QUESTION #5 (10 minutes) – 20 points

The dinning philosopher problem is one of the classical process synchronization problems, in which multiple processes simulate dinning philosophers. Multiple ($N$) philosophers are dining on a round table, on which $N$ forks exist (as shown below). The $N$ philosophers share the forks. Each philosopher, who is simulated by a process, needs two forks (one in his left and the other in his right).

Consider the procedure `put_forks` in the solution of the dining philosopher problem (attached at the appendix (APPENDIX) of this exam question). Suppose that the variable `state[i]` was set to `THINKING` after the two calls to `test`, rather than before. How would this change affect the solution?

Questions:

First, select the best solution from the following choices:

(a) No effect (nothing will be changed)
(b) Starvation (some philosophers can starve to death)
(c) Deadlock (all the philosophers will be dead because of a possible deadlock).
(d) Assign a fork to more than one philosopher at a time.

Then, prove your choice by showing an example, if you select option (b), (c), or (d). If you select (a), explain why no effect. **Note:** “up” means “signal”, while “down” means “wait” for semaphore operations.

Each of you is suggested to try this particular question.
APPENDIX (for QUESTION #5):

```c
#define N                  5                             /* number of philosophers */
#define LEFT           (i+N-1) % N           /* number of i’s left neighbor */
#define RIGHT        (i+1) %N                /*  number of i’s right neighbor */
#define THINKING  0                            /* Label for “thinking” state */
#define HUNGRY    1                            /* Label for “hungry” state */
#define EATING      2                            /* Label for “eating” state */

type\_\_def int semaphore;
int state[N];
semaphore mutex = 1;                        /* “mutex” as a binary semaphore */
semaphore S[N];                                    /* One semaphore for each philosopher */

void philosopher (int i)  {                     /* i: philosopher #, from 0 to (N-1) */
    while (TRUE) {                              /* each philosopher loops */
        think ( );                                    /* a philosopher thinks for some time */
        take\_forks (i);                           /* acquire two forks or block */
        eat ( );                                       /* a philosopher eats (the critical section) */
        put\_forks (i);                            /* release the two forks */
    }
}

void take\_forks (int i)   {              /* i: philosopher #, from 0 to (N-1) */
    down (mutex);                        /* enter the critical section */
    state[i] = HUNGRY;              /* philosopher i is now hungry */
    test (i);                                    /* call “test” function */
    up (mutex);                            /* exit the critical section */
    down (S[i]);                            /* blocks if the two forks were not acquired */
}

void put\_forks (int i)     {              /* i: philosopher #, from 0 to (N-1) */
    down (mutex);                        /* enter the critical section */
    state[i] = THINKING;           /* philosopher i finishes eating (thus start thinking) */
    test (LEFT);                           /* see if the left neighbor can eat */
    test (RIGHT);                        /* see if the right neighbor can eat */
    up (mutex);                            /* exit the critical section */
}

void test (int i)  {
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
    {
        state[i] = EATING;    /* Set the state to “EATING” */
        up(S[i]);                     /* Unblock the philosopher (so that it can actually start eating) */
    }
}
```
QUESTION #5 (20 Minutes) – 30 points

The following is the solution for “Producer & Consumer Problem” we built on February 15th (the one that prevents the race condition and spin-waits).

Problem: CFQ (Circular Fifo Queue)

BASE SOLUTION: “SOLUTION #2”

<table>
<thead>
<tr>
<th>Shared memory (&quot;shm&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int Top = 0;</td>
</tr>
<tr>
<td>int Tail = 0;</td>
</tr>
<tr>
<td>int empty = N;</td>
</tr>
</tbody>
</table>

SEMAPHOR S = 1;  // binary semaphore
SEMAPHOR EMPTY = N;  // counting semaphore
SEMAPHOR FULL = 0;  // counting semaphore

void producer (void)
{
    int new_item;         // a place holder for a new item
    for (i = 0; i < NUM_REPEAT; ++i)
    {
        new_item = rand();  // generate a piece of data
        shm->CFQ[shm->Tail] = new_item;  // insert the new item
        shm->Tail = (shm->Tail + 1) % N;  // update the tail pointer
        shm->empty = shm->empty - 1;  // empty slots is decreased by one
        wait(FULL);  // if full, wait
    }
}

void consumer (void)
{
    int new_item;         // a place holder for a new item
    for (i = 0; i < NUM_REPEAT; ++i)
    {
        wait(FULL);  // if full, wait
        new_item = shm->CFQ[shm->Top];  // remove the first item in the CFQ
        shm->Top = (shm->Top + 1) % N;  // update the top pointer
        shm->empty = shm->empty + 1  // empty slots is decreased by one
        wait(S);  // if S is less than 1, wait
        signal(S);  // signal S
        signal(EMPTY);  // signal the empty slot
        do_something(new_item);
    }
}

Producer Process
Consumer Process

Assuming:

① We have only one producer and one consumer

② The computer system has two active processors (or has a multi-core processor that has two processor cores in it).

Question #1: Is it possible to eliminate “S” (mutex) semaphore?

(a) If yes, explain “how” and why is it OK (i.e., prove how it is OK)?

(b) If no, explain why not (i.e., prove how it is not OK)?

Note: Either way, you are expected to prove your conclusion.

[continued to the next page]
Question #2: If it is possible to eliminate “S” (mutex) semaphore, is there any merit (advantage) in eliminating the semaphore?

(a) If yes, explain “why” (or “how is it an advantage”).

(b) If not, explain why not.

Solution:

Question 1: (a) – possible to eliminate semaphore S because of the following reasons:

1. The only process that modifies “Tail” is the producer process (i.e., the consumer process never modified “Tail”).
2. The only process that modifies “Top” is the consumer process (i.e., the producer process never modified “Top”).
3. The common variable, “empty” can be dropped (neither the producer or the consumer process uses it). If it is dropped, it can never be a problem.
4. Although both of the producer and the consumer processes still access (update) “CRQ” in the shared memory, since Top and Tail can never point to the same CRQ slot at the same time except when the CRQ is full (then, the producer can not be active) or when the CRQ is empty (then, the consumer can not be active).

The above four observations conclude that the mutex semaphore S is actually not needed.

Note: If 3 is not mentioned in a solution, it will be OK (but 1, 2, and 4 should be explained for full credit).

Question 2: (a) – there is a merit (advantage)

Eliminating the mutex semaphore (S) will allow the two processes (the producer and the consumer) to use the CRQ (i.e., the producer can add new items to CRQ at (physically) the same time the consumer removes the existing items in CRQ (unless the CRQ is full or the CRQ is empty), which will make the two processes run faster.