Computer Science & Engineering Department
I. I. T. Kharagpur

Operating System: CS33007
3rd Year CSE: 5th Semester (Autumn 2006 - 2007)
Lecture XI

Goutam Biswas *Date:* 29th August - 4th September, 2006

1 Semaphore

A operating system support for process synchronization called *semaphore*, was proposed by Edsger Dijkstra in 1965.

A *semaphore* is a shared integer variable that takes non-negative values (there are implementations where -ve values are permitted). It may be viewed as a data type with three *atomic* operations defined on it and maintained by the OS kernel and system calls are required to perform operations on it. The three operations are

- *Initialization*: A semaphore may be initialized to a non-negative value
- *wait* or *P()*: *probeer* - try to decrement. The P() or wait() operation is performed on a semaphore by a process P to test (atomically) whether the resource is available. If the resource is not available i.e. the value of the semaphore is 0 (or less), the process is blocked.
- *Signal* or *V()*: *verhoog* - increment. The V() or signal() operation is performed on a semaphore by a process P when it releases a resource. If there are processes that wait for this semaphore, one of them is sent to the ready queue. If there is no such process, the value of the semaphore is incremented.

A general semaphore is called *counting semaphore* and a semaphore is called a binary semaphore if it takes values 0 and 1.

A semaphore may be considered as a resource counter for a sharable resource (data structure). If there is one copy of the resource, the availability and non-availability is indicated by the semaphore values 1 and 0 respectively. If there are multiple copies of the resource, the semaphore may be initialized to the number of copies.

The 1/0 values of a binary semaphore with the atomic operations can also be used for control synchronization.

The conceptual view of a semaphore as a datatype is

```
typedef struct {
     int count ;
     semQ queue ;
} semaphore ;

semaphore s ;
```

1. Initialization -

```
void initSem(semaphore *sP, int val) {
     sP -> count = val ;
}
```

For a binary semaphore the value is 0 or 1.

2. Binary semaphore

```
void P(semaphore *sP) {
     if(sP -> count == 1) sP -> count = 0 ;
     else {
          addQ(sP -> queue, process) ;
          remove(readyQ, process) ;
          block the process ;
        }
}

void V(semaphore *sP) {
     if(sP -> queue == NULL) sP -> count = 1 ;
     else {
          addQ(readyQ, frontQ(sP -> queue)) ;
          deleteQ(sP -> queue) ;
     }
}
```

3. Counting semaphore -

```
void P(semaphore *sP) {
     if(sP -> count > 0) sP -> count-- ;
     else {
           addQ(sP -> queue, process) ;
           remove(readyQ, process) ;
           block process ;
     }
}

void V(semaphore *sP) {
     if(!isEmpty(sP -> queue)) {
          addQ(readyQ, frontQ(sP -> queue)) ;
          deleteQ(sP -> queue) ;
     }
     else sP -> count++ ;
}
```

1.1 Mutual Exclusion

Mutual exclusion can be achieved using a binary semaphore in the following way.

```
semaphore s ;
initSem(&s, 1) ;

/*       Process i     */

P(&s) ;
/* Critical section */
V(&s) ;
```

1.2 Semaphore in Unix/Linux

Linux supports two types of semaphore, *kernel semaphore* that are used for synchronization in the kernel and System V IPC semaphore used by user processes.

We start with user semaphore - a semaphore is a *set or vector* of non-negative integer values. The maximum number of semaphores in a set is restricted by SEMMSL (250) (`include/linux/sem.h`). Similarly the maximum number of sets are also restricted to SEMMNI (128) (`include/linux/sem.h`).

Some of the System V API's are -

- Request the OS to create a semaphore set and get a descriptor.
- Update the values of the semaphores in the set.
- Read the values of the semaphors.
- Release the semaphore set.

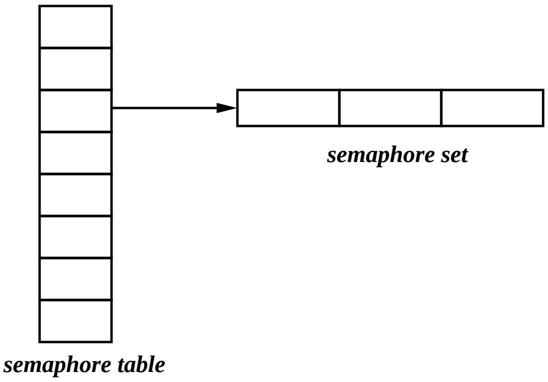
The OS maintains a semaphore table in its own address space to keep track of all the semaphore sets created. Following are some of the data stored in the table.

- The *key value* (an integer identifier) assigned to the created semaphore set. Other processes (with proper permission) may access the set using the same key.
- Read-write access permission for the owner, group and others.
- The number of elements in the set and a pointer to the array of semaphors.
- Each semaphore stores its value, number of processes that are blocked etc.

An entry in the semaphore table is not always removed automatically even if the creator process terminates. It may be necessary to removed it explicitly.

Data structures of the semaphore table: The actual data structure in Linux is different and more complicated.

```
/* One sem_array data structure for each set of semaphores
in the system - linux/sem.h */
struct sem_array {
    struct kern_ipc_perm    sem_perm;   /* permissions */
    time_t                  sem_otime;  /* last semop time */
```




```
    time_t                  sem_ctime;  /* last change time */
    struct sem              *sem_base;  /* ptr to first semaphore
                                            in array */
    struct sem_queue        *sem_pending;   /* pending operations
                                                to be processed */
    struct sem_queue        **sem_pending_last; /* last pending
                                                    operation */
    struct sem_undo         *undo;          /* undo requests on
                                               this array */
    unsigned long           sem_nsems;      /* no. of semaphores
                                                in array */
};
struct sem {
        int     semval;         /* current value */
        int     sempid;         /* pid of last operation */
};
```

Different System V API for semaphores:

1. The API `semget()` for getting a semaphore:

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

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

- The value returned by the `semget()` is a semaphore identifier (a +ve integer) or -1 if there is an error.
- The first parameter specifies an IPC key (use '`ftok()`'). A new private semaphore is created if the key is `IPC_PRIVATE`.
- The second parameter `nsems` specifies the number of elements in the set.
- If the `semflag` is not `IPC_CREAT`, no new semaphore gets created. The flag also holds the read-write permission bits.

A typical semget() call to create a semaphore is

```
#define NUM_SEMS 2
#define PERM (0644)

int semid ;
semid = semget(IPC_PRIVATE, NUM_SEMS, IPC_CREAT | PERM);
```

2. The API `semop()` - used to perform *wait()* signal *signal()* operations one or more semaphores of a set. The operation, if it can be performed, is *atomic*.

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

int semop(int semid, struct sembuf opsPtr[], unsigned int nops) ;
```

- The first parameter is the semaphore identifier of a semaphore.
- The third parameter specifies the number of entries in the `opsPtr[]` array (pointer). There is a limit to this size (SEMOPM) per call (atomic operation).
- Let us look at the `sembuf` structure

```
struct sembuf {
          ushort  sem_num ;     /* semaphor index */
          short   sem_op ;      /* semaphore operation */
          short   sem_flg ;     /* operation flag   */
};
```

Each entry in the `opsPtr[]` array specifies an operation on an element of the set (array) of semaphors. The value of `sem_op` is

  - positive $(+n)$ - increase the indexed semaphore value by $+n$.
  - negative $(-n)$ - decrease the indexed semaphore value by n.
  - test whether the semaphore value is zero (0).

The OS blocks the calling process, if a process calls `semop()` with `sem_flg` $!=$ `IPC_NOWAIT`, and tries to decrease a semaphore vale to -ve, or tests the value of a semaphore for zero (0), but it is not.

3. The API `semctl()` - used to initialize semaphores of a set, removing a semaphore set from the system, getting status information etc.

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

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

- The first argument is the semaphore identifier of the semaphore set.

- The second argument is the semaphore index in the set.
- The third argument is the command e.g. IPS_STAT, IPC_RMID, GETVAL etc.
- If there is a fourth argument, it is of type `union semun arg`.

1. P() operation using semaphore -

```
static int P(int semID) {
     struct sembuf buff ;

     buff.sem_num = 0 ; // On the 0th element
     buff.sem_op  = -1 ;
     buff.sem_flg = 0 ;
     if(semop(semID, &buff, 1) == -1) {
             perror("semop P operation error") ;
             return 0 ;
     }
     return -1 ;
}
```

2. V() operation using semaphore -

```
static int V(int semID) {
     struct sembuf buff ;

     buff.sem_num = 0 ;  // On the 0th element
     buff.sem_op  = 1 ;
     buff.sem_flg = 0 ;
     if(semop(semId, &buff, 1) == -1) {
             perror("semop V operation error") ;
             return 0 ;
     }
     return -1 ;
}
```

Examples of programs using semaphore in Linux -

1. race.c

```
/*********************************************
 * race.c                                    *
 * *******************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main() {
int i, chID ;

if((chID = fork()) != 0) { // Parent
```

```
srand((unsigned int) getpid()) ;
for(i=0; i<=10; ++i) {
printf("\tIndian ") ;  fflush(stdout);
                        sleep(rand() % 2) ;
printf("\tInstitute of ") ;  fflush(stdout) ;
                        sleep(rand() % 2) ;
printf("\tTechnology\n") ; sleep(rand() % 2) ;
}
}
else { // Child
srand((unsigned int) getpid()) ;
for(i=0; i<=10; ++i) {
printf("\t\tKharagpur ") ;  fflush(stdout) ;
                        sleep(rand() % 2) ;
printf("\t\tKanpur\n") ;  fflush(stdout) ;
                        sleep(rand() % 2) ;
printf("\t\tBombay\n") ; sleep(rand() % 2) ;
}
}

return 0 ;
}
```

2. semaphore1.c

```
/*********************************************
 * semaphore1.c: no race                     *
 * *******************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/wait.h>
#define FLAGS (0644)

static int P(int semID) {
struct sembuf buff ;

buff.sem_num = 0 ;
buff.sem_op  = -1 ;
buff.sem_flg = 0 ;
if(semop(semID, &buff, 1) == -1) {
perror("semop P operation error") ;
return 0 ;
}
```

```
return -1 ;
}

static int V(int semID) {
struct sembuf buff ;

buff.sem_num = 0 ;
buff.sem_op  = 1 ;
buff.sem_flg = 0 ;
if(semop(semID, &buff, 1) == -1) {
perror("semop V operation error") ;
return 0 ;
}
return -1 ;
}

int main() {
int i, semID, chID ;

if((semID = semget(IPC_PRIVATE, 1,  IPC_CREAT | FLAGS)) == -1) {
perror("semget fails") ;
        exit(0) ;
}
semctl(semID, 0, SETVAL, 1) ;
if((chID = fork()) != 0) { // Parent
int status ;

srand((unsigned int) getpid()) ;
for(i=0; i<=10; ++i) {
P(semID) ;
printf("\tIndian "); fflush(stdout) ;
                        sleep(rand() % 2) ;
printf("Institute of "); fflush(stdout);
                        sleep(rand() % 2) ;
printf("Technology\n"); sleep(rand() % 2) ;
V(semID) ;
sleep(rand() % 2) ;
}
waitpid(chID, &status, 0) ;
semctl(semID, 0, IPC_RMID) ;
}
else { // Child
srand((unsigned int) getpid()) ;
for(i=0; i<=10; ++i) {
P(semID) ;
printf("\t\tKharagpur "); fflush(stdout) ;
                        sleep(rand() % 2) ;
```

```
printf("Kanpur "); fflush(stdout);
                        sleep(rand() % 2) ;
printf("Bombay\n") ; sleep(rand() % 2) ;
V(semID) ;
sleep(rand() % 2) ;
}
}
return 0 ;
}
```

3. semProdCons1.c -

```
/************************************************************
 * Producer-Consumer Problem on shared memory between
 * processes.  Atomicity by semaphore
 * $ cc -Wall semProdCons1.c queue.o
 * ***********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/sem.h>
#include "queue.h"
#define FLAGS (0644)

void producer(queue *, int) ;
void consumer(queue *, int) ;

static int P(int semID) {
struct sembuf buff ;

buff.sem_num = 0 ;
buff.sem_op  = -1 ;
buff.sem_flg = 0 ;
if(semop(semID, &buff, 1) == -1) {
perror("semop P operation error") ;
return 0 ;
}
return -1 ;
}

static int V(int semID) {
struct sembuf buff ;
```

```
buff.sem_num = 0 ;
buff.sem_op  = 1 ;
buff.sem_flg = 0 ;
if(semop(semID, &buff, 1) == -1) {
perror("semop V operation error") ;
return 0 ;
}
return -1 ;
}


int main() {
int shmID, semID, chID1, chID2, status ;
struct shmid_ds buff ;
queue *qP ;

shmID = shmget(IPC_PRIVATE, sizeof(queue), IPC_CREAT | 0777);
if(shmID == -1) {
printf("Error in shmget") ;
exit(0) ;
}

if((semID = semget(IPC_PRIVATE, 1,  IPC_CREAT | FLAGS)) == -1) {
                perror("semget fails") ;
                exit(0) ;
        }
        semctl(semID, 0, SETVAL, 1) ;

qP = (queue *) shmat(shmID, 0, 0777) ;
initQ(qP) ;

if((chID1 = fork()) != 0) { // Parent after child I
if((chID2 = fork()) != 0) { // Parent after child II

waitpid(chID1, &status, 0) ;
waitpid(chID2, &status, 0) ;

shmdt(qP) ;
shmctl(shmID, IPC_RMID, &buff) ;
semctl(semID, 0, IPC_RMID) ;
}
else { // Child 2
queue *qP ;

qP = (queue *) shmat(shmID, 0, 0777) ;
consumer(qP, semID) ;
```

```
shmdt(qP) ;
}
}
else { // Child I
queue *qP ;

qP = (queue *) shmat(shmID, 0, 0777) ;
producer(qP, semID) ;
shmdt(qP) ;
}

return 0 ;
}

void producer(queue *qP, int semID){
int count = 1 ;

        while(1) {
                int data, added = 1, err ;

if(added) {
data = rand() ;
added = 0 ;
}
P(semID) ;
        err = addQ(qP, data) ;
V(semID) ;
if(err == OK) {
added = 1 ;
        printf("Produced Data %d: %d\n", count++, data) ;
}
        }
}

void consumer(queue *qP, int semID) {
        int count = 1 ;

        while(1) {
                int data, deleted ;

P(semID) ;
        deleted = frontQ(qP, &data) ;
        if(deleted == OK) deleted = deleteQ(qP) ;
V(semID) ;
if(deleted == OK)
          printf("\t\t\tConsumed Data %d: %d\n", count++, data) ;
        }
```

```
}
```

This simple solution has bust wait.

1.3 Bounded Buffer Producer-Consumer Problem

We can solve this problem using only one semaphore as we are doing busy wait for *full* and *empty* queue in cases of i addQ() and deleteQ() respectively. We have to introduce two more semaphores if we want to avoid any busy wait.

We assume that there are 'n' elements in the buffer array. The semaphore for mutual exclusion is *mutex* (essentially a binary semaphore). The semaphore *empty* and *full* (both counting semaphore) keeps track of whether the buffer is empty or full.

The initial value of *mutex* is 1. All n elements of the buffer are empty, so the semaphore *empty* is initialized to n. Nothing is full, so the semaphore *full* is initialized to 0.

```
semaphore mutex = 1, empty = n, full = 0;
```

1. Producer code -

```
/******** Producer Code **************/

while(1){
     .........
     data = ..... // Item is produced
     P(empty) ;   // wait if nothing is empty, and
                  //     reduces emptiness by 1
     P(mutex) ;   // Mutually exclusive
     addQ(qP, data) ;
     V(mutex) ;   // End of mutual exclusion

     V(full) ;    // Increase full by one
}
```

2. Consumer code -

```
/********** Consumer Code *************/

while(1) {
     .....................
     P(full) ;   // Wait if nothing is full, and
              // reduce full by one
     P(mutex) ;
     data = front(qP) ;
     deleteQ(qP) ;
     V(mutex) ;
     V(empty) ;  // One more empty
}
```

3. Following is the code without busy wait.

```
/****** buff.h **********/
          #ifndef _BUFFER_H
          #define _BUFFER_H

          #define MAX 5
          #define ERROR 1
          #define OK 0

          typedef struct {
              int data[MAX] ;
              int front , rear, count ;
          } buffer ;

          /* Queue may contain MAX data.*/

           void initB(buffer *) ;
           void addB(buffer * , int) ;
           void deleteB(buffer *);
           void frontB(buffer *, int *) ;
          #endif
/****** buff.c *******/
#include "buff.h"
#include <unistd.h>
#include <stdlib.h>

void initB(buffer *q) {
q -> front = q -> rear = 0 ;
q -> count = 0 ;
}

void addB(buffer *q, int n) {
int i, lim, m ;

  q -> rear = (q -> rear + 1 ) % MAX ;
  q -> data[q -> rear] = n ;
m = q -> count ;
m = m + 1 ;
lim = rand()%10000000 ;
        for(i=1; i<= lim; ++i) ;
q -> count = m ;
}


void deleteB(buffer *q) {
int n, i, lim ;
```

```
  q -> front = (q -> front + 1 ) % MAX ;
n = q -> count ;
n = n - 1 ;
lim = rand()%10000000 ;
        for(i=1; i<= lim; ++i) ;
q -> count = n ;
}

void frontB(buffer *q , int *v) {
  *v = q -> data[(q -> front + 1) % MAX] ;
}
/**************************************************************
 * Producer-Consumer Problem on shared memory between
 * processes.  Atomicity by three semaphore
 * No bust wait
 * $ cc -Wall semProdCons3.c buff.o
 * ***********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/sem.h>
#include "buff.h"
#define FLAGS (0777)

void producer(buffer *, int) ;
void consumer(buffer *, int) ;

/*********************************************
 * 0 - mutex, 1 - empty, 2 - full            *
 * *******************************************/

static int P(int semID, int num) {
struct sembuf buff ;

buff.sem_num = num ;
buff.sem_op  = -1 ;
buff.sem_flg = 0 ;
if(semop(semID, &buff, 1) == -1) {
perror("semop P operation") ;
return 0 ;
}
```

```
return -1 ;
}

static int V(int semID, int num) {
struct sembuf buff ;

buff.sem_num = num ;
buff.sem_op  = 1 ;
buff.sem_flg = 0 ;
if(semop(semID, &buff, 1) == -1) {
perror("semop V operation\n") ;
return 0 ;
}
return -1 ;
}


int main() {
int shmID, semID, chID1, chID2, chID3, status ;
struct shmid_ds buff ;
buffer *qP ;

shmID = shmget(IPC_PRIVATE, sizeof(buffer), IPC_CREAT | 0777);
if(shmID == -1) {
printf("Error in shmget error") ;
exit(0) ;
}

if((semID = semget(IPC_PRIVATE, 3,  IPC_CREAT | FLAGS)) == -1) {
                perror("semget fails") ;
exit(0) ;
        }
        semctl(semID, 0, SETVAL, 1) ;       // mutex = 1
        semctl(semID, 1, SETVAL, MAX) ;     // empty = MAX
        semctl(semID, 2, SETVAL, 0) ;       // full = 0

qP = (buffer *) shmat(shmID, 0, 0777) ;
initB(qP) ;

if((chID1 = fork()) != 0) { // Parent
if((chID2 = fork()) != 0) { // Parent
if((chID3 = fork()) != 0) { // parent now

waitpid(chID1, &status, 0) ;
waitpid(chID3, &status, 0) ;
waitpid(chID2, &status, 0) ;
```

```
shmdt(qP) ;
shmctl(shmID, IPC_RMID, &buff) ;
semctl(semID, 0, IPC_RMID) ;
}
else { // Child 3
buffer *qP ;

qP = (buffer *) shmat(shmID, 0, 0777) ;
producer(qP, semID) ;
shmdt(qP) ;
}
}
else { // Child 2
buffer *qP ;

qP = (buffer *) shmat(shmID, 0, 0777) ;
consumer(qP, semID) ;
shmdt(qP) ;
}
}
else { // Child I
buffer *qP ;

qP = (buffer *) shmat(shmID, 0, 0777) ;
producer(qP, semID) ;
shmdt(qP) ;
}

return 0 ;
}

void producer(buffer *qP, int semID){
int count = 1 ;

srand(getpid()) ;
        while(count <= 10) {
                int data ;

data = rand() ;
P(semID, 1) ;     // P(empty);
P(semID, 0) ;     // P(mutex);
        addB(qP, data) ;
V(semID, 0) ;     // V(mutex) ;
V(semID, 2) ;     // V(full) ;
        printf("ProcID: %d, Produced Data %d: %d\n",
getpid(), count++, data) ;
        }
```

```
}

void consumer(buffer *qP, int semID) {
        int count = 1 ;

        while(count <= 20) {
                int data ;

P(semID, 2) ;   // P(full) :
P(semID, 0) ;   // P(mutex) ;
        frontB(qP, &data) ;
        deleteB(qP) ;
V(semID, 0) ;   // V(mutex) ;
V(semID, 1) ;   // V(empty) ;
        printf("\t\t\tConsumed Data %d: %d\n", count++, data) ;
        }
}
```

4. Using the semaphore set:

```
/**************************************************************
 * Producer-Consumer Problem on shared memory between
 * processes.  Atomicity by three semaphore
 * semaphore set
 * $ cc -Wall semProdCons4.c buff.o
 * ************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/sem.h>
#include "buff.h"
#define FLAGS (0777)

void producer(buffer *, int) ;
void consumer(buffer *, int) ;

/***********************************************
 * 0 - mutex, 1 - empty, 2 - full             *
 * *********************************************/

static int P2(int semID, int num0, int num1) {
struct sembuf buff[2] ; // Two atomic operations
```

```
buff[0].sem_num = num0 ;
buff[1].sem_num = num1 ;
buff[0].sem_op  = -1 ;
buff[1].sem_op  = -1 ;
buff[0].sem_flg = 0 ;
buff[1].sem_flg = 0 ;
if(semop(semID, buff, 2) == -1) {
perror("semop P operation error") ;
return 0 ;
}
return -1 ;
}

static int V2(int semID, int num0, int num1) {
struct sembuf buff[2] ;

buff[0].sem_num = num0 ;
buff[1].sem_num = num1 ;
buff[0].sem_op  = 1 ;
buff[1].sem_op  = 1 ;
buff[0].sem_flg = 0 ;
buff[1].sem_flg = 0 ;
if(semop(semID, buff, 2) == -1) {
perror("semop V operation error") ;
return 0 ;
}
return -1 ;
}

int main() {
int shmID, semID, chID1, chID2, chID3, status ;
struct shmid_ds buff ;
buffer *qP ;

shmID = shmget(IPC_PRIVATE, sizeof(buffer), IPC_CREAT | 0777);
if(shmID == -1) {
printf("Error in shmget") ;
exit(0) ;
}

if((semID = semget(IPC_PRIVATE, 3,  IPC_CREAT | FLAGS)) == -1) {
                perror("semget fails") ;
exit(0) ;
        }
        semctl(semID, 0, SETVAL, 1) ;       // mutex = 1
        semctl(semID, 1, SETVAL, MAX) ;     // empty = MAX
```

```
        semctl(semID, 2, SETVAL, 0) ;       // full = 0

qP = (buffer *) shmat(shmID, 0, 0777) ;
initB(qP) ;

if((chID1 = fork()) != 0) { // Parent
if((chID2 = fork()) != 0) { // Parent
if((chID3 = fork()) != 0) { // parent now

waitpid(chID1, &status, 0) ;
waitpid(chID3, &status, 0) ;
waitpid(chID2, &status, 0) ;

shmdt(qP) ;
shmctl(shmID, IPC_RMID, &buff) ;
semctl(semID, 0, IPC_RMID) ;
}
else { // Child 3
buffer *qP ;

qP = (buffer *) shmat(shmID, 0, 0777) ;
producer(qP, semID) ;
shmdt(qP) ;
}
}
else { // Child 2
buffer *qP ;

qP = (buffer *) shmat(shmID, 0, 0777) ;
consumer(qP, semID) ;
shmdt(qP) ;
}
}
else { // Child I
buffer *qP ;

qP = (buffer *) shmat(shmID, 0, 0777) ;
producer(qP, semID) ;
shmdt(qP) ;
}

return 0 ;
}

void producer(buffer *qP, int semID){
int count = 1 ;
```

```
srand(getpid()) ;
        while(count <= 10) {
                int data ;

data = rand() ;
P2(semID, 1, 0) ;     // P(empty); P(mutex) ;
        addB(qP, data) ;
V2(semID, 0, 2) ;     // V(mutex) ; V(full) ;
        printf("ProcID: %d, Produced Data %d: %d\n",
getpid(), count++, data) ;
        }
}

void consumer(buffer *qP, int semID) {
        int count = 1 ;

        while(count <= 20) {
                int data ;

P2(semID, 2, 0) ;   // P(full) ; P(full) ;
        frontB(qP, &data) ;
        deleteB(qP) ;
V2(semID, 0, 1) ;   // V(mutex) ; V(empty) :
        printf("\t\t\tConsumed Data %d: %d\n", count++, data) ;
        }
}
```

1.4 Reader-Writer problem

A shared data may be read (not consumed) by some of the processes. Some other processes may update the data. This scenario may give rise to the reader-writer problem.

The correctness of reader-writer problem is defined as follows:

1. More than one reader can read concurrently.
2. Only one writer at a time can update the data
3. Reading and writing are mutually exclusive

The correctness condition is silent about the priority of reader or writer. Different cases may arise depending on this priority. Reader may get the priority over the writer. It may be just the opposite.

We consider the case of *reader priority*: We need two semaphores - one for the mutual exclusion of reader and the writer `mutex`. Another to maintain the atomicity of incrementing or decrementing the reader count (countsem). Both are binary semaphores.

- Initialization:

```
semaphore mutex = 1, countsem = 1 ;
int readerCount = 0 ;  // shared variable
```

- Writer process:

```
P(mutex) ;
/* CS: writer writes */
V(mutex) ;
```

- The reader process:

```
P(countsem) ;
++readerCount ;
if(readerCount == 1) P(mutex) ;
V(countsem) ;

/* CS: reader reading */

P(countsem) ;
--readerCount ;
if(readerCount == 0) V(mutex) ;
V(countsem) ;
```

This solution is unfair to the writer (she may starve). Let us consider the following alternate where

1. More than one readers can read concurrently.
2. A writer writes exclusively. When the writer finishes, all readers waiting or one of the waiting writers are activated.

3. When the last reader finishes, it activates one waiting writer (if there is any).

We maintain the following counts:

- `totReader`: total number of readers reading and waiting.
- `rReader`: number of readers reading i.s. `totReader` − `rReader` is the number of readers waiting.
- `totWriter`: total number of writers writing and waiting.
- `wWriter`: number of writers writing. This number is 1 or 0. `totWriter` − `wWriter` is the number of writers waiting.

These counters are updated atomically using the binary semaphore `mutex`. We use two more counting semaphores `read` and `write`.

```
/* Initialization */

semaphore mutex = 1, read = 0, write = 0 ;
int totReader = 0, rReader = 0, totWriter = 0, wWriter = 0 ; // shared
```

The reader process is

```
/*   Reader Process */
while(1){
   P(mutex);    // Reader enters ME
   ++totReader
   if(wWriter == 0){ // No writer
      ++rReader;     // Reader redy to read
      V(read) ;      // self-scheduling
   }
   V(mutex) ;  // Reader entered ME end
   P(read) ;
   /* CS for reading: no writer */
   P(mutex)    // Reader leavs ME
   --rReader ; --totReader ;
   if(rReader == 0 && totWriter > wWriter){ // No reader,
      wWriter  = 1 ;                        // allow a writer
      V(write) ;
   }
   V(mutex)    // Reader left ME end
}
```

The writer process is

```
/*    Writer Process */
while(1){
   P(mutex)   // Writer enters ME
   ++totWriter ;
```

```
    if(rRead == 0 && wWriter == 0){  // No reader or writer
       wWriter = 1 ; // Writer ready to write
       V(write) ;    // self-scheduling
    }
    V(mutex)   // Writer entered ME end
    P(write) ;
    /* CS for writing: no reader and no other writer */
    P(mutex)   // Writer leavs ME
    --wWrite ; --totWriter ;
    while(rReader < totReader) {   // Pushing waiting readers
        ++rReader;
        V(read) ;
    }
    if(rReader == 0 && totWriter > wWriter) { // No reader waits but
        wWriter = 1 ;                         // writers are waiting
        V(write)
    }
    V(mutex)   // Writer left ME end
}
```

Once again the system prefers the readers.

1.5 Dining-Philosophers Problem

1. Five philosophers seating around a round table, thinking or trying to eat using five chopsticks.

2. Towards a simple solution - one semaphore per chopstick

```
     semaphore cstk[5] = {1, 1, 1, 1, 1} ;
/* Philosopher i */

     while(1) {
          P(cstk[i]) ;
          P(cstk[(i+1) % 5]) ;

          /* CS: philosopher i eats */

          V(cstk[i]) ;
          P(cstk[(i+1) % 5]) ;

          /* philosopher i thinks */
     }
```

3. There is a possibility of dead-lock.

1. Consider two processes P_1 and P_2 accessing the shared variables x and y protected by two binary semaphores Sx and Sy respectively, both initialized to 1. P and V

denote the usual semaphore operators, where P decrements the semaphore value, and V increments the semaphore value. The pseudo-code for P_1 and P_2 is as follows:

```
P1:                              P2:
   while (1) {                      while (1) {
    L1: ..........                   L3: .........
    L2: ..........                   L4: .........
      x = x + 1 ;                      y = y + 1 ;
      y = y - 1 ;                      x = x - 1 ;
      V(Sx) ;                          V(Sy) ;
      V(Sy) ;                          V(Sx) ;
   }                                }
```

In order to avoid deadlock, the correct operators at L_1, L_2, L_3 and L_4 are respectively.

(a) `P(Sy), P(Sx)` and `P(Sx), P(Sy)`

(b) `P(Sx), P(Sy)` and `P(Sy), P(Sx)`

(c) `P(Sx), P(Sx)` and `P(Sy), P(Sy)`

(d) `P(Sx), P(Sy)` and `P(Sx), P(Sy)`

- It cannot be `P(Sy), P(Sx)` and `P(Sx), P(Sy)` as execution of `P(Sy)` in P_1 and `P(Sx)` in P_2 will cause deadlock.
- It cannot be `P(Sx), P(Sy)` and `P(Sy), P(Sx)` as execution of `P(Sx)` in P_1 and `P(Sy)` in P_2 will cause deadlock.
- It cannot be `P(Sx), P(Sx)` and `P(Sy), P(Sy)` as after the first `P(Sx)` and the first `P(Sy)` there cannot be any progress - deadlock.
- The last one is the correct solution.

References

[1] *Operating System: A Concept-Based Approach*, by D M Dhamdhere, 2nd ed., TMH, ISBN 0-07-061194-7, 2006