CS39002 Operating Systems Laboratory
Lab Test 2
06-Apr-2026, 6:15pm – 7:45pm
Maximum Marks: 70

Roll No: ______________________________

Name: ______________________________

---

1. [*Shared memory and semaphores*]

(a) In this part, you write a library to implement the functionality provided by a (synchronization) barrier. A barrier needs to be initialized to a positive integer m which is the number of processes that must join the barrier in order to lift it. The barrier also maintains a count w of how many processes have actually joined. This implementation uses three semaphores MTX, CND, and ACK. MTX is used as a mutex to guard against simultaneous accesses of the shared variable w (m is read-only, and does not need protection). The other two semaphores are needed for the purpose of synchronization (explained later). Implement the library routines following the instructions given below. Notice that the barrier library is independent of any user program, and conversely too any user program should use the interface (only the following function calls) without ever bothering about how barriers are implemented. The library must not implement any function other than the four given below, and must not declare any global variables.

Declaration and creation

The two counts m and w are stored in a single shared-memory segment M with $M[0]$ storing m, and $M[1]$ storing w. The three semaphores mentioned above are created as a single set S of three semaphores, where $S[0]$, $S[1]$, and $S[2]$ are respectively the semaphores MTX, CND, and ACK. We declare a barrier data type to consist of the ID's of M and S. You must use System V constructs only as covered in the lab sessions (not POSIX constructs).

```
    typedef struct {
        int shmid;
        int semid;
    } barrier_t;
```

Write the creation function for the barrier B (to be precise, its components M and S) from the keys supplied by the user. This does not initialize the barrier. **[3]**

```
barrier_t barrier_create ( int shmkey , int semkey )
{


        barrier_t B;

        B.shmid = shmget(shmkey, sizeof(int) * 2, 0666 | IPC_CREAT);
        B.semid = semget(semkey, 3, 0666 | IPC_CREAT);
        return B;


}
```

Removal

Write a function to remove the shared-memory segment M and the semaphore set S associated with a barrier B. **[2]**

```
void barrier_destroy ( barrier_t B )
{

        shmctl(B.shmid, IPC_RMID, NULL);
        semctl(B.semid, 0, IPC_RMID);

}
```

Initialization

Given m, a barrier B is initialized by setting $M[0]$ to m, and $M[1]$ to 0. $S[0]$ (that is, MTX) is for mutual exclusion, and should be initialized to 1. The other two semaphores of S (CND and ACK) are for waiting, and should be initialized to 0. A barrier must first be created, and then initialized. Assume that initialization is done before any process joins the barrier, so this function does not have a need to protect the shared data. Moreover, attaching to M should be opaque to any user program, that is, no user program calling the following functions should attach to M (before calling the functions), or continue being attached to M (after the functions return). **[5]**

```
void barrier_init ( barrier_t B , int m )
{
  /* Initialization of M */



     int *M;

     M = (int *)shmat(B.shmid, NULL, 0);
     M[0] = m; M[1] = 0;
     shmdt(M);
```

```
  /* Initialization of S */



     semctl(B.semid, 0, SETVAL, 1);
     semctl(B.semid, 1, SETVAL, 0);
     semctl(B.semid, 2, SETVAL, 0);

}
```

The joining function

Suppose that a process Q plans to join a barrier B. First, Q locks MTX, and increments the wait count w. If $w < m$, Q releases MTX, and starts waiting on CND. However, if $w = m$, then Q is the last process to join before the barrier is lifted. Q does not release MTX immediately. Q knows that $m - 1$ other processes are waiting (or going to wait) on CND. One by one, Q sends a signal (V() operation) to CND waking up one of the waiting processes; call it R. Q itself starts waiting on the acknowledgment semaphore ACK. The woken up process R sends a signal to ACK, and leaves its call of the join function. After this interaction completes sequentially for all of the $m - 1$ waiting processes, Q resets the barrier to its last initialized state (there is no need to reinitialize a barrier if m remains the same), releases MTX, and returns from its call of the join function. This prolonged holding of MTX by Q prevents processes from rejoining the barrier before it is completely lifted. You need to implement the P() and V() operations on semaphores. **[10]**

```
void barrier_join ( barrier_t B )
{
   /* Local variables and initial bookkeeping */



      int *M, i;
      struct sembuf Pbuf, Vbuf;

      Pbuf.sem_num = 0;
      Pbuf.sem_op = -1; Pbuf.sem_flg = 0;
      Vbuf.sem_num = 0;
      Vbuf.sem_op = 1; Vbuf.sem_flg = 0;

      semop(B.semid, &Pbuf, 1);
      M = (int *)shmat(B.shmid, NULL, 0);
     ++M[1];



   if ( M[1] < M[0]) ) { /* the case w < m */



         semop(B.semid, &Vbuf, 1);
         Pbuf.sem_num = 1;
         semop(B.semid, &Pbuf, 1);
         Vbuf.sem_num = 2;
         semop(B.semid, &Vbuf, 1);
```

```
   } else { /* the case w = m */



        for (i=1; i<M[0]; ++i) {
              Vbuf.sem_num = 1;
              Pbuf.sem_num = 2;
              semop(B.semid, &Vbuf, 1);
              semop(B.semid, &Pbuf, 1);
           }
           M[1] = 0;
           Vbuf.sem_num = 0;
           semop(B.semid, &Vbuf, 1);



   }
   /* Final works (if any) */

       shmdt(M);

}
```

(b) Write an application program that uses the barrier library developed in Part (a). The program accepts three command-line arguments: n (the number of child processes), m (for the initialization of a barrier B), and l (the number of lifts of B). Assume that $m \leqslant n$. The parent process P_0 initializes a barrier B to m, and forks n child processes $P_1, P_2, \ldots, P_n$. Each child process enters a loop in which it gets engaged in lifting the barrier for a total of l times. The body of the loop makes a single call of `barrier_join`(B). As soon as m of the n child processes join the barrier B, it is lifted, and is ready again to wait for the next m joins. After a total of ml joins (by all child processes), the l lifts of the barrier are complete, so all the child processes, and subsequently the parent process too can exit.

Here, ml need not be a multiple of n, so different child processes may be involved in different numbers of lifts of B. This may also be the case even when ml is a multiple of n (because of scheduling which is beyond the control of user processes). User programs interact with any library only through the library calls. Our application program must not use M and S except via the four functions of Part (a). It externally maintains a shared integer count `join_cnt` to store how many join calls are made (in total) by all the child processes. As soon as this count reaches ml, no child process makes any further attempt to join B. Since `join_cnt` is a shared variable, a semaphore JMTX should be used to avoid race conditions to update this count.

The application program should create a shared-memory segment to store `join_cnt`, and a semaphore set of size 1 to implement JMTX. The creation and the removal of these IPC resources are done by the parent process. Immediately after forking, each child jumps to a function `join_loop`() which does not return. A sample run of the program with $n = 4$, $m = 3$, and $l = 2$ is given to the right. It is fine if the printing is a bit garbled.

```
+++ Process 1 is going to join barrier
+++ Process 2 is going to join barrier
+++ Process 3 is going to join barrier
                +++ Process 1 has left barrier
+++ Process 4 is going to join barrier
                +++ Process 2 has left barrier
                +++ Process 3 has left barrier
+++ Process 3 is going to join barrier
+++ Process 1 is going to join barrier
        +++ Process 2 took part in 1 lifts
                +++ Process 4 has left barrier
                +++ Process 3 has left barrier
                +++ Process 1 has left barrier
        +++ Process 3 took part in 2 lifts
        +++ Process 1 took part in 2 lifts
        +++ Process 4 took part in 1 lifts
```

First, write the `main`() function (this is run by the parent P_0). **[10]**

```
barrier_t B;                /* the barrier */
int n = 7, m = 5, l = 3;    /* default values of n, m, and l */
int shmid, semid, *J;       /* IPC resources for the application program */

int main ( int argc , char *argv[] )
{
   /* If the user supplies n, m, and l in the command line, overwrite the default values */


   if (argc > 1) n = atoi(argv[1]);
   if (argc > 2) m = atoi(argv[2]);
   if (argc > 3) l = atoi(argv[3]);


   /* Create and initialize the barrier B */


   B = barrier_create(IPC_PRIVATE, IPC_PRIVATE);   /* You may use ftok() although not needed for this program */
   barrier_init(B, m);


    /* Create and initialize the IPC resources for join_cnt and JMTX */




   shmid = shmget(IPC_PRIVATE, sizeof(int), 0666 | IPC_CREAT);   /* ftok() can be used */
   J = (int *)shmat(shmid, NULL, 0);
   *J = 0;
   shmdt(J);

   semid = semget(IPC_PRIVATE, 1, 0666 | IPC_CREAT);             /* ftok() can be used */
   semctl(semid, 0, SETVAL, 1);



   /* Fork n child processes. Each child jumps to join_loop() which will not return to main() */


   for (i=1; i<=n; ++i) {
      if (!fork()) join_loop(i);
   }
```

```
   /* Wait for the child processes to terminate */

   for (i=0; i<n; ++i) wait(NULL);

   /* Remove the IPC resources including the barrier */

   barrier_destroy(B);
   shmctl(shmid, IPC_RMID, NULL);
   semctl(semid, 0, IPC_RMID);

   exit(0);
}
```

Then, write the function `join_loop()` run by each child process. The function would use a `while (1)` loop in an attempt to join the barrier. As soon as `join_cnt` becomes *ml*, the loop is broken. Use JMTX for the protection of this count. Implement the *P*() and *V*() operations on JMTX. Do <u>not</u> use any `sleep()` or `usleep()` delays. **[10]**

```
int join_loop ( int myid )
{

    int t, *J, endloop, mycount = 0;
    struct sembuf b;

    J = (int *)shmat(shmid, NULL, 0);
    b.sem_num = 0; b.sem_flg = 0;

    t = m * l; endloop = 0;
    while (1) {
       b.sem_op = -1;
       semop(semid, &b, 1);
       if (*J == t) endloop = 1; else ++(*J);
       b.sem_op = 1;
       semop(semid, &b, 1);
       if (endloop) break;
       printf("+++ Process %d is going to join barrier\n", myid);
       ++mycount;
       barrier_join(B);
       printf("\t\t\t\t\t+++ Process %d has left barrier\n", myid);
    }

    shmdt(J);

    printf("\t\t\t+++ Process %d took part in %d lifts\n", myid, mycount);

   exit(0);
}
```

2. [*Multi-threaded programming using the pthread API*]

In this question, you implement the bounded-buffer problem (also called the producer-consumer problem) using the pthread API. The master process creates *p* producer threads and *c* consumer threads. A bounded buffer *B* capable of storing *s* items (assumed integers) is used. *B* is to be implemented as a queue in an array of wrap-around type.

```
Producer ( i )
{
    while (1) {
        Produce the next item (only one).
        Wait so long as the buffer is full.
        Lock the buffer.
        Insert the next item to the buffer (enqueue).
        Unlock the buffer.
        Signal that the buffer is no longer empty.
    }
}
```

```
Consumer ( j )
{
    while (1) {

        Wait so long as the buffer is empty.
        Lock the buffer.
        Extract one item from the buffer (dequeue).
        Unlock the buffer.
        Signal that the buffer is no longer full.
    }
}
```

In order that the program terminates, assume that each producer produces *c* items (one at a time), and that each consumer consumes *p* items (one at a time), so the total number of items produced/consumed is *pc*. Each producer (or consumer) terminates after it produces (or consumes) *c* (or *p*) items. Rewrite the above while (1) loops accordingly.

In the theory class, we have seen a semaphore-based solution to the bounded-buffer problem. But pthread does not support counting semaphores. Moreover, a mutex lock can be released only by the thread that acquires it. The locking and unlocking of the global data (the buffer *B* and associated variables) can be achieved by a mutex MTX. The waits on full and empty buffers are to be implemented using two condition variables FULL and EMPTY. These should be used in conjunction with the same mutex MTX used for guarding the critical sections.

At the end of the loop body, each producer sends a single signal to EMPTY. If no consumer is waiting on EMPTY, the signal is lost, but that does not hurt. Otherwise, only one waiting consumer is woken up. That is fine too because only one item is written to *B*. A single signal to FULL at the end of the consumer loop has similar effects. However, there may be a sequence when two items are produced, and two signals to EMPTY break the waits of two consumers. It is possible that one of these consumers consumes both the items, making the buffer empty again. So the second consumer must restart waiting. A similar situation is possible when two producers wake up (together) by two signals to FULL. The waits on FULL and EMPTY should therefore be in loops (not if statements).

A sample run with $s = 2$, $p = 3$, and $c = 4$ is shown to the right. In case of multiple waits in the loop on FULL or EMPTY, the program prints “waits again” (underlined in the sample).

```
Producer 1 writes 383 to B[0]
Producer 1 writes 777 to B[1]
Producer 1 waits (item = 915) because buffer is full
Producer 3 waits (item = 793) because buffer is full
Producer 2 waits (item = 886) because buffer is full
         Consumer 2 reads 383 from B[0]
         Consumer 2 reads 777 from B[1]
         Consumer 2 waits because buffer is empty
Producer 1 writes 915 to B[0]
Producer 1 writes 335 to B[1]
         Consumer 1 reads 915 from B[0]
         Consumer 4 reads 335 from B[1]
         Consumer 3 waits because buffer is empty
Producer 3 writes 793 to B[0]
Producer 3 writes 386 to B[1]
Producer 3 waits (item = 492) because buffer is full
         Consumer 1 reads 793 from B[0]
         Consumer 1 reads 386 from B[1]
         Consumer 4 waits because buffer is empty
Consumer 2 waits again because buffer is empty
Consumer 3 waits again because buffer is empty
Producer 2 writes 886 to B[0]
Producer 2 writes 649 to B[1]
Producer 2 waits (item = 421) because buffer is full
Producer 3 waits again (item = 492) because buffer is full
         Consumer 2 reads 886 from B[0]
Producer 2 writes 421 to B[0]
Producer 2 waits (item = 362) because buffer is full
         Consumer 4 reads 649 from B[1]
         Consumer 4 reads 421 from B[0]
Producer 2 writes 362 to B[1]
         Consumer 3 reads 362 from B[1]
         Consumer 3 waits because buffer is empty
Producer 3 writes 492 to B[0]
Producer 3 writes 27 to B[1]
         Consumer 3 reads 492 from B[0]
         Consumer 3 reads 27 from B[1]
```

The program uses the following global variables.

```
int *B,            /* B is the buffer used as a queue implemented as a circular array. */
    front, back,   /* The deletion and insertion indices for the queue are front and back. */
    n,             /* n is the number of items waiting in the buffer to be consumed. */
    s,             /* s is the size (capacity) of the buffer */
    p, c;          /* p is the number of producers, and c is the number of consumers. */
```

Globally declare the synchronization primitives MTX, FULL, and EMPTY, and initialize them to default values. **[6]**

```
pthread_mutex_t MTX = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t FULL = PTHREAD_COND_INITIALIZER;
pthread_cond_t EMPTY = PTHREAD_COND_INITIALIZER;
```

Write the `main()` function that implements the working of the master (or main) thread. **[12]**

```
int main ( )
{
    /* Declare local variables if any */

     int i;
     pthread_t PTID[MAX_PROD], CTID[MAX_CONS];
     int parg[MAX_PROD], carg[MAX_CONS];

    scanf("%d%d%d", &s, &p, &c);

    /* Initialize B (int array of size s), n, front, and back */

     B = (int *)malloc(s * sizeof(int));
     n = 0; front = 0; back = s - 1;

    /* Create p producer and c consumer threads. Producers jump to pmain() and consumers to cmain(). */

     for (i=0; i<p; ++i) {
        parg[i] = i + 1;
        pthread_create(PTID + i, NULL, pmain, (void *)(parg + i));
     }

     for (i=0; i<c; ++i) {
        carg[i] = i + 1;
        pthread_create(CTID + i, NULL, cmain, (void *)(carg + i));
     }

    /* Wait for the producers and the consumers to terminate */

     for (i=0; i<p; ++i) pthread_join(PTID[i], NULL);

     for (i=0; i<c; ++i) pthread_join(CTID[i], NULL);

    /* Exit */

     pthread_exit(NULL);   /* Redundant */
     exit(0);              /* Needed to set the exit status of the process */

}
```

Now, write the functions `pmain()` for each producer thread, and `cmain()` for each consumer thread. **[12]**

```
void * pmain ( void *arg )
{
   int myid, i, item, firsttime ;

   myid = *((int *)arg); ;

   /* Producer loop */
   for (i=0; i<c; ++i) {
      item = rand() % 1000; /* Generate an item */
      /* Conditional wait */

      pthread_mutex_lock(&MTX);
      firsttime = 1;
      while (n == s) {
         if (firsttime) printf("Producer %d waits
            (item = %d) because buffer is full\n",
            myid, item);
         else printf("Producer %d waits again
            (item = %d) because buffer is full\n",
            myid, item);
         firsttime = 0;
         pthread_cond_wait(&FULL,&MTX);
      }

      /* Write item to buffer */

      back = (back + 1) % s;
      B[back] = item;
      ++n;
      printf("Producer %d writes %d to B[%d]\n",
              myid, item, back);

      /* send signal to EMPTY */

      pthread_cond_signal(&EMPTY);
      pthread_mutex_unlock(&MTX);

   }
   /* Exit */

   pthread_exit(NULL);

}
```

```
void * cmain ( void *arg )
{
   int myid, i, item, firsttime ;

   myid = *((int *)arg); ;

   /* Consumer loop */
   for (i=0; i<p; ++i) {

      /* Conditional wait */

      pthread_mutex_lock(&MTX);
      firsttime = 1;
      while (n == 0) {
         if (firsttime) printf("\tConsumer %d waits
             because buffer is empty\n", myid);
         else printf("\tConsumer %d waits again
            because buffer is empty\n", myid);
         firsttime = 0;
         pthread_cond_wait(&EMPTY,&MTX);
      }

      /* Read item from buffer */

      item = B[front];
      printf("\tConsumer %d reads %d from B[%d]\n",
              myid, item, front);
      front = (front + 1) % s;
      --n;

      /* send signal to FULL */

      pthread_cond_signal(&FULL);
      pthread_mutex_unlock(&MTX);

   }
   /* Exit */

   pthread_exit(NULL);

}
```

Rough Work