**OSD Practical-12**

**Pre-Lab**

1. **Creating and Destroying Condition Variables: pthread\_cond\_init (condition,attr) pthread\_cond\_destroy (condition)**

**Ans:**

The **pthread\_cond\_destroy()** function shall destroy the given condition variable specified by cond; the object becomes, in effect, uninitialized. An implementation may cause pthread\_cond\_destroy() to set the object referenced by cond to an invalid value. A destroyed condition variable object can be reinitialized using pthread\_cond\_init(); the results of otherwise referencing the object after it has been destroyed are undefined.

It shall be safe to destroy an initialized condition variable upon which no threads are currently blocked. Attempting to destroy a condition variable upon which other threads are currently blocked results in undefined behavior.

The **pthread\_cond\_init()** function shall initialize the condition variable referenced by cond with attributes referenced by attr. If attr is NULL, the default condition variable

attributes shall be used; the effect is the same as passing the address of a default condition variable attributes object. Upon successful initialization, the state of the condition variable shall become initialized.

Only cond itself may be used for performing synchronization. The result of referring to copies of cond in calls to [pthread\_cond\_wait()](https://pubs.opengroup.org/onlinepubs/009604499/functions/pthread_cond_wait.html), [pthread\_cond\_timedwait()](https://pubs.opengroup.org/onlinepubs/009604499/functions/pthread_cond_timedwait.html), [pthread\_cond\_signal()](https://pubs.opengroup.org/onlinepubs/009604499/functions/pthread_cond_signal.html), [pthread\_cond\_broadcast()](https://pubs.opengroup.org/onlinepubs/009604499/functions/pthread_cond_broadcast.html), and pthread\_cond\_destroy() is undefined.

Attempting to initialize an already initialized condition variable results in undefined behavior.

In cases where default condition variable attributes are appropriate, the macro PTHREAD\_COND\_INITIALIZER can be used to initialize condition variables that are statically allocated. The effect shall be equivalent to dynamic initialization by a call to pthread\_cond\_init() with parameter attr specified as NULL, except that no error checks are performed

1. **Waiting and Signaling on Condition Variables: pthread\_cond\_wait (condition,mutex) pthread\_cond\_signal (condition) pthread\_cond\_broadcast (condition)**

**Ans:**

**pthread\_cond\_wait  Syntax**

int pthread\_cond\_wait(pthread\_cond\_t \*restrict cv,pthread\_mutex\_t \*restrict mutex);

#include <pthread.h>

pthread\_cond\_t cv;

pthread\_mutex\_t mp;

int ret;

/\* wait on condition variable \*/

ret = pthread\_cond\_wait(&cv, &mp);

The blocked thread can be awakened by a pthread\_cond\_signal() , a pthread\_cond\_broadcast() , or when interrupted by delivery of a signal.

Any change in the value of a condition that is associated with the condition variable cannot be inferred by the return of pthread\_cond\_wait(). Such conditions must be reevaluated.

The pthread\_cond\_wait() routine always returns with the mutex locked and owned by the calling thread, even when returning an error.

This function blocks until the condition is signaled. The function atomically releases the associated mutex lock before blocking, and atomically acquires the mutex again before returning.

In typical use, a condition expression is evaluated under the protection of a mutex lock. When the condition expression is false, the thread blocks on the condition variable. The condition variable is then signaled by another thread when the thread changes the condition value. The change causes at least one thread that is waiting on the condition variable to unblock and to reacquire the mutex.

The condition that caused the wait must be retested before continuing execution from the point of the pthread\_cond\_wait(). The condition could change before an awakened thread reacquires the mutes and returns from pthread\_cond\_wait(). A waiting thread could be awakened spuriously. The recommended test method is to write the condition check as a while() loop that calls pthread\_cond\_wait().

pthread\_mutex\_lock();

while(condition\_is\_false)

pthread\_cond\_wait();

pthread\_mutex\_unlock();

The scheduling policy determines the order in which blocked threads are awakened. The default scheduling policy, SCHED\_OTHER, does not specify the order in which threads are awakened. Under the SCHED\_FIFO and SCHED\_RR real-time scheduling policies, threads are awakened in priority order.

### pthread\_cond\_signal Syntax

int pthread\_cond\_signal(pthread\_cond\_t \*cv);

#include <pthread.h>

pthread\_cond\_t cv;

int ret;

/\* one condition variable is signaled \*/

ret = pthread\_cond\_signal(&cv);

Modify the associated condition under the protection of the same mutex used with the condition variable being signaled. Otherwise, the condition could be modified between its test and blocking in pthread\_cond\_wait(), which can cause an infinite wait.

The scheduling policy determines the order in which blocked threads are awakened. The default scheduling policy, SCHED\_OTHER, does not specify the order in which threads are awakened. Under the SCHED\_FIFO and SCHED\_RR real-time scheduling policies, threads are awakened in priority order.

When no threads are blocked on the condition variable, calling pthread\_cond\_signal() has no effect.

### pthread\_cond\_broadcast Syntax

int pthread\_cond\_broadcast(pthread\_cond\_t \*cv);

#include <pthread.h>

pthread\_cond\_t cv;

int ret;

/\* all condition variables are signaled \*/

ret = pthread\_cond\_broadcast(&cv);

When no threads are blocked on the condition variable, pthread\_cond\_broadcast() has no effect.

Since pthread\_cond\_broadcast() causes all threads blocked on the condition to contend again for the mutex lock, use pthread\_cond\_broadcast() with care. For example, use pthread\_cond\_broadcast() to allow threads to contend for varying resource amounts when resources are freed

**INLAB**

1. **Solve producer consumer problem using mutex, binary and counting semaphores, and condition variables.**

**Ans:**

/\* **prodcons-mutex.c** - Producer Consumer problem using mutex and pthreads \*/

/\* include main \*/

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#include <pthread.h>

#include <sys/types.h>

#define MAXNITEMS 1000000

#define MAXNTHREADS 100

int nitems; /\* read-only by producer and consumer \*/

struct {

pthread\_mutex\_t mutex;

int buff[MAXNITEMS];

int nput;

int nval;

} shared ={PTHREAD\_MUTEX\_INITIALIZER};

void \*produce(void \*), \*consume(void \*);

int

main(int argc, char \*\*argv)

{

shared.nput=0;

shared.nval=0;

int i, nthreads, count[MAXNTHREADS];

pthread\_t tid\_produce[MAXNTHREADS], tid\_consume;

if (argc != 3)

printf("usage: prodcons1 <#items> <#threads>");

nitems = atoi(argv[1]);

nthreads = atoi(argv[2]);

pthread\_setconcurrency(nthreads);

/\* 4start all the producer threads \*/

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

count[i] = 0;

pthread\_create(&tid\_produce[i], NULL, produce, &count[i]);

}

/\* 4wait for all the producer threads \*/

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

pthread\_join(tid\_produce[i], NULL);

printf("count[%d] = %d\n", i, count[i]);

}

/\* 4start, then wait for the consumer thread \*/

pthread\_create(&tid\_consume, NULL, consume, NULL);

pthread\_join(tid\_consume, NULL);

exit(0);

}

/\* end main \*/

/\* include producer \*/

void \*

produce(void \*arg)

{

pthread\_t tid;

int i=\*((int \*) arg);

for ( ; ; ) {

pthread\_mutex\_lock(&shared.mutex);

tid=pthread\_self();

printf("threadid=%u\n", (unsigned int) tid);

if (shared.nput >= nitems) {

pthread\_mutex\_unlock(&shared.mutex);

return(NULL); /\* array is full, we're done \*/

}

shared.buff[shared.nput] = shared.nval;

printf("buff[%d] = %d\n", shared.nput, shared.buff[shared.nput]);

shared.nput++;

shared.nval++;

\*((int \*) arg) += 1;

pthread\_mutex\_unlock(&shared.mutex);

printf("shared.nput=%d, shared.nval=%d,count[%u] = %d\n",shared.nput,shared.nval, i, \*((int \*) arg));

}

}

void \* consume(void \*arg)

{

int i;

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

if (shared.buff[i] != i)

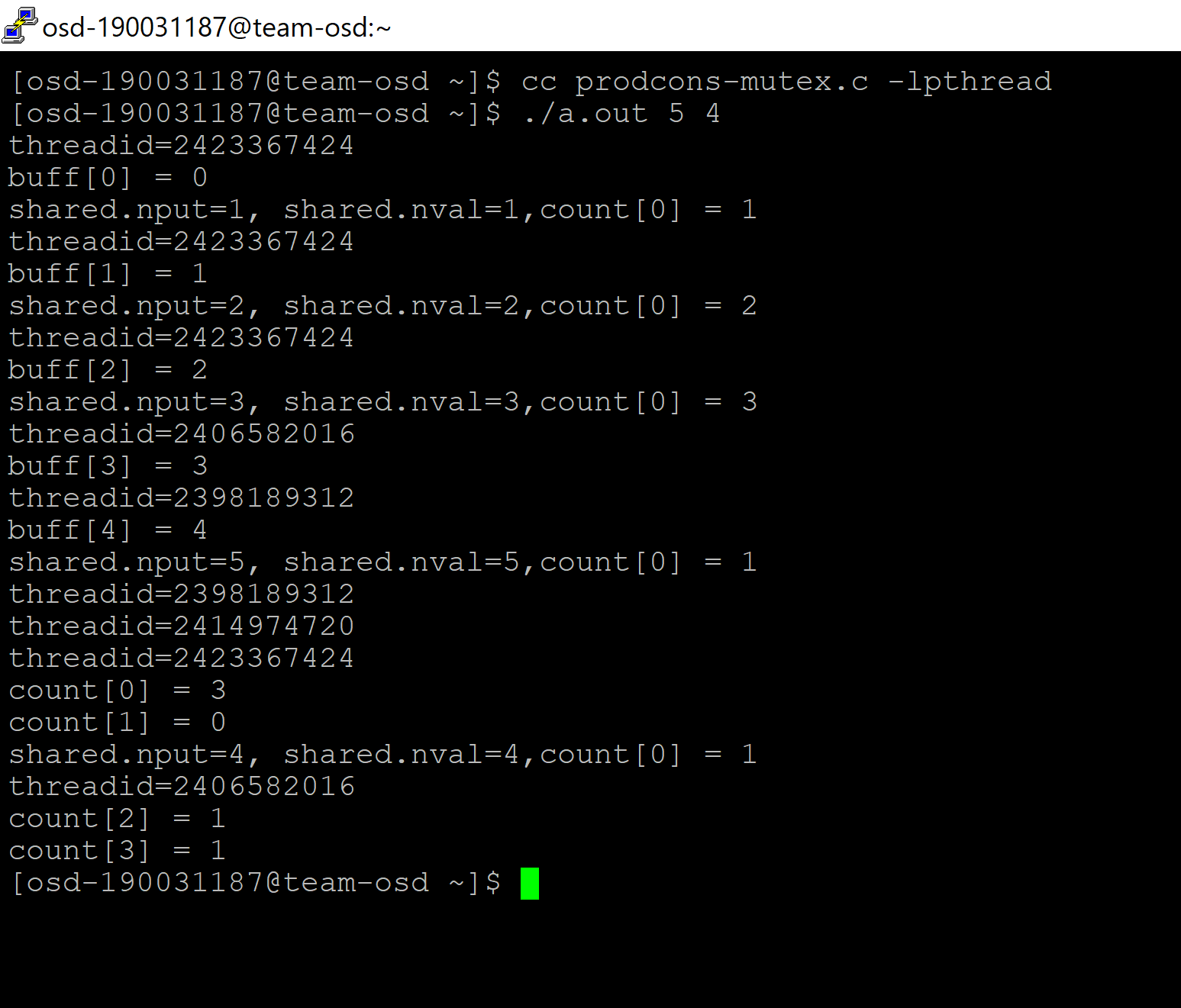
printf("buff[%d] = %d\n", i, shared.buff[i]);

}

return(NULL);

}

/\* end producer \*/



1. **prodcons-cv.c**

/\* Implementation of Producer consumer problem using condition variables \*/

/\* include globals \*/

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#include <pthread.h>

#include <sys/types.h>

#define MAXNITEMS 1000000

#define MAXNTHREADS 100

/\* globals shared by threads \*/

int nitems; /\* read-only by producer and consumer \*/

int buff[MAXNITEMS];

struct {

pthread\_mutex\_t mutex;

int nput; /\* next index to store \*/

int nval; /\* next value to store \*/

} put = { PTHREAD\_MUTEX\_INITIALIZER };

struct {

pthread\_mutex\_t mutex;

pthread\_cond\_t cond;

int nready; /\* number ready for consumer \*/

} nready = { PTHREAD\_MUTEX\_INITIALIZER, PTHREAD\_COND\_INITIALIZER };

/\* end globals \*/

void \*produce(void \*), \*consume(void \*);

/\* include main \*/

int

main(int argc, char \*\*argv)

{

int i, nthreads, count[MAXNTHREADS];

pthread\_t tid\_produce[MAXNTHREADS], tid\_consume;

if (argc != 3)

{

printf("usage: prodcons6 <#items> <#threads>");

}

nitems = atoi(argv[1]);

nthreads = atoi(argv[2]);

pthread\_setconcurrency(nthreads + 1);

/\* 4create all producers and one consumer \*/

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

count[i] = 0;

pthread\_create(&tid\_produce[i], NULL, produce, &count[i]);

}

pthread\_create(&tid\_consume, NULL, consume, NULL);

/\* wait for all producers and the consumer \*/

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

pthread\_join(tid\_produce[i], NULL);

printf("count[%d] = %d\n", i, count[i]);

}

pthread\_join(tid\_consume, NULL);

exit(0);

}

/\* end main \*/

/\* include prodcons \*/

void \*

produce(void \*arg)

{

for ( ; ; ) {

pthread\_mutex\_lock(&put.mutex);

if (put.nput >= nitems) {

pthread\_mutex\_unlock(&put.mutex);

return(NULL); /\* array is full, we're done \*/

}

buff[put.nput] = put.nval;

put.nput++;

put.nval++;

pthread\_mutex\_unlock(&put.mutex);

pthread\_mutex\_lock(&nready.mutex);

if (nready.nready == 0)

pthread\_cond\_signal(&nready.cond);

nready.nready++;

pthread\_mutex\_unlock(&nready.mutex);

\*((int \*) arg) += 1;

}

}

void \*

consume(void \*arg)

{

int i;

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

pthread\_mutex\_lock(&nready.mutex);

while (nready.nready == 0)

pthread\_cond\_wait(&nready.cond, &nready.mutex);

nready.nready--;

pthread\_mutex\_unlock(&nready.mutex);

if (buff[i] == i)

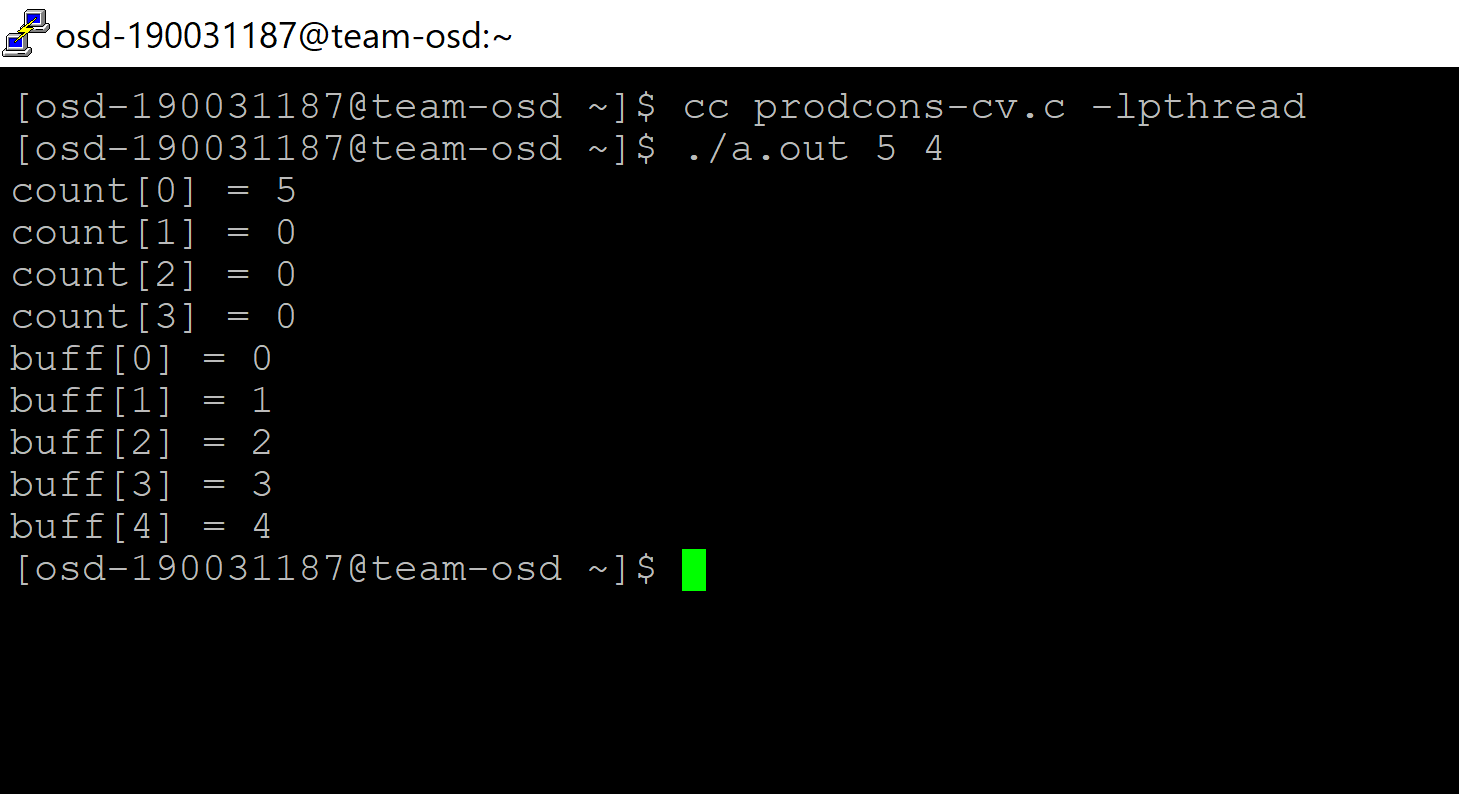
printf("buff[%d] = %d\n", i, buff[i]);

}

return(NULL);

}

/\* end prodcons \*/



**POSTLAB**

**Solve readers writer’s problem using mutex and semaphores**

**Ans:**

#include<pthread.h>

#include <semaphore.h>

#include <stdio.h>

/\*

This program provides a possible solution for first readers writers problem using mutex and semaphore.

I have used 10 readers and 5 producers to demonstrate the solution. You can always play with these values.

\*/

sem\_t wrt;

pthread\_mutex\_t mutex;

int cnt = 1;

int numreader = 0;

void \*writer(void \*wno)

{

sem\_wait(&wrt);

cnt = cnt\*2;

printf("Writer %d modified cnt to %d\n",(\*((int \*)wno)),cnt);

sem\_post(&wrt);

}

void \*reader(void \*rno)

{

// Reader acquire the lock before modifying numreader

pthread\_mutex\_lock(&mutex);

numreader++;

if(numreader == 1) {

sem\_wait(&wrt); // If this id the first reader, then it will block the writer

}

pthread\_mutex\_unlock(&mutex);

// Reading Section

printf("Reader %d: read cnt as %d\n",\*((int \*)rno),cnt);

// Reader acquire the lock before modifying numreader

pthread\_mutex\_lock(&mutex);

numreader--;

if(numreader == 0) {

sem\_post(&wrt); // If this is the last reader, it will wake up the writer.

}

pthread\_mutex\_unlock(&mutex);

}

int main()

{

pthread\_t read[10],write[5];

pthread\_mutex\_init(&mutex, NULL);

sem\_init(&wrt,0,1);

int a[10] = {1,2,3,4,5,6,7,8,9,10}; //Just used for numbering the producer and consumer

for(int i = 0; i < 10; i++) {

pthread\_create(&read[i], NULL, (void \*)reader, (void \*)&a[i]);

}

for(int i = 0; i < 5; i++) {

pthread\_create(&write[i], NULL, (void \*)writer, (void \*)&a[i]);

}

for(int i = 0; i < 10; i++) {

pthread\_join(read[i], NULL);

}

for(int i = 0; i < 5; i++) {

pthread\_join(write[i], NULL);

}

pthread\_mutex\_destroy(&mutex);

sem\_destroy(&wrt);

return 0;

}

