15. POSIX IPC

SOA, Deemed to be University
ITER, Bhubanewar

Book(s)

Text Book(s)

Kay A. Robbins, & Steve Robbins

Unix™ Systems Programming

Communications, concurrency, and Treads

Pearson Education

Reference Book(s)

Brain W. Kernighan, & Rob Pike

The Unix Programming Environment

PHI

Introduction

- The POSIX:XSI Extension standardize the classical UNIX interprocess communication (IPC) mechanisms **shared memory**, **message queues**, and **semaphore sets** respectively.
- These mechanisms allow unrelated processes to exchange information in a reasonably efficient way, use a key to identify, create or access the corresponding entity.
- The entities may persist in the system beyond the lifetime of the process that creates them.
- POSIX:XSI also provides shell commands, (`ipcs`) to list and remove them.

POSIX:XSI Interprocess Communication

- The POSIX interprocess communication (IPC) is part of the POSIX:XSI Extension and has its origin in UNIX System V interprocess communication.

- Types of IPC mechanisms
  - **shared memory**
  - **message queues**
  - **semaphore sets**

- They provides mechanisms for sharing information among processes on the same system.

POSIX:XSI Shared Memory

- Shared memory allows processes to read and write from the same memory segment.Header file: **#include<sys/shm.h>**.
- The kernel maintains a structure, **shmid_ds** with the following members for each shared memory segment:

```
struct shmid_ds {
struct ipc_perm shm_perm; /*operation permission
                             structure */
size_t shm_segsz;          /* size of segment in bytes */
pid_t shm_lpid;          /*process ID of last operation */
pid_t shm_cpid;            /*process ID of creator */
shmatt_t shm_nattch;     /* number of current attaches */
time_t shm_atime;          /* time of last shmat */
time_t shm_dtime;          /* time of last shmdt */
time_t shm_ctime;          /* time of last shctl */
```

Permission Structure

IPC associates an **ipc_perm** structure with each IPC structure. This structure defines the permissions and owner and includes the following members:

```
struct ipc_perm {
    uid_t          cuid;      /* creator user ID */
    gid_t          cgid;      /* creator group ID */
    uid_t          uid;       /* owner user ID */
    gid_t          gid;       /* owner group ID */
    unsigned short mode;      /* r/w permissions */
};
```

Creating/Accessing a Shared Memory Segment

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

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

```
Returns:

(1) If successful, shmget returns a nonnegative integer
    corresponding to the shared memory segment identifier.

(2) If unsuccessful, shmget returns -1 and sets errno.
```

`key` Generation

`key` can be selected one of the three ways:

`IPC_PRIVATE`: System select

Pick a key directly: Select a random kay

Using `ftok()`: System generate using the function **`ftok()`**. **`ftok`** convert a pathname and a project identifier to a System V IPC key

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(const char *pathname, int proj_id);


(1) On success, the generated key_t value is returned.
(1) On failure -1 is returned.
```

The value of `shmflag`

The value of **`shmflag`** is composed of:

`IPC_CREAT`: to create a new segment. If this flag is not used, then **`shmget()`** will find the segment associated with key and check to see if the user has permission to access the segment.

`IPC_EXCL`: used with **`IPC_CREAT`** to ensure failure if the segment exists.

`mode_flags`: (lowest 9 bits) specifying the permissions granted to the owner, group, and other.

Attaching a shared memory segment

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

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

```
Returns:

(1) If successful, shmat returns the starting address of
    the segment.

(2) If unsuccessful, shmat returns -1 [ i.e (void *) -1 ]
    and sets errno
```

- **shmat()** attaches the shared memory segment identified by **shmid** to the address space of the calling process. It can be done in three ways.
- If **shmaddr** is NULL, the system chooses a suitable (unused) address at which to attach the segment.
- By default the shared memory segment is attached for both reading and writing by the calling process, if the process has read-write permissions for the shared memory segment. So, set the **shmflg** to 0.

The value of `shmflag` in `shmat()` OPTIONAL

The value of **`shmflag`** in **`shmat`** is composed of:

`0`: By default the shared memory segment is attached for both reading and writing by the calling process, if the process has read-write permissions for the shared memory segment.

`SHM_RDONLY`: Attach the segment for read-only access.

`SHM_RND`: If **`shmaddr`** isn't `NULL` and **`SHM_RND`** is specified in **`shmflg`**, the attach occurs at the address equal to **`shmaddr`** rounded down to the nearest multiple of **`SHMLBA`**(Segment low boundary address multiple).

`SHM_EXEC`: Allow the contents of the segment to be executed. The caller must have execute permission on the segment.

`SHM_REMAP`: This flag specifies that the mapping of the segment should replace any existing mapping in the range starting at **`shmaddr`** and continuing for the size of the segment.

Detaching a shared memory segment

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

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

```
Returns:

(1) If successful, shmdt returns 0.

(2) If unsuccessful, shmdt returns -1 and sets errno.
```

- The **shmaddr** parameter is the starting address of the shared memory segment.
- **shmdt()** detaches the shared memory segment located at the address specified by **shmaddr** from the address space of the calling process.
- The last process to detach the segment should deallocate the shared memory segment by calling **shmctl**.

Controlling Shared Memory

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

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

```
Returns:

(1) If successful, shmctl returns 0.

(2) If unsuccessful, shmctl returns -1 and sets errno.
```

- The **shmctl** function provides a variety of control operations on the shared memory segment **shmid** as specified by the **cmd** parameter.

- The interpretation of the **buf** parameter depends on the value of **cmd**.

POSIX:XSI Values of `cmd` for `shmctl`

cmd	**Description**
IPC_RMID	remove shared memory segment **shmid** and destroy corresponding **shmid_ds**
IPC_SET	set values of fields for shared memory segment **shmid** from values found in **buf**
IPC_STAT	copy current values for shared memory segment **shmid** into **buf**

Shared Memory Writer

Program 1: shmwriter.c

```
#include<stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include<sys/shm.h>
int main()
{
 int id,*var;
 key_t key;
 key=ftok("key.txt",65);
 id=shmget(key,50,0664|IPC_CREAT);
 printf("Shared memory Identifier=%d\n",id);
 var=(int *)shmat(id, NULL,0);
 *var=50;
 shmdt(var);
 return 0;
}
```

Shared Memory Reader

Program 2: shmreader.c

```
#include<stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include<sys/shm.h>
int main()
{
 int id,*rvar;
 key_t key;
 key=ftok("key.txt",65);
 id=shmget(key,50,0664);
 printf("Shared memory Identifier=%d\n",id);
 rvar=(int *)shmat(id, NULL,SHM_R);
 printf("Value in shared memory=%d\n",*rvar);
 shmdt(rvar);
 shmctl(id,IPC_RMID,NULL);
 return 0;
}
```

Shared Memory after `fork()`, `exec()`, & `exit()`

OPTIONAL

`fork()`: After **`fork`** the child inherits the attached shared memory segments.

`exec()`: After an **`exec`** all attached shared memory segments are detached (not destroyed).

`exit()`: After **`exit`** all shared memory segments are detached (not destroyed).

Shared Memory in Related Processes

```
int main(){
  pid_t pid;int shmid,*shvar;
  key_t key=ftok(".",45);
  shmid=shmget(key,20,0664|IPC_CREAT);
  printf("Key=%x ........Shmid=%d\n",key,shmid);
  shvar=shmat(shmid,NULL,0);
  printf("Default initial value of shvar=%d\n",*shvar);
  *shvar=10;
  pid=fork();
  if(pid==0){
      *shvar=*shvar+90;
      printf("child update=%d\n",*shvar);
      exit(0);
  }
  else{
       wait(NULL);
       *shvar=*shvar+110;
       printf("parent updates=%d\n",*shvar);
 }
 return 0;
}
```

Modification to the above Code

Replace the statement **`wait(NULL);`** after the line (**`printf()`** in the parent part and check the output. State the reason for such output.

```
int main(){
  pid_t pid;int shmid,*shvar;key_t key=ftok(".",45);
  shmid=shmget(key,20,0664|IPC_CREAT);
  printf("Key=%x ........Shmid=%d\n",key,shmid);
  shvar=shmat(shmid,NULL,0);
  printf("Default initial value of shvar=%d\n",*shvar);
  *shvar=10;pid=fork();
  if(pid==0){
       *shvar=*shvar+90;
       printf("child update=%d\n",*shvar);
       exit(0);
  }
  else{
        *shvar=*shvar+110;
        printf("parent updates=%d\n",*shvar);
        wait(NULL);
 }
 return 0;
}
```

System Limits for Shared Memory OPTIONAL

SHMMAX: Maximum size in bytes for a shared memory segment.

```
$ cat /proc/sys/kernel/shmmax
```

SHMMIN: Minimum size in bytes for a shared memory segment: implementation dependent (currently 1 byte, though PAGE_SIZE is the effective minimum size).

SHMMNI: System wide maximum number of shared memory segments:

```
$ cat /proc/sys/kernel/shmmni
```

SHMALL: System wide limit on the total amount of shared memory, measured in units of the system page size.

```
$ cat /proc/sys/kernel/shmall
```

Race Condition and Critical Section

```
do {
    Entry section

    Critical section

    Exit section

    Remainder section

} while(true);
```

Peterson's Solution to Critical Section

do {

```
flag[i] = true ;
turn = j;
while (flag[j] && turn == j);
```

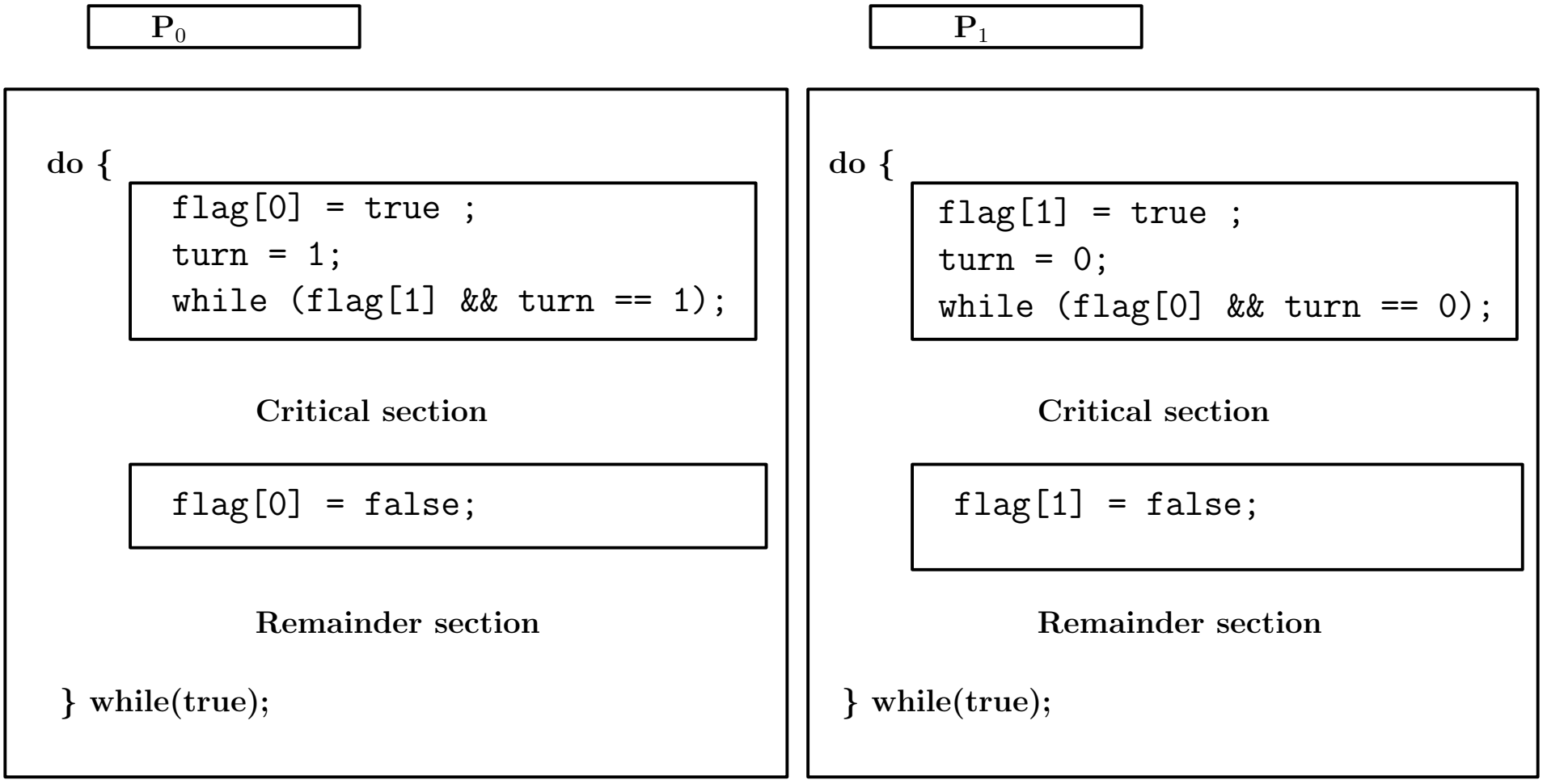
Critical section

```
flag[i] = false;
```

Remainder section

} while(true);

Peterson's Solution for Two Processes



POSIX:XSI Message Queues

Introduction

- The message queue is a POSIX:XSI interprocess communication mechanism that allows a process to send and receive messages from other processes.
- The data structures for message queues are defined in **sys/msg.h**.
- The major data structure for message queues is **msqid_ds**, which has the following members.

```
struct msqid_ds {
  struct ipc_perm  msg_perm;   /* operation permission structure */
  msgqnum_t        msg_qnum;   /* no of messages on queue */
  msglen_t         msg_qbytes; /* bytes max on a queue */
  pid_t            msg_lspid;  /* PID of last msgsnd */
  pid_t            msg_lrpid;  /* PID of last msgrcvcall */
  time_t           msg_stime;  /* last msgsnd time */
  time_t           msg_rtime;  /* last msgrcv time */
  time_t           msg_ctime;  /* last change time */
};
```

Permission Structure

IPC associates an **ipc_perm** structure with each IPC structure. This structure defines the permissions and owner and includes the following members:

```
struct ipc_perm {
    uid_t          cuid;      /* creator user ID */
    gid_t          cgid;      /* creator group ID */
    uid_t          uid;       /* owner user ID */
    gid_t          gid;       /* owner group ID */
    unsigned short mode;      /* r/w permissions */
};
```

Creating/Accessing a Message Queue

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

int msgget(key_t key, int msgflg);
```

```
Returns:

(1) If successful, msgget returns a nonnegative integer
    corresponding to the message queue identifier.

(2) If unsuccessful, msgget returns -1 and sets errno.
```

The value of `msgflag`

The value of **`shmflag`** is composed of:

`IPC_CREAT`: to create a new segment. If this flag is not used, then **`shmget()`** will find the segment associated with key and check to see if the user has permission to access the segment.

`IPC_EXCL`: used with **`IPC_CREAT`** to ensure failure if the segment exists.

`mode_flags`: (lowest 9 bits) specifying the permissions granted to the owner, group, and other.

msgsend() System Call

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

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

```
Returns:

(1) If successful, msgsnd returns 0.

(2) If unsuccessful, msgsnd returns -1 and sets errno.
```

msgsnd Parameters

msqid: The **msqid** parameter identifies the message queue.

msgp: The **msgp** parameter points to a user-defined buffer that contains the message to be sent. The msgp argument is a pointer to caller-defined structure of the following general form:

```
struct msgbuf {
  long mtype;   /* message type, must be > 0 */
  char mtext[SIZE];   /* message data */
};
```

msgz: The **msgsz** parameter specifies the actual size of the message text.

msgflg: The **msgflg** parameter specifies actions to be taken under various conditions.

msgrcv() System Call

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

ssize_t msgrcv(int msqid, void *msgp, size_t
    msgsz,long msgtyp, int msgflg);
```

```
Returns:

(1) If successful, msgrcv returns the number of bytes in
    the text of the message.

(2) If unsuccessful, msgrcv returns (ssize_t) -1 and sets
    errno.
```

`msgrcv` Parameters

A program can remove a message from a message queue with **`msgrcv`** with the following parameters:

`msqid`: The **`msqid`** parameter identifies the message queue.

`msgp`: The **`msgp`** parameter points to a user-defined buffer for holding the message to be retrieved. The format of **`msgp`** is as described in **`msgsnd`** parameters slide.

`msgz`: The **`msgsz`** parameter specifies the actual size of the message text.

`msgtyp`: The msgtyp parameter can be used by the receiver for message selection.

`msgflg`: The **`msgflg`** parameter specifies actions to be taken under various conditions.

`msgtyp` Possible Values

The given table shows how **`msgrcv`** uses the **`msgtyp`** parameter to determine the order in which it removes messages from the queue.

`msgtyp`	**action**
0	remove first message from queue
> 0	remove first message of type **`msgtyp`** from the queue
< 0	remove first message of lowest type that is less than or equal to the absolute value of **`msgtyp`**

msgflag in msgsnd/ msgrcv OPTIONAL

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

0: Default send/receive permission.

IPC_NOWAIT: Return immediately if no message of the requested type is in the queue.

MSG_EXCEPT: Used with **msgtyp** greater than 0 to read the first message in the queue with message type that differs from msgtyp.

MSG_NOERROR: To truncate the message text if longer than **sgsz** bytes.

Controlling Message Queues

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

int msgctl(int msgid, int cmd, struct msqid_ds *buf);
```

```
Returns:

(1) If successful, msgctl returns 0.

(2) If unsuccessful, msgctl returns -1 and sets errno.
```

- The **msgctl** is used to deallocate or change permissions for the message queue identified by **msqid**.
- The **cmd** parameter specifies the action to be taken as listed in the next slide table.
- The **msgctl** function uses its **buf** parameter to write or read state information, depending on **cmd**.

POSIX:XSI Values of `cmd` for `msgctl`

cmd	**Description**
IPC_RMID	remove the message queue **msqid** and destroy the corresponding **msqid_ds**
IPC_SET	set members of the **msqid_ds** data structure from **buf**
IPC_STAT	copy members of the **msqid_ds** data structure into **buf**

Message Sender

Program 3: msgsender.c

```
#define SIZE 512
struct mymsg{
        long mtype;         /* message type, must be > 0 */
        char mtext[SIZE]; /* message data */
};
int main()
{
    key_t key;
    int qid;
    struct mymsg msg1;
    key=ftok(".",65);
    qid=msgget(key,IPC_CREAT|0660);
    printf("Key=%x \t Msg Queue Identifier=%d\n",key,qid);
    msg1.mtype=10;
    strcpy(msg1.mtext,"I am msg-1");
    msgsnd(qid,&msg1,sizeof(msg1.mtext),0);
    return 0;
}
```

Message Receiver

Program 4: msgreceiver.c

```
#define SIZE 512
struct mymsg{
        long mtype;
        char mtext[SIZE];
};
int main()
{
    key_t key;
    int qid;
    struct mymsg msg;
    key=ftok(".",65);
    qid=msgget(key,IPC_CREAT|0660);
    printf("Key=%d \t Msg Queue Identifier=%d\n",key,qid);
    /* Get the message from the queue */
    msgrcv(qid,&msg,SIZE,0,0);
    printf("msg read from the msg queue=%s\n",msg.mtext);
    msgctl(qid,IPC_RMID,NULL);
    return 0;
}
```

Message Queue in Related Processes

```
#define SIZE 512
struct mymsg{
   long mtype;
   char mtext[SIZE];
};
int main(){
 pid_t pid;int mqid;struct mymsg msg,mrcv;key_t key=ftok(".",144);
 mqid=msgget(key,0660|IPC_CREAT);
 printf("key=%x \t Msg Identifier=%d\n",key,mqid);
 msg.mtype=20;
 strcpy(msg.mtext,"message from parent to child");
 pid=fork();
 if(pid>0){
    msgsnd(mqid,&msg,sizeof(msg.mtext),0);
    wait(NULL);
    msgctl(mqid,IPC_RMID,NULL);
 }
 else{
    msgrcv(mqid,&mrcv,sizeof(mrcv.mtext),0,0);
    fprintf(stderr,"Child received=%s\n",mrcv.mtext);
    exit(0);
 }
 return 0;
}
```

System Limits for Message Queue OPTIONAL

MSGMAX: Maximum size for a message text.

```
$ cat /proc/sys/kernel/msgmax
```

MSGMNB: Default maximum size in bytes of a message queue.

```
$ cat /proc/sys/kernel/msgmnb
```

MSGMNI: System wide maximum number of message queues.

```
$ cat /proc/sys/kernel/msgmni
```

Exercise:15.8 Implementing Pipes with Message queues

POSIX:XSI Semaphore Sets

Introduction to Semaphore Set

- A POSIX:XSI semaphore consists of an array of semaphore elements.
- The semaphore elements are similar, but not identical, to the classical integer semaphores proposed by Dijsktra.
- A process can perform operations on the entire set in a single call.
- Each semaphore element includes at least the following information.
  - A nonnegative integer representing the value of the semaphore element (**`semval`**)
  - The process ID of the last process to manipulate the semaphore element (**`sempid`**)
  - The number of processes waiting for the semaphore element value to increase (**`semncnt`**)
  - The number of processes waiting for the semaphore element value to equal 0 (**`semzcnt`**)
- For every set of semaphores in the system, the kernel maintains the structure of information, **`semid_ds`**, defined by including **`<sys/sem.h>`**:

Data Structure for Semaphore Sets & Permission Structure

A semaphore set is uniquely identified by a positive integer (its semid) and has an associated data structure of type **`struct semi_ds`**, defined in **`<sys/sem.h>`**

```
struct msqid_ds {
  struct ipc_perm sem_perm; /* operation permission structure */
  unsigned short sem_nsems; /* number of semaphores in the set */
  time_t sem_otime;   /* time of last semop */
  time_t sem_ctime;  /* time of last semctl */
};
```

IPC associates an **`ipc_perm`** structure with each IPC structure. This structure defines the permissions and owner and includes the following members:

```
struct ipc_perm {
    uid_t             cuid;      /* creator user ID */
    gid_t             cgid;      /* creator group ID */
    uid_t             uid;       /* owner user ID */
    gid_t             gid;       /* owner group ID */
    unsigned short    mode;      /* r/w permissions */
};
```

Semaphore Set

- Let S be the semaphore set defined as $S = \{s_1, s_2, s_3, ..., s_n\}$, where $s_{i,1 \leq i \leq n}$ is an semaphore element.
- Each semaphore element has two queues associated with it, (1) a queue of processes waiting for the value to equal 0 (2) a queue of processes waiting for the value to increase.
- The semaphore element operations allow a process to block until a semaphore element value is 0 or until it increases to a specific value greater than zero.
- This semaphore (i.e. POSIX:XSI semaphore or System V sempahore) refers to *a set of counting semaphores*: one or more semaphores (a set), each of which is a counting semaphore.

Creating/Accessing a Semaphore Set

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

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

Returns:
(1) If successful, semget returns a nonnegative integer
    corresponding to the semaphore identifier.
(2) If unsuccessful, the semget function returns -1 and
    sets errno.
```

- The **semget** function returns the semaphore identifier associated with the **key** parameter.
- The **nsems** parameter specifies the number of semaphore elements in the set.
- If we are not creating a new semaphore set but just for accessing an existing set , we can speclfy this argument as 0.
- The individual semaphore elements within a semaphore set are referenced by the integers 0 through **nsems - 1**.
- Semaphores have permissions specified by the **semflg** argument of **semget**.
- Semaphore elements should be initialized with **semctl** function before they are used.

Creating a New Semaphore Set - I

The following code segment creates a new semaphore set containing three semaphore elements.

```
#include <sys/sem.h>
#define PERMS (S_IRUSR | S_IWUSR)

int semid;
if ((semid = semget(IPC_PRIVATE, 3, PERMS)) == -1)
    perror("Failed to create new private semaphore");
```

- This semaphore can only be read or written by the owner.
- The **IPC_PRIVATE** key guarantees that **semget** creates a new semaphore.
- To get a new semaphore set from a made-up key or a key derived from a pathname, the process must specify by using the **IPC_CREAT** flag that it is creating a new semaphore.
- If both **ICP_CREAT** and **IPC_EXCL** are specified, **semget** returns an error if the semaphore already exists.

Creating a New Semaphore Set - II

- To get a new semaphore set from a made-up key or a key derived from a pathname, the process must specify by using the **IPC_CREAT** flag that it is creating a new semaphore.

```
#include <sys/sem.h>
#include <sys/types.h>
#include <sys/ipc.h>
#define PERMS (S_IRUSR | S_IWUSR)

int semid;
key_t key;
key=ftok("filename",45);
semid = semget(key, 3, IPC_CREAT | PERMS);
if(semget==-1){
    perror("Failed to create new private semaphore");
    return 1;
}
```

- If both **ICP_CREAT** and **IPC_EXCL** are specified, **semget** returns an error if the semaphore already exists.

Accessing an Existing Semaphore Set

The following code segment accesses a semaphore set with a single element identified by the key value 99887.

```
#define PERMS (S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
    S_IROTH | S_IWOTH)
#define KEY ((key_t)99887)

int semid;
if ((semid = semget(KEY, 1, PERMS | IPC_CREAT)) == -1)
      perror ("Failed to access semaphore with key 99887");
```

The **IPC_CREAT** flag ensures that if the semaphore set doesn't exist, **semget** creates it. The permissions allow all users to access the semaphore set.

- Giving a specific key value allows cooperating processes to agree on a common semaphore set. If the semaphore already exists, **semget** returns a handle to the existing semaphore.
- If you replace the **semflg** argument of **semget** with **PERMS | IPC_CREAT | IPC_EXCL**, **semget** returns an error when the semaphore already exists.

Demonstration: To Create a Semaphore Set

The following program demonstrates how to identify a semaphore set by using a key generated from a pathname and an ID, which are passed as command-line arguments.

```
#define PERMS (S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH |
    S_IWOTH)
#define SET_SIZE 2
int main(int argc, char *argv[]) {
 key_t mykey; int semid;
 if (argc != 3) {
   fprintf(stderr, "Usage: %s pathname id\n", argv[0]);
   return 1;
 }
 if ((mykey = ftok(argv[1], atoi(argv[2]))) == (key_t)-1) {
   fprintf(stderr, "Failed to derive key from filename %s:%s\n",
       argv[1], strerror(errno));
   return 1;
 }
 if ((semid = semget(mykey, SET_SIZE, PERMS | IPC_CREAT)) == -1) {
     fprintf(stderr, "Failed to create semaphore with key %d:%s\n"
         ,(int)mykey, strerror(errno));
     return 1;
 }
 printf("semid = %d\n", semid);return 0;
}
```

Semaphore Control

- Each element of a semaphore set must be initialized with **semctl** before it is used.
- The **semctl** function provides control operations in element **semnum** for the semaphore set **semid**.
- The **cmd** parameter specifies the type of operation.
- The optional fourth parameter, **arg**, depends on the value of **cmd**.

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

int semctl(int semid, int semnum, int cmd, ...);

Returns:
(1) If successful, semctl returns a nonnegative value whose
     interpretation depends on cmd.
(2) The GETVAL, GETPID, GETNCNT and GETZCNT values of cmd
    cause semctl to return the value associated with cmd.
(3) All other values of cmd cause semctl to return 0 if
    successful.
(4) If unsuccessful, semctl returns -1 and sets errno.
```

POSIX:XSI values for the `cmd` parameter of `semctl`

cmd	description
GETALL	return values of the semaphore set in **arg.array**
GETVAL	return value of a specific semaphore element GETPID return process ID of last process to manipulate element
GETNCNT	return number of processes waiting for element to increment
GETZCNT	return number of processes waiting for element to become 0
IPC_RMID	remove semaphore set identified by **semid**
IPC_SET	set permissions of the semaphore set from **arg.buf**
IPC_STAT	copy members of **semid_ds** of semaphore set **semid** into **arg.buf**
SETALL	set values of semaphore set from **arg.array**
SETVAL	set value of a specific semaphore element to **arg.val**

Several of these commands, such as GETALL and SETALL, require an **arg** parameter(i.e the 4th parameter to **semctl**) to read or store results. The **arg** parameter is of type **union semun**, which must be defined in programs that use it, as follows.

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
} arg;
```

Setting value to Semaphore Element

Program to set the value of the specified semaphore element and also deletes the semaphore specified by **semid**.

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};
int main(){
 int semid,valr=20; key_t k;
 union semun arg;
 k=ftok(".",453);
 semid=semget(k,1,IPC_CREAT|0644);
 valr=semctl(semid,0,GETVAL);
 printf("Default semaphore element value=%d\n",valr);
 arg.val=100;
 /* setting semephore element value */
 semctl(semid,0,SETVAL,arg);
 /* Semaphore element: value access */
  valr=semctl(semid,0,GETVAL);
  printf("Semaphore element value after setting =%d\n",valr);
  /* removing semaphore set */
  semctl(semid,0,IPC_RMID);
 return 0;
}
```

Semaphore Set Operations

Once a semaphore set is opened with **semget**, operations are performed on one or more of the semaphores in the set using the **semop** function.

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

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

Returns:
(1) If successful, semop returns 0.
(2) If unsuccessful, semop returns -1 and sets errno.
```

- The **semop** function **atomically** performs a user-defined collection of semaphore operations on the semaphore set associated with identifier **semid**.
- The **sops** parameter points to an array of element operations. The **sops** points to an array of the following structures:

```
struct sembuf {
 short sem_num;   /* semaphore number:  1, 2, 3, ..., nsems-1 */
 short sem_op;    /* semaphore operation : <0, 0, >0 */
 short sem_flg;   /* operation flags: 0 , IPC-NOWAIT, SEM-UNDO */
```

- The **nsops** parameter specifies the number of element operations in the **sops** array.

struct sembuf Initialization

Once a semaphore set is opened with **semget**, operations are performed on one or more of the semaphores in the set using the **semop** function.

```
struct sembuf ops[2];

ops[0].sem_num=0; / * wait for [0] to be 0 * /
ops[0].sem_num=0;
ops[0].sem_num=0;
ops[1].sem_num=0; / * then increment [0] by 1 * /
ops[1].sem_num=1;
ops[1].sem_num=SEM_UNDO;
```

The function **setsembuf** initializes the **struct sembuf** structure members **sem_num**, **sem_op** and **sem_flg** in an implementation-independent manner

```
#include <sys/sem.h>
void setsembuf(struct sembuf *s, int num, int op, int flg) {
  s->sem_num = (short)num;
  s->sem_op = (short)op;
  s->sem_flg = (short)flg;
  return;
}
```

Operation of `semop`

The operation of **`semop`** is based on the three possible values of **`sem_op`**: **positive**, **0**, or **negative**. The **`sem_op`** element operations are values specifying the amount by which the semaphore value is to be changed.

- If **`sem_op`** is positive, **`semop`** adds the value to the corresponding semaphore element value and awakens all processes that are waiting for the element to increase. This corresponds to the release of resources that a semaphore controls.
- If **`sem_op`** is 0, the caller wants to wait until semaphore value is 0. If semaphore's value is already 0, return is made immediately.
- If **`sem_op`** is negative, the caller wants to wait until the semaphore's value becomes greater than or equal to the absolute value of **`sem_op`**. This corresponds to the allocation of resources.

Example: Using semop

The function **setsembuf** initializes the **struct sembuf** structure members **sem_num**, **sem_op** and **sem_flg** in an implementation-independent manner

```
#include <sys/sem.h>
void setsembuf(struct sembuf *s, int num, int op, int flg) {
  s→sem_num = (short)num;
  s→sem_op = (short)op;
  s→sem_flg = (short)flg;
  return;
}
```

The following code segment atomically increments element zero of **semid** by 1 and element one of **semid** by 2, using the above **setsembuf** function.

```
struct sembuf myop[2];

setsembuf(myop, 0, 1, 0);
setsembuf(myop + 1, 1, 2, 0);
if (semop(semid, myop, 2) == -1)
    perror("Failed to perform semaphore operation");
```

Complete Program of the previous slide

The following program atomically increments element zero of **semid** by 1 and element one of **semid** by 2, using the above **setsembuf** function.

```
void setsembuf(struct sembuf *s, int num, int op, int flg);
int main(){
 int semid,val1,val2;
 key_t k;k=ftok(".",253);
 union semun arg;
 struct sembuf myop[2];
 semid=semget(k,2,IPC_CREAT|0644);
 val1=semctl(semid,0,GETVAL);
 val2=semctl(semid,0,GETVAL);
 printf("Default values: sem[0]=%d, sem[1]=%d\n",val1,val2);
 setsembuf(myop, 0, 1, 0);
 setsembuf(myop + 1, 1, 2, 0);
 if (semop(semid, myop, 2) == -1)
      perror("Failed to perform semaphore operation");
 /* value after setting */
 val1=semctl(semid,0,GETVAL);
 val2=semctl(semid,1,GETVAL);
 printf("Values after semop: sem[0]=%d, sem[1]=%d\n",val1,val2);
 /*removing semaphore set */
 semctl(semid,0,IPC_RMID);
 return 0;
}
```

Semaphore Set for Ctrical Section Case

Problem Description

Suppose a two-element semaphore set, **S**, represents a tape drive system in which Process 1 uses **Tape A**, Process 2 uses **Tape A and B**, and Process 3 uses **Tape B**. The following pseudocode segment defines semaphore operations that allow the processes to access one or both tape drives in a mutually exclusive manner.

S[0] represents **Tape A**, and **S[1]** represents **Tape B**. We assume that both elements of **S** have been initialized to 1.

```
struct sembuf getTapes[2];
struct sembuf releaseTapes[2];
setsembuf(&(getTapes[0]), 0, -1,0);
setsembuf(&(getTapes[1]), 1, -1,0);
setsembuf(&(releaseTapes[0]), 0,1,0);
setsembuf(&(releaseTapes[1]), 1,1,0);
```

```
Process 1:
 semop(S, getTapes, 1);
   <use tape A>
 semop(S, releaseTapes, 1);
```

```
Process 2:
 semop(S, getTapes, 2);
 <use tapes A and B>
 semop(S, releaseTapes, 2);
```

```
Process 3:
semop(S, getTapes + 1, 1);
 <use tape B>
semop(S, releaseTapes + 1, 1);
```

Implementation of Tape A & B Sharing

The following program shows the critical section problem solution tape based problem.

```
//Semaphore array for ipc technidque
/*Requirement:
     (1) Three processes
     (2) Two Tapes A and B-- as semaphore
*/
#include<stdio.h>
#include<sys/sem.h>
#include<unistd.h>
#include<sys/ipc.h>
#include<sys/wait.h>
void setsembuf(struct sembuf *s,int num, int op, int flg)
{
  s→sem_num=(short)num;
  s→sem_op=(short)op;
  s→sem_flg=(short)flg;
}
union semun {
   int     val;     /* Value for SETVAL */
   struct semid_ds *buf;     /* Buffer for IPC_STAT, IPC_SET */
   unsigned short  *array;   /* Array for GETALL, SETALL */
  };
```

Implementation of Tape A & B Sharing Contd...

```
int main(){
  int sid,v0,v1;
  pid_t pid;
  key_t k;
  struct sembuf getTapes[2];
  struct sembuf releaseTapes[2];
  k=ftok(".",45);
  sid=semget(k,2,IPC_CREAT|0744);
  printf("Semaphore set identifier=%d\n",sid);
  //initializing both semaphore to 1
  union semun arg;
  arg.val=1;
  semctl(sid,0,SETVAL,arg);
  arg.val=1;
  semctl(sid,1,SETVAL,arg);
  ////////////////////////////////
  setsembuf(&(getTapes[0]),0,-1,0);
  setsembuf(&(getTapes[1]),1,-1,0);
  setsembuf(&(releaseTapes[0]),0,1,0);
  setsembuf(&(releaseTapes[1]),1,1,0);
  if(fork()==0){
   //printf("Process-1\n");
   int i=1;
   semop(sid,getTapes,1);
   printf("Now Process-1 gets Tape-A ++++++++++++\n");
   while(i<=15){
        printf("Process-1 holds tape-A\n");
        sleep(2);
    i=i+1;
   }
   semop(sid,releaseTapes,1);
  }
```

Implementation of Tape A & B Sharing Contd...

```
  else if(fork()==0){
      //printf("Process-2\n");
      int i=1;
      semop(sid,getTapes,2);
      printf("\t\tNow Process-2 gets Tape-A-and-B =========\n");
      while(i<=5){
        printf("\t\tProcess-2 holds tape-A and B\n");
        sleep(2);
        i=i+1;
     }
      semop(sid,releaseTapes,2);
  }
  else if(fork()==0){
      //printf("Process-3\n");
      int i=1;
      semop(sid,getTapes+1,1);
      printf("\tNow Process-3 gets Tape-B :::::::::\n");
      while(i<=10){
           printf("\tProcess-3 holds tape-B\n");
           sleep(2);
       i=i+1;
     }
      semop(sid,releaseTapes+1,1);
  }
  else{
    while(wait(NULL)>0);
    printf("Main Process Ends\n");
    semctl(sid,0,IPC_RMID);
 }
 return 0;
}
```

Solve the Problem Using Semaphore Set

Two concurrent processes **P** and **Q** are accessing their critical sections by using boolean variables **S** and **T** as follows;

Proces P

```
while(true)
{
//Entery section ?

print(1);
print(1);

// Exit section ?
}
```

Proces Q

```
while(true)
{
//Entery section ?

print(0);
print(0);

// Exit section ?
}
```

Complete the entry section and exit section of process **P** and **Q** with suitable semaphore operations using the two boolean semaphores **S** and **T**. Also suggest the initial values of **S** and **T**, such that the execution of the processes will print the sequence 11001100.....

Home Work

- Modify the Text Book **Program 14.1** to protect the critical section problem using semaphore set for the chain of *n* processes.
- Differentiate between the POSIX classical semaphore and POSIX:XSI semaphore set.