POLITECNICO
DI TORINO

Inter Process Communication

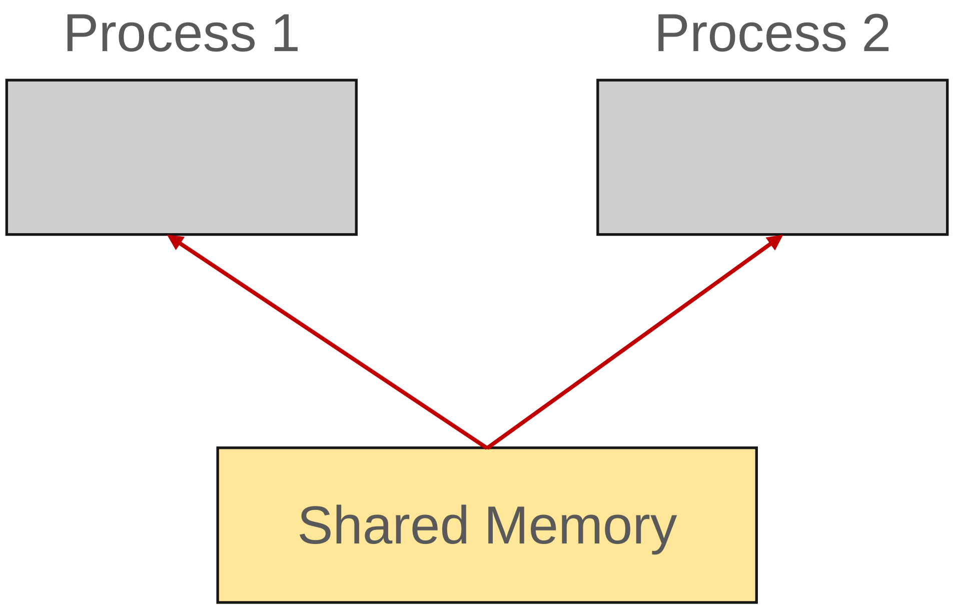
Operating Systems – Sarah Azimi

OUTLINE

- IPC mechanisms:
    - Shared Memory
    - **Message Passing**
    - **Pipes (named and unnamed)**
    - **Signals**

Shared Memory

- Shared memory is a memory shared between two or more processes.



Shared Memory

- Shared memory is a memory shared between two or more processes.

- Create the shared memory segment or use already shared memory segment
  - `shmget()`
- Attached the process to 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()`

- `#include <sys/stat.h>, #include <sys/ipc.h>, #include <sys/types.h>`

Creating a Shared Memory Segment

- The system call which requests a shared memory segment is shmget()
  - Syntax:

```
Shm_id =  shmget  (key_t  key, int  size  , int   flag);
```

Creating a Shared Memory Segment

- The system call which requests a shared memory segment is shmget()
  - Syntax:

```
Shm_id =  shmget  (key_t  key, int  size  , int   flag);
```

- Key is of type key_t. It is an integer that specifies which segment to create:
  - `IPC_PRIVATE` For creating a new private shared memory segment that is accessible only by its creating process or child processes of that creator.
  - Unrelated processes can access the same public shared segment by specifying the same key value.

Creating a Shared Memory Segment

- The system call which requests a shared memory segment is shmget()
  - Syntax:

```
Shm_id =  shmget  (key_t  key, int  size  , int   flag);
```

- The argument Size determines the size in byte of the shared memory segment.

Creating a Shared Memory Segment

- The system call which requests a shared memory segment is shmget()
  - Syntax:

```
Shm_id =  shmget  (key_t  key, int  size  , int  flag);
```

- The argument flag is used to indicate segment creation conditions and access permissions.
  - IPC_CREATE : A new segment should be created.
  - IPC_EXCL : It causes `shmget` to fail if the specified segment key already exists.
  - Mode flags: This value is made of 9 bits indicating permissions granted to owner, group and world to control access to the segment. Execution bits are ignored.

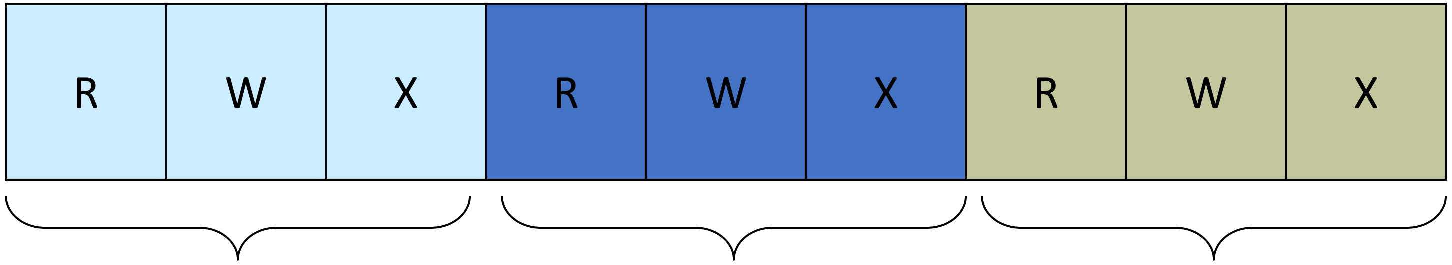
Mode Flags: Access Control

To whom permissions apply:
u >> owner (file’s owner)
g >> group (users who are members of the file’s group)
o >> others (who are neither the owner or member of file’s group)

Access rights are:

read >> r
write >> w
executable >> x

read >> 4
write >> 2
executable >> 1
without right >> 0



R	W	X	R	W	X	R	W	X

user	group	others
S_IRUSR	S_IRGRP	S_IROTH
S_IWUSR	S_IWGRP	S_IWOTH
S_IXUSR	S_IXGRP	S_IXOTH
S_IRWXU	S_IRWXG	S_IRWXO

Generating a unique Key

- Type `key_t` is an integer number. You can use any number you want.
- A special function, `ftok()` has been introduced to generate a unique key, from two arguments:
    - `key_t ftok(const char *path, int id);`
    - `path` is a file that this process can read
    - `id` is usually just set to some arbitrary char, like 'A'.
- The ftok() function uses information about the named file (in particular its inode number) and the id to generate a probably-unique key for the shared memory segment.

Example:

```
key_t key;
int shmid;
key = ftok ("<somefile>", 'A');
shmid = shmget (key, 1024, 0644 | IPC_CREAT);
```

Attach a shared memory segment

- `shmat` is used to attach the referenced shared memory segment into the calling process's data segment.
    - Syntax:

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

    - `shmid` is a valid shared memory identifier.
    - `shmaddr` allows the calling process some flexibility in assigning the location of the shared memory segment
        - If a nonzero value is given, `shmat` uses this as the attachment address for the shared memory segment
        - If `shmaddr` is 0, the system will choose an available address.
    - `shmflg` is used to specify the access permissions for the shared memory segment and to request special attachment conditions, such as a read-only segment

Attach a shared memory segment

- `shmat` is used to attach the referenced shared memory segment into the calling process's data segment.
  - Syntax:

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

  - Return value:
    - If `shmat` is successful, it returns the address of the actual attachment
    - If `shmat` fails, it returns -1.

Detaching a shared memory segment

- When a process finished with a shared memory segment, the segment should be detached using `shmdt()`.
- In order to detach a segment, it is needed to pass the address returned by `shmat()`.

- Syntax:

```
int  shmdt (void  * shmaddr);
```

  - `void *shmaddr`: a reference to an attached memory segment (the shared memory pointer).

- Return value:
  - Success: 0
  - Failure: -1

Controlling shared memory segment

- `shmctl()` call returns information about a shared memory segment and can modify it.

- Syntax:

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

- `shmid` corresponds to the shared memory identifier (i.e., the address returned by `shmat()`)

- To obtain information: pass `IPC_STAT` (see man pages for the details)
- To remove a segment: pass `IPC_RMID` as the second parameter and NULL as the third parameter. The segment is removed when the last process that has attached it, finally detaches it.

Example

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

#define SHMSZ     27
main()
{
    char c;
    int shmid;
    char *shm, *s;
    pid_t pid;

        if ((shmid = shmget(IPC_PRIVATE, SHMSZ, IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR)) < 0) {
               printf("Error in shmget");
               exit(1);
        }
        if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
                perror("Error in shmat");
                exit(1);
    }
   pid= fork();
```

Example (cont.)

```
if (pid < 0) exit(-1);
else if (pid == 0) { /* child process */
            while (shm[SHMSZ-2] != 'z')
                sleep(1);
           printf("Reading: %s\n", shm);
            for (s = shm; *s != NULL ; s++)
                putchar(*s);
            putchar('\n');

            exit (0);
            }
else { /* parent process */
        s = shm;
        for (c = 'a' ; c <= 'z' ; c++)
            *s++ = c;
        *s = NULL;
     printf("Writing: %s\n", shm);
        wait(NULL);
        shmdt(shm);
        shmctl(shmid, IPC_RMID, 0);
        exit(0);
     }
}
```

Client – Server version

SERVER

```
#define SHMSZ 1
main(int argc, char **argv)
{
        char c, tmp;
        int shmid;
        key_t key;
        char *shm;
        key = 1234; /* Shared memory segment at 1234 */
        if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
                perror("shmget");
                return 1;
        }
        if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
                perror("shmat");
                return 1;
        }
        *shm = 0;
         tmp = *shm;
        while (*shm != 'q'){
                sleep(1);
                if(tmp == *shm) continue;
                fprintf(stdout, "You pressed %c\n",*shm);
                tmp = *shm;
        }
        if(shmdt(shm) != 0)  fprintf(stderr, "Could not close memory segment.\n");
        shmctl(shmid, IPC_RMID, 0);
        return 0;
}
```

Client – Server version

CLIENT

```
#define SHMSZ     1
main()
{         int shmid;
          key_t key;
          char *shm;
          key = 1234; /* We need to get the segment named "1234", created by the server. */
          if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
                    perror("shmget");
                    return 1;
          }
          if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
                    perror("shmat");
                    return 1;
          }
          for(;;){
                    char tmp = getchar();
                    getchar(); // Eat the enter key
                    *shm = tmp;
                    if(tmp == 'q')
                        break;
          }
          if(shmdt(shm) != 0) fprintf(stderr, "Could not close memory segment.\n");
          return 0;
}
```

Shared Memory in POSIX

- Shared memory is a memory shared between two or more processes.

- Create or open the existing shared memory object
  - `Shm_open()`
- Map the shared memory region in the address space of a process
  - `mmap()`
- Delete the map
  - `munmap()`
- Remove the shared memory object
  - `Shm_unlink()`

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

Shared Memory in POSIX

- `shm_open()` creates and opens a new, or opens an existing, POSIX shared memory object. It returns a file descriptor to the shared memory.
- A POSIX shared memory object is a handle which can be used by unrelated processes to the same region of shared memory.

  - Syntax:

    ```
    int shm_open (const char * name, int oflag, mode_t mode)
    ```

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

  - Note that when you compile your code, real time library should be used

    ```
    gcc  -o output filename.c –lrt
    ```

Shared Memory in POSIX

- `shm_open()` creates and opens a new, or opens an existing, POSIX shared memory object. It returns a file descriptor to the shared memory.
- A POSIX shared memory object is a handle which can be used by unrelated processes to the same region of shared memory

  - Syntax:

    int fd = int shm_open (const char * name, int oflag, mode_t mode)

  - `Name` is of the shared memory object (`/name`)
  - `Oflag` is a bit mask : `O_RDONLY |O_RDWR | O_CREATE | …`
  - `Mode` established the permissions of the shared memory object (0666)
  - If successful, returns the file descriptor for the shared memory object (`fd`). If it fails, it returns -1.

Shared Memory in POSIX

- To set the size of shared memory object

  - Syntax:

    int ftruncate (int fd, off_t length);

  - `ftruncate` function causes the regular file referenced by `fd` to be truncated to a size of precisely length bytes. If the file previously was larger than this size, the extra data is lost. If the file previously was shorter, it is extended, and the extended part reads as zero bytes. The file pointer is not changed.

Shared Memory in POSIX

- Creating a new mapping in the virtual address space of the calling process.

  - Syntax:

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

  - Addr is the address at which the shared memory should be mapped (NULL).
  - Length is the length of the shared memory object that should be mapped.
  - Prot can have the values, PROT_EXEC, PROT_READ, PROT_WRITE and PROT_NONE.
  - Flags There are several flags, but MAP SHARED is essential for shared memory
  - Fd is the file descriptor for the shared memory received from shm_open call.
  - Offset is the point where the mapping begins in the shared memory entity (the 0 offset value can also be used)

- On success, mmap returns the pointer to the location where the shared memory object has been mapped. In case of error, -1 is returned.

Shared Memory in POSIX

- Deleting the mappings for the specified address range and causes further references to addresses within the range to generate invalid memory references. The region is also automatically unmapped when the process is terminated. On the other hand, closing the file descriptor does not unmap the region.

  - Syntax:

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

  - `addr` is the address at which the shared memory should be mapped.
  - `length` is the length of the shared memory object that should be mapped.

- On success, `munmap` returns 0. In case of error, -1 is returned.

Shared Memory in POSIX

- Removing the shared memory object

  - Syntax:

    int shm_unlinck (const char * name)

  - name is the name of the shared memory object
  - If successful, returns 0. If it fails, it returns -1.

  - Note that when you compile your code, real time library should be used

```
gcc  -o output filename.c –lrt
```

Example

- Write a C program to pass messages to each other in shared memory. Receiving program terminates when it receives the message.

```
#define DEATH(mess) { perror(mess); exit(errno); }
#define SIZE 8196
#define NAME "/my_shm"

int main (int argc, char *argv[])
{
    if (argc > 1) {
        if (!strcasecmp ("create", argv[1]))
            create_it ();
        if (!strcasecmp ("remove", argv[1]))
            remove_it ();
        if (!strcasecmp ("send", argv[1]))
            send_it ();
        if (!strcasecmp ("receive", argv[1]))
            receive_it ();
}
    printf ("Usage: %s  create | remove | receive | send \n", argv[0]);
    exit (-1);
}
```

Example (cont.)

- Write a C program to pass messages to each other in shared memory. Receiving program terminates when it receives the message.

```
void create_it (void)
{
    int shm_fd;

    if ((shm_fd = shm_open (NAME, O_RDWR | O_CREAT | O_EXCL, 0666)) == -1)
        DEATH ("shm_open");
    ftruncate (shm_fd, SIZE);

    printf ("Shared Memory Region successfully created\n");
    exit (EXIT_SUCCESS);
}

void remove_it (void)
{
    int shm_fd;
    if ((shm_fd = shm_open (NAME, O_RDWR, 0) == -1))
        DEATH ("shm_open");
    if (shm_unlink (NAME))
        DEATH ("shm_unlink");
    printf ("Shared Memory Region successfully destroyed\n");

    exit (EXIT_SUCCESS);
}
```

Example (cont.)

- Write a C program to pass messages to each other in shared memory. Receiving program terminates when it receives the message.

```
void send_it (void)
{
    int shm_fd, iflag = 1;
    void *shm_area;

    if ((shm_fd = shm_open (NAME, O_RDWR, 0)) == -1)
        DEATH ("shm_open");

    shm_area = mmap (NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);

    if (shm_area == MAP_FAILED)
        DEATH ("mmap");

    printf ("SEND: Memory attached at %lX\n", (unsigned long) shm_area);
    memcpy (shm_area, &iflag, sizeof(int));

    if (munmap (shm_area, SIZE))
        DEATH ("munmap");

    printf ("SEND has successfully completed\n");
    exit (EXIT_SUCCESS);
}
```

Example (cont.)

- Write a C program to pass messages to each other in shared memory. Receiving program terminates when it receives the message.

```
void receive_it (void)
{
    int shm_fd, iflag = 8;
    void *shm_area;
    if ((shm_fd = shm_open (NAME, O_RDWR, 0)) == -1)
        DEATH ("shm_open");
    shm_area = mmap (NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    if (shm_area == MAP_FAILED)
        DEATH ("mmap");
    printf ("RCV: Memory attached at %lX\n", (unsigned long)shm_area);
    memcpy (shm_area, &iflag, sizeof(int));
    printf ("iflag is now = %d\n", iflag);
    while (iflag == 8) {
        memcpy (&iflag, shm_area, sizeof(int));
        sleep (1);
    }
    printf ("RCV has successfully completed\n");
    printf ("iflag is now = %d\n", iflag);
    if (munmap (shm_area, SIZE))
        DEATH ("munmap");
    exit (EXIT_SUCCESS);
}
```

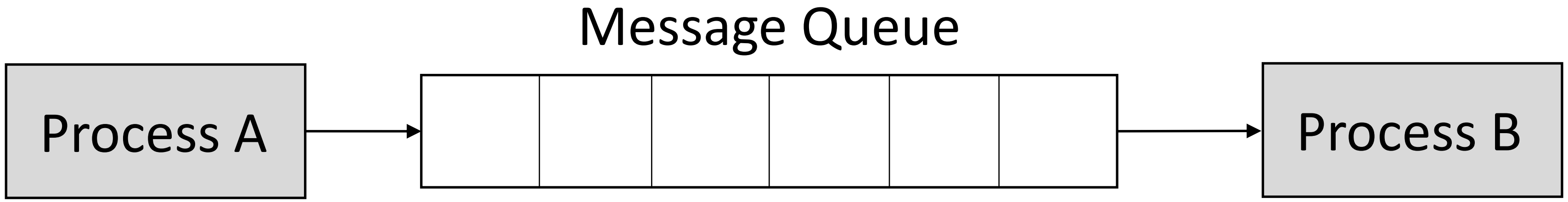
Message Passing

- Message Passing provides a mechanism for processes to communicate and to synchronize their actions without sharing the same address space.
- IPC facility provides two operations:
  - **Send** (message)
  - **Receive** (message)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a communication link between them.
  - Exchange messages via send/receive functions.

POSIX Message Queue

- A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier
- A message is composed of message type and message data.



- It can be either private or public
  - If it is **private**, it can be accessed only by its creating process or child processes of that creator.
  - If it is **public**, it can be accessed by any process that knows the queue’s key.

POSIX Message Queue

- 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())

Create a message queue

- To create or allocate a message queue
  - Syntax:

```
int msgget (key_t key, int flag);
```

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

Create a message queue

- To create or allocate a message queue
    - Syntax:

      int msgget (key_t key, int flag);

    - Key is an integer that specifies the queue key, that may be one of:
        - IPC_PRIVATE: to create a private message queue
        - Positive integer: To create a publicly accessible message queue

Create a message queue

- To create or allocate a message queue
  - Syntax:

    ```
    int msgget (key_t key, int flag);
    ```

  - `flag` is used to indicate creation conditions and access permissions. It is bitwise or of flag values. The flag values include these:
    - `IPC_CREAT`: A new queue should be created
    - `IPC_EXCL`: It causes `msgget` to fail if a queue key that is specified already exists.
    - Mode flags: This value is made of 9 bits indicating permissions granted to owner, group and world to control access to segment. Execution bits are ignored.

- This function returns a message queue identifier (`msgid`) on success and -1 in case of failure.

Send a message queue

- To send or append a message into the message queue
  - Syntax:

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

Send a message queue

- To send or append a message into the message queue
  - Syntax:

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

  - `msgid` is the message queue identifier. It is the return value of `msgget` in case of success.

Send a message queue

- To send or append a message into the message queue
  - Syntax:

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

  - msgp is the pointer to the message. It is defined in the structure of the following form:

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

Send a message queue

- To send or append a message into the message queue
  - Syntax:

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

  - `msgz` is the size of the message (the message should end with a null character).

Send a message queue

- To send or append a message into the message queue
  - Syntax:

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

  - `msgflag` indicates certain flags such as:
    - `IPC_NOWAIT` which returns immediately when no message is found in queue.
    - `MSG_NOERROR` truncates message text if it is more than `msgsz` byte.

- It return 0 in success and -1 in case of failure.

Receive a message on a queue

- Receiving a message from message queue by calling `msgrcv()`.
  - Syntax:

    ```
    int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtype, int msgflg);
    ```

  - `msgid` corresponds to the message queue identifier (the value returned by `msgget()`)
  - `msgp` points to a message received from the sender.
  - `msgsz` specifies the actual size of the message text.

Receive a message on a queue

- Receiving a message from message queue by calling `msgrcv()`.
  - Syntax:

    ```
    int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtype, int msgflg);
    ```

  - `msgtype` can be used by the receiver for message selection.
    - If `msgtype` is 0 ---> Reads the first message available in a FIFO queue.
    - If `msgtype` is +ve ---> Reads first message on queue whose type equals `msgtype`.
    - If `msgtype` is –ve ---> Reads first message on queue whose type is the lowest value less than or equal to the absolute value of `msgtype`.

Receive a message on a queue

- Receiving a message from message queue by calling `msgrcv()`.
  - Syntax:

```
int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtype, int msgflg);
```

.

  - The `msgflag` is a bit mask constructed by *OR*ing together zero or more of the possible flags (see the man pages):
    - `IPC_NOWAIT`: it returns immediately if no message of the requested type is in the queue. The system call fails with `errno` set to ENOMSG.

Receive a message on a queue

- Receiving a message from message queue by calling `msgrcv()`.
  - Syntax:

    ```
    int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtype, int msgflg);
    ```

  - Return value:
    - If successful, `msgrcv` returns the number of bytes in the text of the message.
    - If unsuccessful, it returns -1.

Remove a message on a queue

- To perform control operations on a message queue.
  - Syntax:

    int msgctl (int msqid, int *cmd, struct msqid_ds *buf);

  - msgid corresponds to the message queue identifier (the value returned by msgget())
  - cmd is the command to perform the required control operation on the message queue.
    - IPC_RMID to removed the message queue immediately. The removal is immediate and any other process still using the message queue will get an error on its next attempted operation on the queue.
  - buf is the pointer to the message queue structure named struct msgid_ds.

Sender

```
#define MAX_TEXT 512

struct my_msg_st {
    long int my_msg_type;
    char some_text[MAX_TEXT];
};

int main()
{
    int running = 1;
    struct my_msg_st some_data;
    int msgid, len;
    char buffer[MAX_TEXT];
    int mykey = getuid();

    msgid = msgget((key_t)mykey, 0666 | IPC_CREAT);

    if (msgid == -1) {
        fprintf(stderr, "msgget failed with error: %d\n", errno);
        exit(EXIT_FAILURE);
    }

some_data.my_msg_type = 1;
```

Receiver

```
#define MAX_TEXT 512

struct my_msg_st {
   long int my_msg_type;
   char some_text[MAX_TEXT];
};

int main(int argc, char * argv[])
{
   int running = 1;
   int msgid;
   struct my_msg_st some_data;
   long int msg_to_receive = 0;
   int mykey = getuid();

   msgid = msgget((key_t)mykey, 0666 | IPC_CREAT);

   if (msgid == -1) {
      fprintf(stderr, "msgget failed with
       error: %s\n", strerror(errno));
      exit(EXIT_FAILURE);
   }
```

```
while(running) {
     if (msgrcv(msgid, &some_data, MAX_TEXT,
              msg_to_receive, 0) == -1)
      {
      fprintf(stderr, "msgrcv failed with error:
      %s\n", strerror(errno));
       exit(EXIT_FAILURE);
     }
     printf("You wrote: %s", some_data.some_text);

     if (strncmp(some_data.some_text,"end", 3) == 0)
      {
        running = 0;
      }
  }

  if (msgctl(msgid, IPC_RMID, 0) == -1) {
     fprintf(stderr, "msgctl(IPC_RMID) failed\n");
     exit(EXIT_FAILURE);
  }

  exit(EXIT_SUCCESS);
}
```

Example

```
#include <sys/msg.h>
#include <sys/stat.h>
#include "tlpi_hdr.h"
#define KEY_FILE "/some-path/some-file“  /* Should exist  or be created*/
int
main(int argc, char *argv[])
{
    int msqid;
    key_t key;
    const int MQ_PERMS = S_IRUSR | S_IWUSR | S_IWGRP;
    key = ftok(KEY_FILE, 1);
    …
   msqid = msgget(key, IPC_CREAT | IPC_EXCL | MQ_PERMS)
… one side client  writes … msgsnd(msqid, &msg, len, 0)
}
```

In the sever program after fork’ing and msgget’ing uses msgrec(rec_qid, &entry, BUFSIZE, msg_to_receive, 0) to read message

System Calls for I/O

- There are 5 basic system calls that Unix provides for file I/O
  - int open(char *path, int flags [, int mode]);
  - int close(int fd);
  - int read(int fd, char *buf, int size);
  - int write(int fd, char *buf, int size);
  - off_t lseek(int fd, off_t offset, int whence);

Open ()

- To make a request to the operating system to use a file.
    - Syntax:

```
int open(char *path, int flags [ , int mode ] )
```

- The `path` argument specifies the file you would like to use
- The `flags` and 'mode' arguments specify how you would like to use it.
- The allowable `flags` as defined in `#include <fcntl.h>` are:

```
#define  O_RDONLY 0            /* open the file for reading only */
#define  O_WRONLY 1            /* open the file for writing only */
#define  O_RDWR   2      /* open the file for both reading and writing*/
#define  O_APPEND 010          /* append (writes guaranteed at the end) */
#define  O_CREAT 00400         /* open with file create (uses third open arg) */
#define  O_EXCL   02000        /* error if create and file exists */
```

- The `mode` is optional.

Open ()

- To make a request to the operating system to use a file.
    - Syntax:

```
int open(char *path, int flags [ , int mode ] )
```

    - If the operating system approves your request, it will return a file descriptor to you. This is a non-negative integer. Any future accesses to this file needs to provide this file descriptor
    - If it returns -1, then you have been denied access; check the value of global variable "errno" to determine why (or use perror() to print corresponding error message).

Read ()

- To read `size` bytes from the file specified by `fd` into the memory location pointed to by `buf`.
  - Syntax:

```
int read(int fd, char *buf, int size)
```

  - It returns how many bytes were actually read.
    - 0 : at end of the file
    - < size : fewer bytes are read to the buffer
    - == size : read the specified # of bytes
  - Things to be careful about
    - `buf` must point to valid memory not smaller than the specified size
    - `fd` should be a valid file descriptor returned from `open()` to perform read operation.

Write ()

- To writes the bytes stored in `buf` to the file specified by `fd`
  - Syntax:

    ```
    int write(int fd, char *buf, int size)
    ```

  - It returns the number of bytes actually written, which is usually `size` unless there is an error.

  - Things to be careful about
    - `buf` must be at least as long as `size.`
    - The file must be open for write operations.

Close ()

- Tells the operating system you are done with a file descriptor.
    - Syntax:

```
int close(int fd)
```

Example 1

```
#include <fcntl.h>
main(int argc, char** argv) {
  char *c;
  int fd, sz;

  c = (char *) malloc(100 * sizeof(char));

  fd = open(“foo.txt", O_RDONLY);
  if (fd < 0) { perror(”foo.txt"); exit(1); }

  sz = read(fd, c, 10);
  printf("called read(%d, c, 10), which read %d bytes.\n”, fd, sz);
  c[sz] = '\0';
  printf("Those bytes are as follows: %s\n", c);

  close(fd);
}
```

Example 1

```
#include <fcntl.h>
main()
{
  int fd, sz;

  fd = open("out3", O_RDWR | O_CREAT | O_APPEND, 0644);
  if (fd < 0) { perror("r1"); exit(1); }

  sz = write(fd, "04JEZOQ\n", strlen("04JEZOQ\n"));

  printf("called write(%d, \"04JEZOQ\\n\", %d), which returned %d\n",          fd,  strlen("04JEZOQ\n"),  sz);

  close(fd);
}
```

Perror()

- Interprets the value of `errno` as an error message, and prints it to stderr (the standard error output stream, usually the console), optionally preceding it with the custom message specified in str.
  - Syntax:

```
void perror(const char *s);
```

```
#include <stdio.h>
int main ()
{
FILE * pFile;
pFile=fopen ("unexist.ent","r");
if (pFile==NULL)
perror ("The following error occurred");
else fclose (pFile);
return 0;
}
```

If the file unexist.ent does not exist, something similar to this may be expected as program output:

```
The following error occurred: No such file or directory
```

Standard Input, Output and Error

- Every process in Unix starts out with three file descriptors predefined:
    - File descriptor 0 is standard input.
    - File descriptor 1 is standard output.
    - File descriptor 2 is standard error.

    - You can read from standard input, using read(0, ...), and write to standard output using write(1, ...) or using two library calls
        - printf
        - scanf

Iseek

- All open files have a "file pointer" associated with them to record the current position for the next file operation.
  - When a file is opened, file pointer points to the beginning of the file
  - After reading/write m bytes, the file pointer moves m bytes forward

  - Syntax:

```
off_t lseek(int fd, off_t offset, int whence)
```

  - The `whence` argument specifies how the seek is to be done
  - from the beginning of the file (`SEEK_SET`)
  - from the current value of the pointer (`SEEK_CUR`)
  - from the end of the file (`SEEK_END`)
  - The return value is the offset of the pointer after the `lseek`

Example 1

```
c = (char *) malloc(100 * sizeof(char));
 fd = open(“foo.txt", O_RDONLY);
 if (fd < 0) { perror(”foo.txt"); exit(1); }

 sz = read(fd, c, 10);
 printf("We have opened foo.txt, and called read(%d, c, 10).\n”, fd);
 c[sz] = ‘\0’;
 printf(“Those bytes are as follows: %s\n”, c);

 i = lseek(fd, 0, SEEK_CUR);
 printf(“lseek(%d, 0, SEEK_CUR) returns the current offset = %d\n\n”, fd, i);

 printf(“now, we seek to the beginning of the file and call read(%d, c, 10)\n”, fd);
 lseek(fd, 0, SEEK_SET);
 sz = read(fd, c, 10);
 c[sz] = ‘\0’;
 printf("The read returns the following bytes: %s\n", c);
```

Example 2

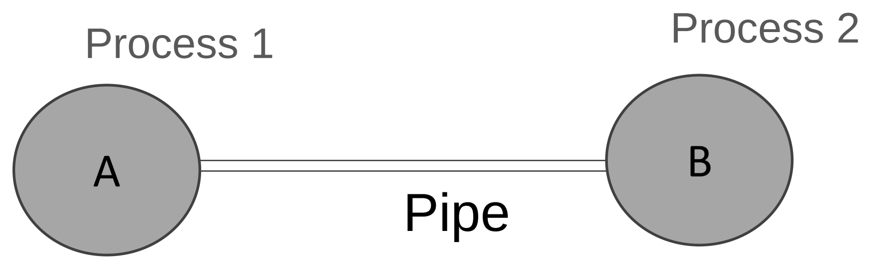
```
int main()
{
        int file=0;
        if((file=open("testfile.txt",O_RDONLY)) < -1)
               return 1;
         char buffer[19];
        if(read(file,buffer,19) != 19)  return 1;
        printf("%s\n",buffer);

        if(lseek(file,10,SEEK_SET) < 0) return 1;
        if(read(file,buffer,19) != 19)  return 1;
        printf("%s\n",buffer);
        return 0;
}
```

The output of the preceding code is:

```
$ cat testfile.txt
This is a test file that will be used
to demonstrate the use of lseek.
$ ./testing
This is a test file
test file that will
```

Unix Pipes

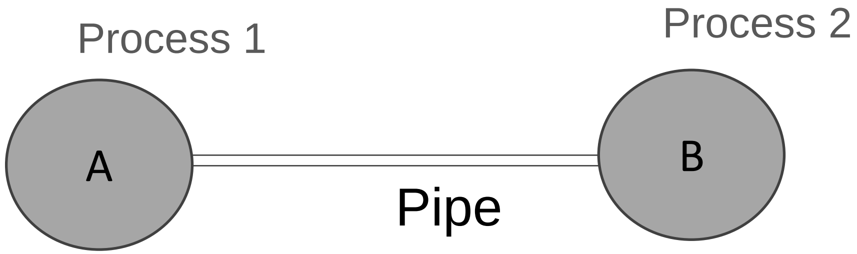
- Pipe sets up communication channel between two processes (related or unrelated):
  - Pipes are uni-directional.
    - They can only transfer data in one direction.
    - Both the writer and reader process of a pipeline execute concurrently.
    - A pipe automatically buffer the output of the writer and suspends the writer if the pipe gets too full.
    - Similarly, if a pipe is empty, the reader is suspended until some more output becomes available.



Two processes connected by a pipe

Unix Pipes

- There are two kinds of pipes.
  - **Unnamed piped** for communication between a parent and its child, with one process writing and the other process reading.
  - **Named pipes** for communication between unrelated processes. Any process can communicate with another one using named pipes.



Two processes connected by a pipe

Unnamed Pipe: pipe()

- An unnamed pipe is a unidirectional communications link that automatically buffers its data.
  - It is created using the `pipe()` system call.
  - Each end of a pipe has an associated file descriptor:
    - The writer uses the “*write*” end of the pipe (using “`write()`”).
    - The reader uses the “read” end of the pipe (using “`read()`”).

Unnamed Pipe: pipe()

- An unnamed pipe is a unidirectional communications link that automatically buffers its data.
  - Syntax:

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

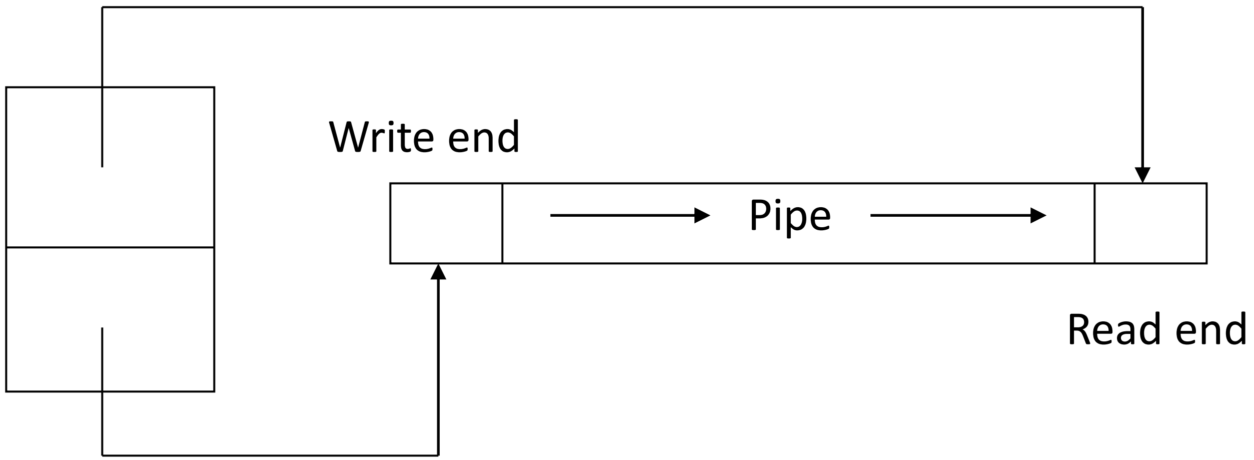
  - `pipe()` creates an unnamed pipe and returns two file descriptors:
    - The descriptor associated with the read end of pipe is stored in fd [0].
    - The descriptor associated with the write end of the pipe is stored in fd [1].
    - Return value: If the kernel cannot allocate enough space for a new pipe, `pipe()` returns a value of -1; otherwise, it returns a value of 0.

Unnamed Pipe: pipe()

- Assume that the following code was executed:

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

  - The following data structure would be created:



  - The maximum size of the pipe varies with different versions of UNIX, but is approximately 5K.

Read() & Write() and Close() system calls

- The read() system call does all inputs and the write() system call does all outputs.
- Read and Write are atomic operations.

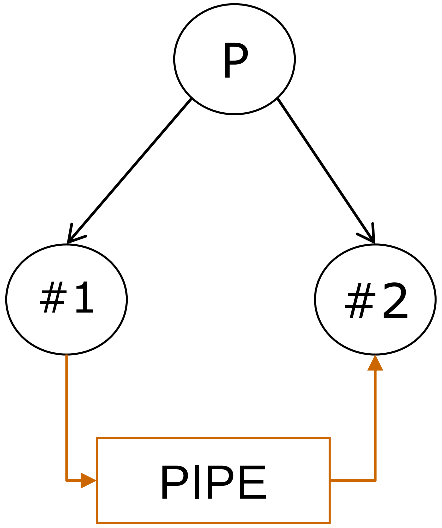
- Read:
  - If a process reads from a pipe whose `write` end has been closed, the `read()` call returns a value of zero, indicating the end of input.
  - If a process reads from an empty pipe whose `write` end is still open, it sleeps until some input becomes available.

  - Write:
    - If a process writes to a pipe whose `read` end has been closed, the `write` fails and a SIGPIPE signal is sent to the writer.

- When a process has finished with a pipe’s file descriptor, it should close it using `close()`.

Scheme

- The typical sequence of events for a communication is as follows:
  - The parent process creates an unnamed pipe using `pipe()`.
  - The parent process `forks`.
  - The processes communicate by using `write()` and `read()` calls.
  - Each process closes its active pipe descriptor when it’s finished with it.



Pipe between 2 child processes

Scheme

- The typical sequence of events for a communication is as follows:

“pipeFD[2]”:

- The descriptor associated with the “read” end of the pipe is stored in pipeFD [0].
- The descriptor associated with the “write” end of the pipe is sored at pipeFD [1].

“pipe()” creates an un named pipe and returns two file descriptions: pipeFD [0], pipeFD[1].

int write (int fd, char * buf, int size)

int read (int fd, char * buf, int size)

```
# include <sys/wait.h>     /*wait*/
# include <stdio.h>
# include <stdlib.h>      /*exit functions*/
# include <unistd.h>     /*read, write, pipe*/

Int main (){
  int pipeFDs [2];        /*two file descriptors*/

   pipe(pipeFDs);
   pid =  fork ();
   If ( pid == 0){   // in the child
       write (pipeFD [1], yourMessage, strlen (yourMessage));
       close (pipeFD [1]);}
  else { // in the father
       byteRead = read (pipeFD [0], msg, 100);
       close (pipeFD [0]);}
}
```

Example

```
#include <stdio.h>
#define  READ   0   /* The index of the “read” end of the pipe */
#define  WRITE  1    /* The index of the “write” end of the pipe */
char*  phrase ="This is a message!!!";
main()
{
int fd[2], bytesRead;
char message[100];   /* Parent process’ message buffer */

pipe(fd);  /* Create  an unnamed pipe */
if ( fork() == 0 )  /* Child, write */
    {
     write(fd[WRITE], phrase, strlen(phrase)+1); /* Send */
     close(fd[WRITE]); /* Close used end */
     }
   else      /* Parent, reader */
    {
     bytesRead = read( fd[READ], message, 100 ); /* Receive */
     printf(“Read %d bytes: %s \n”, bytesRead, message );
     close(fd[READ]);  /* Close used end */
    }
  }
 $ talk        ---> run the program.
  Read 21 bytes: This is a message!!!
```

Dup() system call

- The `dup()` system call duplicates an open file descriptor and returns the new file descriptor.
  - Syntax:

```
int dup (oldfd);
int oldfd;
```

- The dup()system call creates a copy of the file descriptor `oldfd`, using the lowest-numbered unused file descriptor for the new descriptor.
- After a successful return, the old and new file descriptors may be used interchangeably.
- The new file descriptor has the following properties in common with the original file descriptor:
  - Refers to the same open file or pipe.
  - has the same access mode, whether read, write, or read and write
  - has the same file pointer that is, both file descriptors share one file pointer (i.e., if the file offset is modified by using lssek() on one of the file descriptors, the offset is also changed for the other).
- `dup()` is exploited to accomplish I/O redirection.

Example

```
int main()
{
        int fd[2];
        pid_t pid1, pid2;
        char message[100];
        int bytes_read;

        pipe(fd);

        /* Create the first child */
        pid1 = fork();
        if (pid1<0) {
                perror("First fork() failed!");
                return -1;
        }
if (pid1==0) {
                /* Set the process output to the input of the pipe. */
                close(1);
                dup(fd[1]);
                close(fd[0]);
                write(fd[1], "Ciao", sizeof("ciao"));
                exit(0);
        }
```

Example

```
/* Create the second child */
        pid2 = fork();
        if (pid2<0) {
                perror("Second fork() failed!");
                return -1;
        }
        if (pid2==0) {
                /* Set the process input to the output of the pipe. */
                close(0);
                dup(fd[0]);
                close(fd[1]);
                bytes_read = read(fd[0], message, 100);
                printf("%d read %s\n", bytes_read, message);
                exit(0);
        }
        close(fd[0]);
        close(fd[1]);
        /* Wait for the children to finish, then exit. */
        waitpid(pid1,NULL,0);
        waitpid(pid2,NULL,0);
        return 0;
}
```

Named Pipes or FIFOs

- Named pipes, often referred to as *FIFOs (first in, first out),* are less restricted than unnamed pipes and offer the following advantages:
  - They have a name that exists in the file system.
  - They may be used by unrelated processes.
  - They exist until explicitly deleted.
  - They have a larger buffer capacity, typically about 40K.

- Like an unnamed pipe, a named pipe is intended only for use as a unidirectional link.

Named Pipes or FIFOs

- `mkfifo()` allows you to create a FIFO special file (a named pipe).
  - Syntax:

```
int mkfifo (const char *path, mode_t mode);
```

  - `Path` corresponds to the name of the pipe.
  - `Mode` corresponds to the permission mode flags.

  - A named pipe is first opened using `open()`.
  - `Write()` adds data at the start of the FIFO queue.
  - `Read()` removes data from the end of the FIFO queue.

Close and Remove a Pipe

- `mkfifo()` allows you to create a FIFO special file (a named pipe).
  - Syntax:

    ```
    int mkfifo (const char *path, mode_t mode);
    ```

  - When a process has finished using a named pipe, it should close it using `close()`.
  - Writer processes should open a named pipe for writing only
  - Reader processes should open a pipe for reading only.
- when a named pipe is no longer needed, it should be removed from the file system using `unlink()`.
  - Syntax:

    ```
    int unlink (const char *path);
    ```

Example

- A single reader process that creates a named pipe called *aPipe* is executed.
- It then reads and displays NULL-terminated lines from the pipe until the pipe is closed by all of the writing processes.
- One or more writer processes are executed, each of which opens the named pipe called “aPipe” and sends three messages to it.
- If the pipe does not exist when a writer tries to open it, the writer retries every second until it succeeds.
- When all the writer’s messages are sent, the writer closes the pipe and exits.

Reader Program

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>             /* For SIFIFO */
#include <fcntl.h>

main()
{
 int fd;
 char str[100];


 mkfifo(“aPipe”, 0660);  /* Create named pipe */

 fd = open(“aPipe”, O_RDONLY);  /* Open it for reading */
 while(readLine(fd, str) )   /* Display received messages */
         printf(“%s\n”, str);
 close(fd);   /* Close pipe */

 unlink(“aPipe”);  /* Remove named pipe */

}
```

```
int readLine( int fd, char* str )
/* Read s single NULL-terminated line into str from fd */
/* Return 0 when the end of input is reached and 1 otherwise */
{
  int n;

  do /* Read characters until NULL or end of input */
     {
      n = read( fd, str, 1);  /* Read one character */
     }
  while ( n>0 && *str++ != NULL );

  return ( n > 0 );  /* Return false if end of input */
 }
```

Writer Program

```
#include <stdio.h>
#include <fcntl.h>
main()
  {
   int fd, messageLen, i;
   char message[100];

   /* Prepare message */
   sprintf( message, “Hello from PID %d”, getpid() );
   messageLen = strlen( message ) +1;
   do /* Keep trying to open the file until successful */
   {
     fd = open( “aPipe”, O_WRONLY );    /*Open named pipe for writing */
     if ( fd == -1 ) sleep(1);     /* Try again in 1 second */
   } while ( fd == -1 );
         for ( i=1; i<=3; i++)     /* Send three messages */
       {
       write( fd, message, messageLen );     /* Write message down pipe */
       sleep(3);     /* Pause a while */
       }
        close(fd);     /* Close pipe descriptor */
 }
```

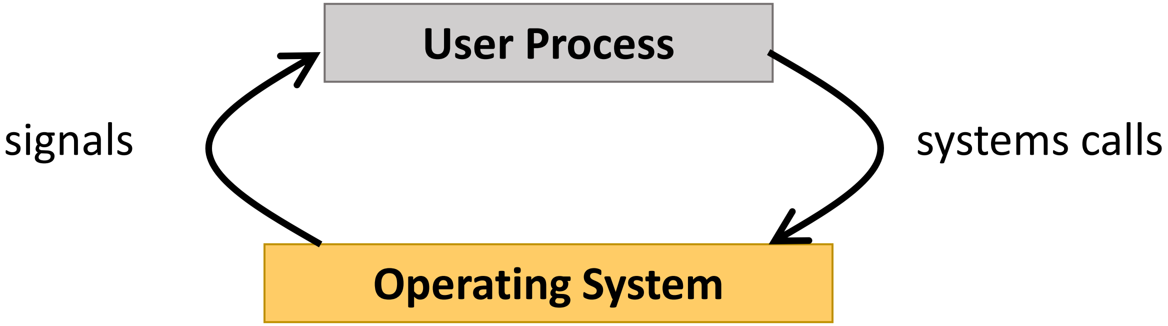
Sample Output

```
$ reader & writer & writer &    ---> start 1 reader, 2 writers.
  [1] 4698                        ---> reader process.
  [2] 4699                        ---> first writer process.
  [3] 4700                        ---> second writer process.
  Hello from PID 4699
  Hello from PID 4700
  Hello from PID 4699
  Hello from PID 4700
  Hello from PID 4699
  Hello from PID 4700
  [2] Done  writer      ---> first writer exists.
  [3] Done  writer      ---> second writer exists.
  [4] Done  reader      ---> reader exists.
  $ _
```

Unix Signals

- A signal is an asynchronous event which is delivered to a process.
- A UNIX signal corresponds to an event
  - It is raised by one process (or hardware) to call another process's attention to an event
  - It can be caught (or ignored) by the subject process

- The OS can communicate to an application process through Signals.



Standard Signals

- Typing certain key combinations at the terminal of a running process causes the system to send it certain signals:
  - `CTRL-C` sends an INT signal (SIGINT); by default, this causes the process to terminate.
  - `CTRL-Z` sends a TSTP signal (SIGTSTP); by default, this causes the process to suspend execution.
  - `CTRL-\` sends a QUIT signal (SIGQUIT); by default, this causes the process to terminate and store the content of the memory (core dump).

Standard Signals

- Typing certain key combinations at the terminal of a running process causes the system to send it certain signals:
  - `CTRL-C` sends an INT signal (SIGINT); by default, this causes the process to terminate.
  - `CTRL-Z` sends a TSTP signal (SIGTSTP); by default, this causes the process to suspend execution.
  - `CTRL-\` sends a QUIT signal (SIGQUIT); by default, this causes the process to terminate and store the content of the memory (core dump).

- `kill -sig pid` command
  - OS sends a sig signal to the process whose id is pid.

Signal

- A signal is a software notification to a process of an event:
  - Signal is generated by a particular event.
  - Signal is delivered to a process.
  - Signal is handled.

Signal

- A signal is a software notification to a process of an event:
  - Signal is generated by a particular event.
  - Signal is delivered to a process.
  - Signal is handled.

- There are three ways in which a process can respond to a signal:
  - Ignoring the signal
    - `signal (SIG#, SIG_IGN)`
  - Executing the default action associated with the signal
    - `signal (SIG#, SIG_DFL)`
  - Catching the signal by invoking a corresponding signal-handler function
    - `signal(SIG#, myHandler)`
- OS provides a facility for writing your own event handlers in the style of interrupt handlers.

Signal Handler

- A *signal handler* is used to process signals
- Corresponding to each signal there is a signal handler.
- Called when a process receives a signal.
- The function is called “asynchronously”.
- When the signal handler returns the process continues, as if it was never interrupted.

- Note: Signal are different from interrupts as:
  - Interrupts are sent to OS by H/W
  - Signals are sent to a process by the OS, or by other processes
  - Note that signals have nothing to do with software interrupts, which are still sent by the hardware (the CPU itself, in this case).

Send a Signal

- To Raise a signal
  - Syntax:

    kill (pid, signal);

  - Pid is an input parameter, id of the process that receives the signal.
  - Signal is a signal number.
  - returns 0 to indicate success, error code otherwise.

  - Example
    - `pid_t iPid = getpid();` /* Process gets its id.*/
    - `kill(iPid, SIGINT);` /* Process sends itself a SIGINT signal (commits suicide) */

Handling Signals

- Each signal type has a default action
    - For most signal types, default action is *terminate*.
- A program can install a signal handler to change action of (almost) any signal type.

- Special cases: A program cannot install a signal handler for signals of type:
    - 9/SIGKILL
        - Default action is "terminate"
        - The catchable termination signal is 15/SIGTERM
    - 19/SIGSTOP
        - Default action is "stop until next 18/SIGCONT"
        - The catchable suspension signal is 20/SIGTSTP

Handling Signals

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

Void handle-sigint (){
    printf (“Caught signals\n”);
    exit (0);
}
Int main (){
pid_t pid;
pid = fork ();
If (pid ==0){
       signal (SIGINT, handle-sigint);
        while (1);
}
   else {
        printf (“I am killing my child\n”);
        kill (pid, SIGINT)
    }
}
```

Example 1

```
#include <stdio.h>
#include <signal.h>
 void sigproc()
{
    signal(SIGINT, sigproc); /* NOTE some versions of UNIX will reset signal to default after each call. So for portability reset signal each time */
    printf(“you have pressed ctrl-c - disabled \n”);
}
 void quitproc()
{
        printf(“ctrl-\\ pressed to quit\n”);   /* this is “ctrl” & “\” */
        exit(0); /* normal exit status */
}

main()
{
        signal(SIGINT, sigproc);     /* ctrl-c : DEFAULT ACTION: term */
        signal(SIGQUIT, quitproc);  /* ctrl-\ : DEFAULT ACTION: term */
        printf(“ctrl-c disabled use ctrl-\\ to quit\n”);

        for(;;);
}
```

Example 2

```
int main()
{   pid_t  child_pid, p;

   if((child_pid = fork()) == 0 )
   {
     printf("Child(pid=%d): I will send SIGINT to my parent (Pid=%d)\n",
                                        getpid(), getppid());

     kill(getppid(), SIGINT);  /* send interrupt signal to parent */

   }
   else /* parent */
   {
      printf("Parent(pid=%d): I will loop forever until I die \n", getpid());

      for(;;);
   }
   return 0;
}
```

Example 3

```
int main()
{ int pid;
    printf("alarm application starting\n");
    if((pid = fork()) == 0) {
        sleep(5);
        kill(getppid(), SIGALRM);
        exit(0);
    }
    printf("waiting for alarm to go off\n");
    signal(SIGALRM, ding);
    pause();
    printf("done\n");
    exit(0);
}
```

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


void ding (int sig)
{
printf("alarm has gone off\n");
}
```

pause system call causes program to suspend until a signal is received

Note

- Problem with signals:
  - Can cause race conditions
- Example:
  - If signal occurs before the call to pause, then your program may wait indefinitely.

```
int main()
{ int pid;
    printf("alarm application starting\n");
    if((pid = fork()) == 0) {
    kill(getppid(), SIGALRM);
    exit(0);
  }
  printf("waiting for alarm to go off\n");
  signal(SIGALRM, ding);
  sleep(5);
  pause();
  printf("done\n");
  exit(0);
}
```

Example 4

```
void sighup()
{
        signal(SIGHUP,sighup); /* reset signal */
        printf("CHILD: I received a SIGHUP\n");
}

void sigint()
{
        signal(SIGINT,sigint); /* reset signal */
        printf("CHILD: I received a SIGINT\n");
}

void sigquit()
{
        printf("My DADDY has Killed me!!!\n");
        exit(0);
}
```

```
#include <stdio.h>
#include <signal.h>
void sighup();
void sigint();
void sigquit();
main()
{
        int pid;
        /* get child process */
        if ((pid=fork()) < 0)
        { perror("fork"); exit(1); }
        if (pid == 0) {       /* child */
                signal(SIGHUP, sighup);
                signal(SIGINT, sigint);
        signal(SIGQUIT, sigquit);
        for(;;);
    } else {   /* parent */
                printf("\nPARENT: sending SIGHUP\n\n");
        kill(pid,SIGHUP);
        sleep(3);
                printf("\nPARENT: sending SIGINT\n\n");
        kill(pid,SIGINT);
        sleep(3);
        printf("\nPARENT: sending SIGQUIT\n\n");
        kill(pid,SIGQUIT);
        sleep(3);
    }}
```

Alarm ()

- It sets an alarm timer that will ring after a specified number of seconds.
- It provides a mechanism for a process to interrupt itself at some future time. It does it by setting a timer. When the timer expires, the process receives a `SIGALRM` signal.
  - Syntax:

```
long  alarm (long seconds);
# include <unistd.h>
```

  - Example
    - `alarm(10);`   /* The process will get SIGALRM signal after 10 seconds (real-time).*/

Command Line Generates Signals

- You can send a signal to a process from the command line using kill
  - `kill -l`
    - will list the signals the system understands
  - `kill [-signal] pid`
    - will send a signal to a process.
      - The optional argument may be a name or a number.
      - The default is SIGTERM.
- To unconditionally kill a process, use:
  - `kill -9 pid`
  - which is
  - `kill -SIGKILL pid`

Examples of POSIX Required Signals

Signal	Description	default action
SIGABRT	process abort	implementation dependent
SIGALRM	alarm clock	abnormal termination
SIGBUS	access undefined part of memory object	implementation dependent
SIGCHLD	child terminated, stopped or continued	ignore
SIGILL	invalid hardware instruction	implementation dependent
SIGINT	interactive attention signal (usually ctrl-C)	abnormal termination
SIGKILL	terminated (cannot be caught or ignored)	abnormal termination

Examples of POSIX Required Signals

Signal	Description	default action
SIGSEGV	Invalid memory reference	implementation dependent
SIGSTOP	Execution stopped	stop
SIGTERM	termination	Abnormal termination
SIGTSTP	Terminal stop	stop
SIGTTIN	Background process attempting read	stop
SIGTTOU	Background process attempting write	stop
SIGURG	High bandwidth data available on socket	ignore
SIGUSR1	User-defined signal 1	abnormal termination