CS39002 Operating Systems Laboratory
Lab Test 2
12-Apr-2025, 6:15pm – 7:15pm
Maximum Marks: 60

Roll No: _____

Name: _____

---

1. Consider the bounded-buffer problem with a buffer B capable of storing n items, with p producers, and with c consumers. The buffer B is a circular queue of integers in the shared memory, stored in an array of size $n + 2$. The first n indices store the queue (in a circular wraparound fashion), $B[n]$ stores the index of the front of the queue, and $B[n + 1]$ stores the index of the back (or rear) of the queue. Each producer produces c items, and each consumer consumes p items (so the total number of items produced/consumed is pc). Each consumer can consume items produced by any producer(s). The solution covered in the class is reproduced below. One `mutex` and two semaphores `empty` and `full` are used in the solution.

```
/* shared items */
int n, p, c;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0;
```

```
/* code for each producer */
    repeat c times:
        produce item;
        P(empty);
        P(mutex);
        Insert item to B;
        Update B[n+1];
        V(mutex);
        V(full);
```

```
/* code for each consumer */
    repeat p times:
        P(full);
        P(mutex);
        Extract item from B;
        Update B[n];
        V(mutex);
        V(empty);
```

A single C program is to be written to implement the above code. The parent process starts by reading n, p, and c from the user (or from command-line arguments). The process then creates and initializes the IPC resources. A shared-memory segment is to be created to accommodate the array B of $n + 2$ int variables (see `shmid` below). The three semaphores `mutex`, `empty`, and `full` are created as a single semaphore set storing these semaphores at indices 0, 1, and 2, respectively. Finally, the parent process creates p producer and c consumer processes, and waits for these child processes to terminate. Write the initialization steps (`shmid` and `semid`) first. **System V shared memory** and **semaphores** must be used. No credit for using any other type of IPC primitives.

```
int n, p, c, *B, i, j, item;
int shmid, semid;

scanf("%d%d%d", &n, &p, &c);
/* create shared-memory segment shmid and attach to B */
```

[2]

```
shmid = shmget(IPC_PRIVATE, (n + 2) * sizeof(int), 0777 | IPC_CREAT | IPC_EXCL);
B = (int *)shmat(shmid, NULL, 0);
```

```
/* initialize the front and back of the queue */
```

[2]

```
B[n] = 0; B[n + 1] = n - 1;

/* create a semaphore set semid with three semaphores, and initialize the semaphores as indicated above */
```

[4]

```
semid = semget(IPC_PRIVATE, 3, 0777 | IPC_CREAT | IPC_EXCL);
semctl(semid, 0, SETVAL, 1);  /* mutex */
semctl(semid, 1, SETVAL, n);  /* empty semaphore */
semctl(semid, 2, SETVAL, 0);  /* full semaphore */
```

Write the functions `P(semid, semno)` and `V(semid, semno)` to implement the wait and the signal operations on the `semno`-th semaphore in the semaphore set `semid`. Use the syntax of System V semaphores. **[2 + 2]**

```
P ( int semid, int semno )
{

    struct sembuf pbuf;

    pbuf.sem_num = semno;
    pbuf.sem_op = -1;
    pbuf.sem_flg = 0;
    semop(semid, &pbuf, 1);

}
```

```
V ( int semid, int semno )
{

    struct sembuf vbuf;

    vbuf.sem_num = semno;
    vbuf.sem_op = 1;
    vbuf.sem_flg = 0;
    semop(semid, &vbuf, 1);

}
```

The rest of the code is to be written below. The parent process creates the producer and the consumer processes. Each child runs its own loop of producing or consuming, and then terminates.

```
for (i=0; i<p; ++i) {

    if ( _____!fork()_____ ) { /* Producer process */                    [1]
        srand((unsigned int)getpid());
        /* Attach the shared memory to B */                             [1]

        B = (int *)shmat(shmid, NULL, 0);

        /* Production loop */
        for (i=0; i<c; ++i) {
            item = 10 + rand() % 90 ;  /* Generate a random item in the range 10 - 99 */
            printf("Producer %d produces item %d\n", i, item);
            /* Write to B */                                            [4]



            P(semid,1);
            P(semid,0);
            B[n+1] = (B[n+1] == n - 1) ? 0 : B[n+1] + 1;
            B[B[n+1]] = item;
            V(semid,0);
            V(semid,2)



        }
        /* Last works by producer */                                    [2]

        shmdt(B);
        exit(0);

    }

    for (i=0; i<c; ++i) {

    if ( _____!fork()_____ ) { /* Consumer process */                    [1]
        /* Attach the shared memory to B */                             [1]

        B = (int *)shmat(shmid, NULL, 0);

        /* Consumption loop */
        for (i=0; i<p; ++i) {
            /* Read from B */                                           [4]



            P(semid,2);
            P(semid,0);
            item = B[B[n]];
            B[n] = (B[n] == n - 1) ? 0 : B[n] + 1;
            V(semid,0);
            V(semid,1)


            printf("Consumer %d reads item %d\n", i, item);
        }
        /* Last works by consumer */                                    [2]

        shmdt(B);
        exit(0);
    }
}

/* Only the parent process comes here */
/* Wait for all the child processes to exit */                          [1]

for (i=0; i<p+c; ++i) wait(NULL);

/* Remove the IPC resources */                                          [2]


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


exit(0);
```

2. Majestique Roadlines plans a bus trip for n travelers of Bargaluru for a day's visit to the Fooysuru Palace. The bus waits until all of the n travelers arrive (this is not an OS class, so there are no absentees). Upon reaching the Fooysuru Palace, the travelers are released. Each traveler visits the palace for a duration of two to four hours (a random integer, in minutes). When all the travelers are done with the sightseeing, the bus takes them to a nearby self-help restaurant. There are three counters (named CCB, MSR, and MNT) in the restaurant serving chow-chow-bath meals, mosaranna meals, and meen-thali meals, respectively. Each traveler makes one of the four choices: skip lunch altogether or take food from one of the three counters. The counters run in parallel. However, each counter can serve only one traveler at a time. Moreover, each counter keeps track of how many travelers it serves (the restaurant needs to prepare an appropriate bill for the travel company). A traveler who does not skip lunch takes 20 to 30 minutes for finishing the lunch (and complaining about the horrible quality of food served). When all travelers are done, the bus starts its return journey back to Bargaluru.

In what follows, we write a thread-level implementation of the above situation. The master thread creates n worker threads, each simulating the behavior of a traveler. The travelers synchronize using barriers and condition variables. Moreover, the counters in the restaurant are guarded by three mutexes to ensure mutual exclusion. Fill in the empty spaces below to complete the code. The **pthread API** must be used.

The following synchronization primitives are used. These are to be initialized by the master thread.

- A barrier `BOJ` (begin of journey) that the n travelers use for boarding the bus during the onward and the return journeys. The master thread initializes it to n.
- Synchronization for moving from Fooysuru Palace to the lunch venue is to be done by a condition variable `lcv` and an associated mutex `lmtx`. The condition will be on a shared int variable `yettojoinforlunch` (initialized to n). Each traveler, after visiting the palace, decrements this value (by 1). The last traveler to decrement this value (to 0) broadcasts a signal to the condition variable.
- The three counters serving food must be protected by three mutexes called `ccbmtx`, `msrmtx`, and `mntmtx`. The counters run in parallel, so a single mutex does not capture the actual situation. These mutexes should be used to increment the three global int counts `ccbcnt`, `msrcnt`, and `mntmtx`, in a mutually exclusive manner.
- Implement the visiting and eating times by proportional usleep's (1 minute → 100ms).
- The master thread does not wait on any barrier (or condition variable). Instead after creating the traveler threads, it waits for the termination of the traveler threads using `pthread_join()`.

Declare the above synchronization primitives. Also initialize them to default settings during the declarations. The barrier `BOJ` cannot be initialized here, because the value of n is not known until `main()` reads it. **[4]**

```
#define PMI PTHREAD_MUTEX_INITIALIZER
#define PCI PTHREAD_COND_INITIALIZER

pthread_barrier_t BOJ;
pthread_cond_t lcv = PCI;
pthread_mutex_t lmtx = PMI, ccbmtx = PMI, msrmtx = PMI, mntmtx = PMI;
int ccbcnt = 0, msrcnt = 0, mntcnt = 0, yettojoinforlunch;
```

Complete the `main()` function below.

```
int n, targ[MAX_SIZE];    /* these must be local variables, targ[] stores arguments for traveler threads */
pthread_t tid[MAX_SIZE];  /* to be populated during creation of traveler threads */
scanf("%d", &n);          /* number of travelers */
/* initialize BOJ and yettojoinforlunch here */                                          [2]
```

```
pthread_barrier_init(&BOJ, NULL, n);
yettojoinforlunch = n;
```

```
/* create n traveler threads. The function for each traveler thread will be tmain(). */  [4]
```

```
for (i=0; i<n; ++i) {
    targ[i] = i + 1;
    pthread_create(tid+i, NULL, tmain, (void *)(targ + i));
}
```

```
/* wait for the traveler threads to terminate */                                         [2]
```

```
for (int i=0; i<n; ++i) pthread_join(tid[i],NULL);

printf("%d %d %d\n", ccbcnt, msrcnt, mntcnt);
```

Now, complete the function `tmain()` for each traveler thread.

```
        void *                                    void *arg
______________________ tmain ( ___________________________________________ )      [1]
{
    int i;                       /* thread number passed by the master thread */
    int vtime, ltime, lchoice; /* visit time, lunch time, choice of lunch */
    /* set i from the argument of this function (no compiler warning allowed) */   [1]
    i = *((int *)arg);

    /* wait on BOJ for onward journey */                                             [1]

    pthread_barrier_wait(&BOJ);

    vtime = 120 + rand() % 121;
    /* visiting palace: sleep for scaled down vtime */                               [1]
    usleep(vtime * 100000);

    /* wait on or send broadcast signal to the condition variable lcv */            [4]

    pthread_mutex_lock(&lmtx);
    --joinforlunch;
    if (joinforlunch > 0) pthread_cond_wait(&lcv, &lmtx);
    else pthread_cond_broadcast(&lcv);
    pthread_mutex_unlock(&lmtx);

    /* Proceed to lunch */
    lchoice = rand() % 4;
    if (lchoice == 0) ltime = 0; /* skip lunch */
    else ltime = 20 + rand() % 11;
    switch (lchoice) {
        case 1: /* chow chow bath */                                                 [2]

            pthread_mutex_lock(&ccbmtx);
            ++ccbcnt;
            pthread_mutex_unlock(&ccbmtx);
            break;

        case 2:  /* mosaranna */                                                     [2]

            pthread_mutex_lock(&msrmtx);
            ++msrcnt;
            pthread_mutex_unlock(&msrmtx);
            break;

        case 3:  /* meen thali (IITKGP professors join this counter) */             [2]

            pthread_mutex_lock(&mntmtx);
            ++mntcnt;
            pthread_mutex_unlock(&mntmtx);
            break;

    }
    /* complete lunch: simulate a delay of ltime (scaled down) */                    [1]
    usleep(ltime * 100000);

    /* wait on BOJ for the return journey */                                         [1]

    pthread_barrier_wait(&BOJ);

    /* explicit exit of traveler thread */                                           [1]
    pthread_exit(NULL);
}
```