Xi'an Jiaotong-Liverpool University
西交利物浦大学

CPT104 - Operating Systems Concepts

LAB 10

POSIX semaphores in C

A **critical section** means the common resources shared by multiple processes.

Requirements for **Mutual Exclusion**

Any facility or capability that is to provide support for mutual exclusion should meet the following requirements:

1. Mutual exclusion must be enforced: Only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or shared object.
2. A process that halts in its noncritical section must do so without interfering with other processes.
3. It must not be possible for a process requiring access to a critical section to be delayed indefinitely: no deadlock or starvation.
4. When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.
5. No assumptions are made about relative process speeds or number of processors.
6. A process remains inside its critical section for a finite time only.

A **semaphore** achieves process synchronization. When two or more processes are using the same resources to perform a task, it may result in improper output. To avoid that problem, a lock keeps the critical section. A lock uses synchronization mechanisms to prevent multiple threads from accessing the same data at the same time.

A semaphore is a positive integer variable that is shared between threads and processes. A semaphore allows or blocks the resources of a process or thread based on conditions. Semaphores are classified into two types: **binary** and **counting**.

The POSIX semaphore mechanism is one IPC mechanism that originated with the real-time extensions to POSIX.1. #include <semaphore.h>

int sem_wait(sem_t *sem);

sem_wait() decrements (**locks**) the semaphore pointed to by *sem*.

If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

int sem_post(sem_t *sem);

sem_post() - **unlock** a semaphore, increments (unlocks) the semaphore pointed to by *sem*.

int sem_init(sem_t *sem, int pshared, unsigned int value);

To create an unnamed semaphore, we call the **sem_init** function. The pshared argument indicates if we plan to use the semaphore with multiple processes. If so, we set it to a nonzero value. The value argument specifies the initial value of the semaphore.

int sem_destroy(sem_t *sem);

When we are done using the unnamed semaphore, we can discard it by calling the **sem_destroy** function.

Example 1 – Semaphore and Critical Section

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>


sem_t mutex;

void* thread(void* arg)    //function which act like shared resources for threads
{

  sem_wait(&mutex);

  printf("\nEntered into Critical Section...\n");
 //Critical Section
  sleep(2);

  printf("\nCompleted...\n");       //coming out from Critical section

  sem_post(&mutex);
}


int main()
{
  sem_init(&mutex, 0, 1);

  pthread_t th1,th2;                  // create threads
  pthread_create(&th1,NULL,thread,NULL);
  sleep(2);

  pthread_create(&th2,NULL,thread,NULL);

  //Join threads with the main thread
  pthread_join(th1,NULL);
  pthread_join(th2,NULL);

  sem_destroy(&mutex);
  return 0;
}
```

declare a semaphore

decrement a semaphore

increment a semaphore

initialize a semaphore

destroy a semaphore

Example 2 – Binary semaphore

Three threads are trying to access the global data of **A** and **B**.
The ***two_numbers*** function acts like a critical section.
A = A * 3;
B = B - 1;
The integers **A = 12** and **B = 45.**
Three threads are trying to access the global data of **A** and **B**. The problem that occurred due to a race condition is solved with a binary semaphore.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>


sem_t mutex;

int A, B;                    //declare global variables A and B


// Function to access the global data

void* two_numbers(void* arg)
{

sem_wait(&mutex);

A = A + 3;
B = B - 1;


printf("A value is: %d and B value is: %d\n", A, B);

sleep(1);

sem_post(&mutex);
}


int main()
{

 A = 12;
 B = 45;
```

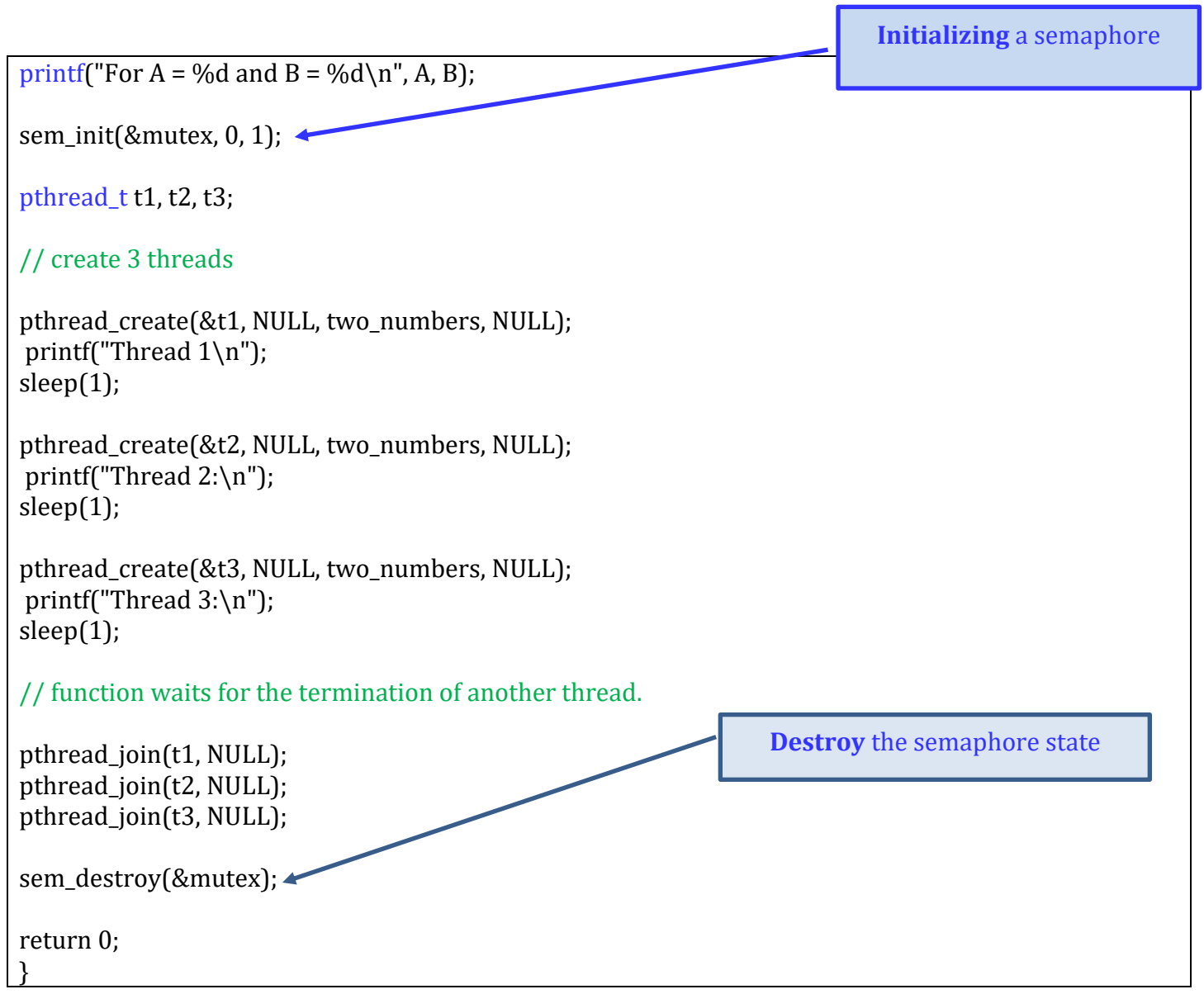
The **header file** to use **semaphore**

declare a semaphore

decrement a semaphore

increment a semaphore

```
printf("For A = %d and B = %d\n", A, B);

sem_init(&mutex, 0, 1);

pthread_t t1, t2, t3;

// create 3 threads

pthread_create(&t1, NULL, two_numbers, NULL);
 printf("Thread 1\n");
sleep(1);

pthread_create(&t2, NULL, two_numbers, NULL);
 printf("Thread 2:\n");
sleep(1);

pthread_create(&t3, NULL, two_numbers, NULL);
 printf("Thread 3:\n");
sleep(1);

// function waits for the termination of another thread.

pthread_join(t1, NULL);
pthread_join(t2, NULL);
pthread_join(t3, NULL);

sem_destroy(&mutex);

return 0;
}
```



PV Primitives

- Counting semaphores are equipped with two operations, historically denoted as P() and V (). Operation V(S) increments the semaphore S, and operation P(S) decrements it.
- P Operations and V operations are non-interruptible program segments, called Primitives. The concept of PV primitives and Semaphores was proposed by E.w.dijkstra, a Dutch scientist.

- We now continue with the **Lab Exercise** (see LMO)
- Just like the Lab Example, you will use the https://remisharrock.github.io/sysbuild/#/VM to write and test your code;
- You will also need to submit your code in **LMO**.

Reference

For more details about semaphore, reference book chapter **15.10**

Some typical process synchronizations

- produce-consumer problem;
- barbershop problem;
- readers-writers problem;
- ...