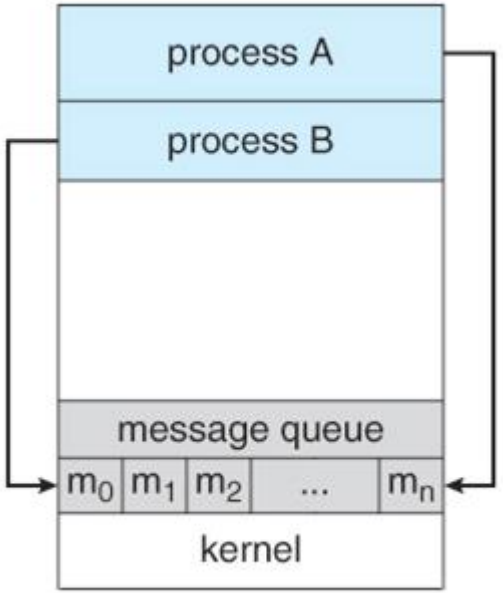
Linux System Programming Part 6 - IPC (synchronization)

IBA Bulgaria
2021

Message queue

- Message queues can be described as an internal linked list within the kernel's addressing space.
- Messages can be sent to the queue in order and retrieved from the queue in several different ways.
- Each message queue (of course) is uniquely identified by an IPC identifier.



Messaging in C

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

int msgget(key_t key, int msgflg);
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

The **msgget**() system call returns the System V message queue identifier associated with the value of the ***key*** argument.
The **msgsnd**() and **msgrcv**() system calls are used, respectively, to send messages to, and receive messages from, a System V message queue. The calling process must have write permission on the message queue in order to send a message, and read permission to receive a message.
msgctl() and ***cmd*** = **IPC_RMID** removes a message queue.

Define common data

- Define **KEY** - the queue unique ID
- Define **MAXLEN** - max length of message
- Define **msg_1_t** for the client message
- Define **msg_1_2** for the server response

```
#ifndef MSG_TYPES
#define MSG_TYPES

#define KEY 1274
#define MAXLEN 512

struct msg_1_t
{
    long mtype;
    int snd_pid;
    char body[MAXLEN];
};

struct msg_2_t
{
    long mtype;
    int snd_pid;
    int rcv_pid;
    char body[MAXLEN];
};

#endif
```

msgsrv.c

Message Server

- Create a message queue with a given **KEY**
- Wait for a message of type **msg_1_t** from the queue
- Print the received message data
- Prepare and send answer of type **msg_2_t**
- Wait for confirmation **msg_1_t**
- Remove the message queue

```c
struct msg_1_t message1;
struct msg_2_t message2;
int msgid;
char * response = "Ok!";

msgid = msgget(KEY, 0777 | IPC_CREAT);
msgrcv(msgid, &message1, sizeof(struct msg_1_t), 1, 0);
printf("Client (pid = %i) sent: %s", message1.snd_pid,
message1.body);

message2.mtype = 2;
message2.snd_pid = getpid();
message2.rcv_pid = message1.snd_pid;
strcpy(message2.body, response);
msgsnd(msgid, &message2, sizeof(struct msg_2_t), 0);

msgrcv(msgid, &message1, sizeof(struct msg_1_t), 1, 0);

msgctl(msgid, IPC_RMID, 0);
return EXIT_SUCCESS;
```

Message Client

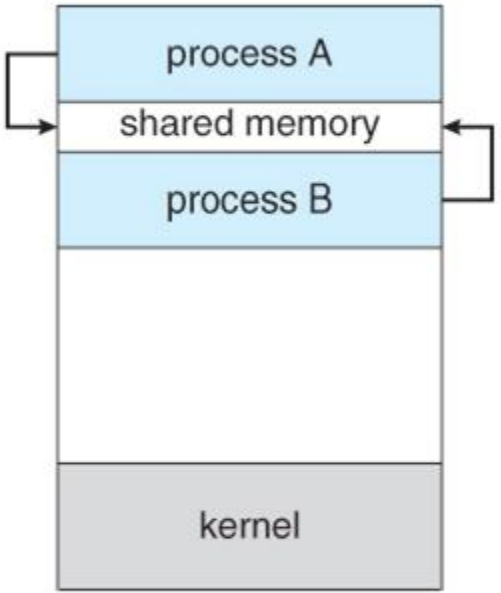
- Open a message queue with a given **KEY** and notify and exit if failed
- Read a message body from the keyboard and set the other parameters of **msg_1_t**
- Send **msg_1_t** message to the queue
- Wait for **msg_2_t** and print it
- Send back a confirmation **msg_1_t**

```c
int msgid;
int i;
struct msg_1_t message1;
struct msg_2_t message2;
char buf[MAXLEN];
msgid = msgget(KEY, 0666);
if (msgid == -1){
    printf("Server is not running!\n", msgid);
    return EXIT_FAILURE;
}
i = 0;
while ( (i < (MAXLEN - 1)) &&
        ((message1.body[i++] = getchar()) !=  '\n') );

message1.body[i] = '\0';
message1.mtype = 1;
message1.snd_pid = getpid ();
msgsnd(msgid, &message1, sizeof(struct msg_1_t), 0);
msgrcv(msgid, &message2, sizeof(struct msg_2_t), 2, 0);
printf("Server (pid= %i) responded: %s\n",
message2.snd_pid, message2.body);
message1.mtype = 1;
msgsnd(msgid, &message1, sizeof(struct msg_1_t), 0);
return EXIT_SUCCESS;
```

Shared Memory

- Shared memory can be described as the mapping of an area (segment) of memory that will be mapped and shared by more than one process.
- This is by far the fastest form of IPC, because there is no intermediation - information is mapped directly from a memory segment into the addressing space of the calling process.
- A segment can be created by one process, and subsequently written to and read from by any number of processes.



(b) Shared Memory

Shared memory in C

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

key_t ftok(const char *pathname, int proj_id);
int shmget(key_t key, size_t size, int shmflg);
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

ftok() - convert a pathname and a project identifier to a System V IPC key.
shmget() - allocates a System V shared memory segment.
shmat() - attaches the System V shared memory segment identified by ***shmid*** to the address space of the calling process.
shmdt() - detaches the shared memory segment located at the address specified by ***shmaddr*** from the address space of the calling process.
shmctl() with ***cmd*** = **IPC_RMID** - marks the segment to be destroyed.

Define common data

Define **FTOK_FILE** - the queue unique ID

Define the shared memory structure **memory_block:**

server_lock is **1** when server uses the memory

client_lock is **1** when client uses the memory

turn is **0** when waiting for client message
turn is **1** when waiting for server message

readlast is **0** when client received last message
readlast is **1** when server received last message

string holds the current message

```
#ifndef SHMEM_TYPES
#define SHMEM_TYPES

#define FTOK_FILE "./shmemserv"

#define MAXLEN 512
#define FREE 1
#define BUSY 0
#define SERVER 1
#define CLIENT 0

struct memory_block
{
  int server_lock;
  int client_lock;
  int turn;
  int readlast;
  char string[MAXLEN];
};

#endif
```

shmemserv.c

Memory Server

- Create token '**./shmemserv**', exit on error
- Allocate and attach shared memory
- Configure the memory for client turn and write 'Hello!' message in it
- While current message is not '**q**':
  - Lock memory for server, set client's turn
  - Wait if client is using the memory
  - If client processed last message - indicate server processed the current one, print the client's message, return 'Ok!', and release the server lock
- Detach and remove the shared memory

```
key = ftok(FTOK_FILE, 1);
if (key == -1)
...
shmid = shmget(key, sizeof(struct memory_block), 0666 | IPC_CREAT);
mblock = (struct memory_block *) shmat(shmid, 0, 0);
mblock->turn = CLIENT;  mblock->server_lock = FREE;
mblock->client_lock = FREE;  mblock->readlast = SERVER;
strcpy(mblock->string, "Hello!");
while (strcmp("q\n", mblock->string) != 0){
   mblock->server_lock = BUSY;
   mblock->turn = CLIENT;
   while ((mblock->client_lock == BUSY) && (mblock->turn == CLIENT));
   if (mblock->readlast == CLIENT){
     mblock->readlast = SERVER;
     printf("String sent by the client is: %s", mblock->string);
     if (strcmp("q\n", mblock->string) != 0)
         strcpy(mblock->string, "Ok!");
     mblock->server_lock = FREE;
   }
}
printf("Server got q and exits\n");
shmdt((void *) mblock);
shmctl(shmid, IPC_RMID, 0);
```

shmemcli.c

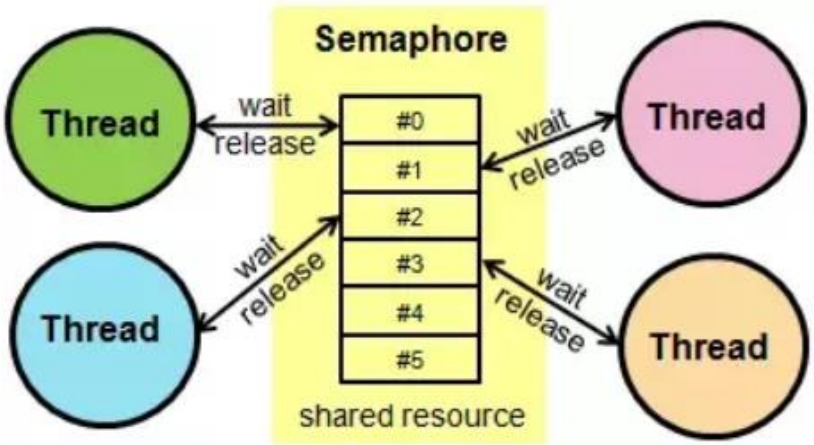
Memory Client

- Create token '**./shmemserv**', exit on error
- Attach the shared memory
- While current message is not '**q**':
  - Lock memory for client, set server's turn
  - Wait if server is using the memory
  - If server processed last message - indicate client processed the current one, print the server's message, read a line from keyboard, and release the client lock
- Detach the shared memory

```
struct memory_block * mblock;
key = ftok(FTOK_FILE, 1);
if (key == -1) //ERROR
...
shmid = shmget(key, sizeof(struct memory_block), 0666);
if (shmid == -1) //ERROR
...
mblock = (struct memory_block *) shmat(shmid, 0, 0);
while (strcmp("q\n", mblock->string) != 0){
    int i = 0;
    mblock->client_lock = BUSY;
    mblock->turn = SERVER;
    while ((mblock->server_lock == BUSY)
       && (mblock->turn == SERVER));
    if (mblock->readlast == SERVER){
       mblock->readlast = CLIENT;
       printf("Server sends %s\n", mblock->string);
       while ((i < (MAXLEN - 1)) &&
           ((mblock->string[i++] = getchar()) !=  '\n') );
       mblock->string[i] = 0;
       mblock->client_lock = FREE;
    }
}
printf("Client exits\n");
shmdt((void *) mblock);
```

Semaphore

- **Semaphores** are counters used to control access to shared resources by multiple processes (resource counters).
- They are used as a locking mechanism to prevent processes from accessing a particular resource while another process is performing operations on it.
- A resource counter is decreased when a process starts using a resource and increased back, when the resource is released.
- When the counter = 0, the resource is unavailable at the moment.



Using semaphores

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

int semget(key_t key, int nsems, int semflg);
int semctl(int semid, int semnum, int cmd, ...);
int semop(int semid, struct sembuf *sops, size_t nsops);
```

semget() - returns the System V semaphore set identifier associated with the argument ***key***.

semctl() with ***cmd*** = **SETVAL** - Sets the value of ***semval*** to ***arg.val*** for the ***semnum***-th semaphore of the set.

semctl() with ***cmd*** = ***IPC_RMID*** - removes the semaphore set.

semop() performs operations on selected semaphores in the set indicated by ***semid***.

Define common data

Define **FTOK_FILE** = '**./semserv**'

Define the shared memory structure **memory_block:**

string holds the current message

```
#ifndef SHMEM_TYPES
#define SHMEM_TYPES

#define FTOK_FILE "./semserv"

#define MAXLEN 512

struct memory_block
{
    char string[MAXLEN];
};



#endif
```

Semaphore Server

- Set 2 semaphores - first indicating that the server has to read, second for the client
- Create token '**./shmemserv**', exit on error
- Allocate and attach shared memory and write 'Hello!'
- Release the client semaphore
- While current message is not '**q**':
  - Request server semaphore resource
  - If current message is not '**q**', write 'Ok!' in memory and release the client sem.
- Detach and remove the shared memory and the semaphores

```
...
struct sembuf buf[2];
buf[0].sem_num = 0;
buf[0].sem_flg = SEM_UNDO;
buf[1].sem_num = 1;
buf[1].sem_flg = SEM_UNDO;
semid = semget(key, 3, 0666|IPC_CREAT);
semctl(semid, 0, SETVAL, 0);
...
buf[0].sem_op = -1;
buf[1].sem_op = 1;
semop(semid, (struct sembuf*) &buf[1], 1);
while (strcmp("q\n", mblock->string) != 0) {
    semop(semid, (struct sembuf*) &buf, 1);
    printf("String sent by the client is: %s",
                mblock->string);
    if (strcmp("q\n", mblock->string) != 0)
    strcpy(mblock->string, "Ok!");
    buf[0].sem_op = -1;
    buf[1].sem_op = 1;
    semop(semid, (struct sembuf*) &buf[1], 1);
}
...
semctl(semid, 2, IPC_RMID);
return EXIT_SUCCESS;
```

Semaphore Client

- Set 2 semaphores - first indicating that the server has to read, second for the client
- Create token '**./shmemserv**', exit on error
- Attach the shared memory
- While current message is not '**q**':
  - Request client semaphore resource and print the next message
  - Read a new line from the keyboard to the shared memory
  - Release the server semaphore
- Detach the shared memory

```
...
struct sembuf buf[2];
semid = semget(key, 2, 0666);
...
buf[1].sem_op = -1;
while (strcmp("q\n", mblock->string) != 0) {
    int i = 0;
    semop(semid, (struct sembuf*) &buf[1], 1);
    printf("Server sends %s\n", mblock->string);
    while ((i < (MAXLEN - 1)) &&
        ((mblock->string[i++] = getchar()) !=  '\n') );
    mblock->string[i] = 0;
    buf[0].sem_op = 1;
    buf[1].sem_op = -1;
    semop(semid, (struct sembuf*) &buf, 1);
}
printf("Client exits\n");
shmdt((void *) mblock);
return EXIT_SUCCESS;
```

Exercise

Project *DoubleSharedMemory*:

Write a server/client pair of programs (‘**dblmemsrv.c**’ and ‘**dblmemcli.c**’), which share blocks of memory between them. The server supports up to 2 client instances in parallel execution, which should not interfere with each other. The server initializes the shared memory with welcoming messages for the clients and waits for them to modify the memory with their messages. When a new message from the client is written, the server answers with the same message, but with ‘Confirmed!” at the end.

The client is started with a single argument: **client_number**, which identifies the client (1 or 2) for the server. The client should continuously print the message from the shared memory, read a line from the keyboard and write it to the shared memory. The program should quit when ‘**q**’ message is entered.

Program *DoubleDumper*:

Write a program (‘**dbldump.c**’), which in real time shows the changes in the shared memory from the ***DoubleSharedMemory*** project. You may need to modify the server and/or client programs to support this functionality.