MIDDLE EAST TECHNICAL UNIVERSITY

NORTHERN CYPRUS CAMPUS

DEPARTMENT
OF COMPUTER ENGINEERING

CNG 334
Introduction to Operating Systems
Assignment-1

Name: Mert Can Bilgin
Student ID: 2453025

Table of Contents

Task 1: Processes Synchronization

Q1)

a)

“available_resources” variable holds the number of available licenses. It is accessed and modified by multiple processes concurrently. The race condition arises because multiple processes try to access and modify simultaneously. For example:

1. Process A check that there are enough resources, let’s say 3, available to acquire.
2. Before Process A decrements “available_resources”, Process B also checks the availability of resources, and finds that there are enough.
3. Both Process A and Process B then decrement “available_resources” to allocate the resources, and it causes that “available_resources” being decremented below zero, allowing more resources to be allocated than are available.

This concurrent access can lead to inconsistencies, hence causes a race condition.

b)

The race condition arises in the “decrease_count()” and “increase_count()” functions because of concurrent access for the “available_resources” variable.

1. In the “decrease_count()” function, the race condition occurs when multiple processes call it simultaneously. If multiple processes check the “available_resources” at the same time, they may both determine that there are enough available resources, and it leads incorrect results.
2. In the “increase_count()” function, if multiple processes attempt to increase “available_resources” at the same time, the race condition occurs. If multiple processes execute this function concurrently, they may overwrite each other’s changes to “available_resources”, and it causes inconsistent results.

c)

c.1)

We can use semaphores to ensure mutual exclusion so that only one process can access and modify “available_resources” at a time. We first need to initialize a semaphore as 1 so that only one process can access the critical section at a time to control access to the critical section of code where “available_resources” is accessed and modified. Before entering the critical section, a process should wait on the semaphore. If the semaphore value is greater than zero, the process can go on. Once the process has acquired the semaphore, it can safely enter the critical section. After completing its job in the critical section, the process should release the semaphore to allow other processes to enter the critical section.

c.2) Pseudocode

```
Define MAX_RESOURCES as the maximum number of resources available
Initialize available_resources to MAX_RESOURCES
Define a semaphore called mutex to control access to the critical section
Initialize mutex with an initial value of 1

Function decrease_count(count):
        Wait on mutex
        if available_resources >= count:
                Decrease available_resources by count
```

Release mutex
Function increase_count(count):
Wait on mutex
Increase available_resources by count
Release mutex

Task 2: Processes and Threads Creation

A)mystery Code

A.1)

Both the parent and child processes have access to the shared memory region created by “mmap()”. The parent process initializes the array elements in the shared memory to “334”. After the initialization, “fork()” creates a new child process. The parent process prints the values of the array. Then, the child process modifies the values of the array to 462. After that, the parent process prints the updated values of the array “462”. In conclusion, both the parent and child processes have access to the shared memory region, and changes made by one process are immediately visible to the other process. However, the difference in the timing of printing the array values leads to the printing different values by the parent and child processes.

A.2)

Firstly, in the original code using processes, shared memory was allocated using “mmap()”. In the modified code using threads, I’ve used “malloc()” because threads share the same memory space, so there’s no need for explicit shared memory allocation. Secondly, in the original code the processes are created using “fork”, but in the new code the threads are creaded using “pthread_create”. Thirdly, in the original code, the parent process waits for the child process to complete using “wait”, but in the new code, the main thread waits for the child using “pthread_join”. This ensures that the parent does not proceed until the child thread has finished executing. In summary, the main difference lies in the concurrency model process-based vs. thread-based. (The source code is attached in the submission)

B)CNG334_Task4 Code

B.1 & B.2)

The code is attached in the submission.

B.3)

Before the mutex, the main function creates multiple threads where each thread calls the worker function, and each thread receives the number of darts as a parameter. In the worker function, each thread generates random coordinates within a range, and threads increment “hit_count” if the generated coordinates fall within the unit circle. After all threads finish their work, the main function calculates the estimated PI. After the mutex, “circle_count” is protected by locking before updating and releasing after updating. It ensures that only one thread can access and update it at a time to prevent any race condition. After all threads finish their job, the main function destroys the mutex. Without mutex, multiple threads could simultaneously access and update the shared variable. It might cause incorrect results.

Functions:

- pthread_create(): Creates a new thread.
- pthread_join(): Waits for a thread to terminate.
- pthread_mutex_lock(): Locks the mutex.
- pthread_mutex_unlock(): Unlocks the mutex.
- pthread_mutex_destroy(): Destroy the mutex.

C)Write a POSIX C program

C.1)

The code is attached in submission.

C.2)

There are some differences between using processes and threads. First of all, threads share the same memory space, so they can access and modify the same data without needing to explicitly share it. However, processes have their own memory spaces, so they do not share data by default. Secondly, thread creation is faster than process creation. Thirdly, threads are more prone to synchronization issues such as race conditions and deadlocks because they execute concurrently and share the same memory space. On the other hand, processes are more isolated from each other. Even though we use shared memory for processes, we wait for the child process to complete its work to avoid race condition. When we used threads, if the sorting and executing threads access the shared array of processes concurrently, it could cause inconsistent sorting and execution results because synchronization is not implemented.