Inter Process Communication - I

Isolation and Interaction

- Different processes running on an OS are logically independent and isolated entities.
- They have separate logical memory spaces, CPU states, open files etc.
- An event in one process does not interfere with another process. One process may crush but the other processes will continue to run.

Isolation and Interaction

- But often it is also necessary for two processes or a process and the OS to interact. There are several reasons for that.
- It may be necessary to pass the output of one process as the input to another process.
- In a multiple processor system, dividing a job in several processes may achieve faster completion through parallelism.

Different Models

- So it is necessary to share information between two processes.
- There are three fundamental models of sharing information between processes.
- One is through shared memory between communicating processes and the other one is data transfer through the kernel buffer.

Shared Memory

- The address spaces of two processes are mutually disjoint.
- But a process may request the OS for some memory that it can share with other process.
- OS provides a physical memory where portions of logical address spaces of both the process are mapped.

Shared Memory

- Both processes can read from and write in the shared memory space. This allows them to communicate without any further interaction with the OS.
- But writing on the same memory location by more than one processes has the problem of data integrity of the memory location.

Data Transfer

- A data transfer may be a pure byte stream or in the form of a message.
- There is no shared memory in the user space. But there may be buffer maintained by the kernel to store byte stream or message.

Interprocess Communication on Linux

- We shall talk about some of the interprocess communication mechanisms available on Linux platform.
- These are pipe, named pipe, shared memory, Unix domain socket and signal.

A Note on File Descriptor

- We have already mentioned that a file descriptor is available for every open file; and a child process inherits the file descriptors of its parent at the time of creation.
- But in Unix/Linux many objects such as pipes, sockets, devices etc. are also treated as files.

A Note on File Descriptor

- A file descriptor is returned when these objects are opened by a `open()` system call.
- Data can be read from or write to these objects using the descriptors.

Unnamed Pipe

- A pipe is a unidirectional communication channel for byte stream[a] given by kernel to a requesting process.
- Data of any block size can be written in a pipe and read from a pipe. There is no concept of message.

[a] The kernel maintains a FIFO buffer in its space.

Unnamed Pipe

- A pair of file descriptors are associated to a pipe. One of them is used to read from and the other one is to write into the pipe.
- If two processes share the file descriptors of a pipe, then the data of one can be passed to the other.

Unnamed Pipe

- In the following example the command interpreter `bash` redirects the output of `/bin/ls` as input to `/bin/less` using `pipe`.
- `ls -l` displays the files and subdirectories under the current directory.
- `less` facilitates the display of the stream of data on the VDU screen.

Unnamed Pipe

```
$ ls -l /usr/include | less
total 1236
-rw-r--r--.  1 root root   7445 Mar  6  2015 aio.h
-rw-r--r--.  1 root root   2050 Mar  6  2015 alia
drwxr-xr-x.  2 root root   4096 May 15  2015 asm
.......................
-rw-r--r--.  1 root root   2268 Mar  6  2015 cpio
-rw-r--r--.  1 root root   5938 May 13  2015 cpuf
:
```

Unnamed Pipe

- The shell opens a pipe, and creates two child processes using `fork()`. One (c_1) is loaded with `/bin/ls` and the other one (c_2) with `/usr/bin/less` using `exec()` calls.
- The `ls` writes its output on `stdout` and the `less` takes input from the `stdin`.

Unnamed Pipe

- The shell before `exec()` redirects the output descriptor of c_1 to the write-end of the pipe. It also redirects the input descriptor of c_2 to the read-end of the pipe.
- After `exec()` calls `ls` (c_1) and `less` (c_2) are loaded. They inherit the descriptors (but not 'aware' of redirections) and act normally.

Unnamed Pipe

- Following program gives a system call to open an unnamed pipe
- Creates a child process so that the parent and the child share the file descriptors of the pipe.
- Then they communicate through the pipe.

Communication Through Pipe

```
#include <iostream>
using namespace std;
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {  // pipe1.c++
```

```
int chpid, fd[2], err, status ;

err = pipe(fd) ;
if(err == -1) {
   cerr << "pipe open error\n" ;
   return 0;
}
chpid = fork();
if(chpid == -1){
  cerr << "fork() error\n";
  return 0;
```

```
    }
    if(chpid > 0){ // write in parent
       char buffP[100] = "IIIT Kalyani";
       close(fd[0]);
       write(fd[1], buffP, strlen(buffP));
       cout << "Parent has written in pipe\n";
       close(fd[1]);
       waitpid(chpid, &status,0);
    }
    else { // child
       char buffC[100]={0};
```

```
        close(fd[1]);
        sleep(5);
        read(fd[0], buffC, 100);
        cout << "Child: " << buffC << endl;
        close(fd[0]);
    }
    return 0;
}
```

Communication Through Pipe

Output:

```
$ a.out
Parent has written in pipe
Child: IIIT Kalyani
```

Communication Through Pipe

- The system call `pipe(fd)` creates a FIFO data channel that can be used for interprocess communication.
- Two file descriptors are available in the two-element integer array `fd[2]` - `fd[1]` refers to write into and `fd[0]` refers to read from the pipe.
- Data written is buffered by the Kernel.

Communication Through Pipe

- During `fork()` the file descriptors of a pipe are copied to the child process along with other open file descriptors e.g. `0 (stdin)`, `1 (stdout)`, `2 (stderr)`.
- The parent process closes the input descriptor `fd[0]` and uses `fd[1]` to write in the pipe. On the other hand the child process closes the output descriptor and uses `fd[0]` to read data.

Communication Through Unnamed Pipe in Python

```
#!/usr/bin/python
# pipe2.py creates a pipe, parent-child
#                  communicates through it
import os,sys, time
def main():
    try:
        fdr,fdw = os.pipe()
    except:
        OSError
```

```
        print "Pipe-open fails"
        sys.exit(1)
    try:
        chPID = os.fork()
    except:
        OSError
        print "fork() fails"
        sys.exit(1)
    if chPID > 0:
        os.close(fdr)
        n = os.write(fdw, 'IIT Kalyani')
```

```
        print 'Parent has written in pipe'
        os.waitpid(chPID,0)
    else:
        os.close(fdw)
        data = os.read(fdr, 100)
        time.sleep(5)
        print 'child:', data
main()
```

Communication Through Unnamed Pipe in Python

Output:

```
$ ./pipe2.py
Parent has written in pipe
child: IIIT Kalyani
```

Communication Through Pipe

- The call `os.pipe` returns a 2-tuple of file descriptors. The first one is for read and the second one is for write.
- The call `os.write(fdw, str)` writes the byte string of `str` to the file of the descriptor `fdw`.
- The call `os.read(fdr, n)` reads n bytes and returns the byte string.

Close Unused Descriptor

- It is necessary for a process reading from a pipe to close its write descriptor (`fd[1]`). (`pipe4a.c++`)
- Similarly it is also necessary for a process writing in a pipe to close its read descriptor (`fd[0]`). (`pipe4.c++`)

State of Reader/Writer Process

- What is the state of the reader process (child in our example) if the writer (parent in this case) is not writing in the pipe? (`pipe5a.c++`)
- What is the state of the writer if the reader is not reading? (`pipe5b.c++`)

Can There be More than One Reader/Writer

- Can more than one process write in a pipe and similarly can more than one process read from a pipe? (`pipe6.c++`)
- Will the write operation be atomic for a process?

`close()` and `dup()`

- The system call close(fd) closes the open file corresponding to the file descriptor fd.
- The slot corresponding to fd in the file descriptor table is free.
- The system call dup(fd1) copies the file descriptor of fd1 in the least index available in the file descriptor table.

Redirecting Output

```
#include <iostream>
using namespace std;
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
```

```
int main(int ac, char *av[]){
    int fd1;    // dupTOstdout1.c++
                // $ ./a.out dupOut
    if(ac < 2){
      cerr << "File name not specified\n";
      exit(1);
    }
    fd1 = open(av[1], O_CREAT | O_WRONLY, 0666);
    if(fd1 == -1){
      cerr << "File open error\n";
      exit(1);
```

```
    }
    cout << "Line before close(fileno(stdout))\n"
    close(fileno(stdout));
    cout << "Line after close(fileno(stdout))\n";
    dup(fd1);
    cout << "Line after dup(fd1)\n";

    close(fd1);
    return 0;
}
```

Redirecting Output

```
$ a.out dupOut
Line before close(fileno(stdout))
dupOut: Line after dup(fd1)
```

`close()` and `dup2()`

- There is a similar system call `dup2(ofd, nfd)` makes `nfd` a copy of the old file descriptor `ofd`.
- If there is an open file with the file descriptor nfd, it is closed.
- If the call succeeds, both ofd and nfd refers to the same entry of the open file table.

Standard IO and IPC on Pipe

- As an example we use `close()` the file descriptor of `stdin` (stdout).
- Then call `dup2()` to duplicate the input (output) file descriptor of the opened pipe to the file descriptor of `stdin` (stdout).
- Now the `stdio` library functions can be used to read from (write to) the pipe.

stdio, dup2(), pipe()

```
#include <iostream>
using namespace std;
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {  // pipe3.c++
```

```
int chpid, fd[2], err, status ;

err = pipe(fd) ;
if(err == -1) {
   cerr << "pipe open error\n" ;
   exit(1) ;
}
chpid = fork();
if(chpid == -1){
  cerr << "fork() error\n";
  exit(1);
```

```
        }
        if(chpid > 0){ // in parent
          int n;
          close(fd[0]);
          cout << "parent: Enter a +ve integer: ";
          cin >> n;
          cout << "parent: " << n << " is the input
//          dup2(fileno(stdout), fd[1]+1);
                // copy stdout (1) to fd[1]+1
          close(fileno(stdout)) ; // close stdout
          cout << "Cannot be printed\n";
```

```
        dup2(fd[1], fileno(stdout));
        cout << n << "\n";
        waitpid(chpid, &status,0);
      }
      else { // child process
        int m;
        close(fd[1]);
//        dup2(fileno(stdin), fd[1]+1);
        close(fileno(stdin)) ;
        dup2(fd[0], fileno(stdin));
        cin >> m;
```

```
            cout << "data " << m << " received in chi
        }
    return 0;
}
```

Output:

```
$ ./a.out
parent: Enter a +ve integer: 100
parent: 100 in the input
data 100 received in child
```

Named Pipe

- The system call `mkfifo()` creates a named pipe.
- The special file created by this call is similar to anonymous communication channel pipe, but is entered in the file system as a named object.
- Once created, any process with proper permission can open it for read or write.

Named Pipe

```
#include <iostream>
using namespace std;
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
```

```
#define MAX 100
// namedPipe1.c++
// $ ./a.out r <pipeName> &  $ ./a.out w <pipeNam
int main(int count, char *vect[]) {
    int err, pd ;
    char wBuff[] = "This text will be written in
         rBuff[MAX] = {0};

    if(count < 3) {
         cerr << "Less number of arguments\n" ;
         exit(1) ;
```

```
    }
    err = mkfifo(vect[2], 0666) ;
    if(err == -1 && errno != EEXIST){
       cerr << "errno: " << errno << "\n";
       exit(1);
    }
    if(strcmp(vect[1], "r") == 0) { // Reader pr
         pd = open(vect[2], O_RDONLY) ;
         read(pd, rBuff, MAX);
         cout << "OutData: " << rBuff << "\n" ;
         close(pd);
```

```
    }
    else if(strcmp(vect[1], "w") == 0) { // Writ
            pd = open(vect[2], O_WRONLY) ;
            write(pd, wBuff, strlen(wBuff)) ;
            close(pd);
    }    else {
            cerr << "Wrong 2nd argument\n" ;
            exit(1) ;
         }
    return 0;
}
```

Named Pipe

- If a process opens a FIFO for reading (`O_RDONLY`), gets blocked, if it is not opened by another process for writing. This is true for opening in writing mode also.
- A named FIFO can be opened from a shell - `$ mkfifo -m mode pathname`.

Named Pipe

```
/*
 * fifoRead.c++ shows that the process is blocked
 *                    as there is no writing process
 * $ mkfifo -m 0666 myFIFO
 * $ g++ -Wall fifoRead.c++ -o fifoRead
 * $ g++ -Wall fifoWrite.c++ -o fifoWrite
 * $ ./fifoRead myFIFO &
 * $ ./fifoWrite myFIFO &
 */
```

```
#include <iostream>
using namespace std;
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#define MAX 100
```

```
int main(int ac, char *av[]) {
    int pd;
    char buff[MAX];

    if(ac < 2){
      cerr << "FIFO name not specified\n";
      exit(1);
    }
    pd = open(av[1], O_RDONLY);
    if(pd == -1){
      cerr << "FIFO open error\n";
```

```
        exit(1);
    }
    cout << "Not printed until fifoWrite\n";
    read(pd, buff, 100);
    cout << "Data read: " << buff << endl;

    close(pd);
    return 0;
}
```

The `fifoWrite.c++` is similar.

Named Pipe in Python

```
#!/usr/bin/python
# namedPipe2.py creates a named pipe
# $ ./namedPipe2.py r <fileName> &
# $ ./namedPipe2.py w <fileName> &
import os
import sys
def main():
    try:
        os.mkfifo(sys.argv[2], 0666)
```

```
    except: OSError
    try:
        if sys.argv[1] == 'r':
            fd = os.open(sys.argv[2], os.O_RDONLY)
            data = os.read(fd, 100)
            print data
        elif sys.argv[1] == 'w':
            fd = os.open(sys.argv[2], os.O_WRONLY)
            os.write(fd, "\nWrittten in the named p
    except:  print 'wrong argument'
main()
```

Shared Memory

- A process can send a request to the OS to allocate a block of shared memory.
- It can be attached to the virtual address spaces of two or more cooperating processes.
- Once the shared memory is attached, process can access the memory for read and write without any intervention of the kernel.

Shared Memory

- This makes communication through a shared memory more efficient than a pipe where data is buffered in the kernel space, and every access requires a system call.
- But then there is a price to pay - it is necessary to synchronize read and write operations of different processes for data consistency.

Shared Memory

- The original shared memory API on Linus is from System V.
- Subsequently the POSIX (Portable Operating System Interface) API was implemented.
- System V shared memory is identified by a key and an identifier. The POSIX shared memory API is similar to that of a file.

Shared Memory

- A key and an identifier is associated with a System V shared memory segment.
- The key is the name of the shared memory, and the identifier is used within the program by other related functions.

Shared Memory

```
/*
 Creating a shared memory segment and attaching it
 to the logical address space.  sharedMem1.c++
 $ g++ -Wall sharedMem1.c++
 $ ./a.out w
 $ ./a.out r
*/
#include <iostream>
using namespace std;
```

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SIZE 4

int main(int count, char *vect[]) {
int shmID, *p ;

if(count < 2) {
           cerr << "No 2nd argument\n";
```

```
            exit(1) ;
}
shmID = shmget(ftok("/home/goutam", 1234), SIZE,
if(shmID == -1) {
cerr << "Error in shmget" ;
exit(1) ;
}
p = (int *) shmat(shmID, 0, 0777) ;
        cout << "Attached at VA: " << p << endl;
if(vect[1][0] == 'w') {
            cout << "Enter an integer: ";
```

```
              cin >> *p ;  // Write data
    shmdt(p) ;
}
         else if(vect[1][0] == 'r'){
                 cout << "The data is:" << *p << "
         shmdt(p) ;
             }
// The shared memory segment remains in the syste
// $ ipcs $ ipcrm -m<number>
return 0 ;
}
```

Output

```
$ ./a.out w
Attached at VA: 0x7fea66c67000
Enter an integer: 100
$ ./a.out r
Attached at VA: 0x7f44e118e000
The data is:100
```

Shared Memory

- The function `ftok()` creates a key from its parameters.
- The system call `shmget()` takes three parameters - a key, the size of the requested memory[a], and a set of flags.

[a]The actual size of the shared memory is normally the smallest multiple of the page size $\leq$ the requested size.

Shared Memory

- The return value of `shmget()` is either a +ve integer, an identifier of the allocated shared memory segment, or -1 in case of a failure.
- The identifier is used in the subsequent calls.

Shared Memory

- The system call shmat() attaches the shared memory specified by the first parameter (`shmID`) to an unused portion of the logical address space of the process[a].
- The third parameter specifies the access permission to the shared memory.

[a] Often it is the space between the stack and the heap. This may be modified by the second parameter.

Shared Memory

- The call returns the logical address of the point of attachment, which then is bound to some local variable (`p` in the example).
- Finally the memory can be detached from the process by the system call `shmdt()`.

Shared Memory

- Even though the shared memory is not attached to any process, it remains available in the system. It can be identified by its key.
- It can be viewed by the command `$ ipcs` and can be removed by the command `$ ipcrm -m <shmid>`.

Shared Memory

- It also can be removed using the system call `shmctl()`.
- In our program the requested shared memory is only 4 bytes. But OS does not deal with this granularity. It allocates in multiple of pages.

Shared Memory

```
/*
 Creating a shared memory segment and attaching it
 to the logical address space.  sharedMem2.c++
 Its logical address, size and removal
*/
#include <iostream>
using namespace std;
#include <stdlib.h>
#include <sys/types.h>
```

```
#include <sys/ipc.h>
#include <sys/shm.h>
#define SIZE 4
#define MAXSIZE 4095 // 16KB

int main() {
int shmID, *p;
        struct shmid_ds buff;

shmID = shmget(ftok("/home/goutam", 1234), SIZE,
                           IPC_CREAT | 0777);
```

```
if(shmID == -1) {
cerr << "Error in shmget";
exit(1) ;
}
p = (int *) shmat(shmID, 0, 0777);
        cout << "Shared memory address: "
             << (void *) p << "\n";
        p[0]=0; p[MAXSIZE]=MAXSIZE;
        cout << "data: " << p[0] << "-"
             << p[MAXSIZE] << "\n";
shmdt(p) ;
```

```
        shmctl(shmID, IPC_RMID, &buff);

return 0 ;
}
```

```
$ a.out
Shared memory address: 0x7f6955903000
data: 0-4095
```

Size of Shared Memory

- MAXSIZE is changed from 4095 to 4096.

```
$ a.out
Shared memory address: 0x7f238c4a3000
Segmentation fault (core dumped)
```

- 16KB shared memory allocated.

POSIX Shared Memory APIs

```
/*
 * Creating a shared memory segment with POSIX AP
 * attaching it to the logical address space.
 $ g++ -Wall sharedMem1a.c++ -lrt
 $ ./a.out w
 $ ./a.out r
*/
#include <iostream>
using namespace std;
```

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#define SIZE 4

// sharedMem1a.c++
int main(int count, char *vect[]) {
```

```
int *p, shmD ;

if(count < 2) {
   cerr << "No 2nd argument\n";
   exit(1) ;
}
shmD = shm_open("/myShm", O_CREAT | O_RDWR,
                                     0777);
if(shmD == -1){
   cerr << "shm_open() error\n";
   exit(1);
```

```
}
if(ftruncate(shmD, SIZE) == -1){
   cerr << "ftruncate() error\n";
   exit(1);
}
p = (int *)mmap(NULL, SIZE,
                PROT_READ | PROT_WRITE,
                MAP_SHARED, shmD, 0) ;
if(p == MAP_FAILED){
   cerr << "mmap() error\n";
   exit(1);
```

```
    }
    cout << "Attached at VA: " << p << endl;
    if(vect[1][0] == 'w') {
       cout << "Enter an integer: ";
       cin >> *p ;  // Write data
    }
    else if(vect[1][0] == 'r') // read data
            cout << "The data is:" << *p << "\n";
    // shm_unlink("/myShm");
   return 0 ;
}
```

POSIX Shared Memory APIs

- `shm_open()`: opens a shared memory and returns the descriptor.
- `ftruncate()`: used to set the size of the shared memory[a]
- `mmap()`: maps the shared memory in the virtual space and returns the attachment address. Subsequently the memory locations can be accessed using the address.

[a] The call `shm_open()` opens a shared memory with size zero.

Output

```
$ a.out w
Attached at VA: 0x7fc34034f000
Enter an integer: 100
 $ a.out r
Attached at VA: 0x7ff9f6bdf000
The data is:100
 $ ls -l /dev/shm
-rwxrwxr-x 1 goutam goutam 4 Jul 24 15:40 myShm
 .....
```

Race in Shared Memory

- Following example shows race in the shared memory.
- The shared location `p[0]` is initialized to 0.
- A child process is created. The location `p[0]` is decremented 5×10^6 times in the child process.

Race in Shared Memory

- The location `p[0]` is incremented 5×10^6 times in the parent process.
- The expected final result is 0.
- But every run gives different output.

Race in Shared Memory

```
/*
 Race in shared memory
  $ g++ -Wall sharedMem4.c++
  $ ./a.out 5000000
*/
#include <iostream>
using namespace std;
#include <stdlib.h>
#include <sys/types.h>
```

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <unistd.h>
#define SIZE 4

int main(int count, char *vect[]) {
    int shmID, *p, cPID, n, status ;
    struct shmid_ds buff;

    if(count < 2) {
```

```
        cerr << "No 2nd argument\n";
        exit(1) ;
    }
    shmID = shmget(ftok("/home/goutam", 1234), SI
                                IPC_CREAT | 0777);
    if(shmID == -1) {
        cerr << "Error in shmget" ;
        exit(1) ;
    }
    p = (int *) shmat(shmID, 0, 0777);
    p[0] = 0; // shared memory initialized to 0
```

```
n = atoi(vect[1]);
cPID = fork();
if(cPID == -1){
    cerr << "fork() error\n";
    shmdt(p);
    shmctl(shmID, IPC_RMID, &buff);
    exit(1);
}
if(cPID > 0){ // parent
    int i;
```

```
        for(i=1; i<=n; ++i) p[0]=p[0]+1;
        waitpid(cPID, &status, 0);
        cout << "p[0]: " << p[0] << "\n";
    }
    else { // child
        int i;
        for(i=1; i<=n; ++i) p[0]=p[0]-1;
    }
    shmdt(p);
    shmctl(shmID, IPC_RMID, &buff);
    return 0 ;
```

```
}
```

Race in Shared Memory

```
$ a.out 5000000
p[0]: 12440
$ a.out 5000000
p[0]: -2043936
$ a.out 5000000
p[0]: -1069027
```

Race in Shared Memory and Synchronization

- The reason for this peculiar output is due to race condition.
- Two concurrent processes are accessing the shared location `p[0]`. But in different runs the access are interleaved in different ways to produce different results.

Race in Shared Memory and Synchronization

- It is necessary to avoid interleaving of low level operations of increment and decrement.
- It is necessary to make these operations atomic i.e. one cannot take place unless the other is complete.

Concurrent Access of Shared Resource

- Race condition - computation is not deterministic.
- Critical section - portion of code that access a shared resource.
- Mutual exclusion - no two critical sections executed concurrently.
- Atomic - execution of critical section is logically uninterruptible.

Message Queue

- Message queue is another method for communication between two processes.
- It is similar to pipe and FIFO, but it is message oriented. The reader receives the whole message sent by the writer.
- Unlike pipe, it is not possible to read a part of it (a few bytes) leaving the rest in the queue.

POSIX Message Queue

```
/*
 * msgQ1.c++ POSIX message queue
 * $ g++ -Wall msgQ1.c++ -lrt
 * $ sudo ./a.out w; ./a.out r
 */
#include <iostream>
using namespace std;
#include <fcntl.h>
#include <sys/stat.h>
```

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

#define MSGSIZE 1024
#define MAXMSG  16
```

```
int main(int ac, char *av[]){
    struct mq_attr attr;
    int err, msgLen;
    mqd_t mqd;

    if(ac < 2){
      cerr << "r/w not specified\n";
      exit(1);
    }
    attr.mq_maxmsg = MAXMSG;
    attr.mq_msgsize = MSGSIZE;
```

```
attr.mq_flags = 0;
attr.mq_curmsgs = 0;

if(av[1][0] == 'w'){
   char buff[MSGSIZE];
   int prio=0;

   mqd = mq_open("/myMq", O_WRONLY | O_CREAT,
   if(mqd == -1){
     cerr << "mq_open() problem: " << errno <
     exit(1);
```

```
}
cout << "Enter message (terminate with Ctr
while(1) {
   cin.getline(buff, MSGSIZE);
   err = mq_send(mqd, buff, strlen(buff), 
   if(err == -1){
     cerr << "mq_send() fails\n";
     exit(1);
   }
   if(cin.eof()) break;
}
```

```
}
if(av[1][0] == 'r'){
   char buff[MSGSIZE];
   mqd = mq_open("/myMq", O_RDONLY | O_CREAT,
   if(mqd == -1){
     cerr << "mq_open() problem: " << errno <
     exit(1);
   }
   cout << "Reader reads message: \n";
   while((msgLen = mq_receive(mqd, buff, MSGS
     buff[msgLen]='\0';
```

```
            if(msgLen != 0)
              cout << "Received message: " << buff <
        }
    }
    mq_close(mqd);
    return 0;
}
```

struct mq_attr

```
struct mq_attr
{
 long mq_flags;   /* Message queue flags.  */
 long mq_maxmsg;  /* Maximum number of messages.
 long mq_msgsize; /* Maximum message size.  */
 long mq_curmsgs; /* Number of messages currently
};
```

Note

- Messages are ordered in the queue in descending order of priority, a non-negative integer where zero (0) is of lowest priority.
- If the queue is empty, the process of `mq_receive()` is blocked unless the queue is opened with `O_NONBLOCK` flag.

Signals

- A signal is a mechanism to notify a process about an event.
- It is a short message, a number, sent to a process or a set of processes through the OS. It does not have any other parameter.
- A signal may be raised (sent) explicitly by a process for another process through a system call e.g. `kill()`.

Signals

- It may be raised due to some event e.g. memory permission violation, divide-by zero, illegal instruction etc. from a running process.
- It may also be raised by external events e.g. keyboard interrupt e.g. `Ctrl-C` or `Ctrl-Z`.

Signals

- Any occurrence of such event suspends the normal execution of the running process, and the control is transferred to the kernel.
- The kernel updates the data structure of the target process for the signal.
- A signal is delivered when the process starts running.

Signals

- So a signal may remain pending for a suspended process.
- There can be only one pending signal of a particular type per process (no queue).
- The OS checks for pending signals of the process before it going to be scheduled.

Signals

- Every time the mode switches from the **kernel** to the **user** the check for pending signal is done for the scheduled process.
- If the pending signal cannot be ignored, it is **handled** by switching to the corresponding signal handler or taking **default** action.

Signals

- Once the signal handler finishes its job, the original execution of the process may be restarted.
- There are three possible responses on a delivered signal - it may be ignored, some default action may be taken, or handled by the corresponding signal-handler.

Ctrl-C Ctrl-Z Ctrl-\

```
/*
  Ctrl-C terminates the current process:
  $ ./a.out
  Press Ctrl-C to terminate
  Execute again
  Press Ctrl-Z to suspend
 $ fg    to restart
 Try Ctrl-\
*/
```

```
#include <iostream>
using namespace std;
int main(){ // ctrlC.c++
    while(1)
        cout << "What next...\n";
    return 0 ;
}
```

Ctrl-C Ctrl-Z Ctrl-\

- `Ctrl-C` sends SIGINT signal to the foreground process. The default action is to terminate the process.
- `Ctrl-Z` sends SIGTSTP (terminal stop) signal to the foreground process. The default action is to suspend the process.
- The command `fg` resumes the current job in the foreground.

Ctrl-\ and kill

- Ctrl-\ sends SIGABORT aborts the foreground process. The default action is to terminate the process.
- $ kill PID terminates a process.

kill

```
$ ps
9114 pts/2    00:00:00 bash
9709 pts/2    00:00:00 a.out
9711 pts/2    00:00:00 ps
$ kill 9709
```

kill

- `kill 9709` sends the signal SIGKILL to the process with PID 9709.

```
$ ps
  PID TTY          TIME CMD
 9114 pts/2    00:00:00 bash
 9709 pts/2    00:00:00 a.out
 9716 pts/2    00:00:00 ps
```

- But it is not killed!

kill

```
$ fg
a.out
Terminated
```

The command `fg` restarts `a.out` and the signal `SIGKILL` is delivered.

```
$ ps
  PID TTY          TIME CMD
 9114 pts/2    00:00:00 bash
 9752 pts/2    00:00:00 ps
```

System Call `kill()`

- The system call `kill(pid, sig)` can be used to send signal `sig` to a process of `pid`.
- Following is an example.

System Call kill()

```
/*
  kill1.c++ signal from child to parent
*/
#include <iostream>
using namespace std;
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
```

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

int main() { // kill1.c++
    int cPID, status ;

    cPID = fork();
    if(cPID == -1){
      perror("fork() failed\n");
      exit(1);
    }
    if(cPID > 0) {
```

```
        while(1){
            cout << "Parent running...\n";
            sleep(1);
        }
        waitpid(cPID, &status, 0) ;
    }
    else {  // child
         int pPID = getppid();

         sleep(5);
         kill(pPID, SIGTSTP);
```

```
        cout << "SIGTSTP sent to parent\n";
        sleep(5);
        cout << "SIGCONT sent to parent\n";
        kill(pPID, SIGCONT);
        sleep(5);
        cout << "SIGINT sent to parent\n";
        kill(pPID, SIGINT);
    }
    return 0 ;
}
```

Signal Handling

- Each signal has its default action. Often it terminates the receiving process[a].
- But most of the signals can be caught and handled by the signal handler supplied by the user.
- SIGKILL and SIGSTOP cannot be caught.

[a] SIGVHLD is ignored by default. SIGCONT resumes the stopped process.

Signal Handling

- A program can use the library function `signal`.

```
typedef void (*sighandler_t)(int)
sighandler_t signal(int sig,
sighandler_t handler)
```

- `signal` is a function that takes two parameters.

Signal Handling

- The first parameter `sig` is the signal to catch.
- The second parameter `handler` is the function to be called when the signal specified by the first parameter is received.
- `handler` can also take special values `SIG_IGN` or `SIG_DFL`.

Signal Handling

- If `handler` is set to `SIG_IGN`, the signal is ignored.
- If it is set to `SIG_DFL`, the default action associated with the signal takes place.
- If it is a function, then it is invoked with `sig` as the argument.

Signal Handling

- The return type of `signal()` is same as that of its second parameter.
- It returns the previous value of the signal handler or error.

Ignoring SIGINT

```
/*
 sigHand1.c++ Ignoring SIGINT (Ctrl-C)
*/
#include <iostream>
using namespace std;
#include <signal.h>
#include <unistd.h>

void mySigHandler(int n) {
```

```
     static int m = 1;
     if(m > 2) signal(SIGINT, SIG_DFL);
     else signal(SIGINT, mySigHandler);
             // <ctrl-C> default
     cout << "In handler: "<< m << "\n";
     ++m;
}
int main() {
    signal(SIGINT, mySigHandler) ;
          // <Ctrl-C> mySignalHandler()
    while(1) {
```

```
        cout << "What next?...\n";
        sleep(1);
    }
    return 0 ;
}
```

Ignoring SIGINT

- The program `sigHand1.c++` ignores the signal SIGINT (`Ctrl-C`) three times.
- Then SIGINT takes its default action.
- The name of the signal handler is `mySigHandler()`.

Memory Violation

- Access to illegal memory segment generates the signal `SIGSEGV`.
- We often encounter this while using pointer variable.
- This exception cannot be ignored as the offending instruction will be tried again.

SIGSEGV

```
/*
 sigHand2.c++ SIGSEGV handler
*/
#include <iostream>
using namespace std;
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
```

```
void mySEGVhandler(int sig){
     signal(sig, SIG_IGN);
                    // SEGV
     sleep(1);
     cout << "In Handler\n" ;
}
int main() {
        int *p = (int *)100 ;

        signal(SIGSEGV, mySEGVhandler);
                        // SEGV mySEGVhandler()
```

```
        *p = 10 ;
        cout << "Not printed\n" ;
        return 0 ;
}
```

Bibliography

1. https://www.tutorialspoint.com/python/os_pipe.htm
2. *Beginning Linux Programming* by Neil Mathew & Richard Stones, 3^{rd} ed., Wiley Pub., 2004, ISBN 81-265-0484-6.
3. *Understanding the Linux Kernel* by Daniel P Bovet & Marco Cesati, 3^{rd} ed., O'Reilly, ISBN 81-8404-083-0.
4. http://www.comptechdoc.org/os/linux/programming/linux_pgsignals.html