CNG 334

Assignment 2 Report

Name: Shayan

Nadeem

Student ID: 2542413

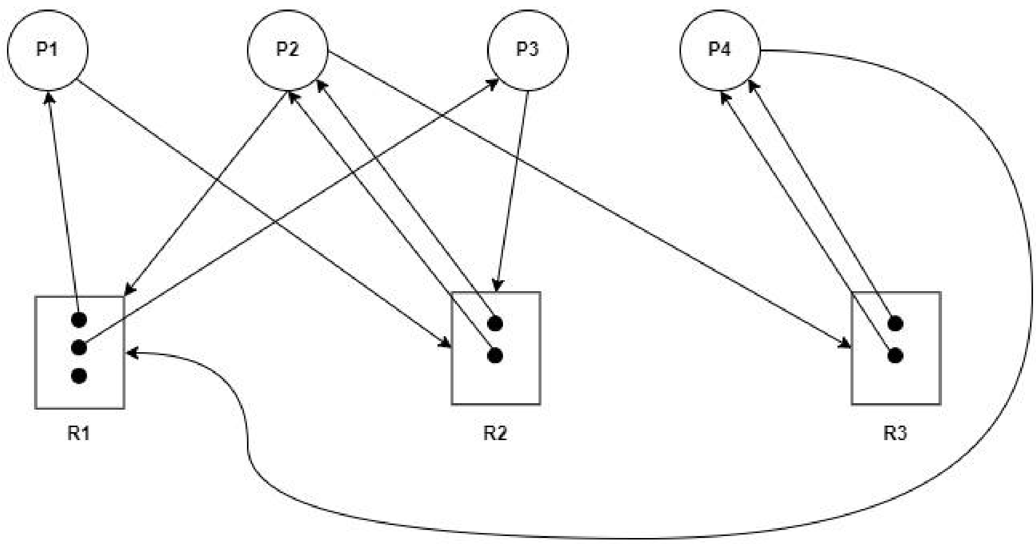
TASK 1:

Processes	Allocation	Maximum	Need/Work	Available
A	0 0 1 2	0 0 1 2	0 0 0 0	1520 0000 <= 1520 → T
B	1 0 0 0	1 7 5 0	0 7 5 0	1532 0750 <= 1532 → F
C	1 3 5 4	2 3 5 6	1 0 0 2	1532 1002 <= 1532 → T
D	0 6 3 2	0 6 5 2	0 0 2 0	2886 0020 <= 2886 → T
E	0 0 1 4	0 6 5 6	0 6 4 2	2 14 11 8 0642 <= 2 14 11 8 → T
Safe Sequence = <A, C, D, E, B>				2 14 12 12 0750 <= 2 14 12 12 → T
				Final Available: <3 14 12 12>

// In Available box, available is updated accordingly if true For the last header, B is evaluated again and since its true this time (arrow is shown to indicate that), available is updated accordingly//

TASK 2:

Resource Allocation Graph



There is **No Deadlock** illustrated in this system. We'll start with P4, as P4 executes first as one instance of R1 is available from the start. P4 releases both instances of R3 in this way after being executed. P2 executes next as one instance of R1 and R3 are available as per its needs. After execution, P2 releases both instances of R2 after execution. P1 now has one instance of R1 available/free and one of R2 so it executes next. After execution, P1 releases one instance of R1. Finally, P3 needs one instance of R2 and P1 needs one instance of R2 to execute which are both available, so they both execute simultaneously without any deadlocks..

TASK 3:

Firstly, using **nmap(),** a shared memory space for an array of "5 elements" is allocated through mapping. If it fails, an error message is printed. Now using a for loop, all the 5 array elements are given the value **"334"**. Then using **fork()** , the main process is forked into a child and a parent process. They both execute simultaneously, but the child process updates all the 5 array elements to the value **"462"**, while the parent process waits for it to execute first using the **wait()** function. At the end of the code, **munmap()** is used to deallocate/unmap the shared memory space and return an error message in-case munmap returns a non-zero value.

Output:

```
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
```

TASK 4:

1)

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/mman.h>
```

```
#define N 5 // Number of elements for the array

void* update_thread(void* arg) {
    int* array = (int*)arg;

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

    pthread_exit(NULL);
}

void* print_thread(void* arg) {
    int* array = (int*)arg;

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

    pthread_exit(NULL);
}

int main() {
    int* array = mmap(NULL, N * sizeof(int), PROT_READ | PROT_WRITE,
MAP_SHARED | MAP_ANONYMOUS, 0, 0);

    if (array == MAP_FAILED) {
        printf("Mapping Failed\n");
        return 1;
    }

    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]);
    }


    pthread_t update_thread_id, print_thread_id;

    int thread_create_result = pthread_create(&update_thread_id, NULL, update_thread, array);
    if (thread_create_result != 0) {
        printf("Update Thread creation failed\n");
        return 1;
    }

    thread_create_result = pthread_create(&print_thread_id, NULL, print_thread, array);
    if (thread_create_result != 0) {
        printf("Print Thread creation failed\n");
```

```
        return 1;
    }

    pthread_join(update_thread_id, NULL);
    pthread_join(print_thread_id, NULL);

    int err = munmap(array, N * sizeof(int));

    if (err != 0) {
        printf("Unmapping Failed\n");
        return 1;
    }

    return 0;
}
```

Note#1: Using mapping in my opinion was not necessary as threads already share a shared memory space.
Note#2: Since we are using two threads to update and print, we will have different outputs every time we run. Sometimes, it will print and then update, and sometimes it will update and print (which is correct).

//Difference between old code and this code would be that this one requires less overhead in terms of context switching as it is using multiple threads to execute the required functions. This in turn will also mean it is faster.//

2)

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

void *worker(void *param);

#define NUMBER_OF_DARTS 50000000
#define NUMBER_OF_THREADS 4

// Global variables
double circle_count = 0.0; // Number of darts that landed inside the circle
pthread_mutex_t* mutex = NULL; // Pointer to mutex

/*
 * Generates a double precision random number
 */
double random_double()
{
    return random() / ((double)RAND_MAX + 1);
}

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

```
    /* seed the random number generator */
    srandom((unsigned)time(NULL));

    pthread_t threads[NUMBER_OF_THREADS]; // Array to store thread IDs
    int thread_args[NUMBER_OF_THREADS]; // Array to store thread arguments

    mutex = (pthread_mutex_t*)malloc(sizeof(pthread_mutex_t)); // Allocate memory for
mutex

    int i;
    int darts_per_thread = NUMBER_OF_DARTS / NUMBER_OF_THREADS;

    // Create threads
    for (i = 0; i < NUMBER_OF_THREADS; i++) {
        thread_args[i] = darts_per_thread;
        pthread_create(&threads[i], NULL, worker, &thread_args[i]);
    }

    // Wait for threads to finish
    for (i = 0; i < NUMBER_OF_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    /* estimate Pi */
    double estimated_pi = 4.0 * circle_count / NUMBER_OF_DARTS;

    printf("Pi = %f\n", estimated_pi);

    // Destroy mutex
    pthread_mutex_destroy(mutex);
    free(mutex);

    return 0;
}

void *worker(void *param)
{
    int number_of_darts = *((int *)param);
    int i;
    int hit_count = 0;
    double x, y;

    for (i = 0; i < number_of_darts; i++) {
        /* generate random numbers between -1.0 and +1.0 (exclusive) */
        x = random_double() * 2.0 - 1.0;
        y = random_double() * 2.0 - 1.0;

        if (sqrt(x * x + y * y) < 1.0)
            ++hit_count;
    }

    // Lock the mutex to protect access to the shared variable circle_count
    pthread_mutex_lock(mutex);
```

```
    circle_count += hit_count;

    // Unlock the mutex after updating the shared variable
    pthread_mutex_unlock(mutex);

    pthread_exit(0);
}
```

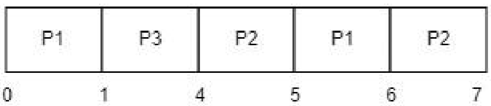
//The code uses a mutex (mutual exclusion) to ensure that only one thread can access and update the shared variable circle_count at a time.

Before the mutex, multiple threads could simultaneously access and modify circle_count, leading to data races and inconsistent results. The mutex provides synchronization by allowing only one thread to lock it and access the critical section (the update operation) at a time. Other threads attempting to lock the mutex will be blocked until it becomes available.

By using the mutex, the code guarantees that only one thread can update the shared variable **"circle_count"** at any given time, preventing data races and ensuring the integrity of the variable.//

BONUS TASK:

FIFO



Waiting Time

P1: 3 ms
P2: 3 ms
P3: 1 ms

Average Waiting Time: 2.33 ms

Round Robin

P1	P3	P2	P1	P3	P2

0 1 3 4 5 6 7

Waiting Time

P1: 2 ms
P2: 3 ms
P3: 3 ms

Average Waiting Time: 2.67 ms

SPN

P1	P2	P1	P2	P3

0 1 2 3 4 7

Waiting Time

P1: 0 ms
P2: 0 ms
P3: 4 ms

Average Waiting Time: 1.33 ms