Operating systems

Interprocess communication (IPC)
Part 2 of 3: Shared Memory and Message Queue

Created by
Enrico Fraccaroli
enrico.fraccaroli@gmail.com

Table of Contents

Shared memory

Shared memory

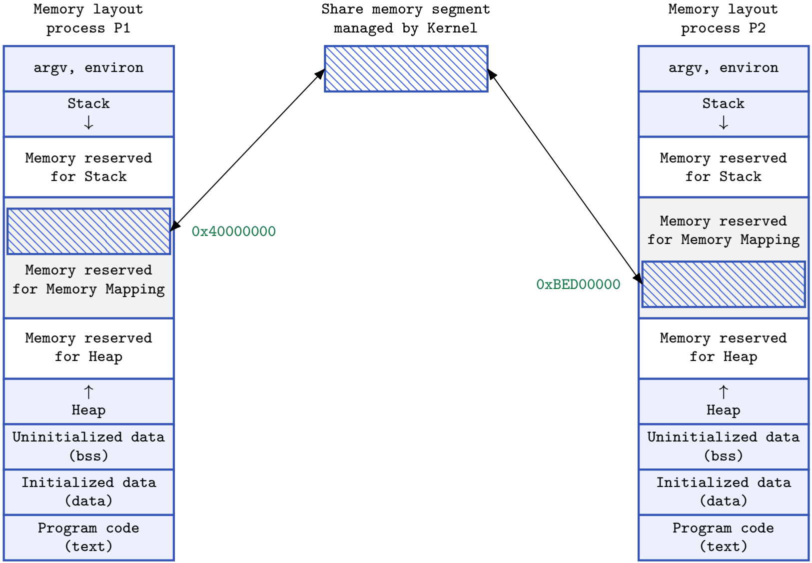
Fundamental concepts

Fundamental concepts

A shared memory is a **memory segment** of **physical memory** managed by Kernel, which allows two or more processes to **exchange data**.
Once attached, even more then once, the shared memory is **part of the process's virtual address space**, and no kernel intervention is required.

Data written in a shared memory is **immediately** available to all other process sharing the same segment. Typically, some method of **synchronization** is required so that processes **don't simultaneously access** the shared memory (for instance, semaphores!).

Fundamental concepts



Shared memory

Creating and Opening

Creating/Opening a shared memory segment

The `shmget` system call creates a new shared memory segment or obtains the identifier of an existing one. The content of a newly created shared memory segment is initialized to 0.

```
#include <sys/shm.h>

// Returns a shared memory segment identifier on success, or -1 on error
int shmget(key_t key, size_t size, int shmflg);
```

The key arguments are:

- an IPC `key`.
- `size` specifies the desired size [1] of the of segment, in bytes.
- if we are using `shmget` to obtain the identifier of an existing segment, then size has no effect on the segment, but it must be less than or equal to the size of the segment.

[1] `size` is rounded up to the next multiple of the system page size

Creating/Opening a shared memory segment

`shmflg` is a bit mask specifying the permissions (see `open(...)` system call, `mode` argument) to be places on a new shared memory segment or checked against an existing segment. In additions, the following flags can be ORed (|) in `shmflg`:

- `IPC_CREAT`: If no segment with the specified `key` exists, create a new segment
- `IPC_EXCL`: in conjunction with IPC_CREAT, it makes `shmget` fail if a segment exists with the specified `key`.

Creating/Opening a shared memory segment

Example showing how to create a shared memory segment

```
int shmid;
ket_t key = //... (generate a key in some way, i.e. with ftok)
size_t size = //... (compute size value in some way)

// A) delegate the problem of finding a unique key to the kernel
shmid = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);

// B) create a shared memory with identifier key, if it doesn't already exist
shmid = shmget(key, size, IPC_CREAT | S_IRUSR | S_IWUSR);

// C) create a shared memory with identifier key, but fail if it exists already
shmid = shmget(key, size, IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
```

Shared memory

Attaching a segment

Attaching a shared memory segment

The `shmat` system call attaches the shared memory segment identified by `shmid` to the calling process's virtual address space.

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

// Returns address at which shared memory is attached on success
//  or (void *)-1 on error
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

- `shmaddr` `NULL`: the segment is attached at a suitable address selected by the kernel (`shmaddr` and `shmflg` are ignored)
- `shmaddr` not `NULL`:
  the segment is attached at `shmaddr` address (, but if also)
  - `shmflg` `SHM_RND`: shmaddr is rounded down to the nearest multiple of the constant `SHMLBA` (shared memory low boundary address)

Attaching a shared memory segment

Normally, `shmaddr` is `NULL`, for the following reasons:

- It increases the portability of an application. An address valid on one UNIX implementation may be invalid on another.
- An attempt to attach a shared memory segment at a particular address will fail if that address is already in use.

In addition to `SHM_RND`, the flag `SHM_RDONLY` can be specified for attaching a the shared memory for reading only. If `shmflg` has value zero, the shared memory is attached in read and write mode.

A child process inherits its parent's attached shared memory segments. Shared memory provides an easy method of IPC between parent and child!

Attaching a shared memory segment

Example showing how to attach a shared memory segment (twice)[2]

```
// attach the shared memory in read/write mode
int *ptr_1 = (int *)shmat(shmid, NULL, 0);
// attach the shared memory in read only mode
int *ptr_2 = (int *)shmat(shmid, NULL, SHM_RDONLY);
// N.B. ptr_1 and ptr_2 are different!
// But they refer to the same shared memory!
// write 10 integers to shared memory segment
for (int i = 0; i < 10; ++i)
    ptr_1[i] = i;
// read 10 integers from shared memory segment
for (int i = 0; i < 10; ++i)
    printf("integer: %d\n", ptr_2[i]);
```

What will code program print?

Can we use ptr_2 to write in the shared memory segment? Why?

[2] error checking statements were omitted

Detaching a shared memory segment

When a process no longer needs to access a shared memory segment, it can call `shmdt` to detach the segment from its virtual address space. The `shmaddr` argument identifies the segment to be detached, and it is a value returned by a previous call to `shmat`.

```
#include <sys/shm.h>

// Returns 0 on success, or -1 on error
int shmdt(const void *shmaddr);
```

During an `exec`, all attached shared memory segments are detached. Shared memory segments are also automatically detached on process termination.

Detaching a shared memory segment

Example showing how to detach a shared memory segment

```
// attach the shared memory in read/write mode
int *ptr_1 = (int *)shmat(shmid, NULL, 0);
if (ptr_1 == (void *)-1)
    errExit("first shmat failed");
// attach the shared memory in read only mode
int *ptr_2 = (int *)shmat(shmid, NULL, SHM_RDONLY);
if (ptr_2 == (void *)-1)
    errExit("second shmat failed");
//...
// detach the shared memory segments
if (shmdt(ptr_1) == -1 || shmdt(ptr_2) == -1)
    errExit("shmdt failed");
```

Shared memory control operations

The `shmctl` system call performs control operations on a shared memory segment.

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

// Returns 0 on success, or -1 error
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

The `shmid` argument is a shared memory identifier. The `cmd` argument specifies the operation to be performed on the shared memory:

- `IPC_RMID`: Mark for deletion the shared memory. The segment is removed as soon as all processes have detached from it
- `IPC_STAT`: Place a copy of the shmid_ds data structure associated with this shared memory in the buffer pointed to by `buf`
- `IPC_SET`: Update selected fields of the shmid_ds data structure associated with this shared memory using values provided in the

Shared memory control operations - Example

Example showing how to remove a shared memory segment

```
if (shmctl(shmid, IPC_RMID, NULL) == -1)
    errExit("shmctl failed");
else
    printf("shared memory segment removed successfully\n");
```

Shared memory control operations

For each shared memory segment the kernel has an associated `shmid_ds` data structure of the following form:

```
struct shmid_ds {
    struct ipc_perm shm_perm; /* Ownership and permissions */
    size_t shm_segsz;          /* Size of segment in bytes */
    time_t shm_atime;          /* Time of last shmat() */
    time_t shm_dtime;          /* Time of last shmdt() */
    time_t shm_ctime;          /* Time of last change */
    pid_t shm_cpid;            /* PID of creator */
    pid_t shm_lpid;            /* PID of last shmat() / shmdt() */
    shmatt_t shm_nattch;       // Number of currently attached
};                             // processes
```

With `IPC_STAT` and `IPC_SET` we can respectively get and update[3] this data structure.

[3] Only the field shm_perm can be modified

Message Queue

Message Queue

Creating and Opening

Creating/Opening a Message Queue

The `msgget` system call creates a new message queue, or obtains the identifier of an existing queue.

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

// Returns message queue identifier on success, or -1 error
int msgget(key_t key, int msgflg);
```

The `key` argument is an IPC key, `msgflg` is a bit mask specifying the permissions (see `open(...)` system call, `mode` argument) to be places on a new message queue, or checked against an existing queue. In additions, the following flags can be ORed (|) in `msgflg`:

- `IPC_CREAT`: If no message queue with the specified `key` exists, create a new queue
- `IPC_EXCL`: in conjunction with `IPC_CREAT`, it makes `msgget` fail if a queue exists with the specified `key`

Creating/Opening a Message Queue

Example showing how to create a message queue

```
int msqid;
ket_t key = //... (generate a key in some way, i.e. with ftok)

// A) delegate the problem of finding a unique key to the kernel
msqid = msgget(IPC_PRIVATE, S_IRUSR | S_IWUSR);

// B) create a queue with identifier key, if it doesn't already exist
msqid = msgget(key, IPC_CREAT | S_IRUSR | S_IWUSR);

// C) create a queue with identifier key, but fail if it exists already
msqid = msgget(key, IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
```

Message Queue

The message structure

The message in a Message Queue

A message in a message queue always follows the following structure:

```
struct mymsg {
    long mtype;     /* Message type */
    char mtext[];  /* Message body */
};
```

The first part of a message contains the message type, specified as a long integer **greater than 0**. The remainder of the message is a **programmer-defined** structure of arbitrary length and content (it is not necessary an array of char). Indeed, it **can be omitted** if not needed.

Message Queue

Sending a message

Sending Messages

The `msgsnd` system call writes a message to a message queue.

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

// Returns 0 on success, or -1 error
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

- `msqid` argument is a message queue identifier
- `msgp` is an address pointing to a message structure
- `msgsz` specifies the number of bytes contained in the `mtext` field of the message
- `msgflg` argument can be 0, or the flag `IPC_NOWAIT`.
  - Normally, if a message queue is full, `msgsnd` **blocks** until enough space has become available to allow the message to be placed on the queue
  - If `IPC_NOWAIT` is specified, `msgsnd` immediately returns with error `EAGAIN` (*i.e., there is no data available right now, try again later*)

Sending Messages - Example 1

```c
// Message structure
struct mymsg {
    long mtype;
    char mtext[100]; /* array of chars as message body */
} m;
// message has type 1
m.mtype = 1;
// message contains the following string
char *text = "Ciao mondo!";
memcpy(m.mtext, text, strlen(text) + 1); // why +1 here?
// size of m is only the size of its mtext attribute!
size_t mSize = sizeof(struct mymsg) - sizeof(long);
// sending the message in the queue
if (msgsnd(msqid, &m, mSize, 0) == -1)
    errExit("msgsnd failed");
```

Sending Messages - Example 2

```c
// Message structure
struct mymsg {
    long mtype;
    int num1, num2; /* two integers as message body */
} m;
// message has type 2
m.mtype = 2;
// message contains the following numbers
m.num1 = 34;
m.num2 = 43;
// size of m is only the size of its mtext attribute!
size_t mSize = sizeof(struct mymsg) - sizeof(long);
// sending the message in the queue
if (msgsnd(msqid, &m, mSize, 0) == -1)
    errExit("msgsnd failed");
```

Sending Messages - Example 3

```c
// Message structure
struct mymsg {
    long mtype;
    /* The message has not got body. It has just a type!*/
} m;
// message has type 3
m.mtype = 3;
// size of m is only the size of its mtext attribute!
size_t mSize = sizeof(struct mymsg) - sizeof(long); // 0!
// sending the message in the queue
if (msgsnd(msqid, &m, mSize, IPC_NOWAIT) == -1) {
    if (errno == EAGAIN) {
        printf("The queue was full!\n");
    } else {
        errExit("msgsnd failed");
    }
}
```

Message Queue

Receiving a message

Receiving Messages

The msgrcv system call reads **and remove** a message from a message queue.

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

// Returns number of bytes copied into msgp on success, or -1 error
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtype, int msgflg);
```

The msqid argument is a message queue identifier. The maximum space available in the mtext field of the msgp buffer is specified by the argument msgsz.

Receiving Messages

The value in the `msgtype` field selects the message retrieved as follow:

- if **equal to 0**, the first message from the queue is removed and returned to the calling process.
- if **greater than 0**, the first message from the queue having `mtype` equals to `msgtype` is removed and returned to the calling process.
- if **less than 0**, the first message of the lowest `mtype` less than or equal to the absolute value of `msgtype` is removed and returned to the calling process.

Given the message definition: (`mtype`, `char`)

And the following queue:

{(300,'a'); (100,'b'); (200,'c'); (400,'d'); (100,'e')}

A series of `msgrcv` calls with `msgtype=-300` retrieve the messages:

(100,'b'), (100,'e'), (200,'c'), (300,'a')

Receiving Messages

The msgflg argument is a bit mask formed by ORing together zero or more of the following flags:

- IPC_NOWAIT: By default, if no message matching msgtype is in the queue, msgrcv blocks until such a message becomes available. Specifying the IPC_NOWAIT flag causes msgrcv to instead return immediately with the error ENOMSG.
- MSG_NOERROR: By default, if the size of the mtext field of the message exceeds the space available (as defined by the msgsz argument), msgrcv fails. If the MSG_NOERROR flag is specified, then msgrcv instead removes the message from the queue, truncates its mtext field to msgsz bytes, and returns it to the caller.

Receiving Messages - Example 1

```c
// message structure definition
struct mymsg {
    long mtype;
    char mtext[100];  /* array of chars as message body */
} m;

// Get the size of the mtext field.
size_t mSize = sizeof(struct mymsg) - sizeof(long);

// Wait for a message having type equals to 1
if (msgrcv(msqid, &m, mSize, 1, 0) == -1)
    errExit("msgrcv failed");
```

Receiving Messages - Example 2

```c
// message structure definition
struct mymsg {
    long mtype;
    char mtext[100];  /* array of chars as message body */
} m;

// Set an arbitrary size for the size.
size_t mSize = sizeof(char) * 50;

// Wait for a message having type equals to 1, but copy its first 50 bytes only
if (msgrcv(msqid, &m, mSize, 1, MSG_NOERROR) == -1)
    errExit("msgrcv failed");
```

Receiving Messages - Example 3

```c
// Message structure
struct mymsg {
    long mtype;
} m;
// In polling mode, try to get a message every SEC seconds.
while (1) {
    sleep(SEC);
    // Performing a nonblocking msgrcv.
    if (msgrcv(msqid, &m, 0, 3, IPC_NOWAIT) == -1) {
        if (errno == ENOMSG) {
            printf("No message with type 3 in the queue\n");
        } else {
            errExit("msgrcv failed");
        }
    } else {
        printf("I found a message with type 3\n");
    }
}
```

Message Queue

Control operations

Message queue control operations

The msgctl system call performs control operations on the message queue.

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

// Returns 0 on success, or -1 error
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

- msqid is a message queue identifier.
- cmd specifies the operation to be performed on the queue:
  - IPC_RMID: Immediately remove the message queue. All unread messages are lost, and any blocked reader/writer awakened (errno set to EIDRM). For this operation, buf is ignored
  - IPC_STAT: Place a copy of the msqid_ds data structure associated with this message queue in the buffer pointed to by buf
  - IPC_SET: Update selected fields of the msqid_ds data structure associated with this message queue using values provided in the buffer pointed to by buf

Message queue control operations - Example

Example showing how to remove a message queue

```
if (msgctl(msqid, IPC_RMID, NULL) == -1)
    errExit("msgctl failed");
else
    printf("message queue removed successfully\n");
```

Message queue control operations

For each message queue the kernel has an associated `msqid_ds` data structure of the following form:

```
struct msqid_ds {
    struct ipc_perm msg_perm;      /* Ownership and permissions */
    time_t msg_stime;              /* Time of last msgsnd() */
    time_t msg_rtime;              /* Time of last msgrcv() */
    time_t msg_ctime;              /* Time of last change */
    unsigned long __msg_cbytes; /* Number of bytes in queue */
    msgqnum_t msg_qnum;            /* Number of messages in queue */
    msglen_t msg_qbytes;           /* Maximum bytes in queue */
    pid_t msg_lspid;               /* PID of last msgsnd() */
    pid_t msg_lrpid;               /* PID of last msgrcv() */
};
```

With `IPC_STAT` and `IPC_SET` we can respectively get and update[4] this data structure.

[4] Only the fields `msg_perm` and `msg_qbytes` can be modified

Message queue control operations

Example showing how to change upper limit size of a message queue.

```
struct msqid_ds ds;
// Get the data structure of a message queue
if (msgctl(msqid, IPC_STAT, &ds) == -1)
    errExit("msgctl");

// Change the upper limit on the number of bytes in the mtext
//  fields of all messages in the message queue to 1 Kbyte
ds.msg_qbytes = 1024;

// Update associated data structure in kernel
if (msgctl(msqid, IPC_SET, &ds) == -1)
    errExit("msgctl");
```

Conclusive Overview of System V IPC interfaces

Conclusive Overview of System V IPC interfaces

Interface	Message queues	Semaphores	Shared memory
Header file	`<sys/msg.h>`	`<sys/sem.h>`	`<sys/shm.h>`
Data structure	`msqid_ds`	`semid_ds`	`shmid_ds`
Create/Open	`msgget(...)`	`semget(...)`	`shmget(...)`
Close	`(none)`	`(none)`	`shmdt(...)`
Control Oper.	`msgctl(...)`	`semctl(...)`	`shmctl(...)`
Performing IPC	`msgsnd(...)` `msgrcv(...)`	`semop(...)` to test/adjust	access memory in shared region