OS ASSIGNMENT-5 REPORT

Lab4 - Lab5 - IPC with Semaphores

: Adarsh Anand : 2003101 :

: Prof - Dr Sharad Sinha : TA - Prachi Kashikar

QUESTIONS

Part B

1. Compile and run lab5_1.c with gcc. Review lab5_1.c to be sure you understand how it works.

Ans) In this program we are creating a shared memory segment and then writing a string to it. The parent process then forks a child process and the child process reads the string from the shared memory segment and prints it on the screen. The child process reads using semaphores and the parent process writes using semaphores. The parent process writes the string to the shared memory segment and then signals the child process to read the string from the shared memory segment. Using semaphores we ensure that the child process reads only when the parent process has written to the shared memory segment and the parent process writes only when the child process has read from the shared memory segment. Thus no data is lost and no data is read before it is written.

2. Use the idea of semaphores, as implemented in lab5_1.c , in place of the spinlocks in lab 4 to handle the synchronization of the reader and writer process from lab5_1.c .

Ans)

Initial imports and constants

```
/* A readers/writers program using a shared buffer and semaphores  */

#include <stdio.h>

#include <sys/mman.h>

#include <sys/sem.h>

#include <sys/types.h>

#include <unistd.h>


#define BUF_SIZE 1                                  /* logical size of buffer */

#define SHARED_MEM_SIZE (BUF_SIZE + 2) * sizeof(int) /* size of shared memory */

#define run_length 10                               /* number of iterations in test run */


#define SIZE sizeof(int) /* size of [int] integer */


#define buf_used 0  /* semaphore array index to check buffer elts used */

#define buf_space 1 /* semaphore array index to check buffer elts empty */
```

Creating semaphores

```


int sem_init(void) { /* procedure to create and initialize semaphores and return semaphore
id,

                       assuming two semaphores defined in the given array of semaphores
*/

    int semid;

    /* create new semaphore set of 2 semaphores */

    if ((semid = semget(IPC_PRIVATE, 2, IPC_CREAT | 0600)) < 0)
/*https://man7.org/linux/man-pages/man2/semget.2.html*/

    {

        perror("error in creating semaphore"); /* 0600 = read/alter by user */

        exit(1);

    }
```

```
    /* initialization of semaphores */

    /* BUF_SIZE free spaces in empty buffer */

    if (semctl(semid, buf_space, SETVAL, BUF_SIZE) < 0)
/*https://man7.org/linux/man-pages/man2/semctl.2.html*/

    {

        perror("error in initializing first semaphore");

        exit(1);

    }

    /* 0 items in empty buffer */

    if (semctl(semid, buf_used, SETVAL, 0) < 0) {

        perror("error in initializing second semaphore");

        exit(1);

    }

    return semid;

}
```

Wait and post operation

```
void P(int semid, int index) { /* procedure to perform a P or wait operation on a semaphore
of given index */

    struct sembuf sops[1];     /* only one semaphore operation to be executed */


    sops[0].sem_num = index; /* define operation on semaphore with given index */

    sops[0].sem_op = -1;     /* subtract 1 to value for P operation */

    sops[0].sem_flg = 0;     /* type "man semop" in shell window for details */


    if (semop(semid, sops, 1) == -1) {

        perror("error in semaphore operation");

        exit(1);

    }

}


void V(int semid, int index) { /* procedure to perform a V or signal operation on semaphore
of given index */

    struct sembuf sops[1];     /* define operation on semaphore with given index */
```

```
    sops[0].sem_num = index; /* define operation on semaphore with given index */
    sops[0].sem_op = 1;      /* add 1 to value for V operation */
    sops[0].sem_flg = 0;     /* type "man semop" in shell window for details */

    if (semop(semid, sops, 1) == -1) {
        perror("error in semaphore operation");
        exit(1);
    }
}
```

Child process

```
if (0 == pid) { /* processing for child */
        printf("The child process begins.\n");
        for (i_child = 0; i_child < run_length; i_child++) {
            P(semid, buf_used); /* wait semaphore for something used */
            value = buffer[*out];
            *out = (*out + 1) % BUF_SIZE;
            printf("Child's report: item %2d == %2d\n", i_child, value);
            V(semid, buf_space); /* signal semaphore for space available */
        }
        printf("The child is done\n");
    }
```

Parent Process

```
else { /* processing for parent */
        printf("The parent process begins.\n");
        for (j_child = 0; j_child < run_length; j_child++) {
            P(semid, buf_space);             /* wait semaphore for space available */
            buffer[*in] = j_child * j_child; /* put data in buffer */
            *in = (*in + 1) % BUF_SIZE;
            printf("Parent's report: item %2d put in buffer\n", j_child);
            V(semid, buf_used); /* signal semaphore for something used */
```

```
    }

    wait(pid); /* wait for child to finish */

    printf("The parent is done\n");

}
```

Output

```
The parent process begins.
Parent's report: item  0 put in buffer
The child process begins.
Child's report: item  0 ==  0
Parent's report: item  1 put in buffer
Child's report: item  1 ==  1
Parent's report: item  2 put in buffer
Child's report: item  2 ==  4
Parent's report: item  3 put in buffer
Child's report: item  3 ==  9
Parent's report: item  4 put in buffer
Child's report: item  4 == 16
Parent's report: item  5 put in buffer
Child's report: item  5 == 25
Parent's report: item  6 put in buffer
Child's report: item  6 == 36
Parent's report: item  7 put in buffer
Child's report: item  7 == 49
Parent's report: item  8 put in buffer
Child's report: item  8 == 64
Parent's report: item  9 put in buffer
Child's report: item  9 == 81
The child is done
The parent is done
```

4. This program contains a third semaphore, mutex. Explain the purpose of this semaphore. Specifically, if semaphore mutex were omitted, give a timing sequence involving the several readers and/or writers showing what might go wrong.

Ans)

- The purpose of the mutex semaphore is to ensure that only one process is accessing the buffer at a time.
- If the mutex semaphore were omitted, then the buffer could be accessed by multiple processes at the same time, which would cause race conditions and incorrect data to be read or written.
- For example, if the mutex semaphore were omitted, then the following sequence could occur:
    - process 1 reads the value of in
    - process 2 reads the value of in
    - process 1 writes to the buffer
    - process 2 writes to the buffer
    - process 1 increments in
    - process 2 increments in
    - process 1 writes to the buffer
    - process 2 writes to the buffer, etc.
- This would cause the buffer to be filled with incorrect data.

5. Program lab5_2.c prevents any reader from working at the same time as any writer. Assuming that the buffer contains several locations, however, writing to one buffer location should not interfere with reading from another. That is, the critical section for readers need not be considered exactly the same as the critical section for writers.

Remove semaphore mutex and add additional semaphores, as needed, so that some reader could work concurrently with some writer (assuming the buffer contained some data but was not full – so both reading and writing made sense).

Ans)

- We create 2 additional mutexes readMutex and writeMutex such that readMutex is used to protect the out pointer and writeMutex is used to protect the in pointer.
- We also create 2 additional semaphores buf_used and buf_space to keep track of the number of elements in the buffer.
- The reader process will wait on buf_used and the writer process will wait on buf_space.
- The reader process will wait on readMutex and the writer process will wait on writeMutex.
- The reader process will signal readMutex and the writer process will signal writeMutex.
- The reader process will signal buf_space and the writer process will signal buf_used.

```
/* set up semaphores */
semid = sem_create(4); /* create 4 semaphores */


/* initialize semaphores */
```

```
   sem_init(semid, buf_used, 0);         /* initially no buffer elements used */
   sem_init(semid, buf_space, BUF_SIZE); /* initially BUF_SIZE buffer elements empty */
   sem_init(semid, readMutex, 1);        /* initially mutex is available */
   sem_init(semid, writeMutex, 1);       /* initially mutex is available */
```

Writer process

```
 /* spawn writer processes */
   for (p_count = 1; p_count <= NUM_WRITERS; p_count++) {
       if (-1 == (pid = fork())) /* spawn child process */
       {
           perror("error in fork");
           exit(1);
       }

       if (0 == pid) { /* processing for parent == writer */
           printf("The writer process %d begins.\n", p_count);

           for (i = 0; i < writer_length; i++) {
               value = 100 * p_count + i; /* writer == first digit of value */
               P(semid, buf_space);       /* wait semaphore for space available */
               P(semid, writeMutex);      /* wait semaphore for space available */
               buffer[*in] = value;       /* put data in buffer */
               *in = (*in + 1) % BUF_SIZE;
               V(semid, writeMutex); /* signal semaphore for space available */
               V(semid, buf_used);   /* signal semaphore for something used */
                                     /*printf ("Writer %d: item %2d put %d in buffer\n",
j_child);*/
           }
           printf("Writer %d done.\n", p_count);
           exit(0);
       } else
           proc[p_count - 1] = pid;
   }
```

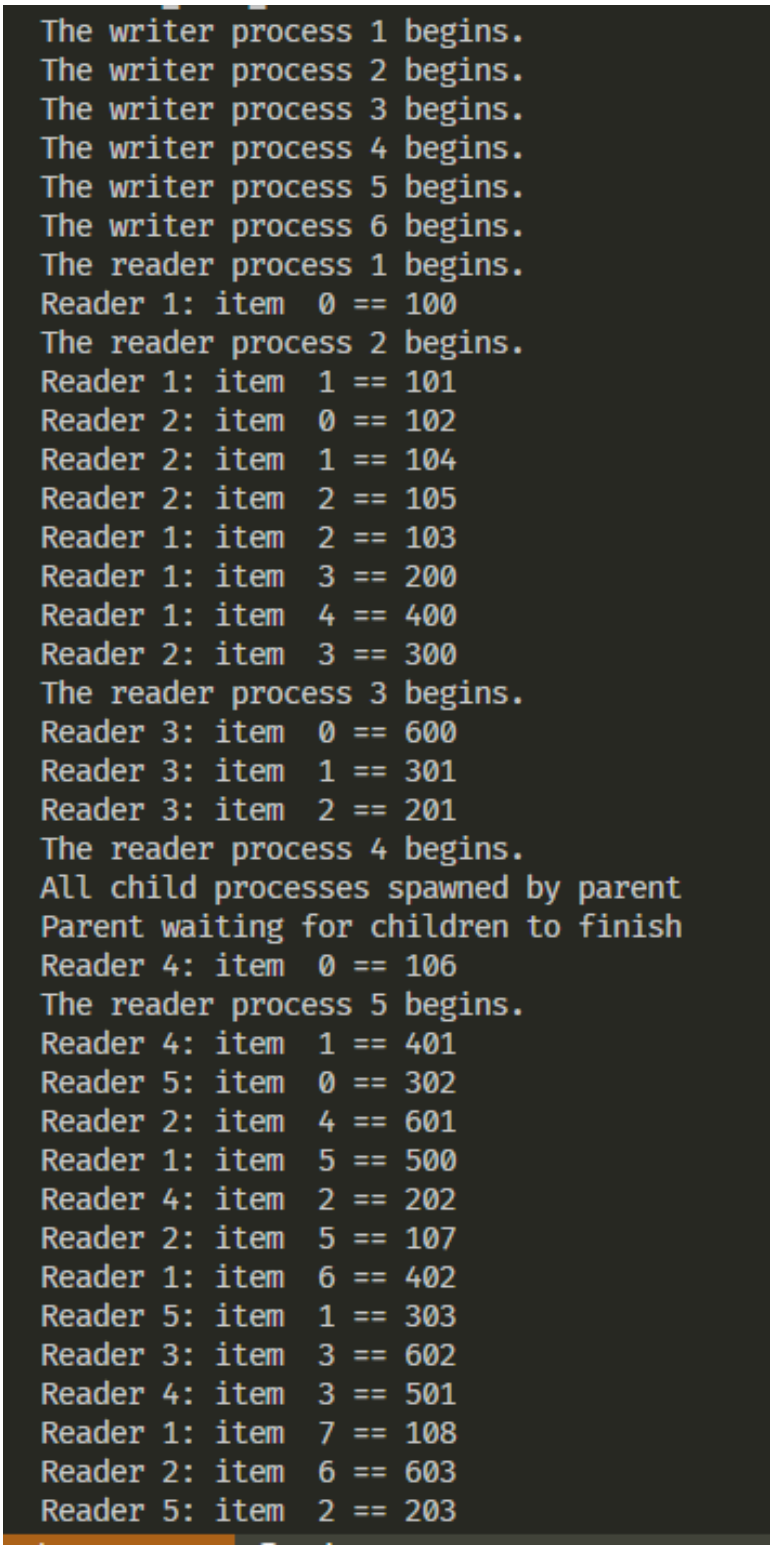
Reader process

```
/* spawn reader processes */
for (p_count = 1; p_count <= NUM_READERS; p_count++) {
    if (-1 == (pid = fork())) /* spawn child process */
    {
        perror("error in fork");
        exit(1);
    }

    if (0 == pid) { /* processing for child == reader */
        printf("The reader process %d begins.\n", p_count);

        for (i = 0; i < reader_length; i++) {
            P(semid, buf_used);   /* wait semaphore for something used */
            P(semid, readMutex);  /* wait semaphore for something used */
            value = buffer[*out]; /* take data from buffer */
            *out = (*out + 1) % BUF_SIZE;
            V(semid, readMutex); /* signal semaphore for something used */
            V(semid, buf_space); /* signal semaphore for space available */
            printf("Reader %d: item %2d == %2d\n", p_count, i, value);
            if ((i + p_count) % 5 == 0) /* pause somewhere in processing */
                sleep(1);               /* to make output more interesting */
        }
        printf("Reader %d done.\n", p_count);
        exit(0);
    } else
        proc[p_count + NUM_READERS - 1] = pid;
}
```

Output



Full output :-

The writer process 1 begins.

The writer process 2 begins.

The writer process 3 begins.

The writer process 4 begins.

The writer process 5 begins.

The writer process 6 begins.

The reader process 1 begins.

Reader 1: item 0 == 100

The reader process 2 begins.

Reader 1: item 1 == 101

Reader 2: item 0 == 102

Reader 2: item 1 == 104

Reader 2: item 2 == 105

Reader 1: item 2 == 103

Reader 1: item 3 == 200

Reader 1: item 4 == 400

Reader 2: item 3 == 300

The reader process 3 begins.

Reader 3: item 0 == 600

Reader 3: item 1 == 301

Reader 3: item 2 == 201

The reader process 4 begins.

All child processes spawned by parent

Parent waiting for children to finish

Reader 4: item 0 == 106

The reader process 5 begins.

Reader 4: item 1 == 401

Reader 5: item 0 == 302

Reader 2: item 4 == 601

Reader 1: item 5 == 500

Reader 4: item 2 == 202

Reader 2: item 5 == 107

Reader 1: item 6 == 402

Reader 5: item 1 == 303

Reader 3: item 3 == 602

Reader 4: item 3 == 501

Reader 1: item 7 == 108

Reader 2: item 6 == 603

Reader 5: item 2 == 203

Reader 1: item 8 == 304

Writer 1 done.

Reader 3: item 4 == 502

Reader 4: item 4 == 604

Reader 1: item 9 == 109

Reader 2: item 7 == 305

Reader 2: item 8 == 605

Reader 3: item 5 == 204

Reader 3: item 6 == 306

Reader 4: item 5 == 503

Reader 3: item 7 == 205

Reader 4: item 6 == 606

Reader 5: item 3 == 307

Reader 5: item 4 == 504

Reader 5: item 5 == 607

Reader 1: item 10 == 206

Reader 4: item 7 == 403

Reader 1: item 11 == 608

Reader 1 done.

Reader 2: item 9 == 308

Reader 3: item 8 == 505

Writer 3 done.

Reader 2: item 10 == 309

Reader 4: item 8 == 404

Reader 3: item 9 == 609

Writer 6 done.

Reader 5: item 6 == 506

Reader 2: item 11 == 207

Reader 4: item 9 == 405

Reader 2 done.

Reader 3: item 10 == 507

Reader 4: item 10 == 208

Reader 5: item 7 == 406

Reader 3: item 11 == 508

Reader 3 done.

Writer 2 done.

Reader 5: item 8 == 209

Writer 5 done.

Reader 4: item 11 == 407

Writer 4 done.

Reader 5: item 9 == 509

Reader 5: item 10 == 408

Reader 4 done.

Reader 5: item 11 == 409

Reader 5 done.

Semaphore cleanup complete.