

|  |  |
| --- | --- |
| **ASSIGNMENT** | |
| **Course Code** | 19CSC304A |
| **Course Name** | Operating Systems |
| **Programme** | B. Tech. |
| **Department** | Computer Science and Engineering |
| **Faculty** | Faculty of Engineering & Technology |

#### 

|  |  |
| --- | --- |
| **Name of the Student** | Subhendu Maji |
| **Reg. No** | 18ETCS002121 |
| **Semester/Year** | 5TH semester / 2018 batch |
| **Course Leader/s** | Ms. Naveeta |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Declaration Sheet** | | | | | | | | |
| Student Name | Subhendu Maji | | | | | | | |
| Reg. No | 18ETCS002121 | | | | | | | |
| Programme | B. Tech. | | | | | Semester/Year | 5th sem / 2018 batch | |
| Course Code | 19CSC304A | | | | | | | |
| Course Title | Operating Systems | | | | | | | |
| Course Date |  | | to | |  | | | |
| Course Leader | Ms. Naveeta | | | | | | | |
| **Declaration**  The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly. | | | | | | | | |
| Signature of the Student | |  | | | | | Date |  |
| Submission date stamp  (by Examination & Assessment Section) | |  | | | | | | |
| Signature of the Course Leader and date | | | | Signature of the Reviewer and date | | | | |
|  | | | |  | | | | |

# **Contents**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[**Declaration Sheet** ii](#_Toc58104396)

[**Contents** iii](#_Toc58104397)

[Marking Scheme 4](#_Toc58104398)

[**Question No. 1** 6](#_Toc58104399)

[1.1 Introduction to multi-programming 6](#_Toc58104400)

[1.2 Effect of multi-programming on CPU utilization 7](#_Toc58104401)

[**Question No. 2** 11](#_Toc58104402)

[2.1 Design and implementation of the application using sequential approach with functions 11](#_Toc58104403)

[2.2 Design and implementation of the application using multithreaded approach 15](#_Toc58104404)

[2.3 Comparison of the execution time of the above two versions of the program and its analysis 19](#_Toc58104405)

[**Question No. 3** 22](#_Toc58104406)

[3.1 Schedule of the processes using a Gantt chart 22](#_Toc58104407)

[3.2 Average waiting time and average turnaround time experienced 23](#_Toc58104408)

[3.3 Scheduling algorithm with better performance and its justification 24](#_Toc58104409)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Assignment** | | | | |  | |  |  |
| Register No. | | | **18ETCS002121** | Name of Student |  | **SUBHENDU MAJI** | | |
| **Sections** |  | Marking Scheme | | | **Max Marks** | | **First Examiner**    **Marks** | **Moderator**    **M**  **arks** |
| **Question 1** | Q1.1 | Introduction to multi-programming | | | 01 | |  |  |
| Q1.2 | Effect of multi-programming on CPU utilisation | | | 04 | |  |  |
|  | **Question 1 Max Marks** | | | **05** | |  |  |
| **Question 2** | Q2.1 | Design and implementation of the application using sequential approach with functions | | | 04 | |  |  |
| Q2.2 | Design and implementation of the application using multithreaded approach | | | 04 | |  |  |
| Q2.3 | Comparison of the execution time of the above two versions of the program and its analysis | | | 02 | |  |  |
|  | **Question 2 Max Marks** | | | **10** | |  |  |
| **Question 3** | Q3.1 | Schedule of the processes using a Gantt chart | | | 04 | |  |  |
| Q3.2 | Average waiting time and average turnaround time experienced | | | 04 | |  |  |
| Q3.3 | Scheduling algorithm with better performance and its justification | | | 02 | |  |  |
|  | **Question 3 Max Marks** | | | **10** | |  |  |
|  |  | **Total Assignment Marks** | | | **25** | |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Course Marks Tabulation** | | | | |
| **Component- 1(B) Assignment** | **First**  **Examiner** | **Remarks** | **Moderator** | **Remarks** |
| Q1 |  |  |  |  |
| Q2 |  |  |  |  |
| Q3 |  |  |  |  |
| **Marks (out of 25 )** |  |  |  |  |
| Signature of First Examiner Signature of Second Examiner | | | | |

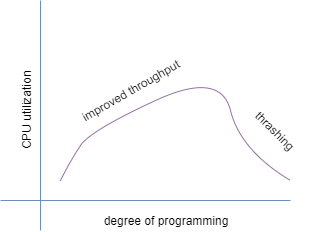
# **Question No. 1**

**Solution to Question No. 1:**

## Introduction to multi-programming

Early computers ran one process at a time. While the process waited for servicing by another device, the CPU was idle. In an I/O intensive process, the CPU could be idle as much as 80% of the time. Advancements in operating systems led to computers that load several independent processes into memory and switch the CPU from one job to another when the first becomes blocked while waiting for servicing by another device. This idea of **multiprogramming** reduces the idle time of the CPU. Multiprogramming accelerates the throughput of the system by efficiently using the CPU time.

Programs in a multiprogrammed environment appear to run at the same time. Processes running in a multiprogrammed environment are called **concurrent** processes. In actuality, the CPU processes one instruction at a time, but can execute instructions from any active process.



As the illustration shows, CPU utilization of a system can be improved by using multiprogramming.

Let P be the fraction of time that a process spends away from the CPU. If there is one process in memory, the CPU utilization is . If there are processes in memory, the probability of processes waiting for an I/O is The CPU utilization is where N is called the multiprogramming level (MPL) or the degree of multiprogramming. As N increases, the CPU utilization increases. While this equation indicates that a CPU continues to work more efficiently as more and more processes are added, logically, this cannot be true. Once the system passes the point of optimal CPU utilization, it thrashes.

In order to use the multiprogramming concept, processes must be loaded into independent sections or partitions of memory. So, main memory is divided into fixed-sized or variable-sized partitions. Since a partition may not be large enough for the entire process, virtual memory is implemented to keep the processes executing. The answers to several questions are important to implementing an efficient virtual memory system in a multiprogrammed environment.

## 1.2 Effect of multi-programming on CPU utilization

Back in the days when 1 processor contained 1 core capable of running 1 thread, CPU utilization reported by the operating system indicated actual resource consumption (and resource availability) of the processor. In such environment’s CPU utilization grows linearly with increased workload.

**Multi-core CPUs**: 1 processor = 2 or more cores

In multi-core CPUs, where 1 processor contains 2 or more cores, each processing core has its own arithmetic and logic unit, floating point unit, set of registers, pipeline, as well as some amount of cache. However multi-core CPUs also share some resources between the cores (e.g. L3-Cache, memory controller).

**Simultaneous multi-threading CPUs/cores**: 1 processor or core = 2 or more threads (aka "Hyper-Threading", "Chip Multi-threading")

The hardware components of one physical core are shared between several threads. Each thread has at least its own set of registers. Most resources of the core (arithmetic and logic unit, floating point unit, cache) are shared between the threads. Naturally those threads compete for processing resources and stall if the desired units are already busy.

**benefits of resource sharing.**

Resource sharing can increase overall throughput and efficiency by keeping the processing units of a core busy. For instance, hyper-threading can reduce or hide stalls on memory access (cache misses). Instead of wasting many cycles while data is fetched from main memory the current thread is suspended and the next runnable thread is resumed and continues execution.

**disadvantages**

* CPU time accounting measurements (sys/usr/idle) as reported by standard tools do not reflect the side-effects of resource sharing between hardware threads
* It is impossible to correctly measure idle and extrapolate available computing resources

**Idle does not indicate how much more work can be accomplished by the CPU**

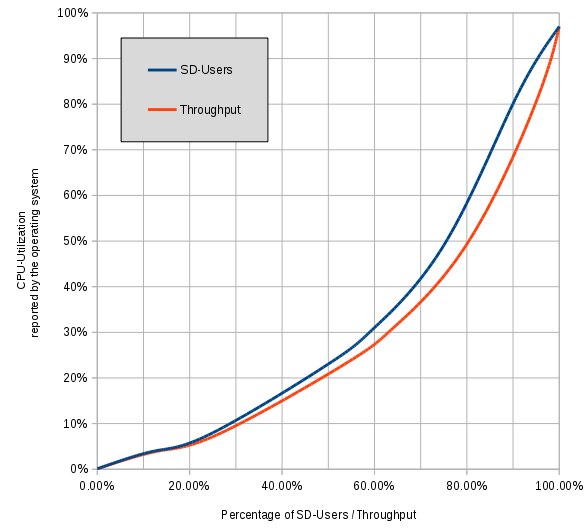
Example:

Assuming 1 CPU core has 4 threads. Currently 2 (single-threaded) processes are scheduled to run on this core and these 2 processes already saturate all available shared compute resources (ALU, FPU, Cache, Memory bandwidth, etc.) of the core. Commonly used performance tools would still report (at least) 50% idle since 2 logical processors (hardware threads) appear completely idle.

In order to correctly estimate how much work can be added until the system approaches full saturation the operating system would need to get detailed utilization information of all shared core processing units (ALU, FPU, Cache, Memory bandwidth, etc.) as well as knowing the characteristics of the workload to be added.

**Measurements with SAP ABAP workload**

To illustrate our case, let's look at a very specific but very common workload in Enterprise Computing: SAP-SD ABAP. We took these measurements on a SPARC T5 system running the latest Solaris 11 release. Simulated benchmark users logged onto the SAP system and entered SD transactions. The maximum number of SD-Users and SAP transaction throughput the system could handle are represented by the 100% mark on the X-Axis. A series of test runs was carried out in order to measure CPU utilization (Y-Axis) as reported by the operating system at 0%, 12.5%, 25%, 50%, 60%, 75%, 90% and 100% of the maximum number of SD-Users.



Unlike what one could naively assume the diagram does not show a straight diagonal line. Instead we see that at 25% of the SD-User / maximum throughput load, the operating system only reports 8% CPU utilization with 92% idle.

At half of the maximum achievable throughput the system only appears to be 21% busy with 79% idle.

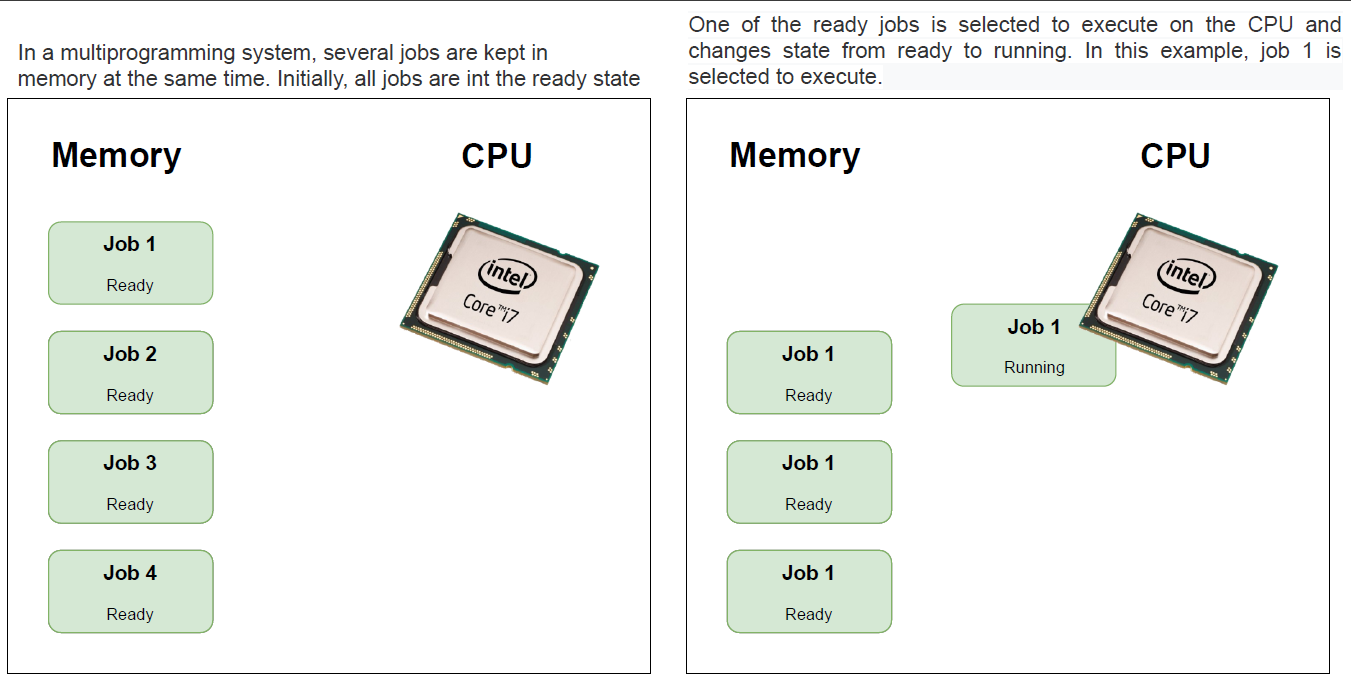
Put it another way, when the OS reports 50% CPU utilization, we are already at 80% of maximum throughput, and cannot assume that adding the same load again would double the throughput with the same response time, yet this is a very common mistake we see reported by customers.

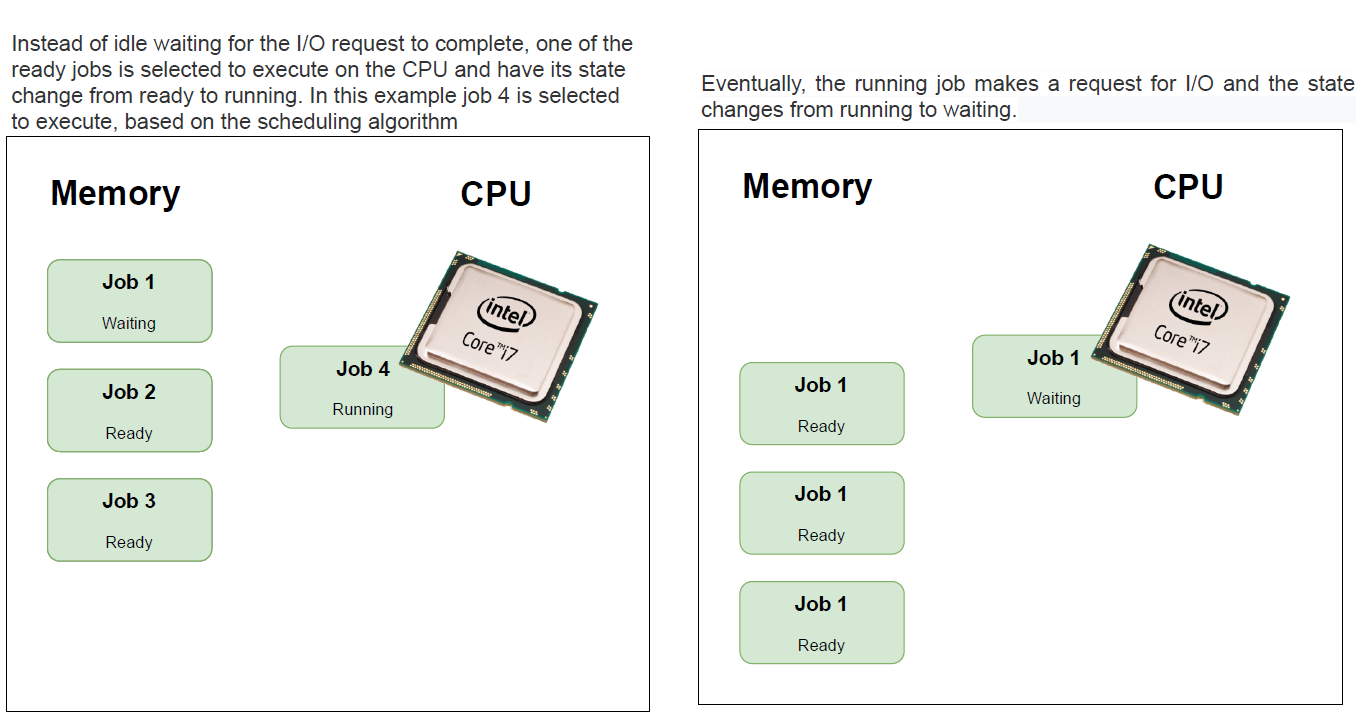
The curve shown on the diagram is highly dependent on the workload (application or application mix) and CPU architecture (number of hardware threads, shared computing resources, etc.). It can be assumed that most applications running on multi-threaded architectures will show this non-linear behavior (more or less pronounced).

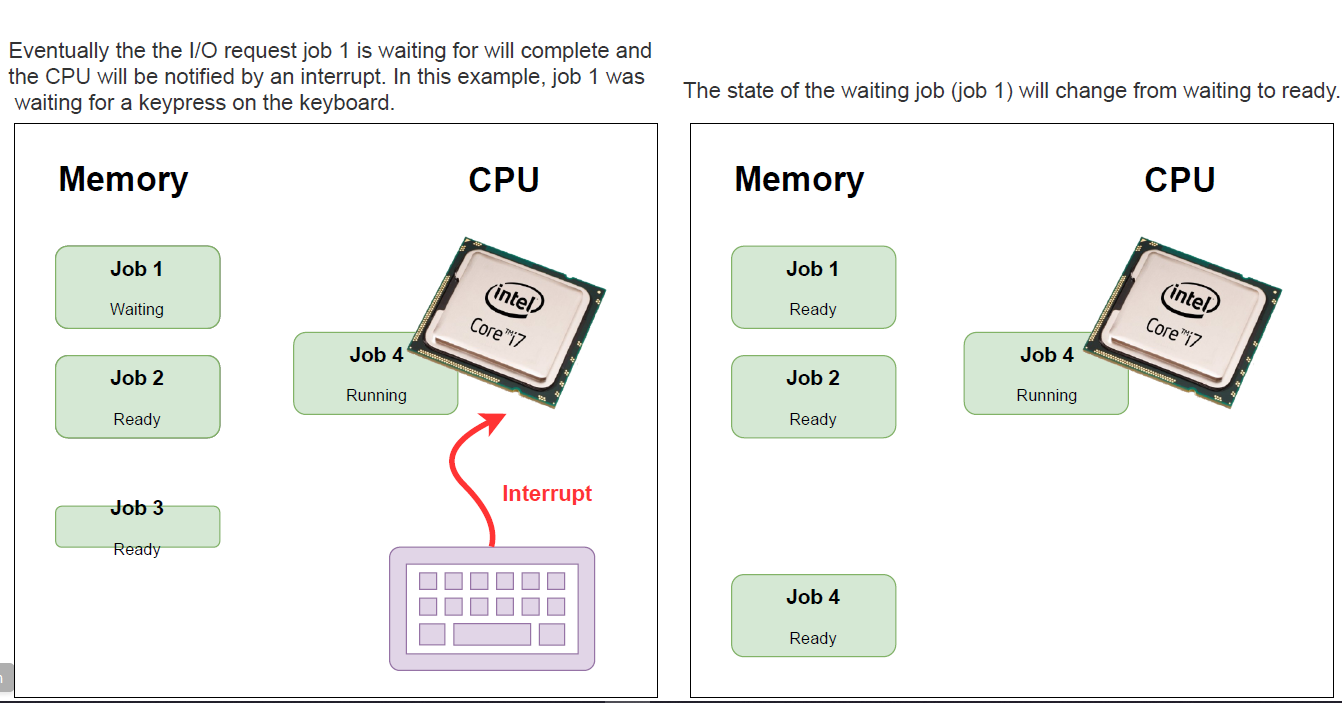
Capacity Planning has become a much more complex affair with the advent of multi-thread/multi-core CPU architectures, and to answer the question of how much more load one can add to an existing system, one has to analyze the workload to be added as well as the current resource consumption.

**Multi programmed system’s working**

In a multi-programmed system, as soon as one job goes for an I/O task, the Operating System interrupts that job, chooses another job from the job pool (waiting queue), gives CPU to this new job and starts its execution. The previous job keeps doing its I/O operation while this new job does CPU bound tasks. Now say the second job also goes for an I/O task, the CPU chooses a third job and starts executing it. As soon as a job completes its I/O operation and comes back for CPU tasks, the CPU is allocated to it.







In this way, no CPU time is wasted by the system waiting for the I/O task to be completed.  
Therefore, the ultimate goal of multi programming is to keep the CPU busy as long as there are processes ready to execute. This way, multiple programs can be executed on a single processor by executing a part of a program at one time, a part of another program after this, then a part of another program and so on, hence executing multiple programs. Hence, the CPU never remains idle.

# 

# **Question No. 2**

**Solution to Question No. 2:**

|  |
| --- |
| **Note:** The sequential approach and the multi-threaded approach are implemented in one program, where command line arguments are used to execute a selected approach. |

## 2.1 Design and implementation of the application using sequential approach with functions

Algorithm for adding two matrices with Sequential Approach:

STEP 1: Start

STEP 2: define MAX = 15,000

STEP 3: generate matrixA[MAX][MAX]and matrixB[MAX][MAX] with random values

STEP 4: initialize a thread thread\_id

STEP 5: Start time using clock\_gettime

STEP 6: pthread\_create(&thread\_id, NULL, &sequential\_add, NULL);

STEP 7: join thread

STEP 8: End time

STEP 9: print sum [][]

STEP 10: print difference of start and end time

STEP 11: Stop

**sequential\_add**(void \*arg)

STEP 1: for i = 0 to MAX

1.1 for j = 0 to MAX

1.1.1 sum[i][j] = matrixA[i][j] + matrixB[i][j]

STEP 2: pthread\_exit

Program For adding two matrices with sequential approach



Figure Driver Code

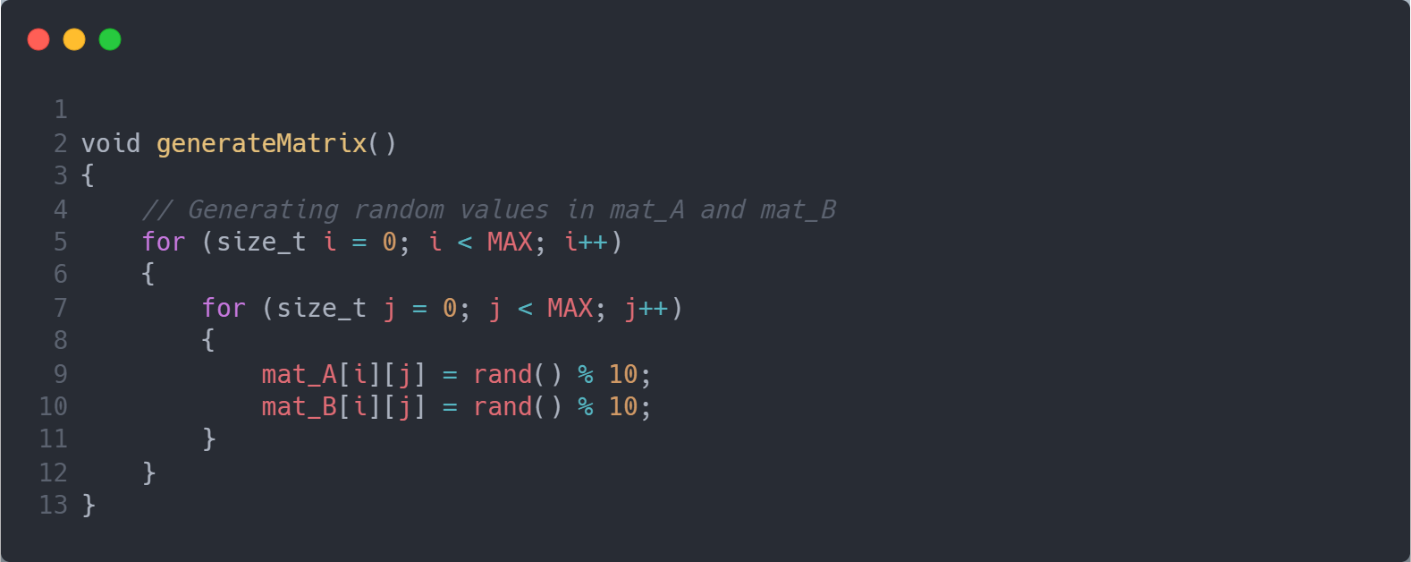


Figure Function to generate random values and store in matrixA and matrixB



Figure Driver Function to add, print and record time in matrix addition

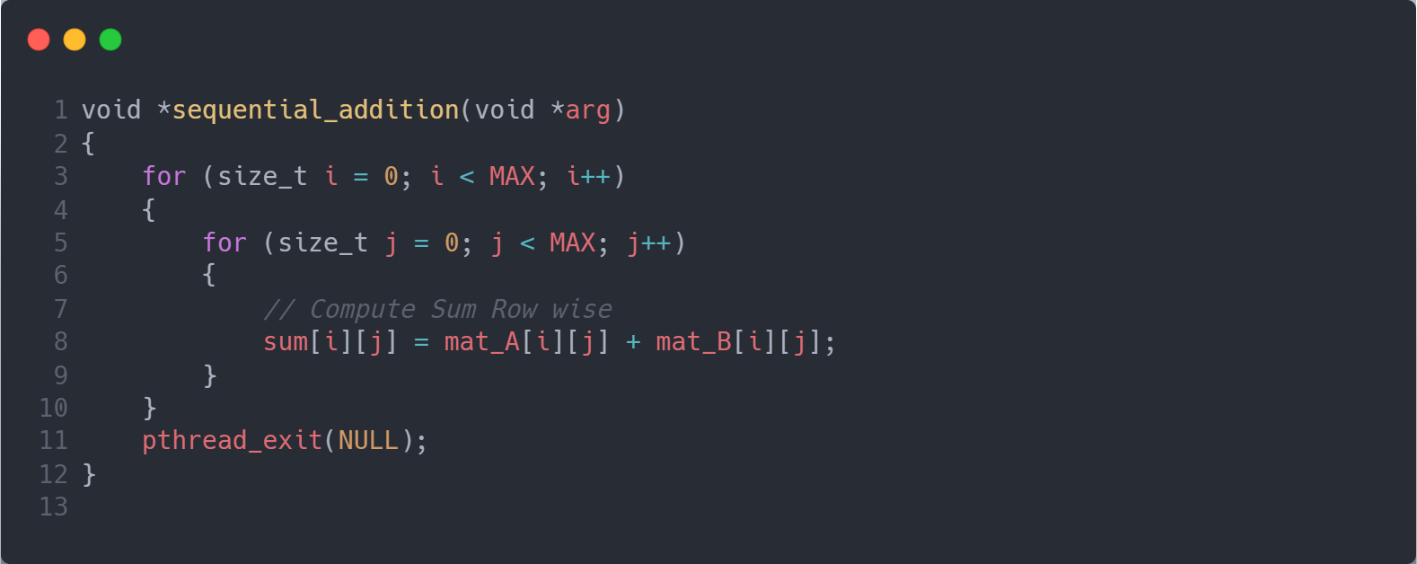


Figure function to add two matrices

Output

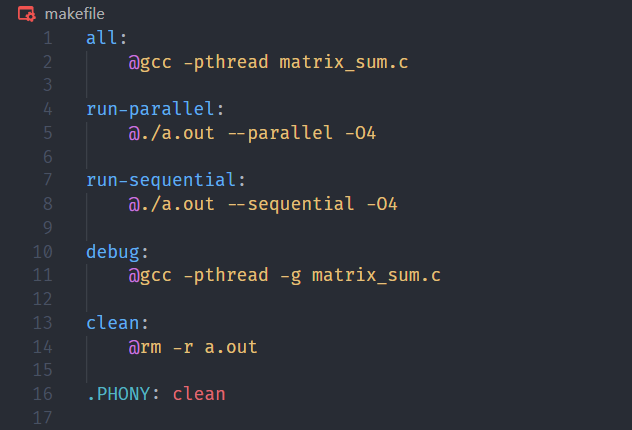


Figure makefile

**Note:** This output is to check whether the program is adding two matrices correctly hence running the program with MAX = 3.

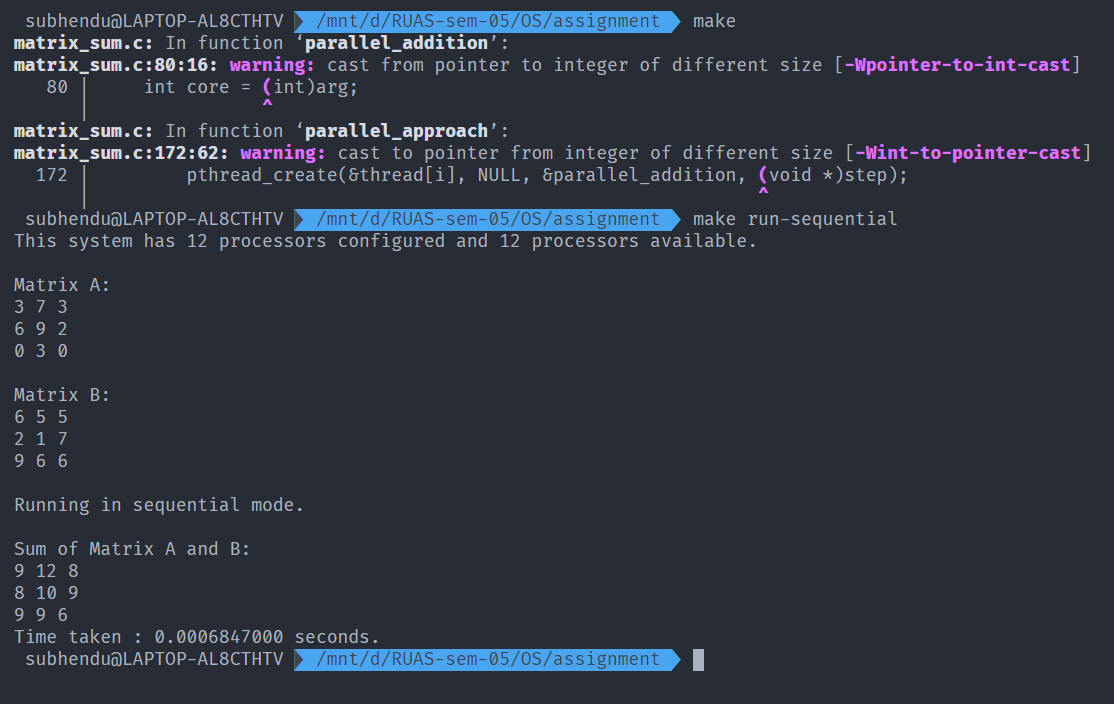


Figure execution of adding two matrices with sequential approach

## 2.2 Design and implementation of the application using multithreaded approach

Algorithm for adding two matrices with Parallel/Multithreaded Approach:

STEP 1: Start

STEP 2: define MAX = 15000 and CORE = 12

STEP 3: generate matrixA[MAX][MAX]and matrixB[MAX][MAX] with random values

STEP 4: initialize threads thread\_id[CORE]

STEP 5: Start time using clock\_gettime

STEP 6: for i = 0 to CORE

6.1 pthread\_create(&thread\_id[i], NULL, &parallel\_add, step);

STEP 7: for i = 0 to CORE

7.1join thread[i]

STEP 8: End time

STEP 9: print sum [][]

STEP 10: print difference of start and end time

STEP 11: Stop

**parallel\_add**(void \*arg)

STEP 1: int core = (int)arg

STEP 2: each thread computes 1/COREth of matrix addition

2.1 for i = (core \* MAX / CORE) to ((core + 1) \* MAX / CORE)

2.1.1 for j = 0 to MAX

2.1.1.1 sum[i][j] = matrixA[i][j] + matrixB[i][j]

STEP 3: pthread\_exit

Program for adding two matrices with parallel/multithreaded approach



Figure Driver Code

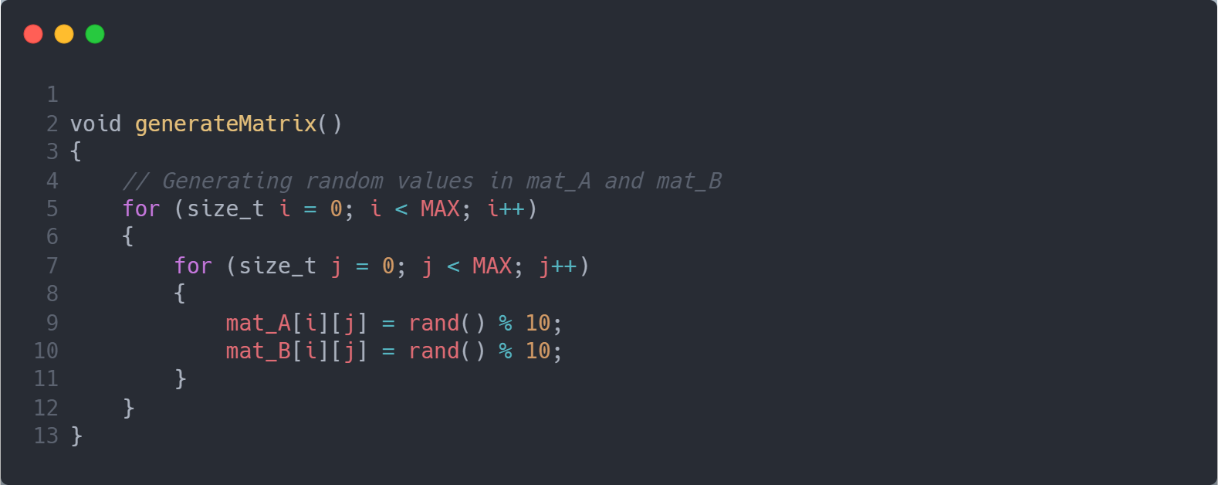


Figure Function to generate random values and store in matrixA and matrixB



Figure Driver Function to add, print and record time in matrix addition

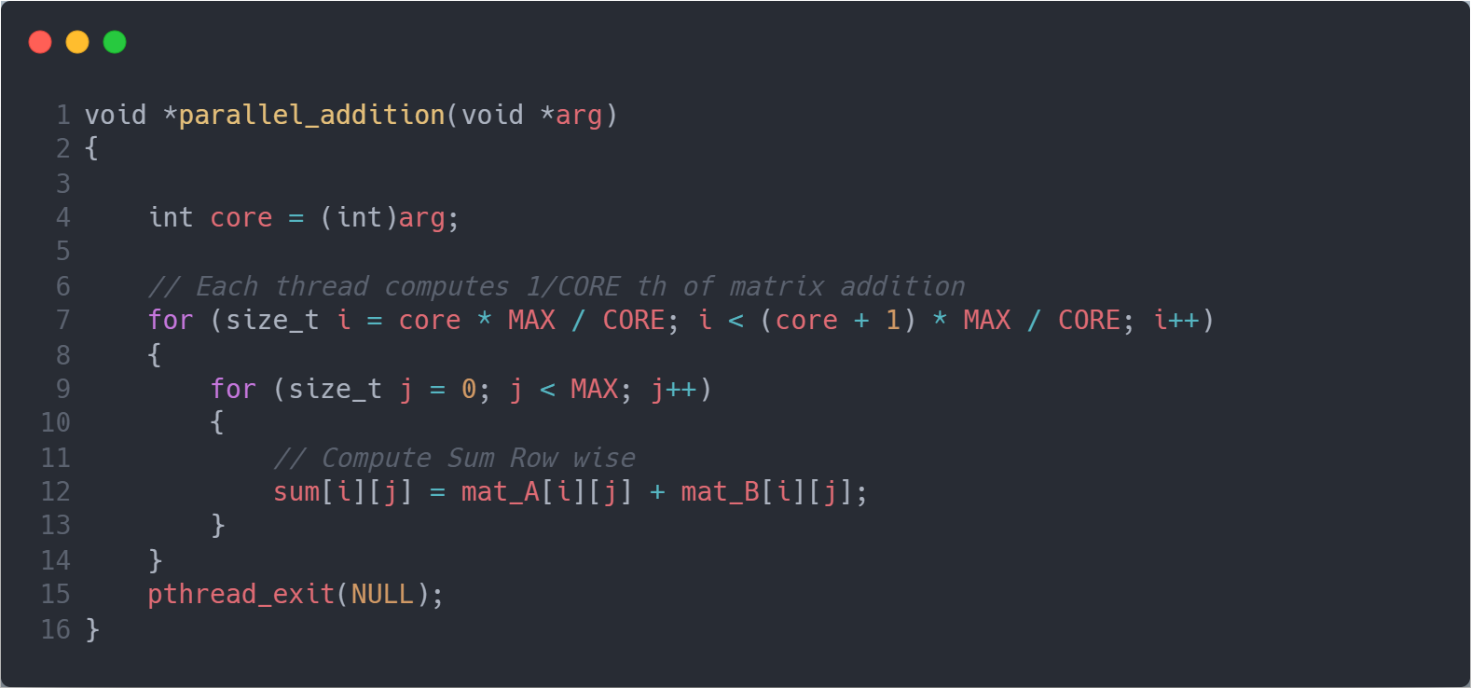


Figure function to add two matrices in multithreads

Output

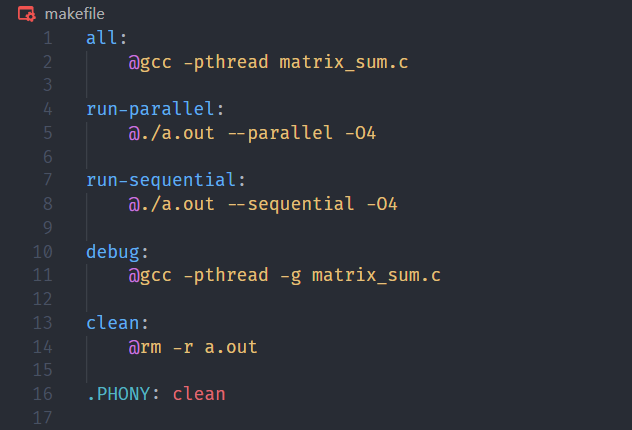


Figure makefile

**Note:** This output is to check whether the program is adding two matrices correctly hence running the program with MAX = 3.

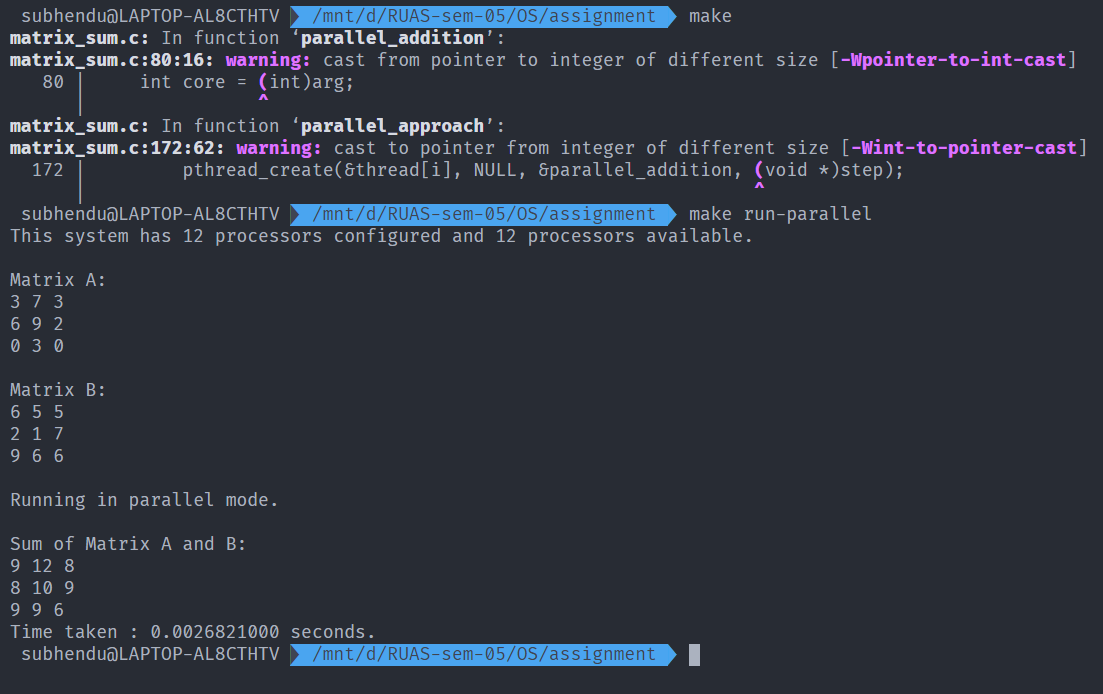


Figure execution of adding two matrices with multithreaded approach

## 2.3 Comparison of the execution time of the above two versions of the program and its analysis

As I have a system with 12 cores.

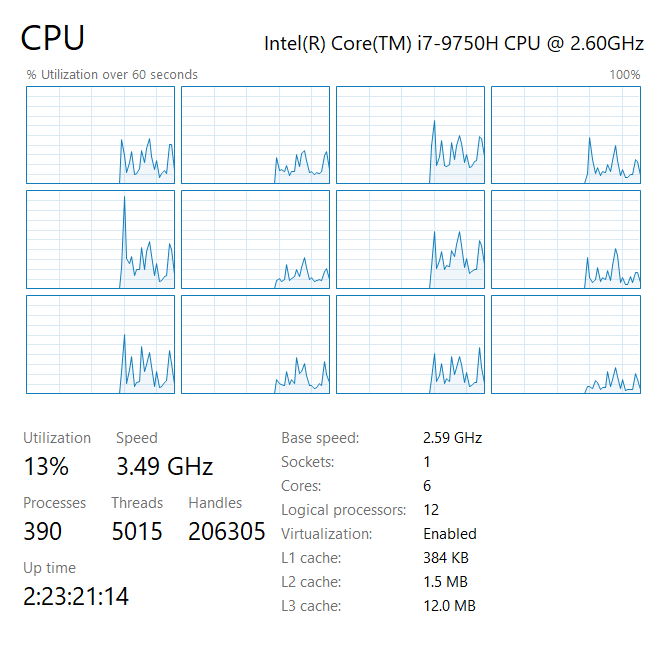


Figure screenshot of CPU config

To get a good comparison of the execution we have to take a larger value of MAX, hence we are taking MAX = 15000 and CORE = 12. This will create 15000 X 15000 matrices and in multithreaded approach each thread will add 1250 row of matrix.

As the matrix size is humungous, it is not possible to display the matrices hence commenting out the code which displaying the matrices and only displaying the time taken in execution.

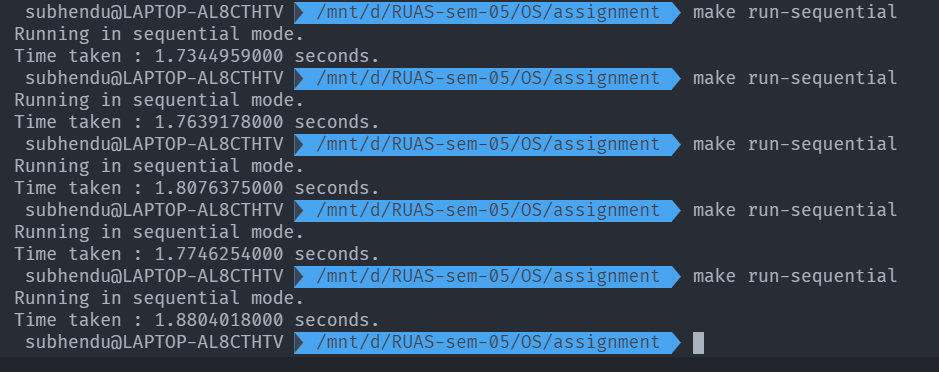


Figure execution of sequential matrix addition with matrix size 15000 X 15000

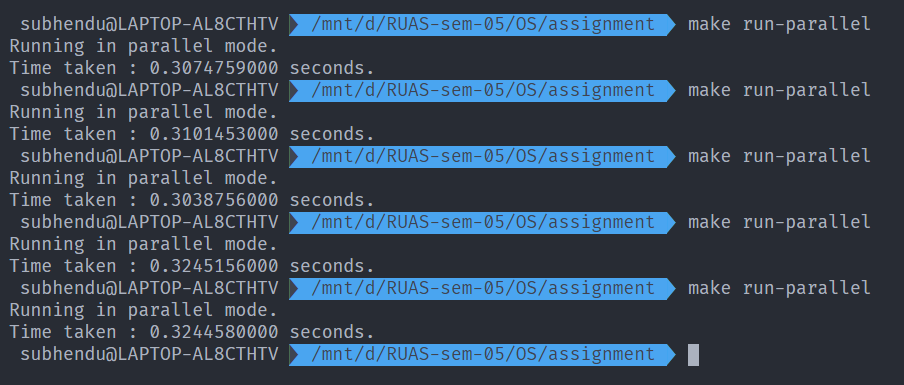


Figure execution of multi-threaded matrix addition with matrix size 15000 X 15000

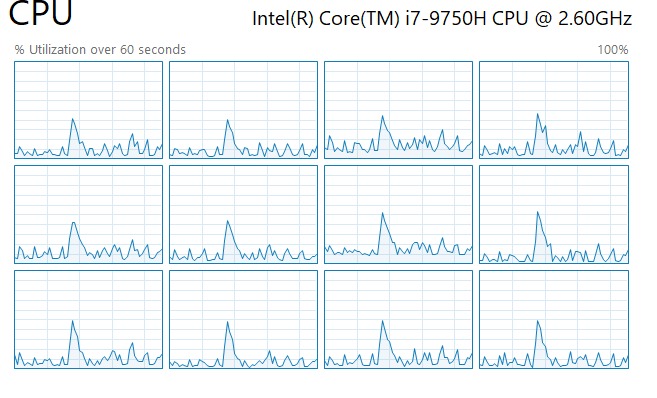


Figure (Hike in all 12 CPU utilization) proof that all the CPU cores are working when executing multithreaded matrix addition

|  |  |  |
| --- | --- | --- |
| Experiment  Number | Sequential Mode  Execution Time (seconds) | Parallel Mode  Execution Time (seconds) |
| 1 | 1.734495 | 0.307475 |
| 2 | 1.763917 | 0.310145 |
| 3 | 1.807637 | 0.303875 |
| 4 | 1.774625 | 0.324515 |
| 5 | 1.880401 | 0.324458 |
|  |  |  |
| Average Execution time | **1.792215** | **0.314093** |

Therefore,

Average execution time in parallel mode .

Average execution time in sequential mode .

Hence, parallel mode is 6 times(approx.) faster than sequential mode.

Thus, the parallel mode execution is faster. This is obvious from the fact that in parallel mode, the addition of 15000 rows divided into 12 threads and processed simultaneously.

In parallel mode, each thread is assigned 1250 rows of matrix to add. But in sequential mode, the main thread processes the whole matrix line by line. Hence, the sequential execution is slower.

# **Question No. 3**

**Solution to Question No. 3:**

## 3.1 Schedule of the processes using a Gantt chart

**Non-Pre-emptive priority scheduling algorithm:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| processes | Burst time  (ns) | Arrival time  (ns) | Priority | Start  time | Completion time | Turnaround time | Waiting time | delay |
| 1 | 10 | 15 | 6 | 37 | 47 | 32 | 22 | 3.2 |
| 2 | 15 | 20 | 8 | 22 | 37 | 17 | 2 | 1.134 |
| 3 | 5 | 25 | 2 | 47 | 52 | 27 | 22 | 5.4 |
| 4 | 12 | 10 | 4 | 10 | 22 | 12 | 0 | 1 |

**Gantt chart for non-pre-emptive:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 4 | 2 | 1 | 3 |

0 10 22 37 47 52

Ready Queue: 4 2 1 3

Number of context switches: 3

**Pre-emptive priority scheduling algorithm:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| processes | Burst time  (ns) | Arrival time  (ns) | Priority | Start  time | Completion time | Turnaround time | Waiting time | delay |
| 1 | 10->5->0 | 15 | 6 | 15 | 40 | 25 | 15 | 2.5 |
| 2 | 15->10->0 | 20 | 8 | 20 | 35 | 15 | 0 | 1 |
| 3 | 5->0 | 25 | 2 | 47 | 52 | 27 | 22 | 5.4 |
| 4 | 12->7->0 | 10 | 4 | 10 | 47 | 37 | 25 | 3.08 |

**Gantt chart for pre-emptive:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 4 | 1 | 2 | 2 | 1 | 4 | 3 |

0 10 15 20 25 35 40 47 52

Ready Queue: 4 1 3 2

Number of context switches: 5

## 3.2 Average waiting time and average turnaround time experienced

Formulae for calculating turnaround time, waiting time and delay:

**Non pre-emptive:**

* **Average turnaround time experienced in non-pre-emptive scheduling is:**

Average turnaround time:

* **Average waiting time experienced in non-pre-emptive scheduling is:**

Average waiting time:

* **Average delay experienced in non-pre-emptive scheduling is:**

Average delay:

**Pre-emptive:**

* **Average turnaround time experienced in pre-emptive scheduling is:**

Average turnaround time:

* **Average waiting time experienced in pre-emptive scheduling is:**

Average waiting time:

* **Average delay experienced in pre-emptive scheduling is:**

Average delay:

## 3.3 Scheduling algorithm with better performance and its justification

By looking at the above solved problem we can say that **non pre-emptive** priority-based scheduling algorithm has better performance than pre-emptive scheduling algorithm as it has

* Less average turnaround time
* Less average awaiting time
* Less average delay
* Less contest switches

than pre-emptive scheduling.

**Bibliography**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. <https://blogs.oracle.com/solaris/cpu-utilization-of-multi-threaded-architectures-explained-v2>
2. <http://www.it.uu.se/education/course/homepage/os/vt18/module-1/multiprogramming/>
3. <https://www.geeksforgeeks.org/difference-between-multitasking-multithreading-and-multiprocessing/>