In the name of God

Operating Systems: Homework #3

Due on November 06, 2019 at 00:00am

Professor Iman GholamPoor

Alireza Shirzad
95101847
ee.sharif.ir/∼alireza.shirzad

Inter-Process Communication (IPC)

Inter Process Communication (IPC) refers to a mechanism, where the operating systems allow various processes to communicate with each other.There are numerous reasons for providing an environment or situation which allows process co-operation:

- **Information sharing:** Since some users may be interested in the same piece of information (for example, a shared file), you must provide a situation for allowing concurrent access to that information.
- **Computation speedup:** If you want a particular work to run fast, you must break it into sub-tasks where each of them will get executed in parallel with the other tasks. Note that such a speed-up can be attained only when the computer has compound or various processing elements like CPUs or I/O channels.
- **Modularity:** You may want to build the system in a modular way by dividing the system functions into split processes or threads.
- **Convenience:** Even a single user may work on many tasks at a time. For example, a user may be editing, formatting, printing, and compiling in parallel.

Also Communications can be of two types in nature:

- Between **related processes** initiating from only one process, such as parent and child processes
- Between **unrelated processes**, or two or more different processes.

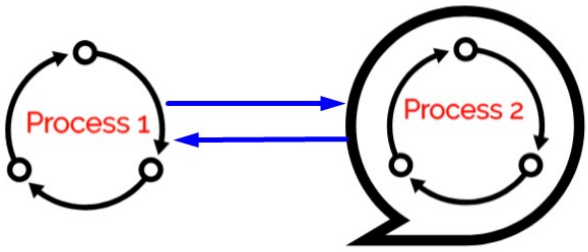

Figure 1: Inter-Process Communication

There are two primary models of interprocess communication :

- **message passing/Pipes**
- **shared memory**

We continue to explain both approaches to IPC :

message passing/Pipes

In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follow:

- Establish a communication link (if a link already exists, no need to establish it again)
- Start exchanging messages using basic primitives

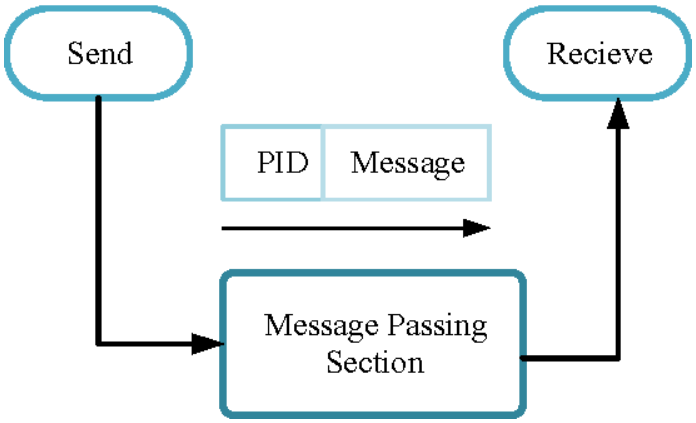

Figure 2: Message Passing

We need at least two primitives:

- **send**(message, destinaion) or **send**(message)
- **receive**(message, host) or **receive**(message)

Example Program

Algorithm:

1. Create a pipe
2. Send a message to the pipe
3. Retrieve the message from the pipe and write it to the standard output
4. Send another message to the pipe
5. Retrieve the message from the pipe and write it to the standard output
6. Retrieving messages can also be done after sending all messages

C code:

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

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

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

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

```
21 read(pipefds[0], readmessage, sizeof(readmessage));
22 return 0;
23 }
```

Shared Memory

Inter-process communication (IPC) usually utilizes shared memory that requires communicating processes for establishing a region of shared memory. Typically, a shared-memory region resides within the address space of any process creating the shared memory segment. Other processes that wish for communicating using this shared-memory segment must connect it to their address space.normally what happens, the operating system tries to check one process from accessing other's process's memory. Shared memory needs that two or more processes agree to remove this limitation. They can then exchange information via reading and writing data within the shared areas. The form of the data and the location gets established by these processes and are not under the control of the operating system. The processes are also in charge to ensure that they are not writing to the same old location simultaneously. We know that to communicate between two or

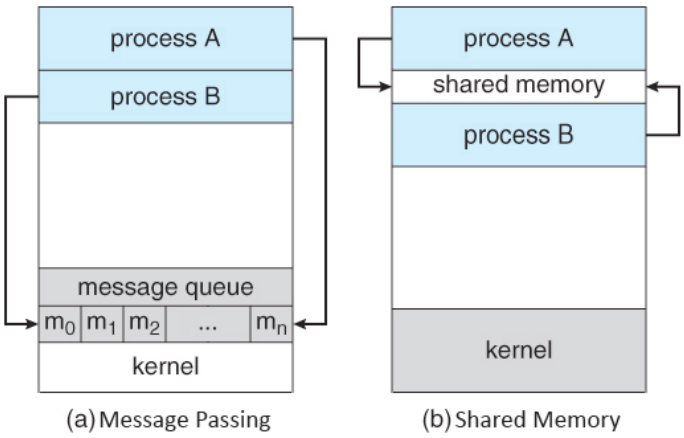

Figure 3: Shared Memory

more processes, we use shared memory but before using the shared memory what needs to be done with the system 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())
- Detach the process from the already attached shared memory segment (shmdt())
- Control operations on the shared memory segment (shmctl())

Let us consider the following sample program:

1. Create two processes, one is for writing into the shared memory (shm write.c) and another is for reading from the shared memory(shm read.c)
2. The program performs writing into the shared memory by write process (shm write.c) and reading from the shared memory by reading process (shm read.c)
3. In the shared memory, the writing process, creates a shared memory of size 1K (and flags) and attaches the shared memory
4. 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

5. Read process would read from the shared memory and write to the standard output
6. Reading and writing process actions are performed simultaneously
7. After completion of writing, the write process updates to indicate completion of writing into the shared memory (with complete variable in struct shmseg)
8. 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)
9. Performs reading and writing process for a few times for simplication and also in order to avoid infinite loops and complicating the program

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

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

#define BUF_SIZE 1024
#define SHM_KEY 0x1234

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

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

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

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

```
spaceavailable = BUF_SIZE;
sleep(3);
}
printf("Writing Process: Wrote %d times\n", numtimes);
shmp->complete = 1;

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

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

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

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

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

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

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

#define BUF_SIZE 1024
#define SHM_KEY 0x1234

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

```

```
19 int main(int argc, char *argv[]) {
20 int shmid;
21 struct shmseg *shmp;
22 shmid = shmget(SHM_KEY, sizeof(struct shmseg), 0644|IPC_CREAT);
23 if (shmid == -1) {
24 perror("Shared memory");
25 return 1;
26 }
27
28 // Attach to the segment to get a pointer to it.
29 shmp = shmat(shmid, NULL, 0);
30 if (shmp == (void *) -1) {
31 perror("Shared memory attach");
32 return 1;
33 }
34
35 /* Transfer blocks of data from shared memory to stdout*/
36 while (shmp->complete != 1) {
37 printf("segment contains : \n\"%s\"\n", shmp->buf);
38 if (shmp->cnt == -1) {
39 perror("read");
40 return 1;
41 }
42 printf("Reading Process: Shared Memory: Read %d bytes\n", shmp->cnt);
43 sleep(3);
44 }
45 printf("Reading Process: Reading Done, Detaching Shared Memory\n");
46 if (shmdt(shmp) == -1) {
47 perror("shmdt");
48 return 1;
49 }
50 printf("Reading Process: Complete\n");
51 return 0;
52 }
```