Assignment

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

Process

Abstract:

C process control refers to a group of functions in the standard library of the C programming language implementing basic process control operations. The process control operations include actions such as termination of the program with various levels of cleanup, running an external command interpreter or accessing the list of the environment operations.

Introduction:

Process is a running instance of a program. Linux is a multitasking operating system, which means that more than one process can be active at once. Use ps command to find out what processes are running on your system. To monitor and control the processes, Linux provides lot of commands such as ps, kill, killall, nice, renice and top commands.

Experimental Procedure.

Problem 1:

```
#include <stdio.h>

 #include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <errno.h>

#include <sys/types.h>

#include <sys/wait.h>

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

int id1 = fork();

int id2 = fork();

if (id1 == 0) {

     if (id2 == 0){

      printf("We are process y\n"); }
      else {

           printf("We are process x\n");

           }

      } else {

           if (id2 == 0) {

                printf("We are process z\n"); }

                else {

                     printf("We are the parent process\n");
                }

                      } while (wait(NULL) != -1 || errno !=
                ECHILD) {

                     printf("Waited for a child to
                finish\n");

                      }

return 0;

}
```

Problem 2:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

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

int pid = fork();
```

```
if (pid == -1)

{

     return 1;

     } if (pid == 0) {

          while (1) {

               printf("Some output\n");

               usleep(50000);

               }

          } else {

          kill(pid, SIGSTOP);

          int t;

          do {

          printf("Time in seconds for execution: ");
          scanf("%d", &t);

          if (t > 0)

          {

               kill(pid, SIGCONT);

               sleep(t);

               kill(pid, SIGSTOP); }

               } while (t > 0);

               kill(pid, SIGKILL);

               wait(NULL);

                    }

return 0;
```

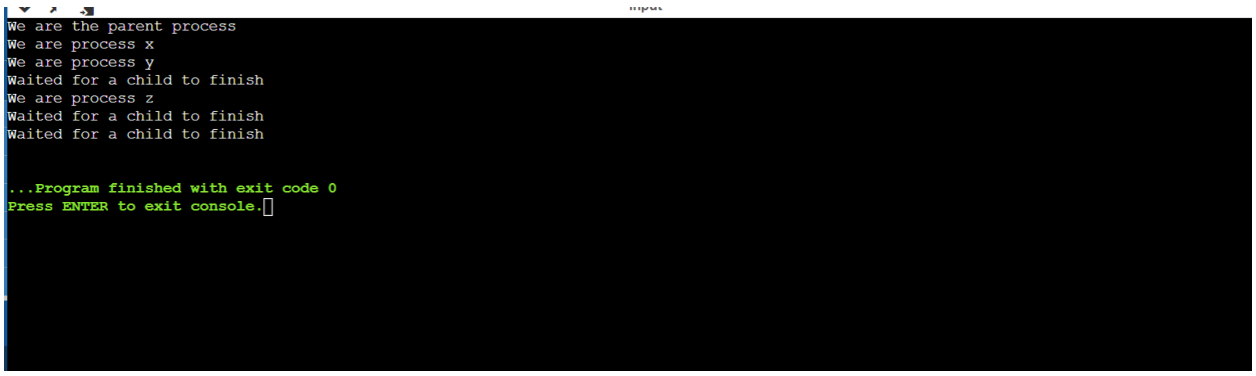
```
}
```

Process 3:

```
#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

void forkexample()

{

     int x = 1;

     if (fork() == 0)

          printf("Child has x = %d\n", ++x);

     else

          printf("Parent has x = %d\n", --x);

}

int main()

{

     forkexample();

     return 0;

}
```

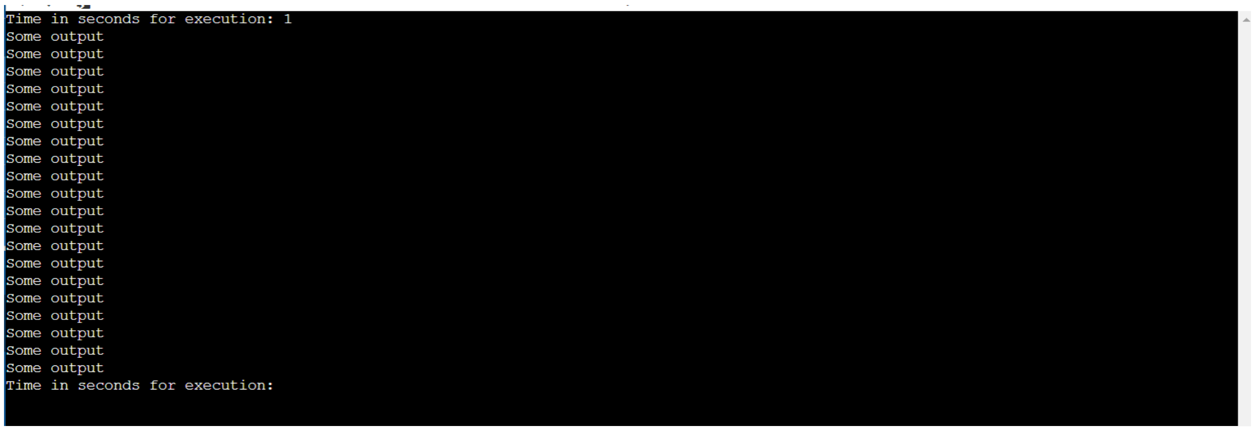
Results:

Output 1:

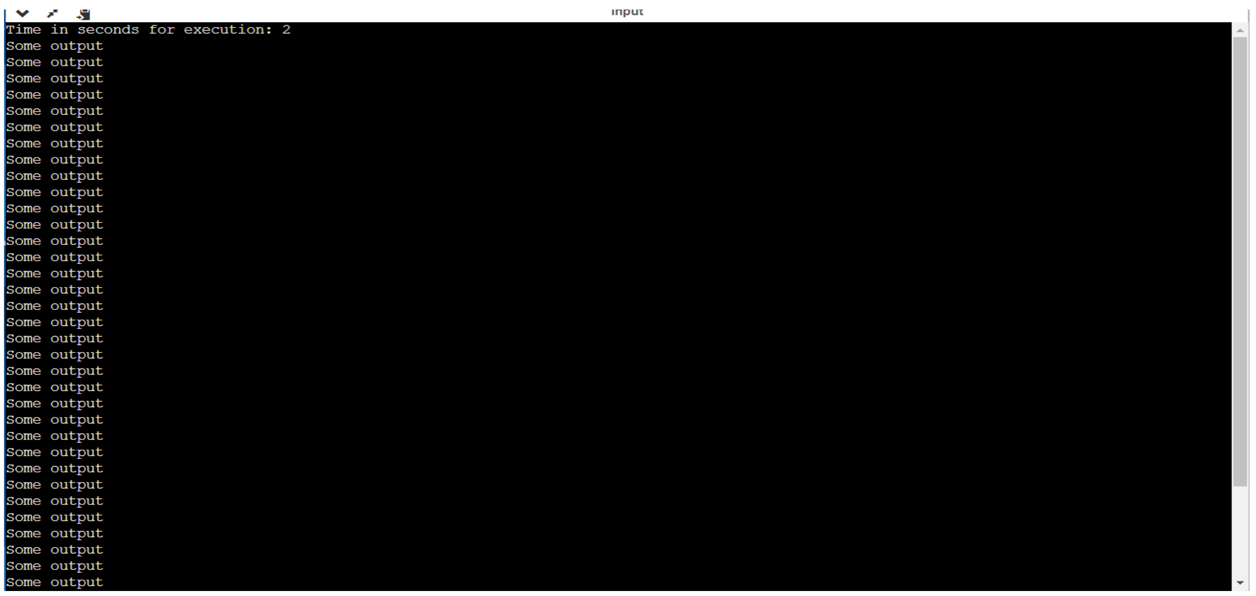


```
We are the parent process
We are process x
We are process y
Waited for a child to finish
We are process z
Waited for a child to finish
Waited for a child to finish


...Program finished with exit code 0
Press ENTER to exit console.
```

Output 2:

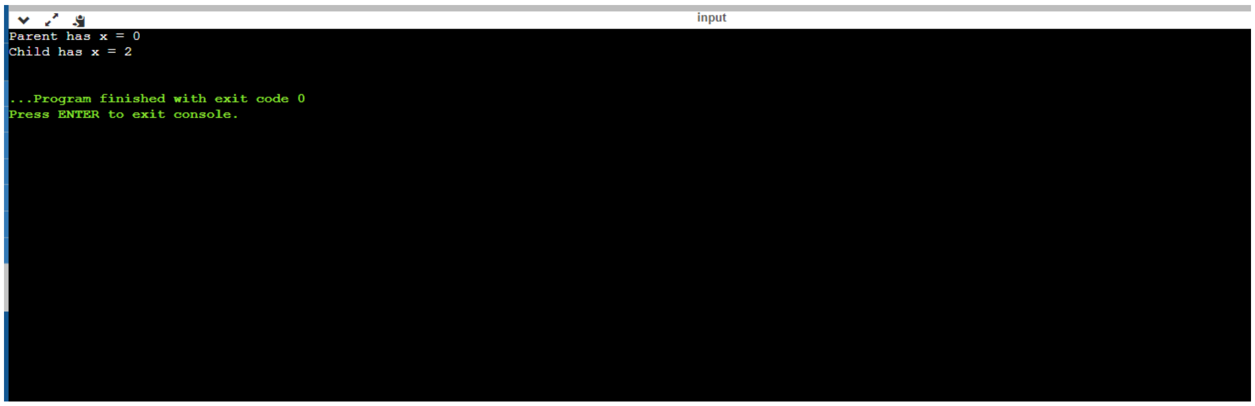


```
Time in seconds for execution: 1
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Time in seconds for execution:
```



```
Time in seconds for execution: 2
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
Some output
```

Output 3:



```
Parent has x = 0
Child has x = 2


...Program finished with exit code 0
Press ENTER to exit console.
```

Discussion

In the computing field, fork() is the primary method of process creation on Unix-like operating systems. This function creates a new copy called the *child* out of the original process, that is called the *parent*. When the parent process closes or crashes for some reason, it also kills the child process.

Conclusion

We have learned what fork() can do, and how to implement it in the C programming language in unique examples. If you are interested more in the operating system abstractions, and how it is working, then I recommend you to start learning about pipes then semaphores.

Thread

Abstract

Threads are not independent of each other because processes allow threads to share their code sections, data sections, and OS resources such as open files and signals with other threads. But, like the process, a thread has its own program counter (PC), a register set and a stack space.

Introduction

Threads are a popular way to improve applications through parallelism. For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads, one thread to format text, another thread to process input, and so on.

Threads work faster than processes for the following reasons:

1) Threads made much faster.

2) Changing the context between threads is much faster.

3) The thread can be closed easily

4) Quick communication between threads.

Experimental Procedure

Problem 1:

```
#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>



pthread_mutex_t mutexFuel;

pthread_cond_t condFuel;

int fuel = 0;



void* fuel_filling(void* arg) {
    for (int i = 0; i < 5; i++) {
        pthread_mutex_lock(&mutexFuel);
        fuel += 15;
        printf("Filled fuel... %d\n", fuel);
        pthread_mutex_unlock(&mutexFuel);
        pthread_cond_signal(&condFuel);
        sleep(1);
    }
}

void* car(void* arg) {
    pthread_mutex_lock(&mutexFuel);
    while (fuel < 40) {
        printf("No fuel. Waiting...\n");
        pthread_cond_wait(&condFuel, &mutexFuel);
        // Equivalent to:
        // pthread_mutex_unlock(&mutexFuel);
        // wait for signal on condFuel
        // pthread_mutex_lock(&mutexFuel);
    }
    fuel -= 40;
    printf("Got fuel. Now left: %d\n", fuel);
    pthread_mutex_unlock(&mutexFuel);
}
```

```
int main(int argc, char* argv[]) {
    pthread_t th[2];
    pthread_mutex_init(&mutexFuel, NULL);
    pthread_cond_init(&condFuel, NULL);
    for (int i = 0; i < 2; i++) {
        if (i == 1) {
            if (pthread_create(&th[i], NULL, &fuel_filling,
NULL) != 0) {
                perror("Failed to create thread");
            }
        } else {
            if (pthread_create(&th[i], NULL, &car, NULL) != 0) {
                perror("Failed to create thread");
            }
        }
    }

    for (int i = 0; i < 2; i++) {
        if (pthread_join(th[i], NULL) != 0) {
            perror("Failed to join thread");
        }
    }
    pthread_mutex_destroy(&mutexFuel);
    pthread_cond_destroy(&condFuel);
    return 0;
}
```

Problem 2:

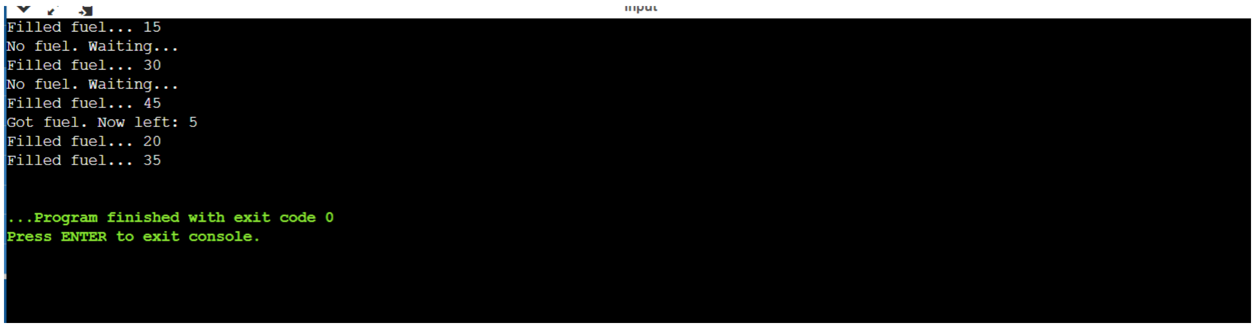
```
#include <stdlib.h>

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>


void* thread_F() {

    printf("This is the starting thread\n");

    sleep(3);

    printf("Bye, this is the ending thread\n");

}


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

    pthread_t p1, p2;

    if (pthread_create(&p1, NULL, &thread_F, NULL) != 0) {

        return 1;

    }

    if (pthread_create(&p2, NULL, &thread_F, NULL) != 0) {

        return 2;

    }

    if (pthread_join(p1, NULL) != 0) {

        return 3;

    }

    if (pthread_join(p2, NULL) != 0) {

        return 4;
    }
    return 0;
}
```

Problem 3:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <time.h>

// chefs = threads
// stove = shared data (+mutex)

pthread_mutex_t stoveMutex[4];
int stoveFuel[4] = { 100, 100, 100, 100 };

void* routine(void* args) {
    for (int i = 0; i < 4; i++) {
        if (pthread_mutex_trylock(&stoveMutex[i]) == 0) {
            int fuelNeeded = (rand() % 30);
            if (stoveFuel[i] - fuelNeeded < 0) {
                printf("No more fuel... going home\n");
            } else {
                stoveFuel[i] -= fuelNeeded;
                usleep(500000);
                printf("Fuel left %d\n", stoveFuel[i]);
            }
            pthread_mutex_unlock(&stoveMutex[i]);
            break;
        } else {
            if (i == 3) {
                printf("No stove available yet, waiting...\n");
                usleep(300000);
                i = 0;
            }
        }
    }
}

int main(int argc, char* argv[]) {
    srand(time(NULL));
    pthread_t th[10];
    for (int i = 0; i < 4; i++) {
        pthread_mutex_init(&stoveMutex[i], NULL);
    }
    for (int i = 0; i < 10; i++) {
        if (pthread_create(&th[i], NULL, &routine, NULL) != 0) {
            perror("Failed to create thread");
        }
    }

    for (int i = 0; i < 10; i++) {
        if (pthread_join(th[i], NULL) != 0) {
            perror("Failed to join thread");
        }
    }
    for (int i = 0; i < 4; i++) {
        pthread_mutex_destroy(&stoveMutex[i]);
    }
    return 0;
}
```

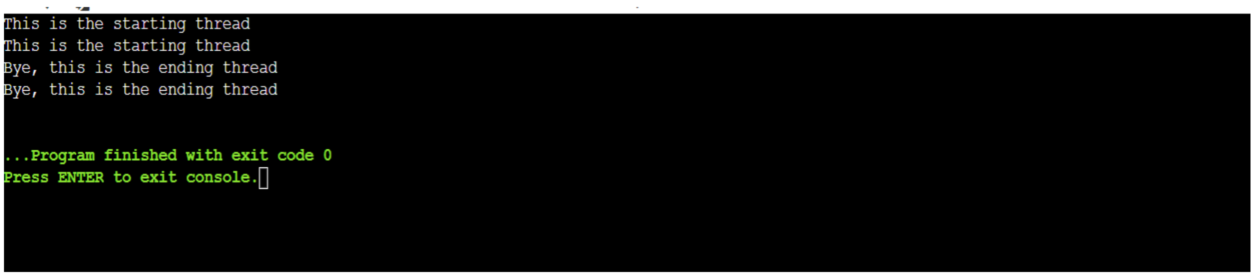
Results

Output 1



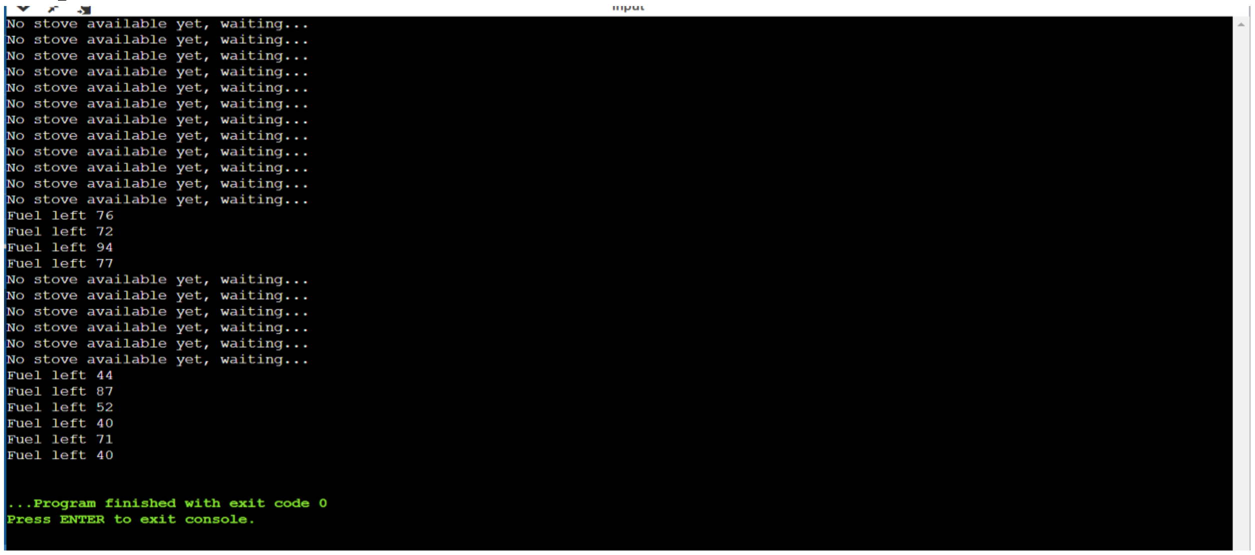
```
Filled fuel... 15
No fuel. Waiting...
Filled fuel... 30
No fuel. Waiting...
Filled fuel... 45
Got fuel. Now left: 5
Filled fuel... 20
Filled fuel... 35


...Program finished with exit code 0
Press ENTER to exit console.
```

Output 2:



```
This is the starting thread
This is the starting thread
Bye, this is the ending thread
Bye, this is the ending thread


...Program finished with exit code 0
Press ENTER to exit console.
```

Output 3:



```
No stove available yet, waiting...
No stove available yet, waiting...
No stove available yet, waiting...
No stove available yet, waiting...
No stove available yet, waiting...
No stove available yet, waiting...
No stove available yet, waiting...
No stove available yet, waiting...
No stove available yet, waiting...
No stove available yet, waiting...
No stove available yet, waiting...
No stove available yet, waiting...
Fuel left 76
Fuel left 72
Fuel left 94
Fuel left 77
No stove available yet, waiting...
No stove available yet, waiting...
No stove available yet, waiting...
No stove available yet, waiting...
No stove available yet, waiting...
No stove available yet, waiting...
Fuel left 44
Fuel left 87
Fuel left 52
Fuel left 40
Fuel left 71
Fuel left 40


...Program finished with exit code 0
Press ENTER to exit console.
```

Discussion

Threading is a specialized form of multitasking and a multitasking is the feature that allows your computer to run two or more programs concurrently. In general, there are two types of multitasking: process-based and thread-based.

Process-based multitasking handles the concurrent execution of programs. Thread-based multitasking deals with the concurrent execution of pieces of the same program.

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

C does not contain any built-in support for multithreaded applications. Instead, it relies entirely upon the operating system to provide this feature.

Conclusion

If you are working on Linux OS and we are going to write multi-threaded C program using POSIX. POSIX Threads, or Pthreads provides API which are available on many Unix-like POSIX systems such as FreeBSD, NetBSD, GNU/Linux, Mac OS X and Solaris.

Semaphore

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;
```

```c
        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;
}
```

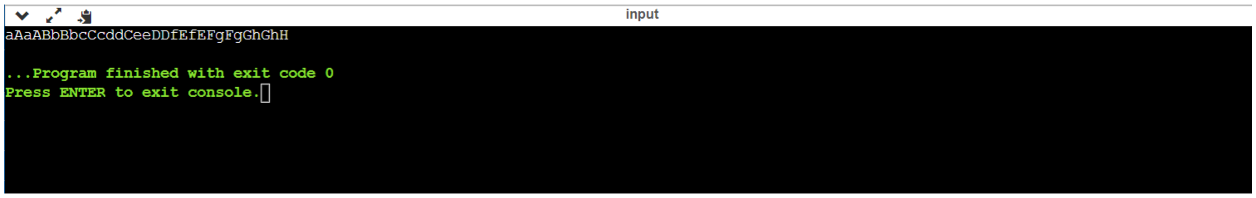
Problem 3:

```c
#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;
```

```c
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;
}
```

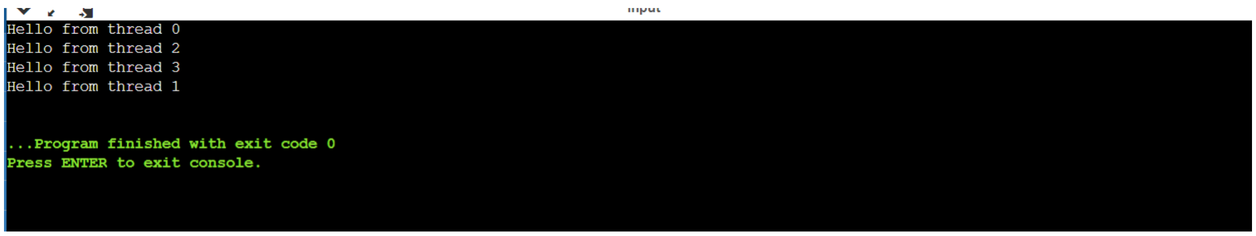
Results

Output 1:



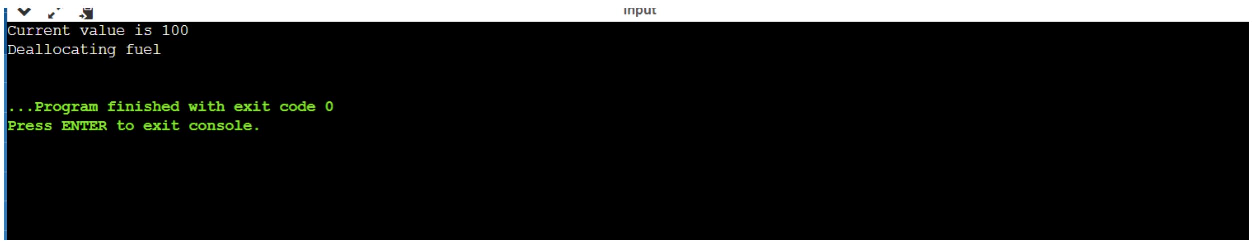
```
aAaABbBbcCcddCeeDDfEfEFgFgGhGhH

...Program finished with exit code 0
Press ENTER to exit console.
```

Output 2:



```
Hello from thread 0
Hello from thread 2
Hello from thread 3
Hello from thread 1


...Program finished with exit code 0
Press ENTER to exit console.
```

Output 3:



```
Current value is 100
Deallocating fuel


...Program finished with exit code 0
Press ENTER to exit console.
```

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.