Interprocess Communication: IPC

IPC is a mechanism where OS allows comunication between processes.
Note that concurrent processes running in the system may be

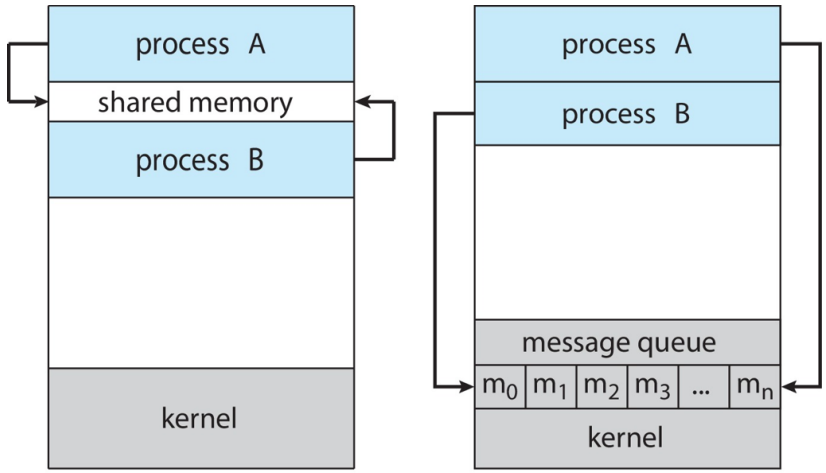
- Independent – not sharing data with any other process
- Cooperating – with other processes (by sharing data) and possibly can affect or be affected by the action of other processes

In IPC there are subtle points to ponder

- How to establish communication?
  - *Shared memory*
  - *Message Passing*
- Avoid race – say, two processes trying to book the last seat of a flight for two different persons – name coming from one process and passport number from the other
- Proper sequencing when dependencies exist – One process is reading a file and other is printing the same – printer cannot start without getting the first record to be printed.

IPC: Basic Models – Shared Memory and Message Passing

- Shared Memeory
  - In this case a shared region in memory is established for the cooperating processes
  - Faster as it does not require kernel intervention once the shared region is established
- Message passing
  - Here the messages are exchanged between the cooperating processes
  - Suitable for sharing a small amout of data
  - Slow as it requires kernel intervention



IPC: Shared Memory

- Shared Memeory region is created by a process in its address space
- Other processes must attach the region in their address space to communicate
- OS, in general, restricts a process to access the address space of other processes
- So, for data exchange at least two processes must remove this restriction and establish a shared region
- Now the data exchange (Read/write the shared location) can take place between processes
- Kernel has no control the form of data being exchanged and the locations iused
- Cooperating processes must ensure not to write to the same location (similar to Bus contention) simultaneously

Producer — Consumer relationship

The concept of cooperating processes and the eventual data exchange would be best comprehended through the common metaphor – *The producer – consumer relationship* that we used to see all around us.

- In IPC; Shared memory may be used to solve the producer – consumer problem; where
  - the producer process and consumer process run concurrently; they
  - maintain a buffer (bounded or unbounded). While producer produces (writes) an item to this buffer; the consumer consumes (reads) another item from this buffer
- The actions must be synchronised so that consumer does not try to consume an item yet to be produced by the producer
- **Bounded buffer: FIXED SIZE** : Here consumer waits if the buffer is empty and the producer waits if the buffer is full.
- **Unbounded buffer: NO PRACTICAL SIZE LIMIT** : Here the producer can always pruduce items and need not wait while the consumer may wait if the buffer is empty.

Bounded buffer : Programming example

- The variables reside in the shared region is shown on the right
- The shared buffer is implemented as a circular array with two pointers; **in** and **out** pointing to the buffer
  - *in* points to the next free position
  - *out* points to the 1st full position
- *in* == *out* implies empty buffer; ((in + 1) % BUFFER SIZE) == out indicates the buffer is full

```
#define BUFFER_SIZE 10
typedef struct {
...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Bounded buffer : Programming example contd.

- The producer has a local variable next_produced to store the new item produced
- This scheme allows at most BUFFER SIZE - 1 items in the buffer at the same time

```
item next_produced;
while (true) {
/* produce an item in next_produced */
while (((in + 1) % BUFFER_SIZE) == out)
; /* do nothing */
buffer[in] = next_produced;
in = (in + 1) % BUFFER_SIZE;
}
```

Bounded buffer : Programming example contd.

- The consumer process has a local variable next_consumed in which the item to be consumed is stored

```
item next_consumed;
while (true) {
while (in == out)
; /* do nothing */
next_consumed = buffer[out];
out = (out + 1) % BUFFER_SIZE;
/* consume the item in next consumed */
}
```

The program segments, however, has no clue for a simultaneous access attempts by the producer and consumer. Such, synchronisation will be discussed elsewhere.

IPC : MESSAGE PASSING

Message passing mechanism allows processes to communicate without using a shared address space.

- It is useful in environment where communicating processes are running in different computing systems connected through a network – e.g., chat participants communicates by exchanging messages.
- Good for exchange of small amount of data; slow as OS intervention is required unlike shared memory
- Message passing provides at least two primitive operations
  - *send(message)*
  - *receive(message)*
- Message may be of fixed size
  - System level implementation is easy; however
  - the restriction makes programming part difficult
- Message may be of variable size
  - System level implementation is complex; and
  - it makes programming part easier

IPC : MESSAGE PASSING contd.

For message passing we need a logical communication link that can be established in several ways to carry out the send() and receive() operations.

- (i) *Direct or indirect communication*
- (ii) *Synchronous or asynchronous communication*
- (iii) *Automatic or explicit buffering*

Direct or indirect communication

- Must explicitly mention the recipient or sender; i.e., the primitives will be
  send(P, message) – and receive(Q, message) for a communication between P and Q.
- A link is established with the following properties
  - Link is estabhished between every pair of processes (say, P and Q) and the processes should know each others name
  - The link is associated with exactly two processes
  - Between a pair there exists only one link

IPC : MESSAGE PASSING contd.

We can have a sort of asymmetry in addressing (naming) in which

- The sender sends a message to process P by send(P, message); however
- The receiver would use receive(id, message) – to receive a message from any process whose *id* – the id of the process from which the message is received is returned in *id*

The disadvantage of the symmetric and asymmetric schemes is that it leads to hardcoding of the identifiers. This is avoided using *indirect communication* where messages are sent and received through *mailboxes* or *ports*.

- each mailbox has a unique *id*
- communication between pocesses can be done through a number of mailboxes, however, two prcess can communicate if there exits a shared mailbox.
- the primitives sharing a mailbox *A* take the form send(A, message) and receive(A, message)

IPC : MESSAGE PASSING contd.

In the mailbox (port) scheme the communication link properties are

- Link is established between a pair of processes if both members have a shared mailbox
- A link may be associated with more than two processes
- Between each pair of communicating processes, a number of different links may exist where each link refers to one mailbox

Suppose P1, P2 and P3 are sharing the mailbox A and P1 has sent a message by send(A, message) – who is going to receive it; P2 or P3? We have a number of methods to choose from.

- Allow a link to be associated with two processes at most
- Allow at most one process at a time to execute receive()
- Allow the system to determine (using some alogorithm like RR) who gets it (P2 or P3; but not the both at a time) – the system may return the receiver-id to the sender

IPC : MESSAGE PASSING contd.

A mailbox may be owned by a process (i.e., it is a part of the address space of that process); in that case

- We can identify the owner who can only receive the message; and the user who can only send the message
- The mailbox vanishses when the owner terminates and Notification should be sent to the subsequent users that the mailbox does not exist

If the mailbox is owned by the OS then it has got an existence of its own and not attached to a particular process
The OS must provide mechanisms allowing a process

- to create a new mailbox (the process is the owner and only receives)
- send and receive messages through the mailbox (ownership and receiving privilege may be passed to other processes); and
- delete a mailbox

IPC : MESSAGE PASSING: SYNCHRONISATION

Message passing through send() and receive() may either be *blocking (synchronous)* or *non-blocking (asynchronous)*

- BLOCKING SEND: The sending process is blocked until the message is received by the receiving process or by the mailbox.
- NONBLOCKING SEND: The sending process sends the message and resumes operation.
- BLOCKING RECEIVE: The receiver blocks until a message is available
- NONBLOCKING RECEIVE: The receiver retrieves either a valid message or a NULL

IPC : MESSAGE PASSING: producer consumer

Different combinations of the send() and receive() are possible. When both the send() and receive() are blocking type the producer consumer problem becomes trivial.
Here, the producer merely invokes the blocking send() call and waits until the message is delivered to either the receiver or the mailbox. The consumer invokes receive() that blocks until the message is available.

```
message next_produced;
while(true)  {
        // produce an item in next_produced
     send(next_produced);
    }
-----------------------------------------
message next_consumed;
    while(true)  {
       // consume an item in next_consumed
            receive(next_consumed);
      }
```

IPC : MESSAGE PASSING: BUFFERING

Irrespective of the communication being direct or indirect the message exchanged by the communicating processes reside in a temporary queue that may be implemented in three ways;

- ZERO CAPACITY (No buffering): The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- BOUNDED CAPACITY (Auto-buffering): The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.
- UNBOUNDED CAPACITY (Auto-buffering): The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

IPC : Examples

PIPES: One of the earliest mechanisms for IPC in UNIX. It provides a simpler way of communication bdtween processes; the issues are

- Unidirectional or bidirectinal
- If bi-directional then – half-duplex or full-duplex
- Relationshhip between the processes (parent – child etc.)
- Communication over a network?

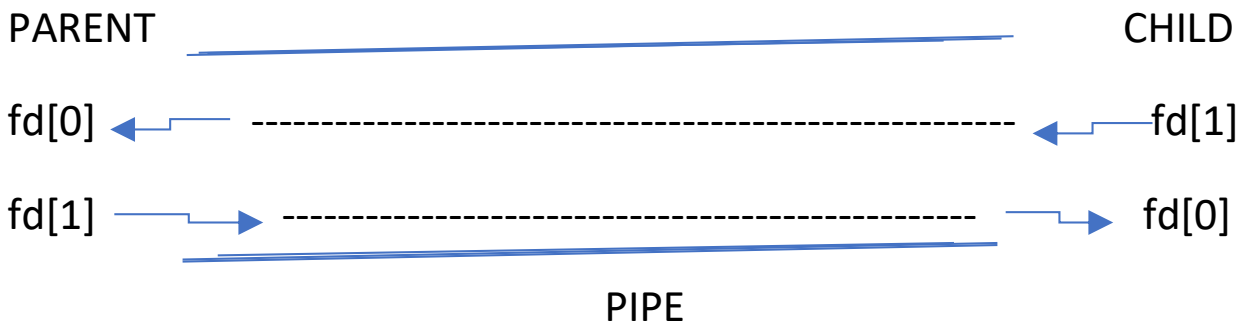
Pipes are: *ORDINARY* or *NAMED*.
ORDINARY PIPE

- Alllows two processes to communicate in producer consumer fashion; the prducer writes at one end (like a reservoir being filled through an inlet) and the consumer gets it from another end (like you drink from an outlet).
- One way communication
- For a two way communication we need two separate pipes – one for each direction

IPC : PIPES

In UNIX the system call pipe(int fd[]) creates an ordinary pipe accessed through file-descriptors fd[].

- fd[0] is the read end of the pipe
- fd[i] is the write end of the pipe
- And the OS treats a pipe as a special type of file. So, read() and write() system calls may be used.
- Pipe cannot be accessed from outside a process who has created it
- A parent process creates a pipe to communicate with the child, created through fork().
- As the child inherits open files of the parent and pipe is a special file so the child inherits the pipe



IPC : PIPES: Program Example

```
#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;
   }
```

IPC : PIPES: Program Example contd.

```
    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;

}
```

Another example – using fork()

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_ END 1
int main(void)
{
char write_msg[BUFFER_ SIZE] = "Greetings";
char read_msg[BUFFER_SIZE];
int fd[2];
pid_t pid;
```

IPC : PIPES: Program Example contd.

```
/* create the pipe */
if (pipe(fd) == -1) {
fprintf(stderr,"Pipe failed");
return 1;
}
/* fork a child process */
pid = fork();
if (pid < 0) { /* error occurred */
fprintf(stderr, "Fork Failed");
return 1;
}
if (pid > 0) { /* parent process */
/* close the unused end of the pipe */
close(fd[READ_END]);
```

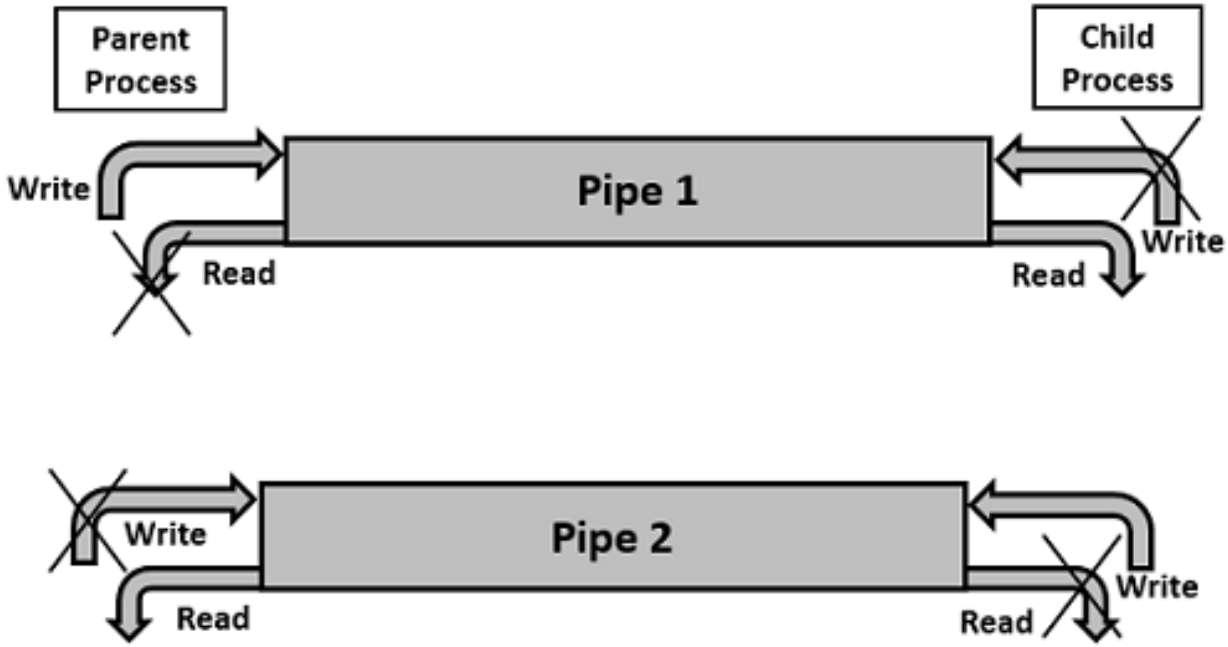
IPC : PIPES: Program Example contd.

```
/* write to the pipe */
write(fd[WRITE_END], write_msg, strlen(write_msg)+1);
/* close the write end of the pipe */
close(fd[WRITE_END]);
}
else { /* child process */
/* close the unused end of the pipe */
close(fd[WRITE_END]);
/* read from the pipe */
read(fd[READ_END], read_msg, BUFFER_SIZE);
printf("read %s", read_msg);
/* close the read end of the pipe */
close(fd[READ_END]);
}
return 0;
}
```

PIPES: Two Way Communication

For a both way communication we need to establish two-pipes and then we perform communication

- Create two pipes. And Create a child process.
- Close unwanted ends in the parent process, read end of pipe1 (write end of pipe 1 for child) and write end of pipe2 (read end of pipe 2 for the child) as you need one way in each direction



PIPES: Two Way Communication: Example

```
#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;
   }
```

PIPES: Two Way Communication: Example

```
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("Parent:Writing>pipe1:Message is %s\n",pipe1writemessage);
  write(pipefds1[1], pipe1writemessage, sizeof(pipe1writemessage));
  read(pipefds2[0], readmessage, sizeof(readmessage));
  printf("Parent:Reading<pipe2: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("Child: Reading<pipe1: Message is %s\n", readmessage);
   printf("Child:Writing>pipe2:Message is %s\n", pipe2writemessage)
   write(pipefds2[1], pipe2writemessage, sizeof(pipe2writemessage))
   }
   return 0;
}
```

Named PIPES: Communication between unrelated processes

Ordinary PIPE cannot be used to communicate other than related processes; NAMED PIPE (also known a FIFO) can and it is bidirectional too

- It is an extension to "pipe" concept in Unix. A traditional pipe is "unnamed" and lasts only as long as the process.
- A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.
- Once a FIFO is established it can be used by any process
- However, it has to be opened at both ends simultaneously before you can proceed to do any input or output operations on it.

Named PIPES: Example program

Program 1 writes first, then reads. Program 2 reads first, then writes – until terminated.

```
// C program to implement one side of FIFO
// This side writes first, then reads
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
int main()
{  int fd;
    char *myfifo = "myfifo";   // FIFO file path
    mkfifo(myfifo, 0666);  // Creating the named file(FIFO)
    char arr1[80], arr2[80];
```

Named PIPES: Example program contd.

```
 while (1) {
   fd = open(myfifo, O_WRONLY); // Open FIFO for write only
   fgets(arr2, 80, stdin); //get input
   write(fd, arr2, strlen(arr2)+1); // write to FIFO & close
   close(fd);
        // Open FIFO for Read only
    fd = open(myfifo, O_RDONLY);
    read(fd, arr1, sizeof(arr1)); // read fifo
    printf("User2: %s\n", arr1);
    close(fd);
    }
    return 0;
}
```

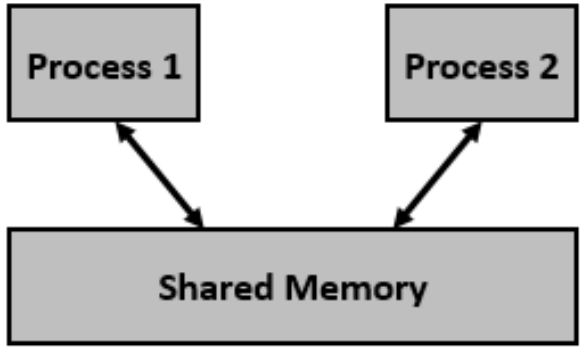
Named PIPES: Example program contd.

2nd program that reads first [include the same header files used in 1st]

```
int main() {
    int fd1;  char *myfifo = "myfifo";
    mkfifo(myfifo, 0666);
    char str1[80], str2[80];
    while (1)  {  // First open in read only and read
        fd1 = open(myfifo,O_RDONLY);
        read(fd1, str1, 80);
        printf("User1: %s\n", str1);
        close(fd1);
        fd1 = open(myfifo,O_WRONLY);
        fgets(str2, 80, stdin);
        write(fd1, str2, strlen(str2)+1);  close(fd1);
    }
    return 0;
}
```

SHARED MEMORY

Unrelated process communication may be done by SHARED memory.



The basic steps are (using system V calls)

- 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())
- Control operations on the shared memory segment (shmctl())
- Detach the process from the already attached shared memory segment (shmdt())

We will delve the system calls in detail in an example program.

SHARED MEMORY: system calls (System V)

The system call shmget() creates or allocates a System V shared memory
int shmget(key_t key, size_t size, int shmflg)

- *key* can be either an arbitrary value or one that can be derived from the library function ftok().
- *size* of the shared memory segment rounded to multiple of PAGE_SIZE.
- *shmflg* describes the required flags; such a IPC_CREATE to create a shared memory segment. This call returns a valid shared memory identifier and -1 if fails.

The shared memeory segment created can be attached to a address space of a process by the call
void * shmat(int shmid, const void *shmaddr, int shmflg)

- shmid is the identifier
- shmaddr specifies the attaching addresss; NULL would choose a default address by the OS
- shmflag specifies the flag such as SHM_RAND or SHM_EXEC etc.

SHARED MEMORY: system calls (System V)

int shmctl(int shmid, int cmd, struct shmid_ds *buf)
is used to peerform a control operation on shared memory. The argument *cmd* is used to specify the action on the segment. For example IPC_STAT Copies the information of the current values of each member of struct shmid_ds to the passed structure pointed by buf. 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.

The system call shmdt() detaches the shared segment specified by *shmaddr from a process
int shmdt(const void *shmaddr)
This call return 0 on success else -1.

SHARED MEMORY : EXAMPLE

Let us consider the following sample program.

- Create two processes, writing and reading to and from the shared memory
- 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

SHARED MEMORY : EXAMPLE

```
/* 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];
};
```

SHARED MEMORY : EXAMPLE contd.

```
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;
    }
```

SHARED MEMORY : EXAMPLE contd.

```
/* 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: SM 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;
```

SHARED MEMORY : EXAMPLE contd.

```
    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;
}
```

SHARED MEMORY : EXAMPLE using POSIX system calls

The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems. Here is an example of shared memory IPC utilising POSIX system calls.

SHARED MEMORY : EXAMPLE (POSIX)

```
// Shared Memory POSIX : Producer program shmPRODUCER.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>
int main()
{
int flag;
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "/OS";
```

SHARED MEMORY : EXAMPLE (POSIX)

```
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";
int fd; /* shared memory file descriptor */
char *ptr; /* pointer to shared memory object */
/* create the shared memory object */
fd = shm_open(name,O_CREAT | O_RDWR, 0666);
if (fd == -1) {
        perror("shm_open() failed!\n");
        exit(1);
}
/* configure the size of the shared memory object */
 flag = ftruncate(fd, SIZE);
if (flag == -1) {
        perror("ftruncate() failed!\n");
        exit(1);
}
```

SHARED MEMORY : EXAMPLE (POSIX)

```
/* memory map the shared memory object */
ptr = (char *) mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd
if (ptr == (void *)-1) {
        perror("mmap() failed!\n");
        exit(1);
}
/* write to the shared memory object */
sprintf(ptr,"%s",message_0);
ptr += strlen(message_0);
sprintf(ptr,"%s"," ");
ptr += 1;
sprintf(ptr,"%s",message_1);
ptr += strlen(message_1);
printf("Producer Done\n");
return 0;
}
```

SHARED MEMORY : EXAMPLE (POSIX)

```
// Consumer program SHARED MEMEORY: shmCONSUMER.c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>
#include <errno.h>
int main()
{
int flag;
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "/OS";
}
```

SHARED MEMORY : EXAMPLE (POSIX)

```
/* shared memory file descriptor */
int fd;
/* pointer to shared memory obect */
char *ptr;
/* open the shared memory object */
// fd = shm_open(name, O_RDONLY, 0666);
fd = shm_open(name, O_RDWR, 0666);
if (fd == -1) {
perror("shm_open() failed!\n");
exit(1);
}
```

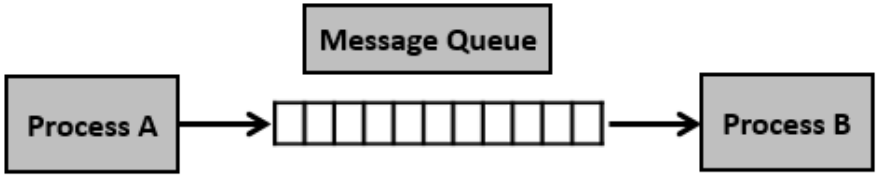
SHARED MEMORY : EXAMPLE (POSIX)

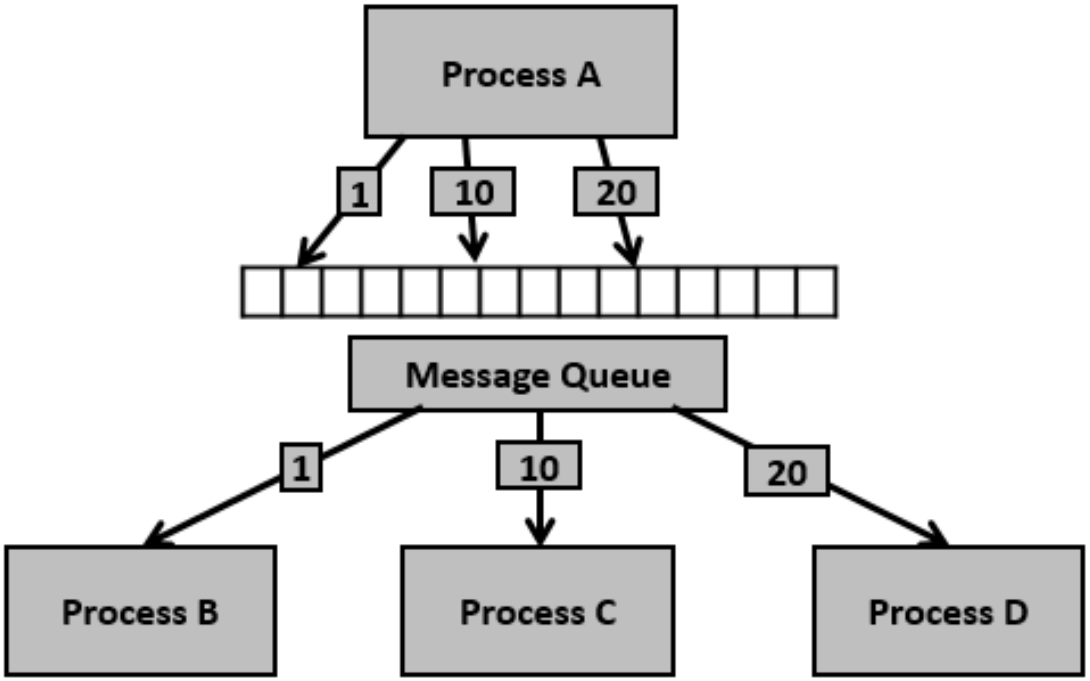
```
 flag = ftruncate(fd, SIZE);
if (flag == -1) {
        perror("ftruncate() failed!\n"); exit(1);
}
ptr = (char *) mmap((void *)
      NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
if (ptr == (void *)-1) {
perror("mmap() failed!\n");
exit(1);
}
/* read from the shared memory object */
printf("%s\n",(char *)ptr);
shm_unlink(name);
printf("\nConsumer Done\n");
return 0;
}
```

MESSAGE QUEUE : WHY?

- Once the message is consumed it is no longer available to others – unlike shared memory.
- Communjcation from one to many with different messages (data items) can be easily implemented using a message queue – accepting 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.
- For infrequent and small message transfers amongst a limited number of processes shared memory overhead does not fit the effort.
- As an extra burden Shared memory data need to be protected with synchronization when multiple processes communicating at the same time.

MESSAGE QUEUE : WHY? contd.





MESSAGE QUEUE : Stepwise Actions & calls

- Create a message queue or connect to an already existing message queue (msgget())
- Write into message queue (msgsnd())
- Read from the message queue (msgrcv())
- Perform control operations on the message queue (msgctl())

MESSAGE QUEUE : programming example

SEND part

```
/* Filename: msgqSEND.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];
};
```

MESSAGE QUEUE : programming example contd.

```
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 */
```

MESSAGE QUEUE : programming example contd.

```
    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;
}
```

MESSAGE QUEUE : programming example contd.

```
/* 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;
```

MESSAGE QUEUE : programming example contd.

```
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 concl
          /* this program ends wuth string of end */
   if (msgrcv(msqid, &buf, sizeof(buf.mtext), 0, 0) == -1) {
      perror("msgrcv");
      exit(1);
   }
```

MESSAGE QUEUE : programming example contd.

```
        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;
}
```

MESSAGE QUEUE : programming example contd.

```
// msgqSEND.c
#include <stdio.h>

#include <stdlib.h>
#include        <sys/types.h>
#include        <sys/ipc.h>
#include        <sys/msg.h>
#include        <errno.h>
#include        <string.h>

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

MESSAGE QUEUE : programming example contd.

```
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 */
```

MESSAGE QUEUE : programming example contd.

```
   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;
}
```

MESSAGE QUEUE : programming example contd.

```
//   msgqRECV.c

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>

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

MESSAGE QUEUE : programming example contd.

```
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");
```

MESSAGE QUEUE : programming example contd.

```
    for(;;) { /* normally receiving never ends but just to make concl
              /* 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;
}
```