

**Lab #3, Semaphore**

**Date: 5th May 2022**

Course Code: **CSE325**

Course Title:

**Operating System**

Section No: **01**

Semester: **Spring2022**

**Submitted By**

**Md. Redwan Ahmed**

**2019-1-60-249**

**Submitted To**

**Dr. Md. Nawab Yousuf Ali**

**Professor**

**Department of Computer Science & Engineering**

**Abstract**

Two operations can be done on a semaphore object - increment or decrement by one, which corresponds to acquiring and releasing the shared resource. POSIX provides a special sem\_t type for an unnamed semaphore, a more common tool in multi-threaded workflows.  sem\_t variable must be initialized with the sem\_init function that also indicates whether the given semaphore should be shared between processes or threads of a process. Once the variable is initialized, we can implement the synchronization using the functions sem\_post and sem\_wait.  sem\_post increments the semaphore, which usually corresponds to unlocking the shared resource. In contrast, sem\_wait decrements the semaphore and denotes the locking of the resource. Thus, the critical section would need to start with sem\_wait and end with sem\_post call. Mind though, that checking for success status code can be essential to debugging the code.

**Introduction**

There are two common semaphore APIs on UNIX-based systems - POSIX semaphores and System V semaphores. The latter is considered to have a less simple interface while offering the same features as POSIX API. Note that semaphores are yet another synchronization mechanism like mutexes and can be utilized in mostly similar scenarios. A semaphore is an integer maintained by the kernel, usually set to the initial value greater or equal to 0.

**Experimental Procedure**

**Problem 1:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

int main(){

int pid;

pid = fork();

srand(pid);

if(pid < 0){

perror("fork"); exit(1);

}

else if(pid){

char \*s = "abcdefgh";

int l = strlen(s);

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

putchar(s[i]);

fflush(stdout);

sleep(rand() % 2);

putchar(s[i]);

fflush(stdout);

sleep(rand() % 2);

}

}

else{

char \*s = "ABCDEFGH";

int l = strlen(s);

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

putchar(s[i]);

fflush(stdout);

sleep(rand() % 2);

putchar(s[i]);

fflush(stdout);

sleep(rand() % 2);

}

}

}

**Problem 2:**

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <semaphore.h>

#define THREAD\_NUM 4

sem\_t semaphore;

void\* routine(void\* args) {

sem\_wait(&semaphore);

sleep(1);

printf("Hello from thread %d\n", \*(int\*)args);

sem\_post(&semaphore);

free(args);

}

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

pthread\_t th[THREAD\_NUM];

sem\_init(&semaphore, 0, 4);

int i;

for (i = 0; i < THREAD\_NUM; i++) {

int\* a = malloc(sizeof(int));

\*a = i;

if (pthread\_create(&th[i], NULL, &routine, a) != 0) {

perror("Failed to create thread");

}

}

for (i = 0; i < THREAD\_NUM; i++) {

if (pthread\_join(th[i], NULL) != 0) {

perror("Failed to join thread");

}

}

sem\_destroy(&semaphore);

return 0;

}

**Problem 3:**

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <semaphore.h>

#define THREAD\_NUM 1

sem\_t semFuel;

pthread\_mutex\_t mutexFuel;

int \*fuel;

void\* routine(void\* args) {

\*fuel += 50;

printf("Current value is %d\n", \*fuel);

sem\_post(&semFuel);

}

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

pthread\_t th[THREAD\_NUM];

fuel = malloc(sizeof(int));

\*fuel = 50;

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

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

int i;

for (i = 0; i < THREAD\_NUM; i++) {

if (pthread\_create(&th[i], NULL, &routine, NULL) != 0) {

perror("Failed to create thread");

}

}

sem\_wait(&semFuel);

printf("Deallocating fuel\n");

free(fuel);

for (i = 0; i < THREAD\_NUM; i++) {

if (pthread\_join(th[i], NULL) != 0) {

perror("Failed to join thread");

}

}

pthread\_mutex\_destroy(&mutexFuel);

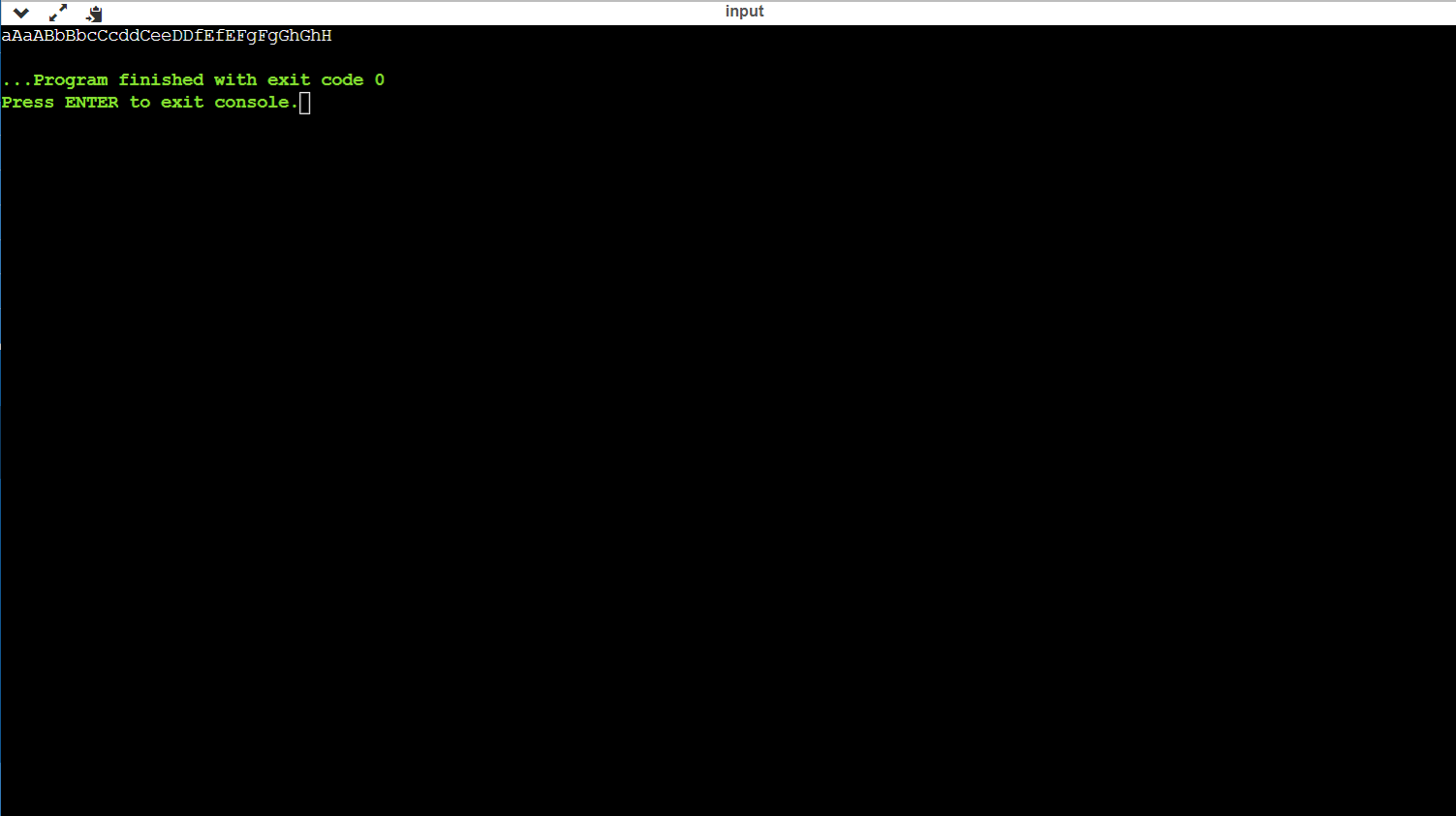
sem\_destroy(&semFuel);

return 0;

}

**Results**

**Output 1:**



**Output 2:**



**Output 3:**



**Discussion**

A semaphore initialized with a sem\_init call must be destroyed using the sem\_destroy function. Note though that sem\_destroy should be called when none of the processes/threads are waiting for it. Omitting the sem\_destroy call may result in a memory leak on some systems. Generally, the semaphores have a similar performance compared to the Pthread mutexes, but the latter is usually preferred for better code structure. Although, there are some scenarios where the lock should be modified from the signal handler, which requires the function to be async-safe, and only sem\_post is implemented as such. There is also a named semaphore in POSIX API, that may persist even after a thread that created it and used it, terminates.

**Conclusion**

Semaphores are a good way to learn about synchronization, but they are not as widely used, in practice, as mutexes and condition variables. Nevertheless, there are some synchronization problems that can be solved simply with semaphores, yielding solutions that are more demonstrably correct.