**EXPERIMENT 5**

**Aim:** Study of functions related to threads (POSIX).

Study functions: pthread\_create(), pthread\_join()

Problem

1. Create two threads that will call function ptask(). Function ptask() will print ‘Job Started’ & ‘Job Stopped’ with counter.
2. Create two threads that will call function add(). Function add() will scan two values and print the addition. Use mutex.
3. Solve producer consumer problem (bounded buffer) using mutex.

**Tools:** Linux OS, gcc complier, libpthread.so library

**Procedure:**

* **Details of the System calls:**

|  |
| --- |
| NAME |
|  | pthread\_create - thread creation |
|  |  |
|  | SYNOPSIS |
|  | #include <pthread.h> |
|  |  |
|  | int pthread\_create(pthread\_t \*restrict thread, |
|  | const pthread\_attr\_t \*restrict attr, |
|  | void \*(\*start\_routine)(void\*), void \*restrict arg); |
|  |  |
|  | DESCRIPTION |
|  | The pthread\_create() function shall create a new thread, with attributes specified by attr, within a process. If attr is NULL, the default |
|  | attributes shall be used. If the attributes specified by attr are modified later, the thread’s attributes shall not be affected. Upon successful |
|  | completion, pthread\_create() shall store the ID of the created thread in the location referenced by thread. |
|  |  |
|  | The thread is created executing start\_routine with arg as its sole argument. If the start\_routine returns, the effect shall be as if there was |
|  | an implicit call to pthread\_exit() using the return value of start\_routine as the exit status. Note that the thread in which main() was origi- |
|  | nally invoked differs from this. When it returns from main(), the effect shall be as if there was an implicit call to exit() using the return |
|  | value of main() as the exit status. |
|  |  |
|  | The signal state of the new thread shall be initialized as follows: |
|  |  |
|  | \* The signal mask shall be inherited from the creating thread. |
|  |  |
|  | \* The set of signals pending for the new thread shall be empty. |
|  |  |
|  | The alternate stack shall not be inherited. |
|  |  |
|  | The floating-point environment shall be inherited from the creating thread. |
|  |  |
|  | If pthread\_create() fails, no new thread is created and the contents of the location referenced by thread are undefined. |
|  |  |
|  | If \_POSIX\_THREAD\_CPUTIME is defined, the new thread shall have a CPU-time clock accessible, and the initial value of this clock shall be set to |
|  | zero. |
|  |  |
|  | RETURN VALUE |
|  | If successful, the pthread\_create() function shall return zero; otherwise, an error number shall be returned to indicate the error. |
|  | ------------------------------------------------------------------------------------------------------------- |
|  | NAME |
|  | pthread\_join - wait for thread termination |
|  |  |
|  | SYNOPSIS |
|  | #include <pthread.h> |
|  |  |
|  | int pthread\_join(pthread\_t thread, void \*\*value\_ptr); |
|  |  |
|  | DESCRIPTION |
|  | The pthread\_join() function shall suspend execution of the calling thread until the target thread terminates, unless the target thread has already terminated. On return from a successful pthread\_join() call with a non-NULL value\_ptr argument, the value passed to pthread\_exit() by the terminating thread shall be made available in the location referenced by value\_ptr. When a pthread\_join() returns successfully, the target thread has been terminated. The results of multiple simultaneous calls to pthread\_join() specifying the same target thread are undefined. If the thread calling pthread\_join() is canceled, then the target thread shall not be detached. |
|  |  |
|  | It is unspecified whether a thread that has exited but remains unjoined counts against {PTHREAD\_THREADS\_MAX}. |
|  |  |
|  | RETURN VALUE |
|  | If successful, the pthread\_join() function shall return zero; otherwise, an error number shall be returned to indicate the error. |
|  | ------------------------------------------------------------------------------------------------------------ |
|  | NAME |
|  | pthread\_cancel - cancel execution of a thread |
|  |  |
|  | SYNOPSIS |
|  | #include <pthread.h> |
|  |  |
|  | int pthread\_cancel(pthread\_t thread); |
|  |  |
|  | DESCRIPTION |
|  | The pthread\_cancel() function shall request that thread be canceled. The target thread’s cancelability state and type determines when the can- |
|  | cellation takes effect. When the cancellation is acted on, the cancellation cleanup handlers for thread shall be called. When the last cancella- |
|  | tion cleanup handler returns, the thread-specific data destructor functions shall be called for thread. When the last destructor function |
|  | returns, thread shall be terminated. |
|  |  |
|  | The cancellation processing in the target thread shall run asynchronously with respect to the calling thread returning from pthread\_cancel(). |
|  |  |
|  | RETURN VALUE |
|  | If successful, the pthread\_cancel() function shall return zero; otherwise, an error number shall be returned to indicate the error. |
|  | ------------------------------------------------------------------------------------------------------------- |
|  | NAME |
|  | pthread\_exit - thread termination |
|  |  |
|  | SYNOPSIS |
|  | #include <pthread.h> |
|  |  |
|  | void pthread\_exit(void \*value\_ptr); |
|  |  |
|  | DESCRIPTION |
|  | The pthread\_exit() function shall terminate the calling thread and make the value value\_ptr available to any successful join with the terminat- |
|  | ing thread. Any cancellation cleanup handlers that have been pushed and not yet popped shall be popped in the reverse order that they were |
|  | pushed and then executed. After all cancellation cleanup handlers have been executed, if the thread has any thread-specific data, appropriate |
|  | destructor functions shall be called in an unspecified order. Thread termination does not release any application visible process resources, |
|  | including, but not limited to, mutexes and file descriptors, nor does it perform any process-level cleanup actions, including, but not limited |
|  | to, calling any atexit() routines that may exist. |
|  |  |
|  | An implicit call to pthread\_exit() is made when a thread other than the thread in which main() was first invoked returns from the start routine |
|  | that was used to create it. The function’s return value shall serve as the thread’s exit status. |
|  |  |
|  | The behavior of pthread\_exit() is undefined if called from a cancellation cleanup handler or destructor function that was invoked as a result of |
|  | either an implicit or explicit call to pthread\_exit(). |
|  |  |
|  | After a thread has terminated, the result of access to local (auto) variables of the thread is undefined. Thus, references to local variables of |
|  | the exiting thread should not be used for the pthread\_exit() value\_ptr parameter value. |
|  |  |
|  | The process shall exit with an exit status of 0 after the last thread has been terminated. The behavior shall be as if the implementation called |
|  | exit() with a zero argument at thread termination time. |
|  |  |
|  | RETURN VALUE |
|  | The pthread\_exit() function cannot return to its caller. |
|  |  |

1. **Create two threads that will call function ptask(). Function ptask() will print ‘Job Started’ & ‘Job Stopped’ with counter.**

#include <pthread.h>

#include <stdio.h>

int counter=0;

void \*ptask(void \*ptr)

{

counter++;

printf("Job %d has started!\n",counter);

sleep(3);

printf("Job %d has stopped!\n",counter);

}

int main()

{

pthread\_t tid1;

//printf("Starting Thread!\n");

pthread\_create(&tid1, NULL, &ptask, NULL);

//sleep(3);

pthread\_t tid2;

pthread\_create(&tid2, NULL, &ptask, NULL);

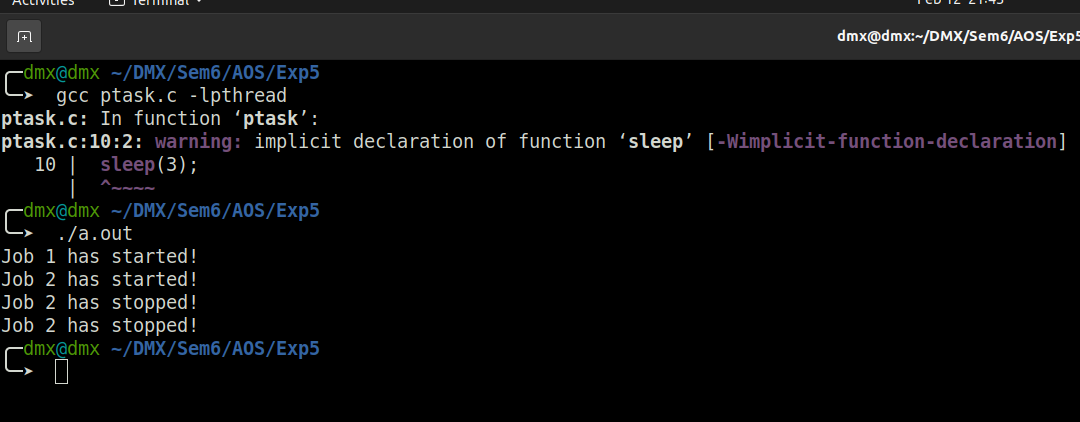
pthread\_join(tid1, NULL);

pthread\_join(tid2, NULL);

//printf("Thread terminated!\n");

return 0;

}



1. **Create two threads that will call function add(). Function add() will scan two values and print the addition. Use mutex.**

#include <pthread.h>

#include <stdio.h>

int a,b;

void \*add(void \*ptr)

{

printf("Addition: %d\n",(a+b));

}

void \*sub(void \*ptr)

{

printf("Substraction: %d\n",(a-b));

}

int main()

{

pthread\_t tid1,tid2;

printf("Enter two numbers: \n");

scanf("%d %d",&a,&b);

printf("Starting Addition Thread!\n");

pthread\_create(&tid1, NULL, &add, NULL);

printf("Starting Substraction Thread!\n");

pthread\_create(&tid2, NULL, &sub, NULL);

sleep(3);

pthread\_join(tid1, NULL);

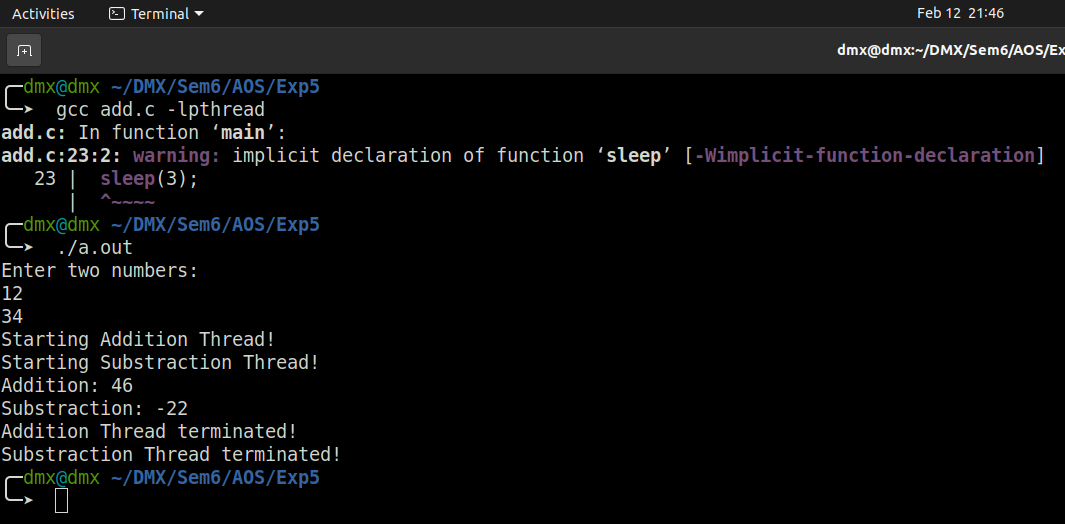
printf("Addition Thread terminated!\n");

pthread\_join(tid2, NULL);

printf("Substraction Thread terminated!\n");

return 0;

}



1. **Solve producer consumer problem (bounded buffer) using mutex.**

#include<stdio.h>

#include<unistd.h>

#include<pthread.h>

#define MAX 6

pthread\_mutex\_t block;

int buffer[MAX];

int in = 0, out = 0, itemID = 0, counter = 0, totalItems;

void \*consume(void \*ptr){

int tmp;

while (1){

while(counter == 0);

pthread\_mutex\_lock(&block);

tmp = buffer[out];

printf("Product %2d is consume by Consumer at %d.\n", tmp, out);

out = (out + 1) % MAX;

counter--;

pthread\_mutex\_unlock(&block);

sleep(1);

if(tmp == totalItems)

return NULL;

}

}

void \*produce(void \*ptr){

while (1){

while(counter == MAX);

pthread\_mutex\_lock(&block);

buffer[in] = ++itemID;

printf("Product %2d is produce by producer at %d.\n", itemID, in);

in = (in + 1) % MAX;

counter++;

pthread\_mutex\_unlock(&block);

sleep(1);

if(itemID == totalItems)

return NULL;

}

}

int main(){

pthread\_t producer, consumer;

printf("Enter total no of Product to Produce/Consume : ");

scanf("%d", &totalItems);

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

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

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

pthread\_join(producer, NULL);

pthread\_join(consumer, NULL);

pthread\_mutex\_destroy(&block);

printf("All threads are terminated.\n\n");

return 0;

}

