Lab 2-2. Interprocess Communication – Message Queues

Objectives

- Understand the structure of message queue, and implement the example of server-client program.

Background

1. Name Space:

   The table is set of possible names for a given type of IPC. A key is a long int, used to identify individual resources.

IPC Type	Name space	Identification
PIPE	No name	File descriptor
FIFO	Pathname	File descriptor
Message Queue	Key	Identifier
Semaphore	Key	Identifier
Shared Memory	Key	Identifier
Socket (Unix Domain)	Pathname	File descriptor
Socket (Internet Domain)	Socket	Socket descriptor

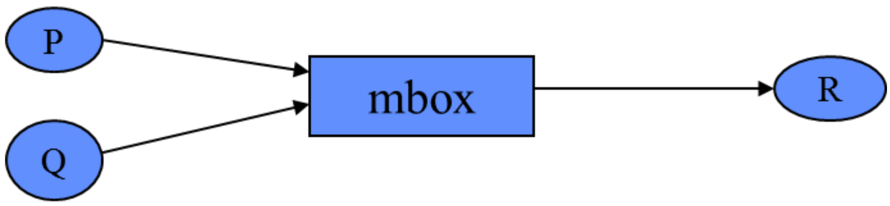
2. Similarities of the 3 types of IPC Channels:

	Message Queue	Semaphore	Shared Memory
include	<sys/msg.h>	<sys/sem.h>	<sys/shm.h>
System calls to create or open	msgget	semget	shmget
System call for control operation	msgctl	semctl	shmctl
System calls for IPC operations	msgsnd, msgrcv	semop	shmat, shmdt

3. IPC Flags:
   - IPC_CREAT: Creates a unique IPC channel if none exists; otherwise use an existing IPC channel.
   - IPC_EXCL: When used with IPC_CREAT, error if the channel exists.

<table>
<tr><th>Flag argument</th><th>Key does not exist</th><th>Key already exist</th><th>Purpose</th></tr>
<tr><td>NO special flags</td><td>Error, errno=ENOENT</td><td>OK</td><td>to open</td></tr>
<tr><td>IPC_CREAT</td><td>OK, create new entry</td><td>OK, returns same id</td><td>To create or to open</td></tr>
<tr><td>IPC_CREAT | IPC_EXCL</td><td>OK, create new entry</td><td>Error, errno=EEXIST</td><td>To distinguish exist or noexist</td></tr>
</table>

4. Necessary of Message Queue: Drawback of PIPE and FIFO
   1) Pipe or FIFO is an unformatted stream. Message queue is a formatted stream consisted of messages.
   2) Pipe or FIFO has to be read in the same order as they are written. Message queue can be accessed randomly.
   3) Pipe or FIFO is unidirectional. Message queue is bidirectional.
   4) Pipe or FIFO is simplex. Message queue is multiplex.
   5) Message queue is faster since it is in kernel.
5. Message Queue
   1) Also known as mailbox, ports
   2) The explicit and symmetric naming of processes in direct naming
   3) Þ Limited modularity since changing the name of a process requires changes elsewhere, i.e., in definitions of other processes



4) P or Q call

   send(mbox-id, message)

   R calls

   receive(mbox-id, message)

- The msgget() function returns the message queue identifier associated with key.
  A message queue identifier and an associated message queue and data structure are created if key is equal to IPC_PRIVATE, or key does not already have a message queue identifier associated with it and (msgflg & IPC_CREAT) is non-zero.

Function	**int msgget (key_t key, int msgflg);**
Function arguments	
key	Specifies the message queue key for which to retrieve the msqid.
msgflg	Is a flag that indicates specific message queue conditions and options to implement.
Return values	
If successful, msgget() returns a message queue identifier. On failure, msgget() returns a value of -1 and sets errno to indicate the error.	

- The msgsnd() function sends a message to the queue associated with message queue identifier msqid.

Function	**int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);**
Function arguments	
msqid	Is a unique positive integer, created by msgget(), that identifies a message queue and its associated data structure.
msgp	Points to a user-defined buffer.
msgsz	Is the length of the message to be sent.
msgflg	Specifies the action to be taken if one or more of the following are true
Return values	
If successful, msgsnd() returns a message queue identifier. On failure, msgsnd() returns a value of -1 and sets errno to indicate the error	

- The msgrcv() function reads a message from the queue associated with the message queue identifier that msqid specifies and places it in the user-defined structure that msgp points to.

Function	**int msgrcv(int msqid, void *msgp, int msgsz, long msgtyp, int msgflg);**
Function arguments	
msqid	Is a unique positive integer, created by a msgget() call, that identifies a message queue
msgflg	Points to a user-defined buffer
msgsz	Specifies the size, in bytes, of mtext
msgtyp	Specifies the type of message requested (3 types: -, 0, +)
Return values	
If successful, msgrcv() returns the number of bytes actually placed into mtext. On failure, msgrcv() returns a value of -1, receives no message, and sets errno.	

6. Semaphore

1) Semaphore

- used for synchronization among processes
- used to synchronize the access of the shared memory segment
- Avoid more than two Processes being Busy waiting at the same time (In User Process, it needs to avoid that more than two processes try to change the same variable)
- Each Semaphore has its own waiting process list.
- Maximum number of Semaphore depends on the number of processes waiting for the corresponding process
- In many cases, you need to synchronize processes
- When they share a resource (shared memory, file descriptor, device, etc.)
- When a process needs to wait for a given event
- Are not used for exchanging large amounts of data as pipes,FIFOs
- Let multiple processes synchronize their operations to synchronize the access to shared memory segments
- Since it provides the resource synchronization between different processes, it must be stored in the kernel

2) Semaphores are positive integers with two methods: UP() and DOWN()

- DOWN():
  - If sem>=1 then sem=sem-1
  - Otherwise, block the process
- UP():
  - If there are blocked processes, then unblock one of them
  - Otherwise, sem=sem+1
- Semaphore operations are atomic

3) **There are generally two usages for semaphores:**

- Mutual exclusion: only one process at a time can be within a section of code
  - Start with sem=1
  - To enter the mutex section: DOWN(sem)
  - To leave the mutex section: UP(sem)
- Wait for a given event:
  - Start with sem=0
  - To wait for the event: DOWN(sem)
  - To trigger the event: UP(sem)
  - What happens if the event is triggered before the other process waits for it?

4) Functions and Structures

- semget()

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

```
int semget(key_t key, int nsems, int semflag);
```

create semaphore or use the existing semaphore

- nsems : number of semaphore
- semflag : combination of the following constants

- semop()

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

int semop(int semid, struct sembuf **opsptr, unsigned int nops);

struct sembuf {
     ushort sem_num;          /* semapore # */
     short    sem_op; /* semaphore operation */
     short    sem_flg;/* operation flags */
};
```

execute the arithmetic functions for more than one semaphore

- sem_op :
  - sem_op > 0 : add sem_val on the value of existing semaphore, It mean to release the resources controlled by semaphore
  - sem_op = 0 : the called process wait when the value of semaphore reduces to zero
  - sem_op < 0 : the called process wait when the value of semaphore be greater than or equal to sem_op. It means to allocate the resources.

- semctl()

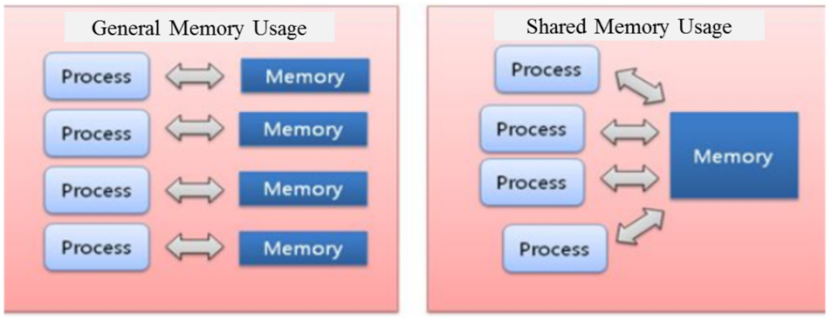
```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, union semun arg);

union semun {
          int val;    /* used for SETVAL only */
          struct semid_ds *buff;          /* used for IPC_STAT and IPC_SET */
          ushort *array;        /* used for IPC_GETALL & IPC_SETALL */
} arg;
```

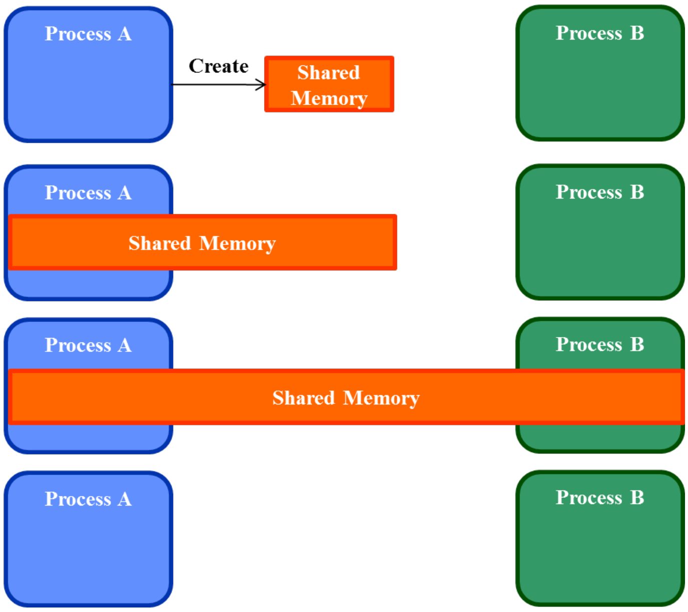
- cmd :
  - IPC_RMID is used to remove semaphore
  - GETVAL is to get the value of semaphore
  - SETVAL is used to change the value of semaphore

7. Shared Memory



1) Processes can share the same segment of memory directly when it is mapped into the address space of each sharing process
2) Faster communication
3) Mutual exclusion must be provided by processes using the shared memory

- ◆ Generation of the shared memory: shmget()
- ◆ Access to the shared memory: shmat()
- ◆ Finishing access of the shared memory: shmdt()
- ◆ Controlling the shared memory: shmctl()



- The shmget() function returns the shared memory identifier associated with key. A shared memory identifier and shared memory segment of at least size bytes is created if key is equal to IPC_PRIVATE, or if key does not already have a shared memory identifier associated with it and if (shmflg & IPC_CREAT) is non-zero.

Function	**int shmget (key_t key, int size, int shmflg);**
Function arguments	
key	Specifies either IPC_PRIVATE or a unique key.
size	Is the size in bytes of the shared memory segment.
shmflg	Specifies both the creation and permission bits (for example, IPC_CREAT | 0666).
Return values	
If successful, shmget() returns a shared memory identifier. On failure, it returns -1 and sets errno.	

- The shmat() function attaches the shared memory segment associated with the shared memory identifier *shmid* to the data segment of the calling process.

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

Function arguments	
shmid	Is a unique positive integer created by a shmget() system call and associated with a segment of shared memory.
shmaddr	Points to the desired address of the shared memory segment.
shmflg	Specifies a set of flags that indicate the specific shared memory conditions and options to implement.
Return values	
If successful, shmat() returns the data segment start address of the attached shared memory segment. On failure, it returns -1, does not attach the shared memory segment, and sets errno	

- The shmdt() function detaches from the calling process's data segment the shared memory segment located at the address specified by shmaddr.

Function	**int shmdt(const void **shmaddr*);**
Function arguments	
shmaddr	Is the data segment start address of a shared memory segment.
Return values	
If successful, shmdt() decrements the shm_nattach associated with the shared memory segment and returns zero. On failure, it returns -1 and set errno.	

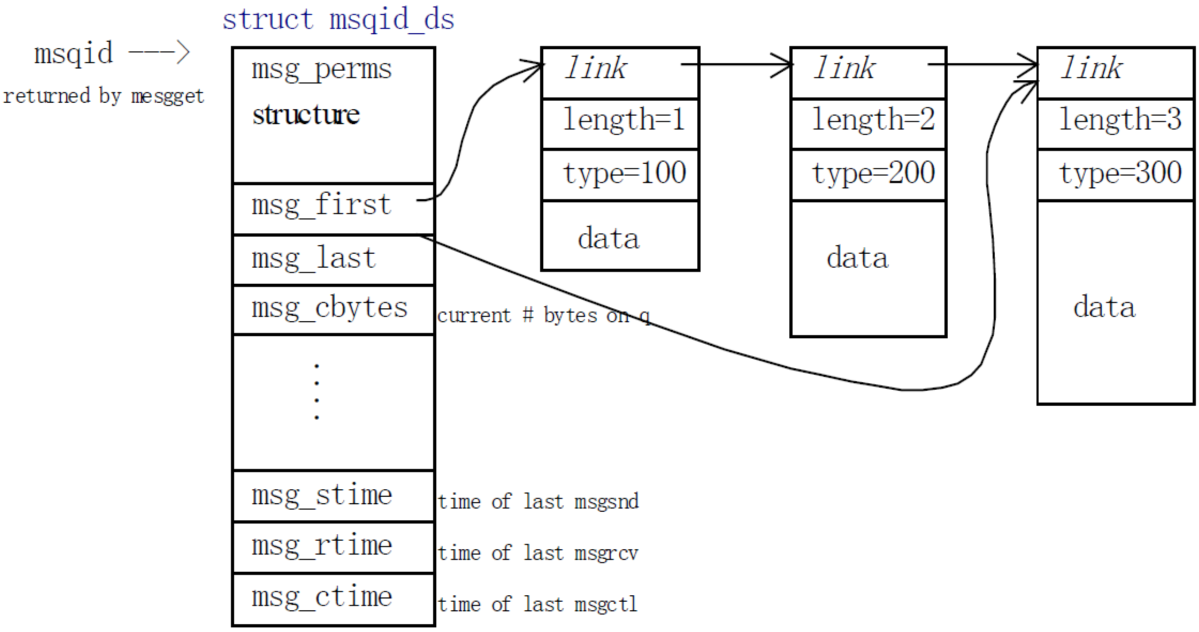
- The shmctl() function provides a variety of shared memory control operations as specified by cmd.

Function	**int shmctl(int *shmid*, int *cmd*, struct shmid_ds **buf*);**
Function arguments	
shmid	Is a unique positive integer returned by the shmget() function and associated with a segment of memory and a data structure.
cmd	Specifies one of IPC_STAT, IPC_SET, or IPC_RMID.
buf	Points to the data structure used for sending or receiving data during execution of shared memory control operations.
Return values	
If successful, shmctl() returns zero. On failure, it returns -1 and sets errno.	

Practice

Message format: it can be user-defined. Every message has

- size (int), i.e., the length of data,
- type (long int),
- data (if length > 0).



Message Queue Structures in Kernel

```
#define MAXMESGDATA (4096-16)
#define MESGHDRSIZE (sizeof(Mesg) - MAXMESGDATA)
typedef struct {
        int mesg_len;
        long mesg_type;
        char mesg_data[MAXMESGDATA];
} Mesg;
```

A message queue is created or opened using: **int msgget(key_t key, int msgflag)**

- msgflag: sets permission + IPC_CREAT / IPC_EXCL. See Comments 1 at page 1.
- msgid: returned by the msgget() function, -1 on error.

Messages are sent using: **int msgsnd(int msgid, struct msgbuf *ptr, int length, int flag)**

- struct msgbuf: message format, has a long int (message type) as a field, immediately followed by the data. Must be filled by caller before calling msgsnd( ).
- length: determine how many bytes of data portion are sent.
- flag: can be IPC_NOWAIT which causes the function to return immediately with error=EAGAIN if no queue space is available. Default action is to block.
- return: 0 if successful; -1 on error.

Messages are received using: **int msgrev(int msgid, struct msgbuf *ptr, int length, long msgtype, int flag)**

- length: specify the size of the data buffer. Error if the message is too long. OK if the length ≥ message length which is determined by msgsnd(…length…).
- flag: if set to MSG_NOERROR, long messages are truncated and the rest is discarded. If set to IPC_NOWAIT,
- function return –1 if no matching messages are found, otherwise block until

- ▲ A message with the appropriate type is placed on the queue.
- ▲ The message queue is removed from the system.
- ▲ The process receives a signal that is not ignored.

- msgtype = 0: read the first msg on the queue.
  >0: read the first msg on the queue with this type.
  <0: read the first msg on the queue with the lowest type ≤ | msgtype|
- return: the length of the data received in bytes if successful; -1 on error.

Control operations use: **int msgctl(int msgid, int cmd, struct msg-id-ds *buff)**

- To remove a msg queue, cmd = IPC_RMID; the 3rd parameter is just a NULL pointer.
- To get/set the queue information in buff, cmd = IPC-STAT/IPC-SET.

Example of message simple client-server using Message Queue:

msg_server.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mesg.h"

#define MKEY1 1234L
#define MKEY2 2345L
#define PERMS 0666

Mesg mesg;

void server(int ipcreadfd, int ipcwritefd) {
        int n, filefd;
        mesg.mesg_type = 1L;
        if((n = mesg_recv(ipcreadfd, &mesg)) <= 0)
                exit(1);
        mesg.mesg_data[n] = '\0';

        if((filefd = open(mesg.mesg_data, 0)) < 0) {
                sprintf(mesg.mesg_data, "can't open the file\n");
                mesg.mesg_len = strlen(mesg.mesg_data);
                mesg_send(ipcwritefd, &mesg);
        }
        else {
                while((n = read(filefd, mesg.mesg_data, MAXMESGDATA)) > 0) {
                        mesg.mesg_len = n;
                        mesg_send(ipcwritefd, &mesg);
                }
                close(filefd);
                if(n < 0)
                        exit(1);
        }
        mesg.mesg_len = 0;
        mesg_send(ipcwritefd, &mesg);
}

int main(void) {
        int readid, writeid;
```

```
	if((readid = msgget(MKEY1, PERMS | IPC_CREAT)) < 0)
		exit(1);
	if((writeid = msgget(MKEY2, PERMS | IPC_CREAT)) < 0)
		exit(1);

	server(readid, writeid);
	return 0;
}
```

msg_client.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mesg.h"

#define MKEY1 1234L
#define MKEY2 2345L

Mesg mesg;

void client(int ipcreadfd, int ipcwritefd) {
	int n;
	if(fgets(mesg.mesg_data, MAXMESGDATA, stdin) == NULL)
		exit(1);
	n = strlen(mesg.mesg_data);
	if(mesg.mesg_data[n-1] == '\n')
		--n;
	mesg.mesg_len = n;
	mesg.mesg_type = 1L;
	mesg_send(ipcwritefd, &mesg);

	while((n = mesg_recv(ipcreadfd, &mesg)) > 0) {
		if(write(1, mesg.mesg_data, n) != n)
			exit(1);
	}
	if(n < 0)
		exit(1);
}

int main(void) {
	int readid, writeid;
	if((writeid = msgget(MKEY1, 0)) < 0)
		printf("client: can't msgget message queue 1\n");
	if((readid = msgget(MKEY2, 0)) < 0)
		printf("client: can't msgget message queue 2\n");

	client(readid, writeid);

	if(msgctl(readid, IPC_RMID, (struct msqid_ds*)0) < 0)
		exit(1);
	if(msgctl(writeid, IPC_RMID, (struct msqid_ds*)0) < 0)
		exit(1);
	return 0;
}
```

mesg.h

```
#include <stdlib.h>
#include <sys/msg.h>

#define MAXMESGDATA (4096-16)
#define MESGHDRSIZE (sizeof(Mesg) - MAXMESGDATA)
typedef struct {
        int mesg_len;
        long mesg_type;
        char mesg_data[MAXMESGDATA];
} Mesg;

void mesg_send(int id, Mesg* mesgptr);
int mesg_recv(int id, Mesg* mesgptr);
```

mesg.c

```
#include "mesg.h"

void mesg_send(int id, Mesg* mesgptr) {
        if(msgsnd(id, (char*)&(mesgptr->mesg_type), mesgptr->mesg_len, 0) != 0)
                exit(1);
}

int mesg_recv(int id, Mesg* mesgptr) {
        int n;
        n = msgrcv(id, (char*)&(mesgptr->mesg_type), MAXMESGDATA, mesgptr->mesg_type,
0);
        if((mesgptr->mesg_len = n) < 0)
                exit(1);
        return n;
}
```

Result:

```
ancl@cloud03:~/lab$ cat msg_queue_test.txt
message queue test
interprocess comunication test

practice
example

ancl@cloud03:~/lab$ ./msg_server
ancl@cloud03:~/lab$
```

```
ancl@cloud03:~/lab$ ./msg_client
/home/ancl/lab/msg_queue_test.txt
message queue test
interprocess comunication test

practice
example

ancl@cloud03:~/lab$
```

Multiplexing Message and bi-directional message queue:

- Only one queue, but used for 2 directions, and multiple clients can read/write the queue.