Message Queues,
Semaphores,
Shared Memory

Message Queues

- Basic idea of a message queue

1. Two processes can exchange information via access to a common system message queue.

2. A process places a message onto the queue which can be accessed and read by another process.

3. Message queue has explicit length, has an identification(so called "type").

4. Blocking message passing is performed in the way that the sender wait for the receiver's acknowledge of receiving a message, so that it can continue sending message again.

5. Non-blocking message passing means that it is not necessary for sender to

wait for the receiving acknowledge, it can continue sending message.

- Programming interface for message queue

  1. Initializing a message queue

     (a) Function msgget() creates and accesses a message queue

     ```
     #include <sys/msg.h>
     #include <sys/ipc.h>
     int msgget(key_t key, int msgflg);
     ```

     Function msgget() returns a postive integer, as message queue identifier.

     (b) Parameters:

     Key: an arbitrary integer number;

     msgflg: the permission mode and creation control flag.

     (c) Example

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

#include <unistd.h>
#include <errno.h>
#include <stdio.h>

int main()
{
/*key to be passed to msgget()*/
key_t key;
/*msgflg to be passed to msgget()*/
int msgflg;
/*return value from msgget()*/
int msgid;

key = 1234;
/* read and write permission for owner,
and create a message queue if not exists*

msgflg = 0666 | IPC_CREAT;
if ((msgid = msgget(key, msgflg))== 1)
{
  perror("msgget: msgget failed");
  exit(1);
 }
 else{
  printf("msgget succeeded\n");
  exit(0);
 }
}
```

2. Controlling a message queue

   (a) Function msgctl() alters the permissions and other characteristics of a message queue.

   ```
   int msgctl(int msgid, int cmd,
                    struct msgid_ds *buf )
   ```

   It returns 0 on success, -1 for failure.

   (b) Parameters:

   msgid: message identifier, which is the return value from msgget().

```
struct msgid_ds {
uid_t msg_perm.uid;
uid_t msg_perm.gid;
mode_t msg_perm.mode;
}
```

cmd: the action to take, can be one of the following settings:

IPC-STAT:Place information about the status of the queue in the data structure pointed to by buf. The process must have read permission for this call to succeed.

IPC-SET: Set the owner's user and group ID, the permissions, and the size (in number of bytes) of the message queue. A process must have the effective user ID of the owner, creator, or superuser for this call to succeed.

IPC-RMID: Remove the message queue.

(c) Example

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

#include <unistd.h>
#include <errno.h>
#include <stdio.h>

int main()
{
/*key to be passed to msgget()*/
key_t key;
/*msgflg to be passed to msgget()*/
int msgflg;
/*return value from msgget()*/
int msgid;

key = 1234;
/* read and write permission for owner,
and create a message queue if not exists*

msgflg = 0666 | IPC_CREAT;
```

```
 if((msgid = msgget(key, msgflg))== 1){
   perror("msgget: msgget failed");
   exit(1);
 }else{
   printf("msgget succeeded\n");
   if(msgctl(msgid, IPC_RMID, 0) == -1){
     perror("msgctl: msgctl failed");
     exit(1);
   }else{
     printf("msgctl succeeded\n");
     exit(0);
   }
 }
}
```

3. Sending and Receiving a message

   (a) The msgsnd() and msgrcv() functions send and receive messages, respectively:

```
int msgsnd(int msgid, const void *msgp,
             size_t msgsz,   int msgflg);
```

```
int msgrcv(int msgid, void *msgp,
           size_t msgsz, long msgpriori,
                          int msgflg);
```

(b) Parameters:

msgp: a pointer to a structure that contains the type of the message and its text.

The structure below is an example of what this user-defined buffer might look like:

```
struct mymsg {
 /* message type */
    long  msgtype;
 /* message text of length MSGSZ */
    char mtext[MSGSZ];
}
```

The structure member msgtype is used in message reception, must be initialized (into a integer value) by the sending process.

msgsz: specifies the length of the message in bytes.

msgpriori: reception priority. Set 0 means you simply want to retrieve the message in the order in which they were sent. Or some other values must be equal to the specified message type.

msgflg: for msgsnd(), controls what happens if either the queue is full, or reaches the system limit on queued messages. Set as 0 means process suspends and wait for the space become available.

msgflg: for msgrcv(), controls what happends if no message of the proper priority is waiting to be received. Set as 0 means process suspends and wait for an message with proper priority to arrive.

- Examples

msgSend.c

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

#include <unistd.h>
#include <errno.h>
#include <stdio.h>

#define MAX_TEXT 512

/*define a message structure*/
struct my_msg{
  long msgtype;
  char message[MAX_TEXT];
};

int main()
{
/*key to be passed to msgget()*/
key_t key;
/*msgflg to be passed to msgget()*/
```

```
int msgflg;
/*return value from msgget()*/
int msgid;
/*declare an instance of structure of msg*/
struct my_msg sometext;
/* declare a buffer for text msg*/
char buffer[BUFSIZ];
/*initialize the key*/
key = 1234;
/* read and write permission for owner,
and create a message queue if not exists*/
msgflg = 0666 | IPC_CREAT;

/*creat a message queue */
 if((msgid = msgget(key, msgflg))== 1){
   perror("msgget: msgget failed");
   exit(1);
 }
 printf("msgget succeeded\n");
 while(1){
   /*read user input from the keyboard */
   printf("Please ener some text:\n");
   fgets(buffer, BUFSIZ, stdin);
```

```
    /*set the msgtype */
    sometext.msgtype = 1;
    /*copy from buffer to message*/
    strcpy(sometext.message, buffer);
    /*send the message */
    if (msgsnd(msgid, &sometext, MAX_TEXT, 0) =
      perror("msgsnd: msgsnd failed\n");
      exit(1);
    }
    if(strncmp(buffer, "end", 3)== 0){
      printf("message ends.\n");
      exit(0);
    }
  }
}
```

msgRcv.c

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

#include <unistd.h>
```

```
#include <errno.h>
#include <stdio.h>

/*define a message structure*/
struct my_msg{
  long msgtype;
  char message[BUFSIZ];
};

int main()
{
/*key to be passed to msgget()*/
key_t key;
/*msgflg to be passed to msgget()*/
int msgflg;
/*return value from msgget()*/
int msgid;
/*declare an instance of structure of msg*/
struct my_msg sometext;

/*initialize the key*/
key = 1234;
/* read and write permission for owner,*
```

```
/*and create a message queue if not exists*/
msgflg = 0666 | IPC_CREAT;

/*initialize the priority of message*/
long int msgpriori = 0;
/*creat a message queue */
 if((msgid = msgget(key, msgflg))== 1){
   perror("msgget: msgget failed");
   exit(1);
 }
 printf("msgget succeeded\n");
 while(1){
   /*receive the message */
   if (msgrcv(msgid, &sometext, BUFSIZ,
                   msg_receive, 0) == -1){
     perror("msgrcv: msgrcv failed\n");
     exit(1);
   }
   printf("You wrote: %s", sometext.message);
   if(strncmp(sometext.message,"end",3)==0){
     printf("message ends.\n");
     /*delete the message queue */
     if(msgctl(msgid, IPC_RMID, 0) == -1){
```

```
     perror("msgctl: msgctl failed");
     exit(1);
     }
     printf("program succeeded\n");
     exit(0);
   }
 }
}
```

Semaphores

- Introduction to semaphores

  1. Memo Semaphores are a programming construct designed by E. W. Dijkstra in the late 1960s.

  2. Basic idea

     Dijkstra's model was the operation of railroads:

     (a) A single railway track, only one train at a time is allowed.

     (b) Guarding this track is a semaphore.

     (c) A train must wait before entering the single track until the semaphore is in a state that permits travel.

     (d) When the train enters the track, the semaphore changes state to prevent other trains from entering the track.

(e) A train that is leaving this section of track must again change the state of the semaphore to allow another train to enter.

3. Semaphores synchronizing processes

(a) sv: value of semaphore

(b) A process waits for permission to proceed(waiting for value of semaphore to be greater than 0).

(c) When it finishes its job, the process changes the semaphore's value by subtracting one, so that the value of integer back to 0.

P(sv): If $sv > 0$, decrement sv by 1; if sv $=$ 0 suspend to wait

(d) It let suspended processes to resume execution or increment sv by 1 telling the critical section is available.

V(sv): If some other process has been waiting for sv, then resume execution; If no process is suspended waiting for sv, increment sv.

- Programming interface of Semaphores

  1. Creating a new semaphores

     (a) The function semget() initializes or gains access to a semaphore. It is prototyped by:

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

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

     When the call succeeds, it returns the semaphore ID (semid).

     (b) Parameters:

key: is a access value associated with the semaphore ID.

nsems: the number of semaphores required(since UNIX provides an array of individual semaphores, to give control of multiple resources). Set 1, means create one semaphore.

semflg: specifies the initial access permissions and creation control flags.

(c) Example

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

#include <unistd.h>
#include <errno.h>
#include <stdio.h>

key_t key; /*key to pass to semget()*/
int semflg; /*semflg to pass tosemget()*/
int nsems; /*nsems to pass to semget()*/
int semid; /*return value from semget()*/

int main()
{
 key = 1234;
 nsems = 1;
 semflg = 0666 | IPC_CREAT;
 if((semid = semget(key, nsems, semflg))
     ==-1){
          perror("semget: semget failed");
          exit(1);
 }
 printf(‘‘semget succeed\n’’);
 exit(0);
}
```

2. Controlling semaphores

   (a) Function semctl() changes permissions and other characteristics of a semaphore set. It's prototyp is as followed:

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

It returns different values depending on the cmd. For setting the value of a single semaphore or remove the specified semaphore set, it returns 0 on success, -1 for failure.

(b) Parameters:

It must be called with a valid semaphore ID, semid.

The semnum value selects a semaphore within an array by its index.

The union semun is optional, depending upon the operation requested. If required it is of type union semun, which must be explicitly declared by the application program. “sem-union” is a member of this union.

The cmd argument is one of the following control flags:

SETVAL: Set the value of a single semaphore.

IPC-RMID: Remove the specified semaphores.

(c) Examples

i. Define a union semun

```
union semun{
  int val;
  struct semid_ds *buf;
  ushort *array;
}sem_union;
```

ii. Set a semaphore value

```
static void set_semvalue
{
  int i;
  int semnum = 0;
  int cmd = SETVAL;

  sem_union.val = 1;
  i= semctl(semid,semnum,cmd,sem_union);
```

```
    if (i == -1){
      perror("semctl: semctl failed");
    }
  }
```

iii. Delete a semaphore

```
  static void del_semvalue
  {
    int i;
    int semnum = 0;
    int cmd = IPC_RMID;

    i= semctl(semid,semnum,cmd,sem_union);
    if (i == -1){
      perror("semctl: semctl failed");
    }
  }
```

3. Semaphores operation

(a) Function semop() performs operations on a semaphore set. It is prototyped by:

```
int semop(int semid,struct sembuf *sops,
                          size_t nsops);
```

(b) Parameters:

semid: the semaphore ID returned by a previous semget() call.

sops: is a pointer to an array of structures, each containing the following information about a semaphore operation: The semaphore number The operation to be performed Control flags, normally set as SEM-UNDO to track on the semaphore changes made by current process.

The sembuf structure specifies a semaphore operation, as defined in $< sys/sem.h >$.

```
struct sembuf {
 ushort_t sem_num;/* semaphore number */
 short sem_op; /* semaphore operation */
 short sem_flg;/* operation flags */
};
```

nsops: number of semaphores in the array.

(c) Examples

i. waiting P(sv)

```
static int semaphore_p(void)
{
  struct sembuf sem_b;

  sem_b.sem_num = 0;
  sem_b.sem_op =-1;/*P(sv)*/
  sem_b.sem_flg = SEM_UNDO;

  if(semop(semid, &sem_b,1)== -1){
    perror("semop: semop failed");
    return(0);
  }
  return(1);
}
```

ii. giving controll V(sv)

```
static int semaphore_v(void)
```

```
{
  struct sembuf sem_b;

  sem_b.sem_num = 0;
  sem_b.sem_op = 1; /*V(sv)*/
  sem_b.sem_flg = SEM_UNDO;

  if(semop(semid, &sem_b, 1)== -1){
    perror("semop: semop failed");
    return(0);
  }
  return(1);
}
```

4. Application

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

#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#define COUNTER_FILE "counter.txt"

union semun{
  int val;
  struct semid_ds *buf;
  ushort *array;
}sem_union;

static int set_semvalue(void);
static void del_semvalue(void);
static int semaphore_p(void);
static int semaphore_v(void);

key_t key; /*key to pass to semget()*/
int semflg; /*semflg to pass tosemget()*/
int nsems; /*nsems to pass to semget()*/
int semid; /*return value from semget()*/

int read_counter(char filename[]) {
  FILE *fp;
  int counter;
  if ((fp=fopen(filename,"r"))==NULL) {
    perror(filename);
    exit(1);
  }
  if(fscanf(fp,"%d",&counter)!=1) {
    printf("integer not found.\n");
    exit(1);
  }
  if(fclose(fp)!=0) {
    perror(filename);
    exit(1);
  }
  return counter;
}
int write_counter(char filename[],
                      int counter)
{
  FILE *fp;
  if ((fp=fopen(filename,"w"))==NULL) {
    perror(filename);
    exit(1);
  }
  if(fprintf(fp,"%d\n",counter)<0) {
    printf("integer not written.\n");
    exit(1);
  }
  if(fclose(fp)!=0) {
    perror(filename);
    exit(1);
  }
  return 0;
}
int main(int argc, char *argv[])
{
  int counter;
  int loop_nr, i;
  int pause_time;

  key = 1234;
  nsems = 1;
  semflg = 0666 | IPC_CREAT;

  if((semid = semget(key, nsems, semflg))
      ==-1){
          perror("semget: semget failed");
          exit(1);
```

```
 }
 if (argc==1 || sscanf(argv[1],"%d",
                      &loop_nr)!=1){
 printf("usage:%s num_of_loop\n",argv[0]);
 exit(1);
}
 if(!set_semvalue()){
   perror("semctl: semctl failed");
   exit(1);
 }
 printf("Looping %d times\n", loop_nr);

 for (i=0; i<loop_nr; i++) {
   if(!semaphore_p()) exit(EXIT_FAILURE);
   /*start of critical section*/
   counter=read_counter(COUNTER_FILE);
   printf("read counter: %d\n", counter);

   counter++;

   write_counter(COUNTER_FILE,counter);
   printf("write counter: %d\n", counter);
```

```
    pause_time= rand() %3;
    usleep(pause_time);

    /*endof critical section*/
    if(!semaphore_v()) exit(EXIT_FAILURE);
    pause_time= rand()%2;
    usleep(pause_time);
 }
 printf("%d finished\n", getpid());
 del_semvalue();
 exit(EXIT_SUCCESS);
}

static int set_semvalue(void)
{
  int i;
  int semnum = 0; /*semaphore number*/
  int cmd = SETVAL;

  sem_union.val = 1;
  i= semctl(semid,semnum,cmd,sem_union);
  if (i == -1) return(0);
  return(1);
```

```
}

static void del_semvalue(void)
{
  int i;
  int semnum = 0; /*semaphore numeber*/
  int cmd = IPC_RMID;

  i= semctl(semid,semnum,cmd,sem_union);
  if (i == -1){
    perror("semctl: semctl failed");
  }
}

static int semaphore_p(void)
{
  struct sembuf sem_b;

  sem_b.sem_num = 0;
  sem_b.sem_op =-1;/*P(sv)*/
  sem_b.sem_flg = SEM_UNDO;

  if(semop(semid, &sem_b,1)== -1){
    perror("semop: semop failed");
    return(0);
  }
  return(1);
}

static int semaphore_v(void)
{
  struct sembuf sem_b;

  sem_b.sem_num = 0;
  sem_b.sem_op = 1; /*V(sv)*/
  sem_b.sem_flg = SEM_UNDO;

  if(semop(semid, &sem_b, 1)== -1){
    perror("semop: semop failed");
    return(0);
  }
  return(1);
}
```

Shared Memory

- Overview

  1. Shared memory is an efficient way of transferring data between two running processes.

  2. It lets multiple processes attach to a segment of physical memory to their virtual address spaces.

  3. If one process writes to a shared memory, the changes immediately become visible to any other processes that has access to the same shared memory.

  4. The shared memory doesn't provide synchronization of accessing, we may use semaphore to prevent inconsistencies and collisions.

- Programming interface of Shared Memory

1. Accessing a segment of memory shmget() is used to obtain access to a shared memory segment. It is prottyped by:

   ```
   int shmget(key_t key,size_t size,int shmflg
   ```

   Parameters:

   key: an access value associated with the semaphore ID.
   size: the size (in bytes) of the requested shared memory.
   shmflg: specifies the initial access permissions and creation control flags.

   A successful function call should return the shared memory segment ID, otherwise, returns -1.

2. Controlling a shared memory shmctl() is used to alter the permissions and other characteristics of a shared memory segment. It is prototyped as follows:

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

The process must have an effective shmid of owner, creator or superuser to perform this command.

Parameters:

cmd argument is one of following control commands:

```
SHM_LOCK
-- Lock the specified shared memory segment
SHM_UNLOCK
-- Unlock the shared memory segment.
IPC_STAT
-- Return the status information contained
   in the control structure and place it in
   the buffer pointed to by buf.
IPC_SET
-- Set the effective user and group
   identification and access permissions.
IPC_RMID
-- Remove the shared memory segment.
```

buf is a pointer to a type structure,

```
struct shmid_ds
```

which is defined in $< sys/shm.h >$

3. Attaching and detaching a shared memory

   shmat() and shmdt() are used to attach and detach shared memory segments. They are prototypes as follows:

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

int shmdt(const void *shmaddr);
```

   shmat() returns a pointer, shmaddr, to the head of the shared segment associated with a valid shmid.

   shmdt() detaches the shared memory segment located at the address indicated by shmaddr

- Application

  1. `shared_stuff.h`

     – define a structure contains the contents of shared memory. A flag

     ```
     written_by_you
     ```

     is used to tell the client whether the data is available.

     ```
     # define SHMSZ  1024
     struct shared_stuff_st{
       int written_by_you;
       char some_text[SHMSZ];
     }
     ```

  2. server.c – allows us to put in some text and create a shared memory portion.

     ```
     #include <stdio.h>
     #include <stdlib.h>
     #include <string.h>

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

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

#include "shared_stuff.h"

main()
{
  int running =1;
  void *shm = NULL;
  struct shared_stuff_st *memo;
  char buffer[SHMSZ];

  int shmid;
  key_t key;

/* We'll name our shared memory
 * segment "1234" */

  key = 1234;

  /*Create the segment.*/
  if ((shmid = shmget(key, SHMSZ,
         IPC_CREAT | 0666)) == -1){
```

```
    perror("shmget");
    exit(EXIT_FAILURE);
  }

/* Now we attach the segment to
 *  our data space.*/
  if ((shm = shmat(shmid, NULL, 0))
                  == (void*) -1) {
    perror("shmat");
    exit(EXIT_FAILURE);
  }
  printf("memorey attached at %X\n",
                          (int)shm);

/* assign the shared memory segment
 * to memo */
  memo = (struct shared_stuff_st*) shm;

  while(running){
    while(memo -> written_by_you == 0){
      sleep(1);
      printf("waiting for client...\n");
    }
    /* Now put some things into the memory
     * for the other process to read.*/
    printf("Enter some text: ");
    fgets(buffer, BUFSIZ, stdin);

    /* copy from the buffer to the
     * shared memo*/
    strncpy(memo->some_text,buffer,SHMSZ);
    /* set the flag telling client
     * that the data is available* /
    memo -> written_by_you == 1;

    /* if user type in "end", then
     * stop the execution */
    if(strncmp(buffer, "end", 3)== 0){
      running =0;
    }
  }
  /* Finally, we detach the shared memory s
  if(shmdt(shm)== -1){
    perror("shmdt");
    exit(EXIT_FAILURE);
  }
  exit(EXIT_SUCCESS);
}
```

3. client.c – attaches itself to the created shared memory portion and reads the contents.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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

#include "shared_stuff.h"

main()
{
  int running = 1;
  void *shm = NULL;
  struct shared_stuff_st *memo;
  int shmid;
  key_t key;
```

```
/* We'll name our shared memory
 * segment "1234" */

  key = 1234;

  /*Create the segment.*/
  if ((shmid = shmget(key, SHMSZ,
         IPC_CREAT | 0666)) == -1){
    perror("shmget");
    exit(EXIT_FAILURE);
  }

 /* Now we attach the segment to
  * our data space.*/
  if ((shm = shmat(shmid, NULL, 0))
                    == (void*) -1) {
    perror("shmat");
    exit(EXIT_FAILURE);
  }
  printf("memorey attached at %X\n",
                        (int) shm);
```

```
/*assign the shared memory segment to memo*
   memo = (struct shared_stuff_st*) shm;

/*initinalize the flag */
   memo -> written_by_you = 0;

/* Now read what the server put in */
   while(running){
     if(memo -> written_by_you){
       printf("you wrote: %s",
                    memo->some_text);
       sleep(1);
     /* clear the flag after read */
       memo -> written_by_you = 0;

      if(strncmp( memo -> some_text,
                          "end", 3)== 0){
 running =0;
       }
     }
   }
/* we detach the shared memory segment*/
  if(shmdt(shm)== -1){
    perror("shmdt");
    exit(EXIT_FAILURE);
  }
/*Finally remove the shared memory segment*/

  if(shmctl(shmid, IPC_RMID, 0)== -1){
    perror("shmctl");
    exit(EXIT_FAILURE);
  }
  exit(EXIT_SUCCESS);
}
```

THE END