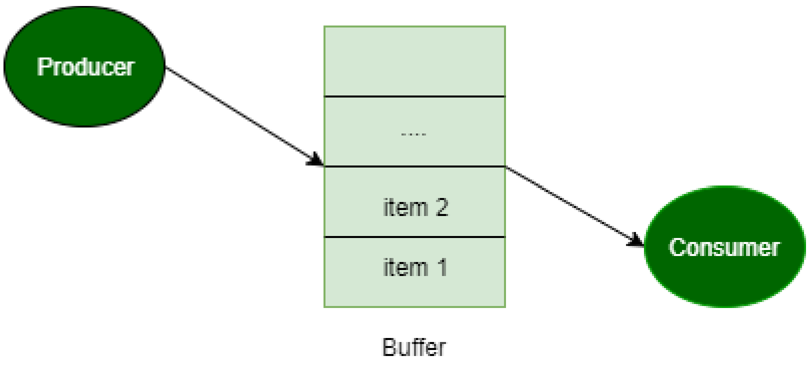
Peterson’s Algorithm in Process Synchronization

The producer-consumer problem (or bounded buffer problem) describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue. Producers produce an item and put it into the buffer. If the buffer is already full then the producer will have to wait for an empty block in the buffer. Consumers consume an item from the buffer. If the buffer is already empty then the consumer will have to wait for an item in the buffer. Implement Peterson’s Algorithm for the two processes using shared memory such that there is mutual exclusion between them. The solution should have free from synchronization problems.



Peterson’s algorithm –

C

```
// code for producer (j)

// producer j is ready
// to produce an item
flag[j] = true;

// but consumer (i) can consume an item
turn = i;

// if consumer is ready to consume an item
// and if its consumer's turn
while (flag[i] == true && turn == i)

    { /* then producer will wait*/ }

    // otherwise producer will produce
    // an item and put it into buffer (critical Section)

    // Now, producer is out of critical section
    flag[j] = false;
    // end of code for producer

    //---------------------------------------------------------
    // code for consumer i

    // consumer i is ready
    // to consume an item
    flag[i] = true;

    // but producer (j) can produce an item
    turn = j;

    // if producer is ready to produce an item
    // and if its producer's turn
    while (flag[j] == true && turn == j)

        { /* then consumer will wait */ }

        // otherwise consumer will consume
        // an item from buffer (critical Section)

        // Now, consumer is out of critical section
        flag[i] = false;
// end of code for consumer
```

Explanation of Peterson's Algorithm

Peterson's Algorithm is used to synchronize two processes. It uses two variables, a bool array **flag** of size 2 and an int variable **turn** to accomplish it. In the solution, i represents the Consumer and j represents the Producer. Initially, the flags are false. When a process wants to execute it's critical section, it sets its flag to true and turn into the index of the other process. This means that the process wants to execute but it will allow the other process to run first. The process performs busy waiting until the other process has finished it's own critical section. After this, the current process enters its critical section and adds or removes a random number from the shared buffer. After completing the critical section, it sets it's own flag to false, indicating it does not wish to execute anymore. The program runs for a fixed amount of time before exiting. This time can be changed by changing value of the macro RT.

C

```
// C program to implement Peterson's Algorithm
// for producer-consumer problem.
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```c
#include <time.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdbool.h>
#define _BSD_SOURCE
#include <sys/time.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) // Returns a random number between 1 and n
{
    time_t t;
    srand((unsigned)time(&t));
    return (rand() % n + 1);
}

int main()
{
    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); // time stamp

    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;

    struct timeval t;
    time_t t1, t2;
    gettimeofday(&t, NULL);
    t1 = t.tv_sec;
```

```
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 = 0;

    while (*state == 1) {
        flag[j] = true;
        printf("Producer is ready now.\n\n");
        *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: ");
        index = 0;
        while (index < BSIZE)
```

```
                printf("%d ", buf[index++]);
            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 = 0;
        flag[i] = false;
        sleep(5);
        while (*state == 1) {
            flag[i] = true;
            printf("Consumer is ready now.\n\n");
            *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) // Shifting 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: ");
            index = 0;
            while (index < BSIZE)
                printf("%d ", buf[index++]);
            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 now for RT seconds before causing child to terminate
    while (1) {
        gettimeofday(&t, NULL);
        t2 = t.tv_sec;
        if (t2 - t1 > RT) // Program will exit after RT seconds
        {
            *state = 0;
            break;
        }
    }
    // Waiting for both processes to exit
    wait();
    wait();
    printf("The clock ran out.\n");
    return 0;
}
```

Output:

```
Producer is ready now.

Job 9 has been produced
Buffer: 9 0 0 0 0 0 0 0
Producer will wait for 1 seconds

Producer is ready now.

Job 8 has been produced
Buffer: 9 8 0 0 0 0 0 0
```

```
Producer will wait for 2 seconds

Producer is ready now.

Job 13 has been produced
Buffer: 9 8 13 0 0 0 0 0
Producer will wait for 1 seconds

Producer is ready now.

Job 23 has been produced
Buffer: 9 8 13 23 0 0 0 0
Producer will wait for 1 seconds

Consumer is ready now.

Job 9 has been consumed
Buffer: 8 13 23 0 0 0 0 0
Consumer will sleep for 9 seconds

Producer is ready now.

Job 15 has been produced
Buffer: 8 13 23 15 0 0 0 0
Producer will wait for 1 seconds

Producer is ready now.

Job 13 has been produced
Buffer: 8 13 23 15 13 0 0 0
Producer will wait for 1 seconds

Producer is ready now.

Job 11 has been produced
Buffer: 8 13 23 15 13 11 0 0
Producer will wait for 1 seconds

Producer is ready now.

Job 22 has been produced
Buffer: 8 13 23 15 13 11 22 0
Producer will wait for 2 seconds

Producer is ready now.

Job 23 has been produced
Buffer: 8 13 23 15 13 11 22 23
Producer will wait for 1 seconds
```

```
The clock ran out.
```

Advantages of the Peterson Solution

1. With Peterson's solution, multiple processes can access and share a resource without causing any resource conflicts.
2. Every process has a chance to be carried out.
3. It uses straightforward logic and is easy to put into practice.
4. Since it is entirely software dependent and operates in user mode, it can be used with any hardware.
   eliminates the chance of a deadlock.

Disadvantages of the Peterson's Solution

1. Waiting for the other processes to exit the critical region may take a long time. We call it busy waiting.
2. On systems that have multiple CPUs, this algorithm might not function.
3. The Peterson solution can only run two processes concurrently.