

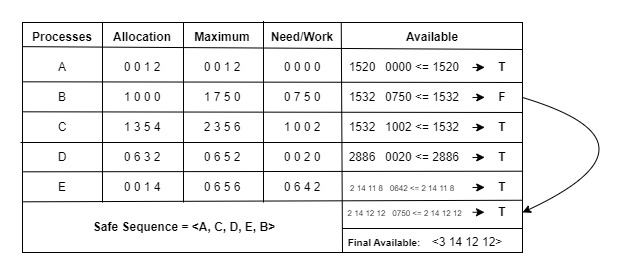
**CNG 334**

**Assignment 2 Report**

**Name:** Shayan Nadeem

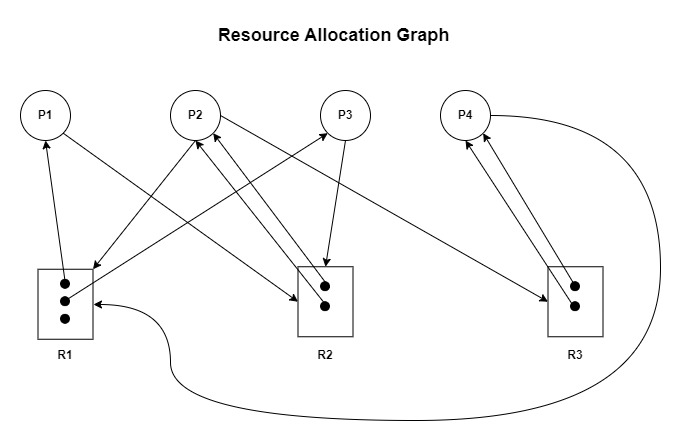
**Student ID:** 2542413

### **TASK 1:**

****

// 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:**

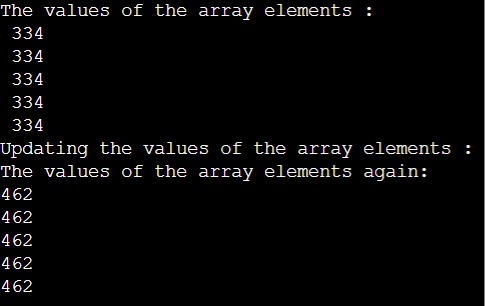


There is **No Deadlock** illustrated in this system. We’ll start with P4, as P4 executes first as one instance of R1 is available from the start. P4 releases both instances of R3 in this way after being executed. P2 executes next as one instance of R1 and R3 are available as per its needs. After execution, 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:**

****

### **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);

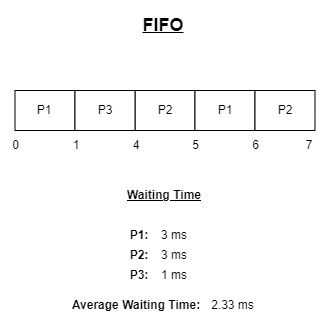
}

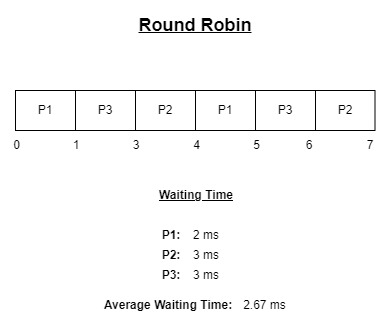
**//**The code uses a mutex (mutual exclusion) to ensure that only one thread can access and update the shared variable circle\_count at a time.

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:**

****

****

