CNG 334

Assignment 1

Raed Alsheikh Amin

2528271

Task 1:

1.a)

The “**available resources**” variable contains the data used in the race condition. This variable may be accessed and modified by multiple processes at the same time, creating a race condition where concurrent modifications could cause the value of “**available resources**” to become inconsistent.

1.b)

The race condition arises when the “**available resources**” variable is accessed and modified without the use of a synchronization mechanism, which happens in both the “**decrease count**” and “**increase count**” functions. Multiple processes may call these functions concurrently, which could result in a race condition if they read and modify “**available resources**” at the same time.

1.c) ch5/ slide 22,23,24 are references

The race condition in the given code can be fixed by using a **semaphore** to make sure that only one process at a time can access and modify the **available resources** variable.
A semaphore: An integer variable for containing the number of pending wakeups.
mutex: To grant mutual exclusion → either producer or consumer is accessing the buffer. Initially set to 1 and known as binary semaphore
empty: counting the number of empty slots. initially is N
full: contains the number of slots that are all full. initially is 0

```
// initialize mutex semaphore with value 1
Semaphore mutex = 1;

// initialize empty semaphore with initial count N
Semaphore empty = 5
// initialize full semaphore with initial count 0
```

```
Semaphore full = 0

// decrease available resources by count resources
// return 0 if sufficient resources available,
// otherwise return -1
int decrease_count(int count) {
    empty.down(); // decrease the count of empty slots
    mutex.down(); // acquire mutual exclusion before accessing shared resource
    if (available_resources < count) {
        // if not enough resources available, release mutex and empty semaphores
        mutex.up();
        empty.up();
        return -1;// flag to indicate there is a problem
    } else {
        available_resources -= count;
        mutex.up(); // release mutual exclusion after accessing shared resource
        full.up(); // increase the count of full slots
        return 0;
    }
}

// increase available resources by count
int increase_count(int count) {
    full.down(); // wait until there's at least one full slot
    mutex.down(); // acquire mutual exclusion before accessing shared resource
    available_resources += count;
    mutex.up(); // release mutual exclusion after accessing shared resource
    empty.up(); // increase the count of empty slots
    return 0;
}
```

Task 2:

A.1)

```
The values of the array elements :
 334
 334
 334
 334
 334
Updating the values of the array elements :
The values of the array elements again:
462
462
462
462
462
```

the values are different because the parent process is not aware of what is happening in the child process => it will just see the global one before updating the values.

the child process updates the values of the elements in the shared memory array **(ptr)** to 462, while the parent process still has the original values (334) because it does not modify the array after the fork() call

A.2)

used as a resource to learn threads in c:
https://www.geeksforgeeks.org/multithreading-in-c/

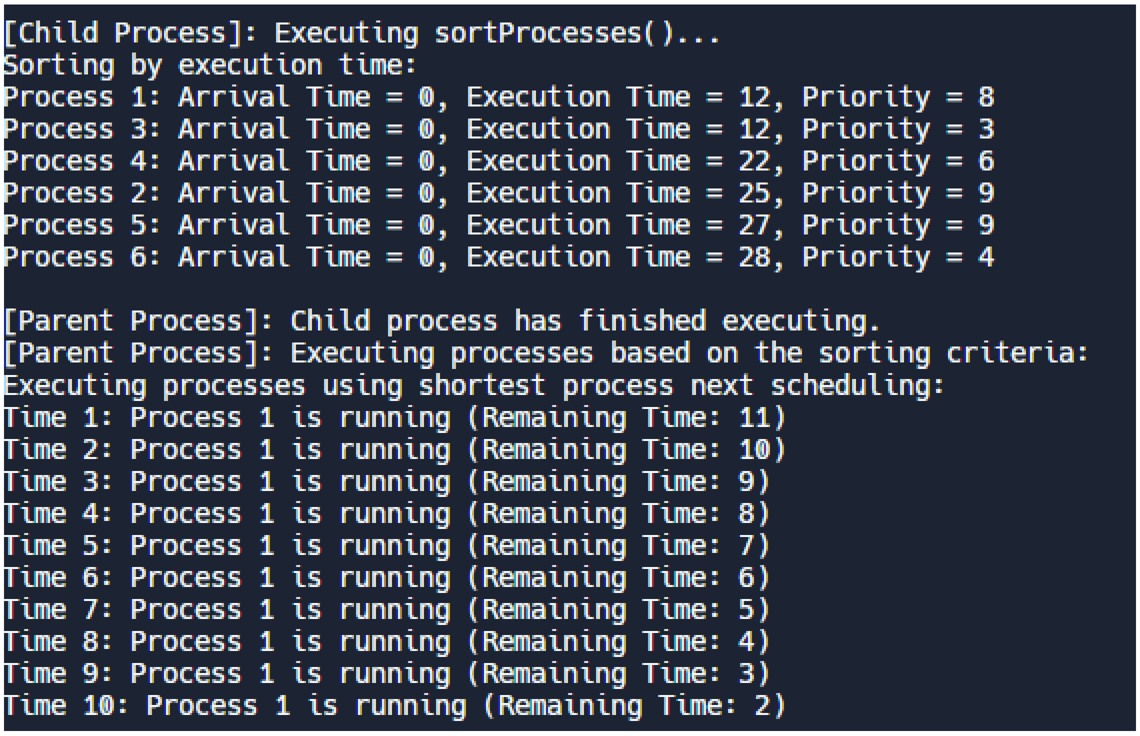
```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define N 5

int array[N];
```

```
void *child_thread(void *arg) {
    printf("Updating the values of the array elements:\n");
    for (int i = 0; i < N; i++)
        array[i] = 462;

    pthread_exit(NULL);
}

int main() {
    pthread_t tid;

    // initialize array
    for (int i = 0; i < N; i++)
        array[i] = 334;

    printf("The values of the array elements:\n");
    for (int i = 0; i < N; i++)
        printf("%d\n", array[i]);

    // create child thread
    if (pthread_create(&tid, NULL, child_thread, NULL) != 0) {
        fprintf(stderr, "error while  creating thread\n");
        return 1;
    }

    // wait for child thread to finish
    if (pthread_join(tid, NULL) != 0) {
        fprintf(stderr, "error while joining thread\n");
        return 1;
    }

    printf("The values of the array elements again:\n");
    for (int i = 0; i < N; i++)
        printf("%d\n", array[i]);

    return 0;
}
```

Using the C pthread library, the modified code uses threads in place of processes. While both parent and child threads in the new code share the same memory space within the process, the original mystery.c code relied on shared memory created by mmap() for communication and data sharing between processes. While in the original code wait(NULL) is used to wait for the child process to finish, synchronization is achieved using pthread_join() to wait for the child thread to finish before the parent thread continues execution. Additionally, the pthread_create() and pthread_join() functions are used, respectively, to implement error handling for thread creation and joining

C.1)

result:



```
[Child Process]: Executing sortProcesses()...
Sorting by execution time:
Process 1: Arrival Time = 0, Execution Time = 12, Priority = 8
Process 3: Arrival Time = 0, Execution Time = 12, Priority = 3
Process 4: Arrival Time = 0, Execution Time = 22, Priority = 6
Process 2: Arrival Time = 0, Execution Time = 25, Priority = 9
Process 5: Arrival Time = 0, Execution Time = 27, Priority = 9
Process 6: Arrival Time = 0, Execution Time = 28, Priority = 4

[Parent Process]: Child process has finished executing.
[Parent Process]: Executing processes based on the sorting criteria:
Executing processes using shortest process next scheduling:
Time 1: Process 1 is running (Remaining Time: 11)
Time 2: Process 1 is running (Remaining Time: 10)
Time 3: Process 1 is running (Remaining Time: 9)
Time 4: Process 1 is running (Remaining Time: 8)
Time 5: Process 1 is running (Remaining Time: 7)
Time 6: Process 1 is running (Remaining Time: 6)
Time 7: Process 1 is running (Remaining Time: 5)
Time 8: Process 1 is running (Remaining Time: 4)
Time 9: Process 1 is running (Remaining Time: 3)
Time 10: Process 1 is running (Remaining Time: 2)
```

the code first initializes the array then the child starts to sort the array according to random generated numbers to decide which type of sorting will be used, after the child is done => the parent will start executing the processor according to the schedules provided, if the sorting is according to priority => priority scheduling will be executed, otherwise SPN will be executed.

C.2)

When using threads, limited arguments were allowed to be passed to the functions that were implemented by threads, while using process (fork) gave the code more flexibility

References:

https://www.geeksforgeeks.org/use-posix-semaphores-c/
https://www.geeksforgeeks.org/multithreading-in-c/
https://www.geeksforgeeks.org/selection-sort/
https://www.geeksforgeeks.org/insertion-sort/
https://www.prepbytes.com/blog/c-programming/priority-scheduling-program-in-c/
https://www.geeksforgeeks.org/program-for-shortest-job-first-or-sjf-cpu-scheduling-set-1-non-preemptive/