Last byte signifies the end of buffer
- Read process would read from the shared memory and write to the standard output
- Reading and writing process actions are performed simultaneously
- After completion of writing, the write process updates to indicate completion of writing into the shared memory (with complete variable in struct shmseg)

Reading process performs reading from the shared memory and displays on the output until it gets indication of write process completion (complete variable in struct shmseg)

- Performs reading and writing process for a few times for simplication and also in order to avoid infinite loops and complicating the program

Following is the code for write process (Writing into Shared Memory – File: shm_write.c)

```
/* Filename: shm_write.c */
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<string.h>
#include<errno.h>
#include<stdlib.h>
#include<unistd.h>
```

```
#include<string.h>

#define BUF_SIZE 1024
#define SHM_KEY 0x1234

struct shmseg {
   int cnt;
   int complete;
   char buf[BUF_SIZE];
};
int fill_buffer(char * bufptr, int size);

int main(int argc, char *argv[]) {
   int shmid, numtimes;
   struct shmseg *shmp;
   char *bufptr;
   int spaceavailable;
   shmid = shmget(SHM_KEY, sizeof(struct shmseg), 0644|IPC_CREAT);
   if (shmid == -1) {
      perror("Shared memory");
      return 1;
   }

 // Attach to the segment to get a pointer to it.
   shmp = shmat(shmid, NULL, 0);
   if (shmp == (void *) -1) {
      perror("Shared memory attach");
      return 1;
   }

   /* Transfer blocks of data from buffer to shared memory */
   bufptr = shmp->buf;
   spaceavailable = BUF_SIZE;
   for (numtimes = 0; numtimes < 5; numtimes++) {
      shmp->cnt = fill_buffer(bufptr, spaceavailable);
      shmp->complete = 0;
      printf("Writing Process: Shared Memory Write: Wrote %d bytes\n", shmp->cnt);
      bufptr = shmp->buf;
      spaceavailable = BUF_SIZE;
      sleep(3);
   }
   printf("Writing Process: Wrote %d times\n", numtimes);
   shmp->complete = 1;

   if (shmdt(shmp) == -1) {
      perror("shmdt");
      return 1;
   }
   if (shmctl(shmid, IPC_RMID, 0) == -1) {
      perror("shmctl");
      return 1;
   }
   printf("Writing Process: Complete\n");
   return 0;
}

int fill_buffer(char * bufptr, int size) {
   static char ch = 'A';
   int filled_count;

  //printf("size is %d\n", size);
   memset(bufptr, ch, size - 1);
   bufptr[size-1] = '\0';
   if (ch > 122)
   ch = 65;
   if ( (ch >= 65) && (ch <= 122) ) {
      if ( (ch >= 91) && (ch <= 96) ) {
         ch = 65;
      }
   }
   filled_count = strlen(bufptr);

   //printf("buffer count is: %d\n", filled_count);
   //printf("buffer filled is:%s\n", bufptr);
   ch++;
   return filled_count;
}
```

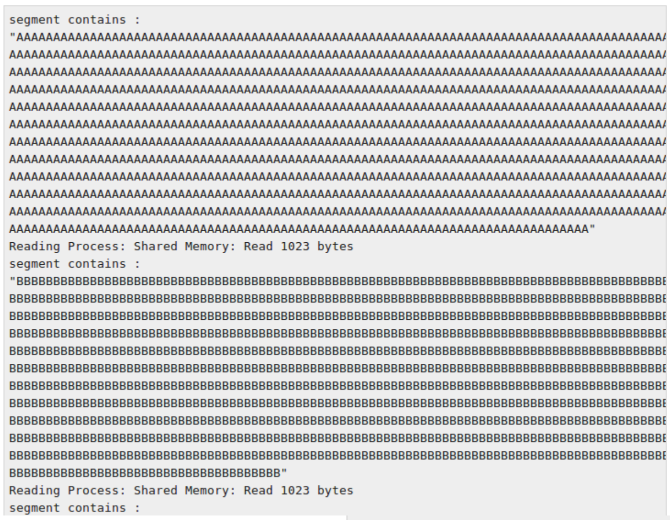
Compilation and Execution Steps

```
Writing Process: Shared Memory Write: Wrote 1023 bytes
Writing Process: Shared Memory Write: Wrote 1023 bytes
Writing Process: Shared Memory Write: Wrote 1023 bytes
Writing Process: Shared Memory Write: Wrote 1023 bytes
Writing Process: Shared Memory Write: Wrote 1023 bytes
Writing Process: Wrote 5 times
Writing Process: Complete
```

Following is the code for read process (Reading from the Shared Memory and writing to the standard output – File: shm_read.c)

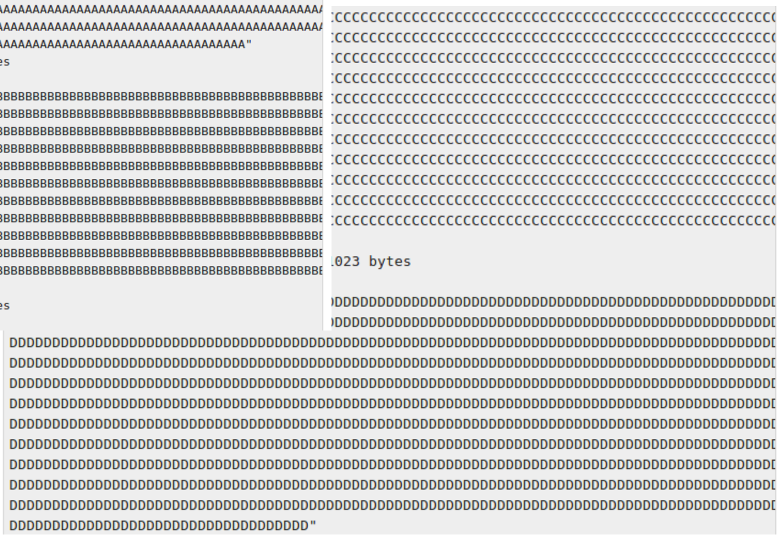
```
/* Filename: shm_read.c */
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
```

```
#include<string.h>
#include<errno.h>
#include<stdlib.h>

#define BUF_SIZE 1024
#define SHM_KEY 0x1234

struct shmseg {
   int cnt;
   int complete;
   char buf[BUF_SIZE];
};

int main(int argc, char *argv[]) {
   int shmid;
   struct shmseg *shmp;
   shmid = shmget(SHM_KEY, sizeof(struct shmseg), 0644|IPC_CREAT);
   if (shmid == -1) {
      perror("Shared memory");
      return 1;
   }

   // Attach to the segment to get a pointer to it.
   shmp = shmat(shmid, NULL, 0);
   if (shmp == (void *) -1) {
      perror("Shared memory attach");
      return 1;
   }

   /* Transfer blocks of data from shared memory to stdout*/
   while (shmp->complete != 1) {
      printf("segment contains : \n\"%s\"\n", shmp->buf);
      if (shmp->cnt == -1) {
         perror("read");
         return 1;
      }
      printf("Reading Process: Shared Memory: Read %d bytes\n", shmp->cnt);
      sleep(3);
   }

 printf("Reading Process: Reading Done, Detaching Shared Memory\n");
   if (shmdt(shmp) == -1) {
      perror("shmdt");
      return 1;
   }
   printf("Reading Process: Complete\n");
   return 0;
}
```

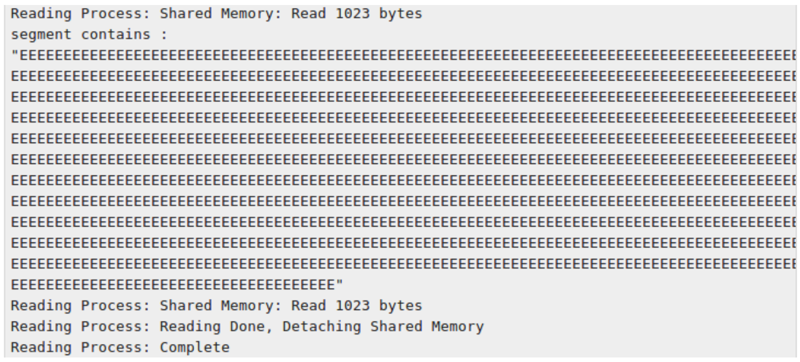


```
segment contains :
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
Reading Process: Shared Memory: Read 1023 bytes
segment contains :
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
Reading Process: Shared Memory: Read 1023 bytes
segment contains :
```

Compilation and Execution Steps



```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

.023 bytes

DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD"
```



```
Reading Process: Shared Memory: Read 1023 bytes
segment contains :
"EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE"
Reading Process: Shared Memory: Read 1023 bytes
Reading Process: Reading Done, Detaching Shared Memory
Reading Process: Complete
```

V. Message Queues

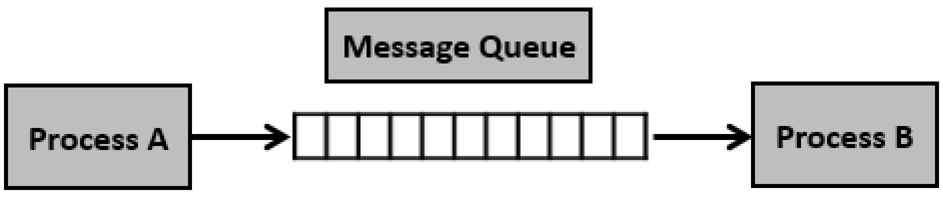
Why do we need message queues when we already have the shared memory? It would be for multiple reasons, let us try to break this into multiple points for simplification

- As understood, once the message is received by a process it would be no longer available for any other process. Whereas in shared memory, the data is available for multiple processes to access.
- If we want to communicate with small message formats.
- Shared memory data need to be protected with synchronization when multiple processes communicating at the same time.
- Frequency of writing and reading using the shared memory is high, then it would be very complex to implement the functionality. Not worth with regard to utilization in this kind of cases.
- What if all the processes do not need to access the shared memory but very few processes only need it, it would be better to implement with message queues.
- If we want to communicate with different data packets, say process A is sending message type 1 to process B, message type 10 to process C, and message type 20 to process D. In this case, it is simplier to implement with message queues. To simplify the given message type as 1, 10, 20, it can be either 0 or +ve or –ve as discussed below.
- Ofcourse, the order of message queue is FIFO (First In First Out). The first message inserted in the queue is the first one to be retrieved.

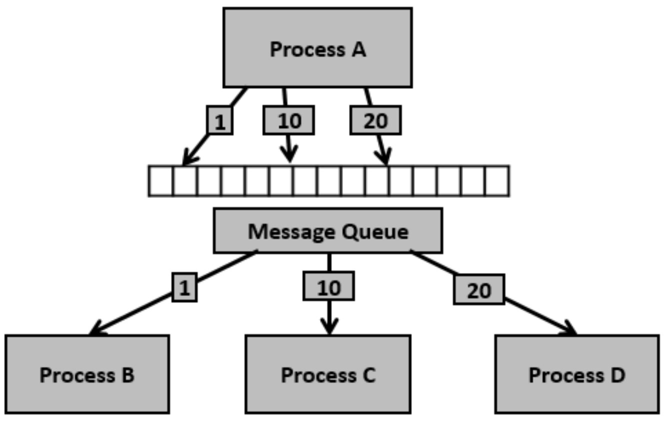
Using Shared Memory or Message Queues depends on the need of the application and how effectively it can be utilized.

Communication using message queues can happen in the following ways −

- Writing into the shared memory by one process and reading from the shared memory by another process. As we are aware, reading can be done with multiple processes as well.



Writing into the shared memory by one process with different data packets and reading from it by multiple processes, i.e., as per message type.



Having seen certain information on message queues, now it is time to check for the system call (System V) which supports the message queues**.**

To perform communication using message queues, following are the steps −

Step 1 − Create a message queue or connect to an already existing message queue (msgget())

Step 2 − Write into message queue (msgsnd())

Step 3 − Read from the message queue (msgrcv())

Step 4 − Perform control operations on the message queue (msgctl())

Now, let us check the syntax and certain information on the above calls.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg)
```

This system call creates or allocates a System V message queue. Following arguments need to be passed −

- The first argument, key, recognizes the message queue. The key can be either an arbitrary value or one that can be derived from the library function ftok().
- The second argument, shmflg, specifies the required message queue flag/s such as IPC_CREAT (creating message queue if not exists) or IPC_EXCL (Used with IPC_CREAT to create the message queue and the call fails, if the message queue already exists). Need to pass the permissions as well.

Note − Refer earlier sections for details on permissions.

This call would return a valid message queue identifier (used for further calls of message queue) on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

Various errors with respect to this call are EACCESS (permission denied), EEXIST (queue already exists can’t create), ENOENT (queue doesn’t exist), ENOMEM (not enough memory to create the queue), etc.

```
#include <sys/types.h>
#include <sys/ipc.h>
```

```
#include <sys/msg.h>

int msgsnd(int msgid, const void *msgp, size_t msgsz, int msgflg)
```

This system call sends/appends a message into the message queue (System V). Following arguments need to be passed −

- The first argument, msgid, recognizes the message queue i.e., message queue identifier. The identifier value is received upon the success of msgget()
- The second argument, msgp, is the pointer to the message, sent to the caller, defined in the structure of the following form −

```
struct msgbuf {
  long mtype;
  char mtext[1];
};
```

The variable mtype is used for communicating with different message types, explained in detail in msgrcv() call. The variable mtext is an array or other structure whose size is specified by msgsz (positive value). If the mtext field is not mentioned, then it is considered as zero size message, which is permitted.

- The third argument, msgsz, is the size of message (the message should end with a null character)
- The fourth argument, msgflg, indicates certain flags such as IPC_NOWAIT (returns immediately when no message is found in queue or MSG_NOERROR (truncates message text, if more than msgsz bytes)

This call would return 0 on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgrcv(int msgid, const void *msgp, size_t msgsz, long msgtype, int msgflg)
```

This system call retrieves the message from the message queue (System V). Following arguments need to be passed −

- The first argument, msgid, recognizes the message queue i.e., the message queue identifier. The identifier value is received upon the success of msgget()

The second argument, msgp, is the pointer of the message received from the caller. It is defined in the structure of the following form −

```
struct msgbuf {
   long mtype;
   char mtext[1];
};
```

The variable mtype is used for communicating with different message types. The variable mtext is an array or other structure whose size is specified by msgsz (positive value). If the mtext field is not mentioned, then it is considered as zero size message, which is permitted.

- The third argument, msgsz, is the size of the message received (message should end with a null character)
- The fouth argument, msgtype, indicates the type of message −
    - **If msgtype is 0** − Reads the first received message in the queue
    - **If msgtype is +ve** − Reads the first message in the queue of type msgtype (if msgtype is 10, then reads only the first message of type 10 even though other types may be in the queue at the beginning)
    - **If msgtype is –ve** − Reads the first message of lowest type less than or equal to the absolute value of message type (say, if msgtype is -5, then it reads first message of type less than 5 i.e., message type from 1 to 5)
- The fifth argument, msgflg, indicates certain flags such as IPC_NOWAIT (returns immediately when no message is found in the queue or MSG_NOERROR (truncates the message text if more than msgsz bytes)

This call would return the number of bytes actually received in mtext array on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(int msgid, int cmd, struct msqid_ds *buf)
```

This system call performs control operations of the message queue (System V). Following arguments need to be passed −

- The first argument, msgid, recognizes the message queue i.e., the message queue identifier. The identifier value is received upon the success of msgget()
- The second argument, cmd, is the command to perform the required control operation on the message queue. Valid values for cmd are −

IPC_STAT − Copies information of the current values of each member of struct msqid_ds to the passed structure pointed by buf. This command requires read permission on the message queue.

IPC_SET − Sets the user ID, group ID of the owner, permissions etc pointed to by structure buf.

IPC_RMID − Removes the message queue immediately.

IPC_INFO − Returns information about the message queue limits and parameters in the structure pointed by buf, which is of type struct msginfo

MSG_INFO − Returns an msginfo structure containing information about the consumed system resources by the message queue.

- The third argument, buf, is a pointer to the message queue structure named struct msqid_ds. The values of this structure would be used for either set or get as per cmd.

This call would return the value depending on the passed command. Success of IPC_INFO and MSG_INFO or MSG_STAT returns the index or identifier of the message queue or 0 for other operations and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

Having seen the basic information and system calls with regard to message queues, now it is time to check with a program.

Let us see the description before looking at the program −

Step 1 − Create two processes, one is for sending into message queue (msgq_send.c) and another is for retrieving from the message queue (msgq_recv.c)

Step 2 − Creating the key, using ftok() function. For this, initially file msgq.txt is created to get a unique key.

Step 3 − The sending process performs the following.

Reads the string input from the user

- Removes the new line, if it exists
- Sends into message queue
- Repeats the process until the end of input (CTRL + D)
- Once the end of input is received, sends the message “end” to signify the end of the process

Step 4 − In the receiving process, performs the following.

- Reads the message from the queue
- Displays the output
- If the received message is “end”, finishes the process and exits

To simplify, we are not using the message type for this sample. Also, one process is writing into the queue and another process is reading from the queue. This can be extended as needed i.e., ideally one process would write into the queue and multiple processes read from the queue.

Now, let us check the process (message sending into queue) – File: msgq_send.c

```
/* Filename: msgq_send.c */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define PERMS 0644
struct my_msgbuf {
   long mtype;
   char mtext[200];
};

int main(void) {
   struct my_msgbuf buf;

  int msqid;
   int len;
   key_t key;
   system("touch msgq.txt");

   if ((key = ftok("msgq.txt", 'B')) == -1) {
      perror("ftok");
      exit(1);
   }

   if ((msqid = msgget(key, PERMS | IPC_CREAT)) == -1) {
      perror("msgget");
      exit(1);
   }
   printf("message queue: ready to send messages.\n");
   printf("Enter lines of text, ^D to quit:\n");
   buf.mtype = 1; /* we don't really care in this case */

   while(fgets(buf.mtext, sizeof buf.mtext, stdin) != NULL) {
      len = strlen(buf.mtext);
      /* remove newline at end, if it exists */
      if (buf.mtext[len-1] == '\n') buf.mtext[len-1] = '\0';
      if (msgsnd(msqid, &buf, len+1, 0) == -1) /* +1 for '\0' */
      perror("msgsnd");
   }
   strcpy(buf.mtext, "end");
   len = strlen(buf.mtext);
   if (msgsnd(msqid, &buf, len+1, 0) == -1) /* +1 for '\0' */
   perror("msgsnd");

   if (msgctl(msqid, IPC_RMID, NULL) == -1) {
      perror("msgctl");
      exit(1);
   }
   printf("message queue: done sending messages.\n");
   return 0;
}
```

Compilation and Execution Steps

```
message queue: ready to send messages.
Enter lines of text, ^D to quit:
this is line 1
this is line 2
message queue: done sending messages.
```

Following is the code from message receiving process (retrieving message from queue) – File: msgq_recv.c

```
/* Filename: msgq_recv.c */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define PERMS 0644
struct my_msgbuf {
  long mtype;
  char mtext[200];
};

int main(void) {
  struct my_msgbuf buf;
  int msqid;
  int toend;
  key_t key;

  if ((key = ftok("msgq.txt", 'B')) == -1) {
    perror("ftok");
    exit(1);
  }

  if ((msqid = msgget(key, PERMS)) == -1) { /* connect to the queue */
    perror("msgget");
```

```
      exit(1);

  }
   printf("message queue: ready to receive messages.\n");

   for(;;) { /* normally receiving never ends but just to make conclusion
             /* this program ends wuth string of end */
      if (msgrcv(msqid, &buf, sizeof(buf.mtext), 0, 0) == -1) {
         perror("msgrcv");
         exit(1);
      }
      printf("recvd: \"%s\"\n", buf.mtext);
      toend = strcmp(buf.mtext,"end");
      if (toend == 0)
      break;
   }
   printf("message queue: done receiving messages.\n");
   system("rm msgq.txt");
   return 0;
}
```

Compilation and Execution Steps

```
message queue: ready to receive messages.
recvd: "this is line 1"
recvd: "this is line 2"
recvd: "end"
message queue: done receiving messages.
```

VI. **Semaphores**

The first question that comes to mind is, why do we need semaphores? A simple answer, to protect the critical/common region shared among multiple processes.

Let us assume, multiple processes are using the same region of code and if all want to access parallelly then the outcome is overlapped. Say, for example, multiple users are using one printer only (common/critical section), say 3 users, given 3 jobs at same time, if all the jobs start parallelly, then one user output is overlapped with another. So, we need to protect that using semaphores i.e., locking the critical section when one process is running and unlocking when it is done. This would be repeated for each user/process so that one job is not overlapped with another job.

Basically semaphores are classified into two types −

Binary Semaphores − Only two states 0 & 1, i.e., locked/unlocked or available/unavailable, Mutex implementation.

Counting Semaphores − Semaphores which allow arbitrary resource count are called counting semaphores.

Assume that we have 5 printers (to understand assume that 1 printer only accepts 1 job) and we got 3 jobs to print. Now 3 jobs would be given for 3 printers (1 each). Again 4 jobs came while this is in progress. Now, out of 2 printers available, 2 jobs have been scheduled and we are left with 2 more jobs, which would be completed only after one of the resource/printer is available. This kind of scheduling as per resource availability can be viewed as counting semaphores.

To perform synchronization using semaphores, following are the steps −

Step 1 − Create a semaphore or connect to an already existing semaphore (semget())

Step 2 − Perform operations on the semaphore i.e., allocate or release or wait for the resources (semop())

Step 3 − Perform control operations on the message queue (semctl())

Now, let us check this with the system calls we have.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg)
```

This system call creates or allocates a System V semaphore set. The following arguments need to be passed −

- The first argument, key, recognizes the message queue. The key can be either an arbitrary value or one that can be derived from the library function ftok().
- The second argument, nsems, specifies the number of semaphores. If binary then it is 1, implies need of 1 semaphore set, otherwise as per the required count of number of semaphore sets.
- The third argument, semflg, specifies the required semaphore flag/s such as IPC_CREAT (creating semaphore if it does not exist) or IPC_EXCL (used with IPC_CREAT to create semaphore and the call fails, if a semaphore already exists). Need to pass the permissions as well.

Note − Refer earlier sections for details on permissions.

This call would return valid semaphore identifier (used for further calls of semaphores) on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

Various errors with respect to this call are EACCESS (permission denied), EEXIST (queue already exists can't create), ENOENT (queue doesn't exist), ENOMEM (not enough memory to create the queue), ENOSPC (maximum sets limit exceeded), etc.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *semops, size_t nsemops)
```

This system call performs the operations on the System V semaphore sets viz., allocating resources, waiting for the resources or freeing the resources. Following arguments need to be passed

The first argument, semid, indicates semaphore set identifier created by semget().

- The second argument, semops, is the pointer to an array of operations to be performed on the semaphore set. The structure is as follows −

```
struct sembuf {
  unsigned short sem_num; /* Semaphore set num */
  short sem_op; /* Semaphore operation */
  short sem_flg; /* Operation flags, IPC_NOWAIT, SEM_UNDO */
};
```

Element, sem_op, in the above structure, indicates the operation that needs to be performed −

- If sem_op is –ve, allocate or obtain resources. Blocks the calling process until enough resources have been freed by other processes, so that this process can allocate.
- If sem_op is zero, the calling process waits or sleeps until semaphore value reaches 0.
- If sem_op is +ve, release resources.

For example −

```
struct sembuf sem_lock = { 0, -1, SEM_UNDO };

struct sembuf sem_unlock = {0, 1, SEM_UNDO };
```

- The third argument, nsemops, is the number of operations in that array.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, …)
```

This system call performs control operation for a System V semaphore. The following arguments need to be passed −

- The first argument, semid, is the identifier of the semaphore. This id is the semaphore identifier, which is the return value of semget() system call.
- The second argument, semnum, is the number of semaphore. The semaphores are numbered from 0.

The third argument, cmd, is the command to perform the required control operation on the semaphore.

- The fourth argument, of type, union semun, depends on the cmd. For few cases, the fourth argument is not applicable.

Let us check the union semun −

```
union semun {
  int val; /* val for SETVAL */
  struct semid_ds *buf; /* Buffer for IPC_STAT and IPC_SET */
  unsigned short *array; /* Buffer for GETALL and SETALL */
  struct seminfo *__buf; /* Buffer for IPC_INFO and SEM_INFO*/
};
```

The semid_ds data structure which is defined in sys/sem.h is as follows −

```
struct semid_ds {
  struct ipc_perm sem_perm; /* Permissions */
  time_t sem_otime; /* Last semop time */
  time_t sem_ctime; /* Last change time */
  unsigned long sem_nsems; /* Number of semaphores in the set */
};
```

Note − Please refer manual pages for other data structures.

union semun arg; Valid values for cmd are −

- **IPC_STAT** − Copies the information of the current values of each member of struct semid_ds to the passed structure pointed by arg.buf. This command requires read permission to the semaphore.
- **IPC_SET** − Sets the user ID, group ID of the owner, permissions, etc. pointed to by the structure semid_ds.
- **IPC_RMID** − Removes the semaphores set.
- **IPC_INFO** − Returns the information about the semaphore limits and parameters in the structure semid_ds pointed by arg.__buf.
- **SEM_INFO** − Returns a seminfo structure containing information about the consumed system resources by the semaphore.

This call would return value (non-negative value) depending upon the passed command. Upon success, IPC_INFO and SEM_INFO or SEM_STAT returns the index or identifier of the highest used entry as per Semaphore or the value of semncnt for GETNCNT or the value of sempid for GETPID or the value of semval for GETVAL 0 for other operations on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function**.**

Before looking at the code, let us understand its implementation −

- Create two processes say, child and parent.
- Create shared memory mainly needed to store the counter and other flags to indicate end of read/write process into the shared memory.
- The counter is incremented by count by both parent and child processes. The count is either passed as a command line argument or taken as default (if not passed as command line argument or the value is less than 10000). Called with certain sleep time to ensure both parent and child accesses the shared memory at the same time i.e., in parallel.
- Since, the counter is incremented in steps of 1 by both parent and child, the final value should be double the counter. Since, both parent and child processes performing the operations at same time, the counter is not incremented as required. Hence, we need to ensure the completeness of one process completion followed by other process.
- All the above implementations are performed in the file shm_write_cntr.c
- Check if the counter value is implemented in file shm_read_cntr.c
- To ensure completion, the semaphore program is implemented in file shm_write_cntr_with_sem.c. Remove the semaphore after completion of the entire process (after read is done from other program)
- Since, we have separate files to read the value of counter in the shared memory and don't have any effect from writing, the reading program remains the same (shm_read_cntr.c)
- It is always better to execute the writing program in one terminal and reading program from another terminal. Since, the program completes execution only after the writing and reading process is complete, it is ok to run the program after completely executing the write program. The write program would wait until the read program is run and only finishes after it is done.

Programs without semaphores.

```
/* Filename: shm_write_cntr.c */
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<string.h>
```

```
#include<errno.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>

#define SHM_KEY 0x12345
struct shmseg {
   int cntr;
   int write_complete;
   int read_complete;
};
void shared_memory_cntr_increment(int pid, struct shmseg *shmp, int total_count);

int main(int argc, char *argv[]) {
   int shmid;
   struct shmseg *shmp;
   char *bufptr;
   int total_count;
   int sleep_time;
   pid_t pid;
   if (argc != 2)
   total_count = 10000;
   else {
      total_count = atoi(argv[1]);
      if (total_count < 10000)
      total_count = 10000;
   }
   printf("Total Count is %d\n", total_count);
   shmid = shmget(SHM_KEY, sizeof(struct shmseg), 0644|IPC_CREAT);

  if (shmid == -1) {
      perror("Shared memory");
      return 1;
   }

   // Attach to the segment to get a pointer to it.
   shmp = shmat(shmid, NULL, 0);
   if (shmp == (void *) -1) {
      perror("Shared memory attach");
      return 1;
   }
   shmp->cntr = 0;
   pid = fork();

   /* Parent Process - Writing Once */
   if (pid > 0) {
      shared_memory_cntr_increment(pid, shmp, total_count);
   } else if (pid == 0) {
    shared_memory_cntr_increment(pid, shmp, total_count);
    return 0;
  } else {
    perror("Fork Failure\n");
    return 1;
  }
  while (shmp->read_complete != 1)
  sleep(1);

  if (shmdt(shmp) == -1) {
    perror("shmdt");
    return 1;
  }

  if (shmctl(shmid, IPC_RMID, 0) == -1) {
    perror("shmctl");
    return 1;
  }
  printf("Writing Process: Complete\n");
  return 0;

}

/* Increment the counter of shared memory by total_count in steps of 1 */
void shared_memory_cntr_increment(int pid, struct shmseg *shmp, int total_count) {
  int cntr;
  int numtimes;
  int sleep_time;
  cntr = shmp->cntr;
  shmp->write_complete = 0;
  if (pid == 0)
  printf("SHM_WRITE: CHILD: Now writing\n");
  else if (pid > 0)
  printf("SHM_WRITE: PARENT: Now writing\n");
  //printf("SHM_CNTR is %d\n", shmp->cntr);

  /* Increment the counter in shared memory by total_count in steps of 1 */
  for (numtimes = 0; numtimes < total_count; numtimes++) {
    cntr += 1;
    shmp->cntr = cntr;

    /* Sleeping for a second for every thousand */
    sleep_time = cntr % 1000;
    if (sleep_time == 0)
    sleep(1);
  }

  shmp->write_complete = 1;
```

```
   if (pid == 0)
   printf("SHM_WRITE: CHILD: Writing Done\n");
   else if (pid > 0)
   printf("SHM_WRITE: PARENT: Writing Done\n");
   return;
}
```

Compilation and Execution Steps

```
Total Count is 10000
SHM_WRITE: PARENT: Now writing

SHM_WRITE: CHILD: Now writing
SHM_WRITE: PARENT: Writing Done
SHM_WRITE: CHILD: Writing Done
Writing Process: Complete
```

Now, let us check the shared memory reading program.

```
/* Filename: shm_read_cntr.c */
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<string.h>
#include<errno.h>
#include<stdlib.h>
#include<unistd.h>

#define SHM_KEY 0x12345
struct shmseg {
   int cntr;
   int write_complete;
   int read_complete;
};

int main(int argc, char *argv[]) {
   int shmid, numtimes;
   struct shmseg *shmp;
   int total_count;
   int cntr;
   int sleep_time;
   if (argc != 2)
   total_count = 10000;

   else {
      total_count = atoi(argv[1]);
      if (total_count < 10000)
      total_count = 10000;
   }

shmid = shmget(SHM_KEY, sizeof(struct shmseg), 0644|IPC_CREAT);

   if (shmid == -1) {
      perror("Shared memory");
      return 1;
   }
   // Attach to the segment to get a pointer to it.
   shmp = shmat(shmid, NULL, 0);

   if (shmp == (void *) -1) {
      perror("Shared memory attach");
      return 1;
   }

   /* Read the shared memory cntr and print it on standard output */
   while (shmp->write_complete != 1) {
      if (shmp->cntr == -1) {
         perror("read");
         return 1;
      }
      sleep(3);
   }
   printf("Reading Process: Shared Memory: Counter is %d\n", shmp->cntr);
   printf("Reading Process: Reading Done, Detaching Shared Memory\n");
   shmp->read_complete = 1;

   if (shmdt(shmp) == -1) {
      perror("shmdt");
      return 1;
   }
   printf("Reading Process: Complete\n");
   return 0;
}
```

Compilation and Execution Steps

```
Reading Process: Shared Memory: Counter is 11000

Reading Process: Reading Done, Detaching Shared Memory
Reading Process: Complete
```

If you observe the above output, the counter should be 20000, however, since before completion of one process task other process is also processing in parallel, the counter value is not as expected. The output would vary from system to system and also it would vary with each execution. To ensure the two processes perform the task after completion of one task, it should be implemented using synchronization mechanisms.

Now, let us check the same application using semaphores.

Note − Reading program remains the same.

```
/* Filename: shm_write_cntr_with_sem.c */
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/sem.h>
#include<string.h>
#include<errno.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>

#define SHM_KEY 0x12345
#define SEM_KEY 0x54321
#define MAX_TRIES 20

struct shmseg {
   int cntr;
   int write_complete;
   int read_complete;
};
void shared_memory_cntr_increment(int, struct shmseg*, int);
void remove_semaphore();

int main(int argc, char *argv[]) {
   int shmid;
   struct shmseg *shmp;
   char *bufptr;
   int total_count;
   int sleep_time;
   pid_t pid;
   if (argc != 2)
   total_count = 10000;
   else {
      total_count = atoi(argv[1]);
      if (total_count < 10000)
      total_count = 10000;
   }
   printf("Total Count is %d\n", total_count);
   shmid = shmget(SHM_KEY, sizeof(struct shmseg), 0644|IPC_CREAT);

   if (shmid == -1) {
      perror("Shared memory");
      return 1;
   }
   // Attach to the segment to get a pointer to it.
   shmp = shmat(shmid, NULL, 0);

   if (shmp == (void *) -1) {
      perror("Shared memory attach: ");
      return 1;
   }
   shmp->cntr = 0;
   pid = fork();

   /* Parent Process - Writing Once */
   if (pid > 0) {
      shared_memory_cntr_increment(pid, shmp, total_count);
   } else if (pid == 0) {
      shared_memory_cntr_increment(pid, shmp, total_count);
      return 0;
   } else {

     perror("Fork Failure\n");
      return 1;
   }
   while (shmp->read_complete != 1)
   sleep(1);

   if (shmdt(shmp) == -1) {
      perror("shmdt");
      return 1;
   }

   if (shmctl(shmid, IPC_RMID, 0) == -1) {
      perror("shmctl");
      return 1;
   }
   printf("Writing Process: Complete\n");
   remove_semaphore();
   return 0;
}

/* Increment the counter of shared memory by total_count in steps of 1 */
void shared_memory_cntr_increment(int pid, struct shmseg *shmp, int total_count) {
   int cntr;
   int numtimes;
   int sleep_time;
   int semid;
   struct sembuf sem_buf;
   struct semid_ds buf;
   int tries;
   int retval;
```

```
  semid = semget(SEM_KEY, 1, IPC_CREAT | IPC_EXCL | 0666);
  //printf("errno is %d and semid is %d\n", errno, semid);

  /* Got the semaphore */
  if (semid >= 0) {
    printf("First Process\n");
    sem_buf.sem_op = 1;
    sem_buf.sem_flg = 0;
    sem_buf.sem_num = 0;

 retval = semop(semid, &sem_buf, 1);
    if (retval == -1) {
      perror("Semaphore Operation: ");
      return;
    }
  } else if (errno == EEXIST) { // Already other process got it
    int ready = 0;
    printf("Second Process\n");
    semid = semget(SEM_KEY, 1, 0);
    if (semid < 0) {
      perror("Semaphore GET: ");
      return;
    }

    /* Waiting for the resource */
    sem_buf.sem_num = 0;
    sem_buf.sem_op = 0;
    sem_buf.sem_flg = SEM_UNDO;
    retval = semop(semid, &sem_buf, 1);
    if (retval == -1) {
      perror("Semaphore Locked: ");
      return;
    }
  }
  sem_buf.sem_num = 0;
  sem_buf.sem_op = -1; /* Allocating the resources */
  sem_buf.sem_flg = SEM_UNDO;
  retval = semop(semid, &sem_buf, 1);

  if (retval == -1) {
    perror("Semaphore Locked: ");
    return;
  }
  cntr = shmp->cntr;
  shmp->write_complete = 0;
  if (pid == 0)
  printf("SHM_WRITE: CHILD: Now writing\n");
  else if (pid > 0)
  printf("SHM_WRITE: PARENT: Now writing\n");
  //printf("SHM_CNTR is %d\n", shmp->cntr);

  /* Increment the counter in shared memory by total_count in steps of 1 */
  for (numtimes = 0; numtimes < total_count; numtimes++) {
    cntr += 1;
    shmp->cntr = cntr;
    /* Sleeping for a second for every thousand */
    sleep_time = cntr % 1000;
    if (sleep_time == 0)
    sleep(1);
  }
  shmp->write_complete = 1;
  sem_buf.sem_op = 1; /* Releasing the resource */
  retval = semop(semid, &sem_buf, 1);

  if (retval == -1) {
    perror("Semaphore Locked\n");
    return;
  }

  if (pid == 0)
    printf("SHM_WRITE: CHILD: Writing Done\n");
    else if (pid > 0)
    printf("SHM_WRITE: PARENT: Writing Done\n");
    return;
}

void remove_semaphore() {
  int semid;
  int retval;
  semid = semget(SEM_KEY, 1, 0);
    if (semid < 0) {
      perror("Remove Semaphore: Semaphore GET: ");
      return;
    }
  retval = semctl(semid, 0, IPC_RMID);
  if (retval == -1) {
    perror("Remove Semaphore: Semaphore CTL: ");

    return;
  }
  return;
}
```

Compilation and Execution Steps

```
Total Count is 10000
```

First Process
SHM_WRITE: PARENT: Now writing
Second Process
SHM_WRITE: PARENT: Writing Done
SHM_WRITE: CHILD: Now writing
SHM_WRITE: CHILD: Writing Done
Writing Process: Complete

Now, we will check the counter value by the reading process.

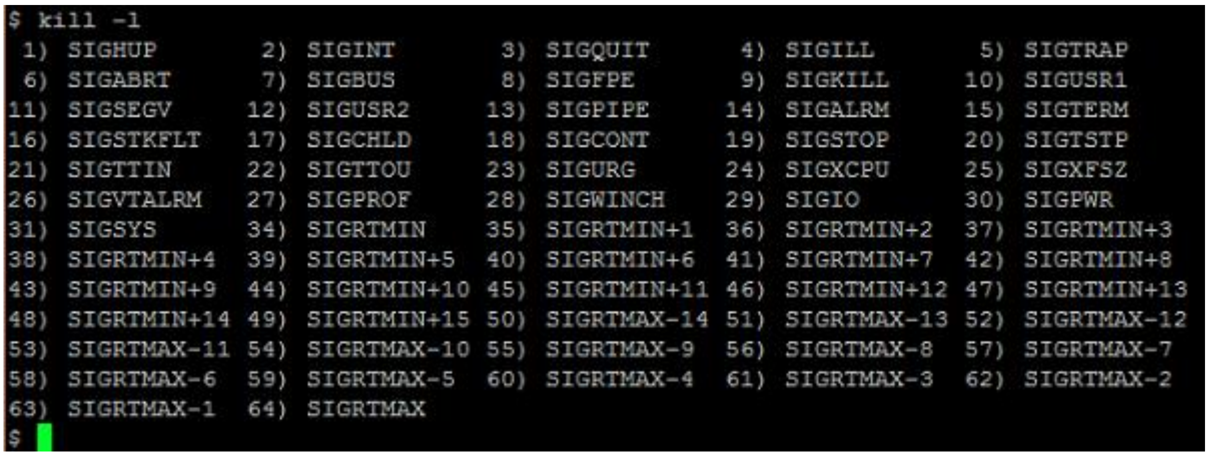
Execution Steps

Reading Process: Shared Memory: Counter is 20000
Reading Process: Reading Done, Detaching Shared Memory
Reading Process: Complete

VII. Signals

A signal is a notification to a process indicating the occurrence of an event. Signal is also called software interrupt and is not predictable to know its occurrence, hence it is also called an asynchronous event.

Signal can be specified with a number or a name, usually signal names start with SIG. The available signals can be checked with the command kill –l (l for Listing signal names), which is as follows



```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
$
```

Whenever a signal is raised (either programmatically or system generated signal), a default action is performed. What if you don’t want to perform the default action but wish to perform your own actions on receiving the signal? Is this possible for all the signals? Yes, it is possible to handle the signal but not for all the signals. What if you want to ignore the signals, is this possible? Yes, it is possible to ignore the signal. Ignoring the signal implies neither performing the default action nor handling the signal. It is possible to ignore or handle almost all the signals. The signals which can’t be either ignored or handled/caught are SIGSTOP and SIGKILL.

In summary, the actions performed for the signals are as follows −

- Default Action
- Handle the signal
- Ignore the signal

As discussed the signal can be handled altering the execution of default action. Signal handling can be done in either of the two ways i.e., through system calls, signal() and sigaction().

```
#include <signal.h>

typedef void (*sighandler_t) (int);
sighandler_t signal(int signum, sighandler_t handler);
```

The system call signal() would call the registered handler upon generation of signal as mentioned in signum. The handler can be either one of the SIG_IGN (Ignoring the Signal), SIG_DFL (Setting signal back to default mechanism) or user-defined signal handler or function address.

This system call on success returns the address of a function that takes an integer argument and has no return value. This call returns SIG_ERR in case of error.

Though with signal() the respective signal handler as registered by the user can be called, fine tuning such as masking the signals that should be blocked, modifying the behavior of a signal, and other functionalities are not possible. This are possible using sigaction() system call.

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)
```

This system call is used to either examine or change a signal action. If the act is not null, the new action for signal signum is installed from the act. If oldact is not null, the previous action is saved in oldact.

The sigaction structure contains the following fields −

Field 1 − Handler mentioned either in sa_handler or sa_sigaction.

```
void (*sa_handler)(int);
void (*sa_sigaction)(int, siginfo_t *, void *);
```

The handler for sa_handler specifies the action to be performed based on the signum and with SIG_DFL indicating default action or SIG_IGN to ignore the signal or pointer to a signal handling function.

The handler for sa_sigaction specifies the signal number as the first argument, pointer to siginfo_t structure as the second argument and pointer to user context (check getcontext() or setcontext() for further details) as the third argument.

The structure siginfo_t contains signal information such as the signal number to be delivered, signal value, process id, real user id of sending process, etc.

Field 2 − Set of signals to be blocked.

```
int sa_mask;
```

This variable specifies the mask of signals that should be blocked during the execution of signal handler.

Field 3 − Special flags.

```
int sa_flags;
```

This field specifies a set of flags which modify the behavior of the signal.

Field 4 − Restore handler.

```
void (*sa_restorer) (void);
```

This system call returns 0 on success and -1 in case of failure.

Let us consider a few sample programs.

First, let us start with a sample program, which generates exception. In this program, we are trying to perform divide by zero operation, which makes the system generate an exception.

```
/* signal_fpe.c */
#include<stdio.h>

int main() {
   int result;
   int v1, v2;
   v1 = 121;
   v2 = 0;
   result = v1/v2;
   printf("Result of Divide by Zero is %d\n", result);
   return 0;
}
```

Compilation and Execution Steps

```
Floating point exception (core dumped)
```

Thus, when we are trying to perform an arithmetic operation, the system has generated a floating point exception with core dump, which is the default action of the signal.

Now, let us modify the code to handle this particular signal using signal() system call.

```
/* signal_fpe_handler.c */
#include<stdio.h>
#include<signal.h>
#include<stdlib.h>

void handler_dividebyzero(int signum);

int main() {
  int result;
  int v1, v2;
  void (*sigHandlerReturn)(int);
  sigHandlerReturn = signal(SIGFPE, handler_dividebyzero);
  if (sigHandlerReturn == SIG_ERR) {
    perror("Signal Error: ");
    return 1;
  }

 v1 = 121;
  v2 = 0;
  result = v1/v2;
  printf("Result of Divide by Zero is %d\n", result);
  return 0;
}

void handler_dividebyzero(int signum) {
  if (signum == SIGFPE) {
    printf("Received SIGFPE, Divide by Zero Exception\n");
    exit (0);
  }
  else
    printf("Received %d Signal\n", signum);
    return;
}
```

Compilation and Execution Steps

```
Received SIGFPE, Divide by Zero Exception
```

As discussed, signals are generated by the system (upon performing certain operations such as divide by zero, etc.) or the user can also generate the signal programmatically. If you want to generate signal programmatically, use the library function raise().

Now, let us take another program to demonstrate handling and ignoring the signal.

Assume that we have raised a signal using raise(), what happens then? After raising the signal, the execution of the current process is stopped. Then what happens to the stopped process? There can be two scenarios – First, continue the execution whenever required. Second, terminate (with kill command) the process.

To continue the execution of the stopped process, send SIGCONT to that particular process. You can also issue fg (foreground) or bg (background) commands to continue the execution. Here, the commands would only re-start the execution of the last process. If more than one process is stopped, then only the last process is resumed. If you want to resume the previously stopped process, then resume the jobs (using fg/bg) along with job number.

The following program is used to raise the signal SIGSTOP using raise() function. Signal SIGSTOP can also be generated by the user press of CTRL + Z (Control + Z) key. After issuing this signal, the program will stop executing. Send the signal (SIGCONT) to continue the execution**.**

In the following example, we are resuming the stopped process with command fg.

```
/* signal_raising.c */
#include<stdio.h>
#include<signal.h>
#include<stdlib.h>

int main() {
   printf("Testing SIGSTOP\n");
   raise(SIGSTOP);
   return 0;
}
```

Compilation and Execution Steps

```
Testing SIGSTOP
[1]+ Stopped ./a.out
./a.out
```

Now, enhance the previous program to continue the execution of the stopped process by issuing SIGCONT from another terminal.

```
/* signal_stop_continue.c */
#include<stdio.h>
#include<signal.h>
#include <sys/types.h>
#include <unistd.h>
```

```
void handler_sigtstp(int signum);

int main() {
   pid_t pid;
   printf("Testing SIGSTOP\n");
   pid = getpid();

   printf("Open Another Terminal and issue following command\n");
   printf("kill -SIGCONT %d or kill -CONT %d or kill -18 %d\n", pid, pid, pid);
   raise(SIGSTOP);
   printf("Received signal SIGCONT\n");
   return 0;
}
```

Compilation and Execution Steps

```
Testing SIGSTOP
Open Another Terminal and issue following command
kill -SIGCONT 30379 or kill -CONT 30379 or kill -18 30379
[1]+ Stopped ./a.out

Received signal SIGCONT
[1]+ Done ./a.out
```

In another terminal

```
kill -SIGCONT 30379
```

So far, we have seen the program which handles the signal generated by the system. Now, let us see the signal generated through program (using raise() function or through kill command). This program generates signal SIGTSTP (terminal stop), whose default action is to stop the execution. However, since we are handling the signal now instead of default action, it will come to the defined handler. In this case, we are just printing the message and exiting.

```
/* signal_raising_handling.c */
#include<stdio.h>
#include<signal.h>
#include<stdlib.h>

void handler_sigtstp(int signum);

int main() {
   void (*sigHandlerReturn)(int);
   sigHandlerReturn = signal(SIGTSTP, handler_sigtstp);
   if (sigHandlerReturn == SIG_ERR) {
      perror("Signal Error: ");
      return 1;
   }
   printf("Testing SIGTSTP\n");
   raise(SIGTSTP);
   return 0;
}

void handler_sigtstp(int signum) {
   if (signum == SIGTSTP) {
      printf("Received SIGTSTP\n");
      exit(0);
   }
   else
      printf("Received %d Signal\n", signum);
      return;
}
```

Compilation and Execution Steps

```
Testing SIGTSTP
Received SIGTSTP
```

We have seen the instances of performing default action or handling the signal. Now, it is time to ignore the signal. Here, in this sample program, we are registering the signal SIGTSTP to ignore through SIG_IGN and then we are raising the signal SIGTSTP (terminal stop). When the signal SIGTSTP is being generated that would be ignored.

```
/* signal_raising_ignoring.c */
#include<stdio.h>
#include<signal.h>
#include<stdlib.h>

void handler_sigtstp(int signum);

int main() {
   void (*sigHandlerReturn)(int);
   sigHandlerReturn = signal(SIGTSTP, SIG_IGN);
   if (sigHandlerReturn == SIG_ERR) {
      perror("Signal Error: ");
      return 1;
   }
   printf("Testing SIGTSTP\n");
   raise(SIGTSTP);
   printf("Signal SIGTSTP is ignored\n");
   return 0;
}
```

Compilation and Execution Steps

```
Testing SIGTSTP
```

Signal SIGTSTP is ignored

So far, we have observed that we have one signal handler to handle one signal. Can we have a single handler to handle multiple signals? The answer is Yes. Let us consider this with a program.

The following program does the following −

Step 1 − Registers a handler (handleSignals) to catch or handle signals SIGINT (CTRL + C) or SIGQUIT (CTRL + \)

Step 2 − If the user generates signal SIGQUIT (either through kill command or keyboard control with CTRL + \), the handler simply prints the message as return.

Step 3 − If the user generates signal SIGINT (either through kill command or keyboard control with CTRL + C) first time, then it modifies the signal to perform default action (with SIG_DFL) from next time.

Step 4 − If the user generates signal SIGINT second time, it performs a default action, which is the termination of program.

```
/* Filename: sigHandler.c */
#include<stdio.h>
#include<unistd.h>
#include<signal.h>

void handleSignals(int signum);

int main(void) {
  void (*sigHandlerInterrupt)(int);
  void (*sigHandlerQuit)(int);
  void (*sigHandlerReturn)(int);
  sigHandlerInterrupt = sigHandlerQuit = handleSignals;
  sigHandlerReturn = signal(SIGINT, sigHandlerInterrupt);
  if (sigHandlerReturn == SIG_ERR) {
    perror("signal error: ");
    return 1;
  }
  sigHandlerReturn = signal(SIGQUIT, sigHandlerQuit);

  if (sigHandlerReturn == SIG_ERR) {
    perror("signal error: ");
    return 1;
  }
  while (1) {
    printf("\nTo terminate this program, perform the following: \n");
    printf("1. Open another terminal\n");
    printf("2. Issue command: kill %d or issue CTRL+C 2 times (second time it terminates)\n",
getpid());
    sleep(10);
  }
  return 0;
}

void handleSignals(int signum) {
  switch(signum) {
    case SIGINT:
    printf("\nYou pressed CTRL+C \n");
    printf("Now reverting SIGINT signal to default action\n");
    signal(SIGINT, SIG_DFL);
    break;
    case SIGQUIT:
    printf("\nYou pressed CTRL+\\ \n");
    break;
    default:
    printf("\nReceived signal number %d\n", signum);
    break;
  }
  return;
}
```

Compilation and Execution Steps

```
To terminate this program, perform the following:
1. Open another terminal
2. Issue command: kill 74 or issue CTRL+C 2 times (second time it terminates)
^C
You pressed CTRL+C
Now reverting SIGINT signal to default action

To terminate this program, perform the following:
1. Open another terminal
2. Issue command: kill 74 or issue CTRL+C 2 times (second time it terminates)
^\You pressed CTRL+\
To terminate this program, perform the following:
1. Open another terminal
2. Issue command: kill 120
Terminated
```

Another Terminal

```
kill 71
```

Second Method

```
To terminate this program, perform the following:
1. Open another terminal
2. Issue command: kill 71 or issue CTRL+C 2 times (second time it terminates)
^C
```

```
You pressed CTRL+C
Now reverting SIGINT signal to default action

To terminate this program, perform the following:
1. Open another terminal
2. Issue command: kill 71 or issue CTRL+C 2 times (second time it terminates)
^C
```

We know that to handle a signal, we have two system calls i.e., either signal() or sigaction(). Till now we have seen with signal() system call, now it is time for sigaction() system call. Let us modify the above program to perform using sigaction() as follows −

```
/* Filename: sigHandlerSigAction.c */
#include<stdio.h>
#include<unistd.h>
#include<signal.h>

void handleSignals(int signum);

int main(void) {
   void (*sigHandlerReturn)(int);
   struct sigaction mysigaction;
   mysigaction.sa_handler = handleSignals;
   sigemptyset(&mysigaction.sa_mask);
   mysigaction.sa_flags = 0;
   sigaction(SIGINT, &mysigaction, NULL);

   if (mysigaction.sa_handler == SIG_ERR) {
      perror("signal error: ");
      return 1;
   }
   mysigaction.sa_handler = handleSignals;
   sigemptyset(&mysigaction.sa_mask);
   mysigaction.sa_flags = 0;
   sigaction(SIGQUIT, &mysigaction, NULL);

   if (mysigaction.sa_handler == SIG_ERR) {
      perror("signal error: ");
      return 1;
   }
   while (-1) {
      printf("\nTo terminate this program, perform either of the following: \n");
      printf("1. Open another terminal and issue command: kill %d\n", getpid());
      printf("2. Issue CTRL+C 2 times (second time it terminates)\n");
      sleep(10);
   }
   return 0;
}
```

```
void handleSignals(int signum) {
  switch(signum) {
    case SIGINT:
    printf("\nYou have entered CTRL+C \n");
    printf("Now reverting SIGINT signal to perform default action\n");
    signal(SIGINT, SIG_DFL);
    break;
    case SIGQUIT:

    printf("\nYou have entered CTRL+\\ \n");
    break;
    default:
    printf("\nReceived signal number %d\n", signum);
    break;
  }
  return;
}
```

Let us see the compilation and execution process. In the execution process, let us see issue CTRL+C twice, remaining checks/ways (as above) you can try for this program as well.

Compilation and Execution Steps

```
To terminate this program, perform either of the following:
1. Open another terminal and issue command: kill 3199
2. Issue CTRL+C 2 times (second time it terminates)
^C
You have entered CTRL+C
Now reverting SIGINT signal to perform default action
To terminate this program, perform either of the following:
1. Open another terminal and issue command: kill 3199
2. Issue CTRL+C 2 times (second time it terminates)
^C
```

VIII. Message mappings

The mmap() system call provides mapping in the virtual address space of the calling process that maps the files or devices into memory. This is of two types

File mapping or File-backed mapping − This mapping maps the area of the process' virtual memory to the files. This means reading or writing to those areas of memory causes the file to be read or written. This is the default mapping type.

Anonymous mapping − This mapping maps the area of the process' virtual memory without backed by any file. The contents are initialized to zero. This mapping is similar to dynamic memory allocation (malloc()) and is used in some malloc() implementations for certain allocations.

The memory in one process mapping may be shared with mappings in other processes. This can be done in two ways −

- When two processes map the same region of a file, they share the same pages of physical memory.
- If a child process is created, it inherits the parent's mappings and these mappings refer to the same pages of physical memory as that of the parent. Upon any change of data in the child process, different pages would be created for the child process.

When two or more processes share the same pages, each process can see the changes of the page contents made by other processes depending on the mapping type. The mapping type can be either private or shared −

Private Mapping (MAP_PRIVATE) − Modifications to the contents of this mapping are not visible to other processes and the mapping is not carried to the underlying file.

Shared Mapping (MAP_SHARED) − Modifications to the contents of this mapping are visible to other processes and mapping is carried to the underlying file.

```
#include <sys/mman.h>

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

The above system call returns the starting address of the mapping on success or MAP_FAILED on error.

The virtual address addr, can be either user specified or generated by the kernel (upon passing addr as NULL). The field length indicated requires the size of mapping in bytes. The field prot indicates memory protection values such as PROT_NONE, PROT_READ, PROT_WRITE, PROT_EXEC meant for regions that may not be accessed, read, write or executed respectively. This value can be single (PROT_NONE) or can be ORd with any of the three flags (last 3). The field flags indicate mapping type either or MAP_PRIVATE or MAP_SHARED. The field 'fd' indicates the file descriptor identifying the file to be mapped and the field 'offset' implies the starting point of the file, if need to map the entire file, offset should be zero.

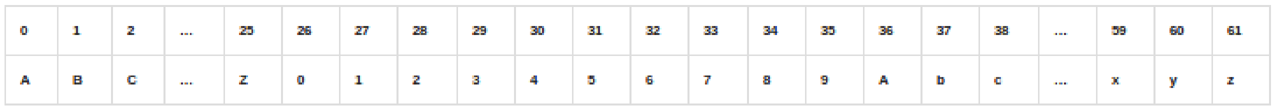
```
#include <sys/mman.h>

int munmap(void *addr, size_t length);
```

The above system call returns 0 on success or -1 on error.

The system call munmap, performs the unmapping of the already memory mapped region. The fields addr indicates the starting address of the mapping and the length indicates the size in bytes of the mapping to be unmapped. Usually, the mapping and unmapping would be for the entire mapped regions. If this has to be different, then it should be either shrinked or cut in two parts. If the addr doesn't have any mappings this call would have no effect and the call returns 0 (success).

Let us consider an example −

Step 1 − Writie into file Alpha Numeric characters as shown below



0	1	2	...	25	26	27	28	29	30	31	32	33	34	35	36	37	38	...	59	60	61
A	B	C	...	Z	0	1	2	3	4	5	6	7	8	9	A	b	c	...	x	y	z

Step 2 − Map the file contents into memory using mmap() system call. This would return the start address after mapped into the memory.

Step 3 − Access the file contents using array notation (can also access with pointer notation) as doesn't read expensive read() system call. Using memory mapping, avoid multiple copying between the user space, kernel space buffers and buffer cache.

Step 4 − Repeat reading the file contents until the user enters "-1" (signifies end of access).

Step 5 − Perform clean-up activities i.e., unmapping the mapped memory region (munmap()), closing the file and removing the file.

```
/* Filename: mmap_test.c */
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/mman.h>
void write_mmap_sample_data();

int main() {
   struct stat mmapstat;
   char *data;
   int minbyteindex;
   int maxbyteindex;
   int offset;
   int fd;
   int unmapstatus;
   write_mmap_sample_data();
   if (stat("MMAP_DATA.txt", &mmapstat) == -1) {
      perror("stat failure");
      return 1;
   }

if ((fd = open("MMAP_DATA.txt", O_RDONLY)) == -1) {
      perror("open failure");
```

```
    return 1;
  }
  data = mmap((caddr_t)0, mmapstat.st_size, PROT_READ, MAP_SHARED, fd, 0);

  if (data == (caddr_t)(-1)) {
    perror("mmap failure");
    return 1;
  }
  minbyteindex = 0;
  maxbyteindex = mmapstat.st_size - 1;

  do {
    printf("Enter -1 to quit or ");
    printf("enter a number between %d and %d: ", minbyteindex, maxbyteindex);
    scanf("%d",&offset);
    if ( (offset >= 0) && (offset <= maxbyteindex) )
    printf("Received char at %d is %c\n", offset, data[offset]);
    else if (offset != -1)
    printf("Received invalid index %d\n", offset);
  } while (offset != -1);
  unmapstatus = munmap(data, mmapstat.st_size);

  if (unmapstatus == -1) {
    perror("munmap failure");
    return 1;
  }
  close(fd);
  system("rm -f MMAP_DATA.txt");
  return 0;
}

void write_mmap_sample_data() {
  int fd;
  char ch;
  struct stat textfilestat;
  fd = open("MMAP_DATA.txt", O_CREAT|O_TRUNC|O_WRONLY, 0666);
  if (fd == -1) {
    perror("File open error ");
    return;
  }
  // Write A to Z
  ch = 'A';

  while (ch <= 'Z') {
    write(fd, &ch, sizeof(ch));
    ch++;
  }
  // Write 0 to 9
```

```
    ch = '0';

    while (ch <= '9') {
      write(fd, &ch, sizeof(ch));
      ch++;
    }
    // Write a to z
    ch = 'a';

    while (ch <= 'z') {
      write(fd, &ch, sizeof(ch));
      ch++;
    }
    close(fd);
    return;
}
```

Output

```
Enter -1 to quit or enter a number between 0 and 61: 3
Received char at 3 is D
Enter -1 to quit or enter a number between 0 and 61: 28
Received char at 28 is 2
Enter -1 to quit or enter a number between 0 and 61: 38
Received char at 38 is c
Enter -1 to quit or enter a number between 0 and 61: 59
Received char at 59 is x
Enter -1 to quit or enter a number between 0 and 61: 65
Received invalid index 65
Enter -1 to quit or enter a number between 0 and 61: -99
Received invalid index -99
Enter -1 to quit or enter a number between 0 and 61: -1
```