Lecture #11 – Interprocess Communication II (Chapter 15)

- Message Queues

Message queues are a linked list of messages stored in kernel memory and identified by a message queue identifier.

A new queue is created, or an existing queue is opened with the "msgget" system call.

New messages are added with "msgsnd", and fetched with "msgrcv".

Each message has a positive long integer type field, a length, and the actual data.

It is not necessary that we retrieve the messages in FIFO order; we can also retrieve messages by type if desired.

Each queue has a "msqid_ds" structure associated with it:

```
struct msqid_ds {
        struct ipc_perm     msg_perm;        /*  permissions */
        struct msg          msg_first;       /*  ptr to first message */
        struct msg          msg_last;        /*  ptr to last message */
        ulong               msg_cbytes;      /*  current # of bytes in queue */
        ulong               msg_qnum;        /*  # of messages in queue */
        ulong               msg_qbytes;      /*  max # of bytes in queue */
        pid_t               msg_lspid;       /*  pid of last msgsnd() */
        pid_t               msg_lrpid        /*  pid of last msgrcv() */
        time_t              msg_stime;       /*  last msgsnd() time */
        time_t              msg_rtime;       /*  last msgrcv() time */
        time_t              msg_ctime;       /*  last change time */
};
```

Kernel Parameters for message queues:

MSGMAX	Size in bytes of largest message
MSGMNB	Max size in bytes of a single queue
MSGMNI	Max number of message queues system wide
MSGTQL	Max number of messages system wide

- msgget

  int msgget(key_t key, int flag);

  Returns: Queue ID if OK, -1 on error

  As mentioned, this is the system call to create or attached to a message queue.

  When a new queue is created, the following members of "msqid_ds" are affected:

  ipc_perm is initialized as described earlier
  msg_qnum, msg_lspid, msg_lrpid, msg_stime, and msg_rtime are set to 0.
  msg_ctime is set to current time
  msg_qbytes is set to the system limit

- msgctl

  int msgctl(int msqid, int cmd, struct msqid_ds *buf);

  "cmd" can be one of:

IPC_STAT	Fetch the msqid_ds structure
IPC_SET	Set the following fields from the msqid_ds structure msg.perm.uid, msg_perm.gid, msg_perm.mode, msg_qbytes
IPC_RMID	Remove the message queue from the system and any remaining data. Removal is immediate.

- msgsnd

  int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);

  This function is used to add messages to a queue.

  "msqid" is the ID of the message queue
  "nbytes" is the message length"
  "flag" is normally 0, but can be set to things like IPC_NOWAIT (non-blocking).
  "ptr" points to a long integer that contains the message type, and it followed by data.

  For example, we can define "ptr" as "struct mymesg *ptr" where:

```
struct mymesg {
        long    mtype;
        char    mtext[512];
};
```

- msgrcv

```
int     msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```

"msqid", "ptr", "nbytes", and "flag" work the same way as with msgsnd

"type" allows us to specify which message we would like to retrieve:

type == 0	Return the first message on the queue
type > 0	Return the first message of type "type"
type < 0	Return the first message whose type is the lowest value <= |type|

- Example: Message Queues

```
/*      kirk.c -- writes to a message queue             */

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

struct my_msgbuf {
        long mtype;
        char mtext[200];
};

int main(void)
{
        struct my_msgbuf        buf;
        int                     msqid;
        key_t                   key;

        if ((key = ftok("kirk.c", 'B')) == -1) {
                perror("ftok");
                exit(1);
        }

        if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1) {
                perror("msgget");
                exit(1);
        }

        printf("Enter lines of text, ^D to quit:\n");
        buf.mtype = 1; /* we don't really care in this case */

        while(gets(buf.mtext), !feof(stdin)) {
                if (msgsnd(msqid, (struct msgbuf *)&buf, sizeof(buf), 0) == -1)
                        perror("msgsnd");
        }

        if (msgctl(msqid, IPC_RMID, NULL) == -1) {
                perror("msgctl");
                exit(1);
        }

        return 0;
}

/*              spock.c -- reads from a message queue           */

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

struct my_msgbuf {
        long mtype;
        char mtext[200];
};

int main(void)
{
        struct my_msgbuf        buf;
        int                     msqid;
        key_t                   key;

        if ((key = ftok("kirk.c", 'B')) == -1) {  /* same key as kirk.c */
                perror("ftok");
                exit(1);
        }

        if ((msqid = msgget(key, 0644)) == -1) { /* connect to the queue */
                perror("msgget");
                exit(1);
        }
        printf("spock: ready to receive messages, captain.\n");

        for(;;) { /* Spock never quits! */
                if (msgrcv(msqid, (struct msgbuf *)&buf, sizeof(buf), 0, 0) == -1) {
                        perror("msgrcv");
                        exit(1);
                }

                printf("spock: \"%s\"\n", buf.mtext);
        }

        return 0;
}
```

- Semaphores

Semaphores are really not a form of IPC, but a control mechanism to provide safe access to shared data resources.

To obtain a shared resource:

1. Test the semaphore that controls the resource.
2. If the semaphore is positive, the process can use the resource. The process decrements the semaphore by 1, indicating it is using the resource.
3. If the value of the semaphore is 0, the process sleeps until the semaphore becomes positive. The process then wakes up, and goes to step 1.

When a process finishes using a resource, the semaphore value is incremented by 1. If any other processes are sleeping for the semaphore, they are awakened.

For semaphores to work correctly, the "test" and the "set" portions must be implemented atomically, so this is often done in the kernel.

System V semaphores are implemented as sets of one or more semaphores.

There is a "semid_ds" structure for each semaphore:

```
struct semid_ds {
        struct ipc_perm     sem_perm;       /*  permissions */
        struct sem          sem_base;       /*  ptr to first semaphore in set */
        ushort              sem_nsems;      /*  # of semaphores in set */
        time_t              sem_otime;      /*  last semop() time */
        time_t              sem_ctime;      /*  last change time */
};
```

```
struct sem {
        ushort      semval;        /*  semaphore value >= 0 */
        pid_t       sempid;        /*  pid for last operation */
        ushort      semncnt;       /*  # processes awaiting semval > curr val */
        ushort      semzcnt;       /*  # processes awaiting semval = 0 */
};
```

Kernel parameters affecting semaphores:

SEMVMX	Max value of any semaphore
SEMAEM	Max value of semaphores adjust-on-exit value
SEMMNI	Max number of semaphore sets
SEMMNS	Max number of semaphores
SEMMSL	Max number of semaphores per set
SEMMNU	Max number of undo structures
SEMUME	Max number of undo entries per undo structure
SEMOPM	Max number of operations per semop call

- semget

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

Returns: semaphore set ID if OK, -1 on error

"nsems" is the number of semaphores in the set.

On creation, the following fields of "semid_ds" are affected:

ipc_perm is initialized
sem_otime is set to 0
sem_ctime is set to current time
sem_nsems is set to "nsems"

- semctl

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

"semctl" is the catchall function for many different semaphore operations.
"semid" is the semaphore set to operate on
"semnum" is the semaphore within the set to operate on

```
union semun {
        int                 val;           /*  for SETVAL */
        struct semid_ds     *buf;          /* for IPC_STAT and IPC_SET */
        ushort          array;      /*  for GETALL and SETALL */
};
```

"cmd" can be one of the following:

IPC_STAT	Fetch the semid_ds structure
IPC_SET	Set the sem_perm.uid, sem_perm.gid, sem_perm.mode
IPC_RMID	Remove the semaphore set
GETVAL	Return the current value of semval for the member semnum
SETVAL	Set the value of semval for the member semnum
GETPID	Return the value of sempid for the member semnum
GETNCNT	Return the value of semncnt for the member semnum
GETZCNT	Return the value of semzcnt for the member semnum
GETALL	Fetch all the semaphores values in the set
SETALL	Set all the semaphore values in the set

- semop

```
int     semop(int semid, struct sembuf semoparray[], size_t nops);
```

The semop function atomically performs an array of operations on a semaphore set.

"semoparray" is a pointer to an array of semaphore operations:

```
struct sembuf {
        ushort      sem_num;    /*  member number in set */
        short       sem_op;     /*  operation (negative, 0, or positive) */
        short       sem_flg;    /*  IPC_NOWAIT, SEM_UNDO */
};
```

"nops" specifies the number of operations in the array

A "sem_op" value that is positive is releasing resources (i.e. signal), and a value that is negative is requesting resources (i.e. wait).

- Semaphore examples

Example: Set creation

```
int             semset;     /*  Value of the semaphore set */
union semun   semval;       /*  Temporary semaphore structure */
int             status;     /*  Return status */

/*  Create a new semaphore set with 4  semaphores */

if ((semset = semget(IPC_PRIVATE, 4, IPC_CREAT | 0600)) < 0)
{
        perror("create_sem: semget");
        exit(1);
}

/*  Initialize the value of the 0th semaphore to 1 */

semval.val = 1;

if ((status = semctl(semset, 0, SETVAL, semval)) < 0)
{
        perror("create_sem: semctl 0");
        return -1;
}
```

Example: Implementation of "wait_sem"

```
void wait_sem(semset, semnum)
int     semset;                 /*  Value of the semaphore set */
int     semnum;                 /*  Value of the semaphore number */
{
    struct sembuf   sem;        /*  Temp semaphore structure */

    /*  Setup semaphore call to decrease value by 1 */

    sem.sem_num = semnum;
    sem.sem_op = -1;
    sem.sem_flg = 0;

    /*  Perform atomic test and decrease */

    while (semop(semset, &sem, 1) < 0);
}
```

Example: Implementation of "signal_sem"

```
void signal_sem(semset, semnum)
int     semset;                 /*  Value of the semaphore set */
int     semnum;                 /*  Value of the semaphore number */
{
     struct sembuf   sem;       /*  Temp semaphore structure */

     /*  Setup semaphore call to increase value by 1 */
     sem.sem_num = semnum;
     sem.sem_op = 1;
     sem.sem_flg = 0;

     while (semop(semset, &sem, 1) < 0);
}
```

- Shared Memory

Shared memory allows two or more processes to share a given region of memory. This is the fastest form of IPC because the data does not need to be copied between the processes.

Since data is not copied, we must take care in synchronizing access to the data (i.e. with the use of semaphores).

Kernel parameters for shared memory:

SHMMAX	Max size in bytes of a shared memory segment
SHMMIN	Min size in bytes of a shared memory segment
SHMMNI	Max number of shared memory segments
SHMSEG	Max number of shared memory segments per process

```
struct shmid_ds {
        struct ipc_perm     shm_perm;        /*  permission */
        struct anon_map     shm_amp;         /*  ptr in kernel */
        int                 shm_seqgz;       /*  size of segment in bytes */
        ushort              shm_lkcnt;       /*  number of times seg is locked */
        pid_t               shm_lpid;        /*  pid of last shmctl */
        pid_t               shm_cpid;        /*  pid of creator */
        ulong               shm_nattch;      /*  number of current attaches */
        ulong               shm_cnattch;     /*  used for shminfo */
        time_t              shm_atime;       /*  last attach time */
        time_t              shm_dtime;       /*  last detach time */
        time_t              shm_ctime;       /*  last change time */
};
```

- shmget

```
int     shmget(key_t key, int size, int flag);
```

Returns: shared memory segment ID if OK, -1 on error

On creation, the following fields are affected:

> ipc_perm is initialized
> shm_lpid, shm_nattach, shm_atime, shm_dtime are set to 0
> shm_ctime is set to the current time

"size" is the minimum size of the shared memory segment

- shmctl

```
int     shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

"cmd" is one of the following:

IPC_STAT	Fetch the shmid_ds structure
IPC_SET	Set shm_perm.uid, shm_perm.gid, shm_perm.mode
IPC_RMID	Remove the shared memory segment (does not occur until the last process using the segment terminates or detaches).
SHM_LOCK	lock the shared memory segment in real memory
SHM_UNLOCK	unlock the shared memory segment

- shmat

```
void    *shmat(int shmid, void *addr, int flag);
```

After creation of the segment, the process must call "shmat" to attach to the segment.

If "addr" is 0, the segment is attached at the first available address selected by kernel.
If "addr" is != 0, and SHM_RND is not specified, the segment is attached at the "addr".
If "addr" is != 0, and SHM_RND is specifed, the segment is attached at an addr multiple.

- shmdt

```
int     shmdt(void *addr);
```

Once we are finished with the segment, we should detach from it with "shmdt"

"addr" is the value of the address returned by "shmat"

- Shared memory examples

Example: Shared memory creation

```
struct buf      buf;        /*  Our shared memory structure */
int             shmid;      /*  Shared memory identifier */

/*  Create a private shared memory segment */

if ((shmid = shmget(IPC_PRIVATE, sizeof(struct buf), IPC_CREAT | 0600)) < 0)
{
        perror("create_shm: shmget");
        return -1;
}

/*  Attach ourselves to the segment */

shmaddr = (struct buf *) shmat(shmid, NULL, 0);

if (!shmaddr)
{
        perror("create_shm: shmat");
        return -1;
}
```

Example: Shared memory removal

```
if (shmdt(shmaddr) < 0)
{
        perror("remove_shm: shmdt");
        return FALSE;
}

 /*  Remove the segment */

if (shmctl(shmid, IPC_RMID, NULL) < 0)
{
        perror("remove_shm: shmctl");
        return FALSE;
}
```

- Combination example (semaphores and shared memory)

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

/*  Global data structures */

#define EMPTY       0
#define FULL 1

union semun {
        int                     val;
        struct semid_ds         *buf;
        unsigned short int      *array;
        struct seminfo          *__buf;
};

struct buf {
            char     msg[128];
            int      flag;
};

struct buf      *shmaddr;       /*  Our shared memory structure */
int     shmid;          /*  Shared memory identifier */

int             semset;         /*  Value of the semaphore set */
union semun     semval;                 /*  Temporary semaphore structure */

int             status;         /*  Return status */

/*  WAIT_SEM  */

void wait_sem(semset, semnum)
int     semset;         /*  Value of the semaphore set */
int     semnum;                 /*  Value of the semaphore number */
{
        struct sembuf   sem;        /*  Temp semaphore structure */

        /*  Setup semaphore call to decrease value by 1 */

        sem.sem_num = semnum;
        sem.sem_op = -1;
        sem.sem_flg = 0;
        /*  Perform atomic test and decrease */

        while (semop(semset, &sem, 1) < 0);
}

/*  SIGNAL_SEM  */

void signal_sem(semset, semnum)
int         semset;             /*  Value of the semaphore set */
int         semnum;                     /*  Value of the semaphore number */
{
        struct sembuf   sem;        /*  Temp semaphore structure */

        /*  Setup semaphore call to increase value by 1 */

        sem.sem_num = semnum;
        sem.sem_op = 1;
        sem.sem_flg = 0;

        while (semop(semset, &sem, 1) < 0);
}

/*  CHILD_CODE  */

int child_code()
{
            int         i;

            for (i = 0; i < 10; i++)
            {
                        wait_sem(semset, FULL);

                        printf("%s\n", shmaddr->msg);

                        signal_sem(semset, EMPTY);
            }

            /*  Detach from shm segment */

            if (shmdt(shmaddr) < 0)
            {
                        perror("remove_shm: shmdt");
                        return(1);
            }
            /*  Remove the segment */

            if (shmctl(shmid, IPC_RMID, NULL) < 0)
            {
                        perror("remove_shm: shmctl");
                        return(1);
            }

            /*  Remove the semaphore set */

            if (semctl(semset, 0, IPC_RMID, NULL) < 0)
            {
                        perror("remove_sem: semctl");
                        return(1);
            }

            return(0);
}

/*  PARENT_CODE  */

int parent_code()
{
            int     i;

            for (i = 0; i < 10; i++)
            {
                        wait_sem(semset, EMPTY);

                        sprintf(shmaddr->msg, "This is number %d", i);

                        signal_sem(semset, FULL);
            }

            /*  Detach from shm segment */

            if (shmdt(shmaddr) < 0)
            {
                        perror("remove_shm: shmdt");
                        return(1);
            }

            return(0);
}

/*  MAIN  */
```

```
int main(int argc, char **argv)
{
        pid_t   pid;

        /*  Create a new semaphore set with 2 semaphores */

        if ((semset = semget(IPC_PRIVATE, 2, IPC_CREAT | 0600)) < 0)
        {
                perror("create_sem: semget");
                return(1);
        }

        /*  Initialize the value of EMPTY semaphore to 1 */

        semval.val = 1;

        if ((status = semctl(semset, EMPTY, SETVAL, semval)) < 0)
        {
                perror("create_sem: semctl EMPTY");
                return(1);
        }

        /*  Initialize the value of FULL semaphore to 0 */

        semval.val = 0;

        if ((status = semctl(semset, FULL, SETVAL, semval)) < 0)
        {
                perror("create_sem: semctl FULL");
                return(1);
        }

        /*  Create a private shared memory segment */

        if ((shmid = shmget(IPC_PRIVATE, sizeof(struct buf),
                IPC_CREAT | 0600)) < 0)
        {
                perror("create_shm: shmget");
                return -1;
        }

        /*  Attach ourselves to the segment */

        shmaddr = (struct buf *) shmat(shmid, NULL, 0);
```

```
	if (!shmaddr)
	{
		perror("create_shm: shmat");
		return(1);
	}

	if ((pid = fork()) < 0)
	{
		perror("fork");
		return(1);
	}
	else if (pid == 0)
		child_code();
	else
		parent_code();

	return(0);
}
```