**OS\_LAB\_09\_Assignment**

**CE\_054**

**Aim :-** Study of semaphores and implementation of producer-consumer problem.

**Theory :-**

**1. sem\_init()**

=> To initialize an unnamed semaphore

**Synopsis:**

#include <semaphore.h>

int sem\_init( sem\_t \* sem, int pshared, unsigned value );

**Arguments:**

Sem => A pointer to the sem\_t object for the semaphore that you want to initialize.

Pshared => Nonzero if you want the semaphore to be shared between processes via shared memory.

Value => The initial value of the semaphore. A positive value (i.e. greater than zero) indicates an unlocked semaphore, and a value of 0 (zero) indicates a locked semaphore. This value must not exceed SEM\_VALUE\_MAX.

**Description:**

The sem\_init() function initializes the unnamed semaphore referred to by the sem argument. The initial counter value of this semaphore is specified by the value argument.

You should allocate synchronization objects only in normal memory mappings. On certain processors (e.g. some PPC ones), atomic operations such as calls to [pthread\_mutex\_lock()](http://www.qnx.com/developers/docs/qnxcar2/topic/com.qnx.doc.neutrino.lib_ref/topic/p/pthread_mutex_lock.html) will cause a fault if the control structure is allocated in uncached memory.

You can use the initialized semaphore in subsequent calls to [sem\_wait()](http://www.qnx.com/developers/docs/qnxcar2/topic/com.qnx.doc.neutrino.lib_ref/topic/s/sem_wait.html), [sem\_trywait()](http://www.qnx.com/developers/docs/qnxcar2/topic/com.qnx.doc.neutrino.lib_ref/topic/s/sem_trywait.html), [sem\_post()](http://www.qnx.com/developers/docs/qnxcar2/topic/com.qnx.doc.neutrino.lib_ref/topic/s/sem_post.html), and [sem\_destroy()](http://www.qnx.com/developers/docs/qnxcar2/topic/com.qnx.doc.neutrino.lib_ref/topic/s/sem_destroy.html). An initialized semaphore is valid until it's destroyed by the sem\_destroy() function, or until the memory where the semaphore resides is released.

If the pshared argument is nonzero, then the semaphore can be shared between processes via shared memory. Any process can then use sem with the sem\_wait(), sem\_trywait(), sem\_post() and sem\_destroy() functions.

**Return value:**

0 => Success. The semaphore referred to by sem is initialized.

-1 => An error occurred (errno is set).

**2. sem\_wait()**

=>Wait on a named or unnamed semaphore

**Synopsis:**

#include <semaphore.h>

int sem\_wait( sem\_t \* sem );

**Arguments:**

Sem => A pointer to the sem\_t object for the semaphore that you want to wait on.

**Description:**

The sem\_wait() function decrements the semaphore referred to by the sem argument. If the semaphore value is not greater than zero, then the calling process blocks until it can decrement the counter, or the call is interrupted by signal.

Some process should eventually call [sem\_post()](http://www.qnx.com/developers/docs/qnxcar2/topic/com.qnx.doc.neutrino.lib_ref/topic/s/sem_post.html) to increment the semaphore.

**Return Value:**

0 => The semaphore was successfully decremented.

-1 => The state of the semaphore is unchanged ([errno](http://www.qnx.com/developers/docs/qnxcar2/topic/com.qnx.doc.neutrino.lib_ref/topic/e/errno.html" \o "Global error variable) is set).

**3. sem\_post()**

=> Increment a named or unnamed semaphore

**Synopsis:**

#include <semaphore.h>

int sem\_post( sem\_t \* sem );

**Arguments:**

Sem => A pointer to the sem\_t object for the semaphore whose value you want to increment.

**Description:**

The sem\_post() function increments the semaphore referenced by the sem argument. If any processes are currently blocked waiting for the semaphore, then one of these processes will return successfully from its call to sem\_wait.

The process to be unblocked is determined in accordance with the scheduling policies in effect for the blocked processes. The highest priority waiting process is unblocked, and if there is more than one highest priority process blocked waiting for the semaphore, then the highest priority process that has been waiting the longest is unblocked.

The sem\_post() function is reentrant with respect to signals, and can be called from a signal handler.

**Return Value:**

0 => Success.

-1 => An error occurred (errno is set).

**Program :-**

1. Write a program to implement solution of bounded buffer producer consumer problem using semaphores.

**Code :-**

// Author : Dhruv B Kakadiya

#include<stdio.h>

#include<pthread.h>

#include<semaphore.h>

#define PC\_BUFFER 6

sem\_t mut\_sema, empty\_sema, full\_sema;

int pcbuff[PC\_BUFFER];

void\* producer(void \*args);

void\* consumer(void \*args);

int main()

{

    pthread\_t pth, cth;

    if (sem\_init(&mut\_sema, 0, PC\_BUFFER) < 0)

    {

        printf("---Error in mut\_sema---");

    }

    if (sem\_init(&empty\_sema, 0, PC\_BUFFER) < 0)

    {

        printf("---Error in empty\_Sema---");

    }

    if (sem\_init(&full\_sema, 0, 0) < 0)

    {

        printf("--Error in full\_sema");

    }

    pthread\_create(&pth, NULL, producer, NULL);

    pthread\_create(&cth, NULL, consumer, NULL);

    pthread\_join(pth, NULL);

    pthread\_join(cth, NULL);

}

void\* producer(void\* args)

{

    int i = 0;

    int j = 0;

    while(1)

    {

        sem\_wait(&empty\_sema);

        sem\_wait(&mut\_sema);

        printf("Producer produced %d items at %d\n", i, j);

        i++;

        j = i % PC\_BUFFER;

        sem\_post(&mut\_sema);

        sem\_post(&full\_sema);

    }

}

void\* consumer(void\* args)

{

    int i = 0;

    int j = 0;

    while(1)

    {

        sem\_wait(&full\_sema);

        sem\_wait(&mut\_sema);

        printf("Consumer consumed %d items at %d\n", i, j);

        i++;

        j = i % PC\_BUFFER;

        sem\_post(&mut\_sema);

        sem\_post(&empty\_sema);

    }

}

Output :-







