**Operating System and Design (19CS2106S)**

**Lab- 11**

**Pre-Lab**

1. **Creating and Destroying Mutexes: pthread\_mutex\_init (mutex,attr) pthread\_mutex\_destroy (mutex)**

#include <stdio.h>

#include <pthread.h>

int done = 0;

void\* worker(void\* arg) {

printf("this should print first\n");

done = 1;

return NULL;

}

int main(int argc, char \*argv[]) {

pthread\_t p;

pthread\_create(&p, NULL, worker, NULL);

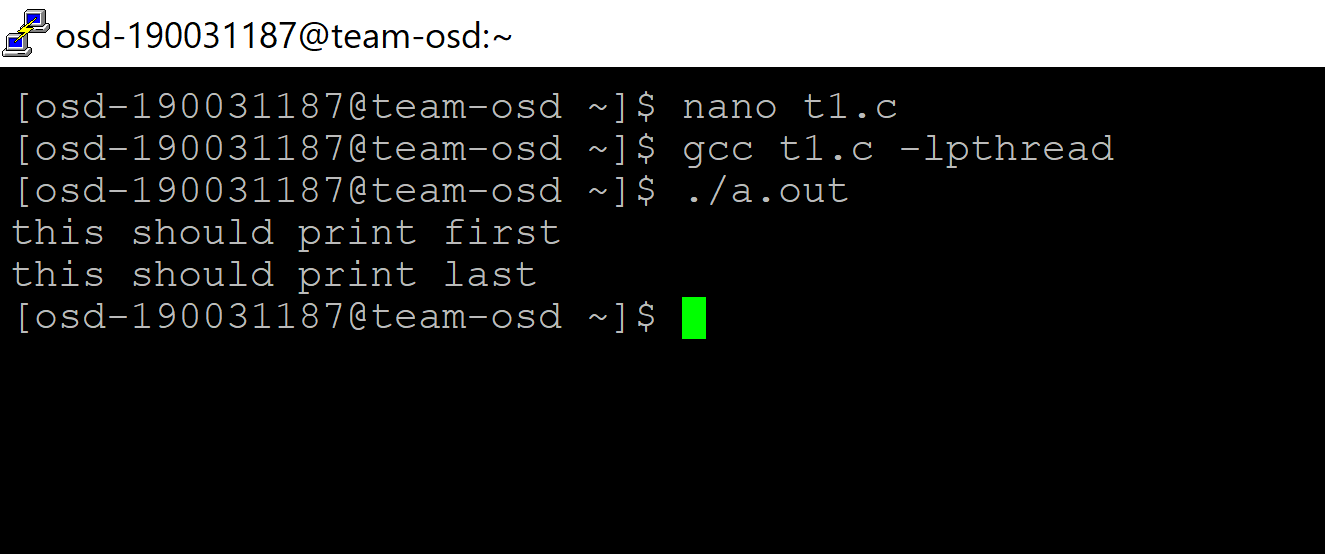
while (done == 0)

;

printf("this should print last\n");

return 0;

}



1. **Locking and Unlocking Mutexes: pthread\_mutex\_lock (mutex) pthread\_mutex\_trylock (mutex)**

#include <stdio.h>

#include <pthread.h>

typedef struct \_\_synchronizer\_t {

pthread\_mutex\_t lock;

pthread\_cond\_t cond;

int done;

} synchronizer\_t;

synchronizer\_t s;

void signal\_init(synchronizer\_t \*s) {

pthread\_mutex\_init(&s->lock, NULL);

pthread\_cond\_init(&s->cond, NULL);

s->done = 0;

}

void signal\_done(synchronizer\_t \*s) {

pthread\_mutex\_lock(&s->lock);

s->done = 1;

pthread\_cond\_signal(&s->cond);

pthread\_mutex\_unlock(&s->lock);

}

void signal\_wait(synchronizer\_t \*s) {

pthread\_mutex\_lock(&s->lock);

while (s->done == 0)

pthread\_cond\_wait(&s->cond, &s->lock);

pthread\_mutex\_unlock(&s->lock);

}

void\* worker(void\* arg) {

printf("this should print first\n");

signal\_done(&s);

return NULL;

}

int main(int argc, char \*argv[]) {

pthread\_t p;

signal\_init(&s);

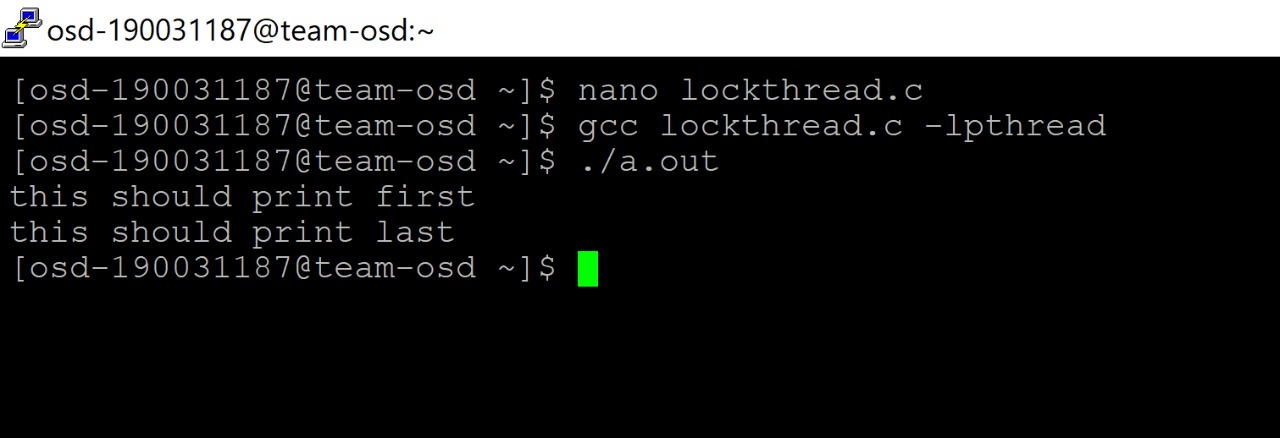
pthread\_create(&p, NULL, worker, NULL);

signal\_wait(&s);

printf("this should print last\n");

return 0;

}



1. **Synchronizing Threads with POSIX Semaphores: sem\_init, sem\_open sem\_wait sem\_post sem\_getvalue sem\_destroy**

**Semaphores:**

[Semaphores](https://www.softprayog.in/programming/semaphore-basics) are used for process and thread synchronization. Semaphores are clubbed with message queues and shared memory under the Interprocess Communication (IPC) facilities in Unix-like systems such as Linux.

**sem\_init**

#include <semaphore.h>

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

sem\_init is the equivalent of sem\_open for unnamed semaphores. One defines a variable of type sem\_t and passes its pointer as sem in the sem\_init call.

**sem\_open**

#include <fcntl.h>

#include <sys/stat.h>

#include <semaphore.h>

sem\_t \*sem\_open (const char \*name, int oflag);

sem\_t \*sem\_open (const char \*name, int oflag,

mode\_t mode, unsigned int value);

sem\_open is the call to get started for a semaphore. sem\_open opens an existing semaphore or creates a new semaphore and opens it for further operations.

**sem\_post**

#include <semaphore.h>

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

sem\_post increments the semaphore. It provides the [V operation](https://www.softprayog.in/programming/semaphore-basics#semop) for the semaphore. It returns 0 on success and -1 on error.

**sem\_wait**

#include <semaphore.h>

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

sem\_wait decrements the semaphore pointed by sem. If the semaphore value is non-zero, the decrement happens right away. If the semaphore value is zero, the call blocks till the time semaphore becomes greater than zero and the decrement is done. sem\_wait returns zero on success and -1 on error.

**sem\_getvalue**

#include <semaphore.h>

int sem\_getvalue (sem\_t \*sem, int \*sval);

sem\_getvalue gets the value of semaphore pointed by sem. The value is returned in the integer pointed by sval. It returns 0 on success and -1 on error, with errno indicating the actual error.

**sem\_destory**

#include <semaphore.h>

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

sem\_destroy destroys the unnamed semaphore pointed by sem.

**In-Lab**

1. **Illustrate how mutex is used for thread synchronization, print the counter variable upon each increment which is in the critical section. (Two threads update a global shared variable with and without synchronization).**

#include <pthread.h>

static volatile int glob = 0; /\* "volatile" prevents compiler optimizations

of arithmetic operations on 'glob' \*/

static void \* /\* Loop 'arg' times incrementing 'glob' \*/

threadFunc(void \*arg)

{

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

int loc, j;

for (j = 0; j < loops; j++) {

loc = glob;

loc++;

glob = loc;

}

return NULL;

}

int

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

{

pthread\_t t1, t2;

int loops, s;

loops = strtol(argv[1], NULL, 10);

s = pthread\_create(&t1, NULL, threadFunc, &loops);

if (s != 0){

perror("pthread\_create fail");

exit(1);}

s = pthread\_create(&t2, NULL, threadFunc, &loops);

if (s != 0){

perror("pthread\_create fail");

exit(2);}

s = pthread\_join(t1, NULL);

if (s != 0){

perror("pthread\_join fail");

exit(1);}

s = pthread\_join(t2, NULL);

if (s != 0){

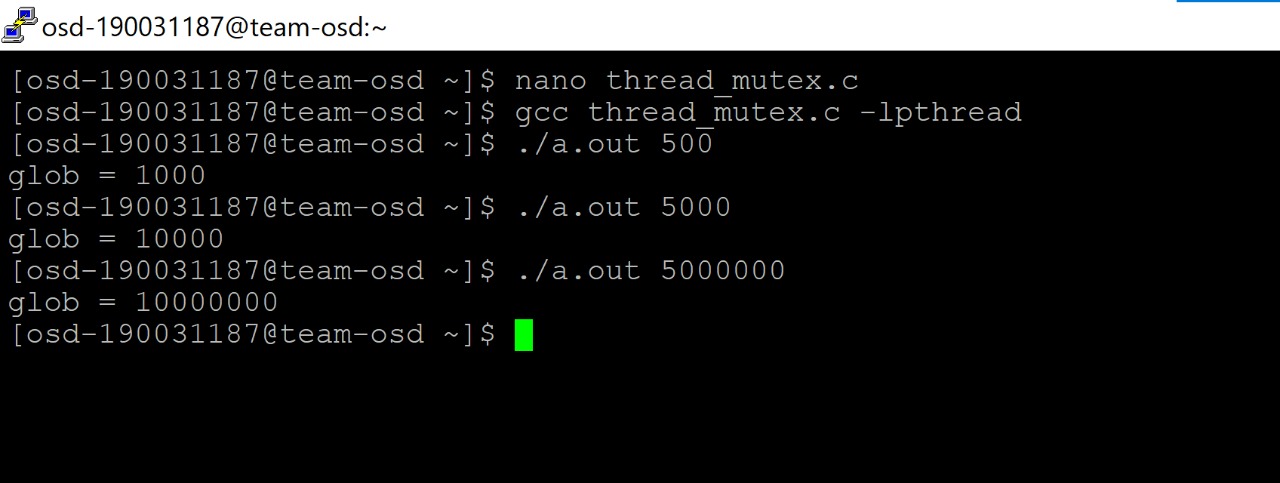
perror("pthread\_join fail");

exit(1);}

printf("glob = %d\n", glob);

exit(0);

}



1. **Write a UNIX system program to implement concurrent Linked List**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

typedef struct \_\_node\_t {

int key;

struct \_\_node\_t \*next;

} node\_t;

// basic list structure (one used per list)

typedef struct \_\_list\_t {

node\_t \*head;

pthread\_mutex\_t lock;

} list\_t;

void List\_Init(list\_t \*L) {

L->head = NULL;

pthread\_mutex\_init(&L->lock, NULL);

}

void List\_Insert(list\_t \*L, int key) {

// synchronization not needed

node\_t \*new = malloc(sizeof(node\_t));

if (new == NULL) {

perror("malloc");

return;

}

new->key = key;

// just lock critical section

pthread\_mutex\_lock(&L->lock);

new->next = L->head;

L->head = new;

pthread\_mutex\_unlock(&L->lock);

}

int List\_Lookup(list\_t \*L, int key) {

int rv = -1;

pthread\_mutex\_lock(&L->lock);

node\_t \*curr = L->head;

while (curr) {

if (curr->key == key) {

rv = 0;

break;

}

curr = curr->next;

}

pthread\_mutex\_unlock(&L->lock);

return rv; // now both success and failure

}

void List\_Print(list\_t \*L) {

node\_t \*tmp = L->head;

while (tmp) {

printf("%d ", tmp->key);

tmp = tmp->next;

}

printf("\n");

}

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

{

list\_t mylist;

List\_Init(&mylist);

List\_Insert(&mylist, 10);

List\_Insert(&mylist, 30);

List\_Insert(&mylist, 5);

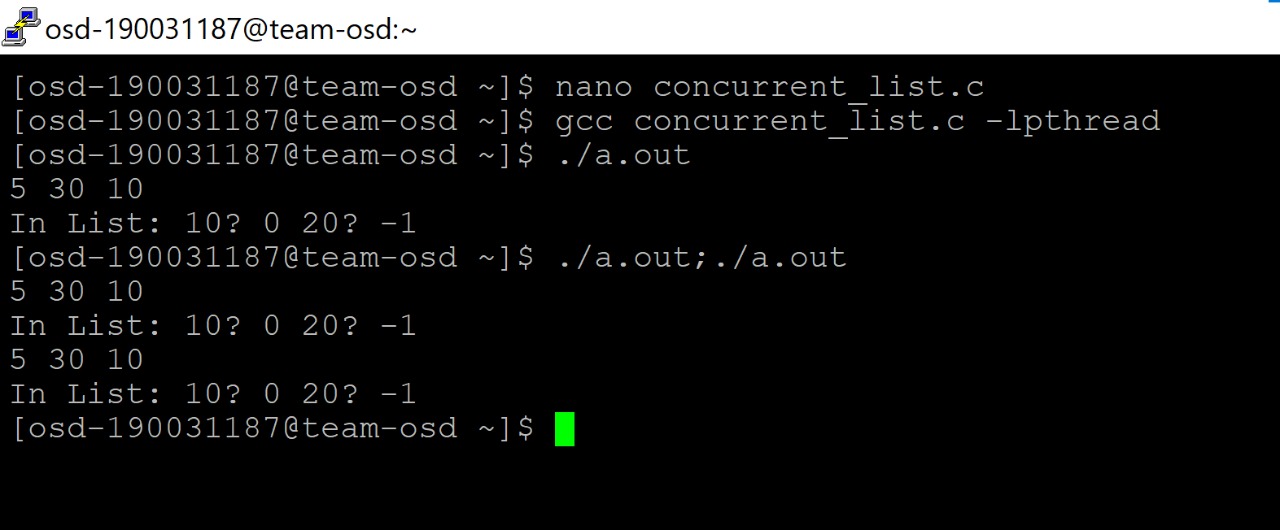
List\_Print(&mylist);

printf("In List: 10? %d 20? %d\n",

List\_Lookup(&mylist, 10), List\_Lookup(&mylist, 20));

return 0;

}



**Post-Lab**

1. **Write a Unix System program to make A Parent Waiting for Its Child using semaphores.**

#include <stdio.h>

#include <unistd.h>

#include <pthread.h>

#include <semaphore.h>

sem\_t done;

void \*

child(void \*arg) {

sleep(5);

printf("child\n");

sem\_post(&done);

return NULL;

}

int

main(int argc, char \*argv[]) {

pthread\_t p;

printf("parent: begin\n");

sem\_init(&done,0, 0);

pthread\_create(&p, NULL, child, NULL);

sem\_wait(&done);

printf("parent: end\n");

return 0;

}

