**1)**

# Peterson Algorithm:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h> // Include this for wait()
#include <stdbool.h>

#define BSIZE 8 // Buffer size
#define PWT 2   // Producer wait time limit
#define CWT 10  // Consumer wait time limit
#define RT 10   // Program run-time in seconds

int shmid1, shmid2, shmid3, shmid4;
key_t k1 = 5491, k2 = 5812, k3 = 4327, k4 = 3213;
bool* SHM1;
int* SHM2;
int* SHM3;

int myrand(int n) {
   return (rand() % n + 1);
}

int main() {
   // Create shared memory segments
   shmid1 = shmget(k1, sizeof(bool) * 2, IPC_CREAT | 0660); // flag
```

```
shmid2 = shmget(k2, sizeof(int) * 1, IPC_CREAT | 0660); // turn
shmid3 = shmget(k3, sizeof(int) * BSIZE, IPC_CREAT | 0660); // buffer
shmid4 = shmget(k4, sizeof(int) * 1, IPC_CREAT | 0660); // state

if (shmid1 < 0 || shmid2 < 0 || shmid3 < 0 || shmid4 < 0) {
   perror("Main shmget error");
   exit(1);
}

SHM3 = (int*)shmat(shmid3, NULL, 0);
int ix = 0;
while (ix < BSIZE) // Initializing buffer
   SHM3[ix++] = 0;

int* state = (int*)shmat(shmid4, NULL, 0);
*state = 1;

int wait_time;
int i = 0; // Consumer
int j = 1; // Producer

if (fork() == 0) { // Producer code
   SHM1 = (bool*)shmat(shmid1, NULL, 0);
   SHM2 = (int*)shmat(shmid2, NULL, 0);
   SHM3 = (int*)shmat(shmid3, NULL, 0);
   if (SHM1 == (bool*)-1 || SHM2 == (int*)-1 || SHM3 == (int*)-1) {
      perror("Producer shmat error");
      exit(1);
   }

   bool* flag = SHM1;
```

```
int* turn = SHM2;
int* buf = SHM3;
int index;

while (*state == 1) {
   flag[j] = true;
   *turn = i;
   while (flag[i] == true && *turn == i) ;

   // Critical Section Begin
   index = 0;
   while (index < BSIZE) {
      if (buf[index] == 0) {
         int tempo = myrand(BSIZE * 3);
         printf("Job %d has been produced\n", tempo);
         buf[index] = tempo;
         break;
      }
      index++;
   }
   if (index == BSIZE)
      printf("Buffer is full, nothing can be produced!!!\n");
   printf("Buffer: ");
   for (int k = 0; k < BSIZE; k++)
      printf("%d ", buf[k]);
   printf("\n");
   // Critical Section End

   flag[j] = false;
   if (*state == 0)
      break;
```

```
        wait_time = myrand(PWT);
        printf("Producer will wait for %d seconds\n\n", wait_time);
        sleep(wait_time);
    }
    exit(0);
}

if (fork() == 0) { // Consumer code
    SHM1 = (bool*)shmat(shmid1, NULL, 0);
    SHM2 = (int*)shmat(shmid2, NULL, 0);
    SHM3 = (int*)shmat(shmid3, NULL, 0);
    if (SHM1 == (bool*)-1 || SHM2 == (int*)-1 || SHM3 == (int*)-1) {
        perror("Consumer shmat error");
        exit(1);
    }

    bool* flag = SHM1;
    int* turn = SHM2;
    int* buf = SHM3;
    int index;
    flag[i] = false;
    sleep(5);
    while (*state == 1) {
        flag[i] = true;
        *turn = j;
        while (flag[j] == true && *turn == j) ;

        // Critical Section Begin
        if (buf[0] != 0) {
            printf("Job %d has been consumed\n", buf[0]);
            buf[0] = 0;
```

```
            index = 1;
            while (index < BSIZE) { // Shift remaining jobs forward
                buf[index - 1] = buf[index];
                index++;
            }
            buf[index - 1] = 0;
        } else {
            printf("Buffer is empty, nothing can be consumed!!!\n");
        }
        printf("Buffer: ");
        for (int k = 0; k < BSIZE; k++)
            printf("%d ", buf[k]);
        printf("\n");
        // Critical Section End

        flag[i] = false;
        if (*state == 0)
            break;
        wait_time = myrand(CWT);
        printf("Consumer will sleep for %d seconds\n\n", wait_time);
        sleep(wait_time);
    }
    exit(0);
}

// Parent process will wait for RT seconds before causing child to terminate
sleep(RT);
*state = 0;

// Waiting for both processes to exit
wait(NULL);
```

```
    wait(NULL);

    printf("The clock ran out.\n");

    return 0;

}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h> // Include this for wait()
#include <stdbool.h>

#define BSIZE 8 // Buffer size
#define PWT 2   // Producer wait time limit
#define CWT 10  // Consumer wait time limit
#define RT 10   // Program run-time in seconds

int shmid1, shmid2, shmid3, shmid4;
key_t k1 = 5491, k2 = 5812, k3 = 4327, k4 = 3213;
bool* SHM1;
int* SHM2;
int* SHM3;

int myrand(int n) {
    return (rand() % n + 1);
}

int main() {
    // Create shared memory segments
    shmid1 = shmget(k1, sizeof(bool) * 2, IPC_CREAT | 0660); // flag
    shmid2 = shmget(k2, sizeof(int) * 1, IPC_CREAT | 0660); // turn
    shmid3 = shmget(k3, sizeof(int) * BSIZE, IPC_CREAT | 0660); // buffer
    shmid4 = shmget(k4, sizeof(int) * 1, IPC_CREAT | 0660); // state

    if (shmid1 < 0 || shmid2 < 0 || shmid3 < 0 || shmid4 < 0) {
        perror("Main shmget error");
        exit(1);
    }

    SHM3 = (int*)shmat(shmid3, NULL, 0);
    int ix = 0;
    while (ix < BSIZE) // Initializing buffer
```

# 22BCE0682 Siddhant Bhagat

```
    while (ix < BSIZE) // Initializing buffer
        SHM3[ix++] = 0;

    int* state = (int*)shmat(shmid4, NULL, 0);
    *state = 1;

    int wait_time;
    int i = 0; // Consumer
    int j = 1; // Producer

    if (fork() == 0) { // Producer code
        SHM1 = (bool*)shmat(shmid1, NULL, 0);
        SHM2 = (int*)shmat(shmid2, NULL, 0);
        SHM3 = (int*)shmat(shmid3, NULL, 0);
        if (SHM1 == (bool*)-1 || SHM2 == (int*)-1 || SHM3 == (int*)-1) {
            perror("Producer shmat error");
            exit(1);
        }

        bool* flag = SHM1;
        int* turn = SHM2;
        int* buf = SHM3;
        int index;

        while (*state == 1) {
            flag[j] = true;
            *turn = i;
            while (flag[i] == true && *turn == i) ;

            // Critical Section Begin
            index = 0;
            while (index < BSIZE) {
                if (buf[index] == 0) {
                    int tempo = myrand(BSIZE * 3);
                    printf("Job %d has been produced\n", tempo);
                    buf[index] = tempo;
                    break;
                }
                index++;
            }
```

# 22BCE0682 Siddhant Bhagat

```
            if (index == BSIZE)
                printf("Buffer is full, nothing can be produced!!!\n");
            printf("Buffer: ");
            for (int k = 0; k < BSIZE; k++)
                printf("%d ", buf[k]);
            printf("\n");
            // Critical Section End

            flag[j] = false;
            if (*state == 0)
                break;
            wait_time = myrand(PWT);
            printf("Producer will wait for %d seconds\n\n", wait_time);
            sleep(wait_time);
        }
        exit(0);
    }

    if (fork() == 0) { // Consumer code
        SHM1 = (bool*)shmat(shmid1, NULL, 0);
        SHM2 = (int*)shmat(shmid2, NULL, 0);
        SHM3 = (int*)shmat(shmid3, NULL, 0);
        if (SHM1 == (bool*)-1 || SHM2 == (int*)-1 || SHM3 == (int*)-1) {
            perror("Consumer shmat error");
            exit(1);
        }

        bool* flag = SHM1;
        int* turn = SHM2;
        int* buf = SHM3;
        int index;
        flag[i] = false;
        sleep(5);
        while (*state == 1) {
            flag[i] = true;
            *turn = j;
            while (flag[j] == true && *turn == j) ;

```

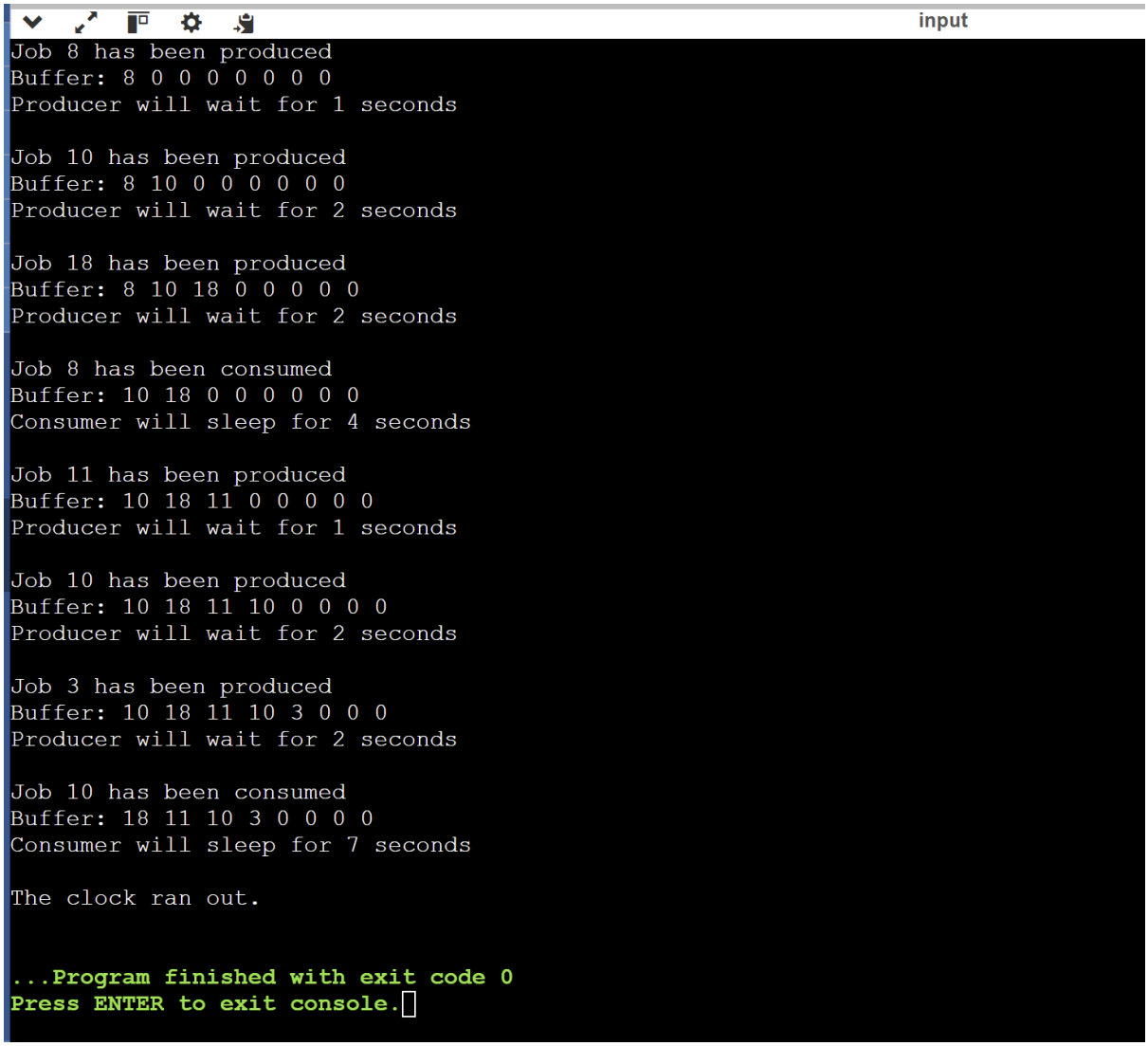
```
            // Critical Section Begin
            if (buf[0] != 0) {
                printf("Job %d has been consumed\n", buf[0]);
                buf[0] = 0;
                index = 1;
                while (index < BSIZE) { // Shift remaining jobs forward
                    buf[index - 1] = buf[index];
                    index++;
                }
                buf[index - 1] = 0;
            } else {
                printf("Buffer is empty, nothing can be consumed!!!\n");
            }
            printf("Buffer: ");
            for (int k = 0; k < BSIZE; k++)
                printf("%d ", buf[k]);
            printf("\n");
            // Critical Section End

            flag[i] = false;
            if (*state == 0)
                break;
            wait_time = myrand(CWT);
            printf("Consumer will sleep for %d seconds\n\n", wait_time);
            sleep(wait_time);
        }
        exit(0);
    }

    // Parent process will wait for RT seconds before causing child to terminate
    sleep(RT);
    *state = 0;

    // Waiting for both processes to exit
    wait(NULL);
    wait(NULL);
    printf("The clock ran out.\n");
    return 0;
}
```

## Output:




## 2)

# Critical Section Solution:

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>


#define NUM_THREADS 5


// Global variable to be accessed in the critical section

int shared_resource = 0;
```

# 22BCE0682 Siddhant Bhagat

```
// Mutex for protecting the critical section
pthread_mutex_t mutex;

// Entry section
void enter_critical_section() {
    pthread_mutex_lock(&mutex);
}

// Critical section
void access_critical_section(int thread_id) {
    printf("Thread %d is entering the critical section.\n", thread_id);
    // Simulate some work in the critical section
    int temp = shared_resource;
    sleep(1); // Simulate time-consuming task
    shared_resource = temp + 1;
    printf("Thread %d has updated shared_resource to %d.\n", thread_id, shared_resource);
}

// Exit section
void exit_critical_section() {
    pthread_mutex_unlock(&mutex);
}

void* thread_function(void* arg) {
    int thread_id = *((int*)arg);

    // Entry section
    enter_critical_section();

    // Critical section
```

```
    access_critical_section(thread_id);

    // Exit section
    exit_critical_section();

    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    // Initialize the mutex
    pthread_mutex_init(&mutex, NULL);

    // Create threads
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, thread_function, &thread_ids[i]);
    }

    // Join threads
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&mutex);

    printf("Final value of shared_resource: %d\n", shared_resource);
    return 0;
}
```

# 22BCE0682 Siddhant Bhagat

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_THREADS 5

// Global variable to be accessed in the critical section
int shared_resource = 0;

// Mutex for protecting the critical section
pthread_mutex_t mutex;

// Entry section
void enter_critical_section() {
    pthread_mutex_lock(&mutex);
}

// Critical section
void access_critical_section(int thread_id) {
    printf("Thread %d is entering the critical section.\n", thread_id);
    // Simulate some work in the critical section
    int temp = shared_resource;
    sleep(1); // Simulate time-consuming task
    shared_resource = temp + 1;
    printf("Thread %d has updated shared_resource to %d.\n", thread_id, shared_resource);
}

// Exit section
void exit_critical_section() {
    pthread_mutex_unlock(&mutex);
}

void* thread_function(void* arg) {
    int thread_id = *((int*)arg);

    // Entry section
    enter_critical_section();

```

# 22BCE0682 Siddhant Bhagat

```c
    // Critical section
    access_critical_section(thread_id);

    // Exit section
    exit_critical_section();

    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    // Initialize the mutex
    pthread_mutex_init(&mutex, NULL);

    // Create threads
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, thread_function, &thread_ids[i]);
    }

    // Join threads
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&mutex);

    printf("Final value of shared_resource: %d\n", shared_resource);
    return 0;
}
```

## Output:

```
Thread 2 is entering the critical section.
Thread 2 has updated shared_resource to 1.
Thread 1 is entering the critical section.
Thread 1 has updated shared_resource to 2.
Thread 0 is entering the critical section.
Thread 0 has updated shared_resource to 3.
Thread 3 is entering the critical section.
Thread 3 has updated shared_resource to 4.
Thread 4 is entering the critical section.
Thread 4 has updated shared_resource to 5.
Final value of shared_resource: 5


...Program finished with exit code 0
Press ENTER to exit console.
```