**CMPE341- OPERATING SYSTEMS**

**PROJECT REPORT (Punto)**

**Project No: 2**

**Scheduling and Process Management (Font)**

Submitted by:

*Öykü Sucuoğlu (CMPE)*

*Yağmur Fatma Birinci (CMPE)*

*Helin Arslan (CMPE)*

*Ozan Şahan (CMPE)*

*Güneş Bade Erdem (CMPE)*

Project Supervisor and Mentors:

Assoc. Prof. Nima Jafari Navimipour

Asst. Prof. Rahim Dehkhargani

Dr. Seyed-Sajad Ahmadpour

Ms. Maryam Zaker

Faculty of Engineering and Natural Sciences

Kadir Has University

Fall 2023

**TABLE OF CONTENTS**

**INTRODUCTION (Punto)**

An operating system acts as the base software for both application programs and system programs. It allows the user to have limited control over the hardware via the application programs or terminals. The (font) constraints of an operating system vary from one another. A common principle for all operating systems can be said as follows: All programs that are part of the OS must become processes for execution. This means that the relationship between the computer hardware and the user interface must be provided by the OS in a way that the computer can understand and work with it. The programs turn into processes for the hardware resources to execute them. Every process requires resources to be executed. These resources are CPU and RAM. The main goals of an operating system are to reduce wait times and enhance throughput by scheduling processes. To reach these goals, CPU utilization for process execution is crucial. Process scheduling algorithms allow the OS to control the execution of processes at the CPU. There are different algorithms used for process scheduling. Before getting into the process scheduling algorithms that were used in our project, explaining process activities will give a better understanding of the algorithms. Process activity is defined by two models. There is a two-state model. A process is either running or not running. The other is the 5-state model. A process can be in one of these five states: new, ready, running, waiting, or terminated. In a multi-programmed system, efficient utilization of the processor is obtained by allowing multiple processes to be ready or waiting for execution. By allowing multiple processes to be in ready or waiting states, when a process is waiting for some I/O operation, the CPU will not remain idle. Meaning that another process in the queue can be executed. To achieve this utilization, CPU scheduling is needed. This leads the way to the process scheduling algorithms. Scheduling algorithms are responsible for the system’s effectiveness. There are preemptive and non-preemptive scheduling algorithms (Rafi, 2017).

In this project, the Round Robin Algorithm is combined with Priority Queue for process scheduling. The RR (Round Robin) Algorithm’s working process is nearly the same as the First Come First Serve (FCFS) Algorithm with two different features: First, RR is a preemption algorithm; secondly, RR uses a Time Quantum (small amount of time) value. (Rafi, 2017) Processes are put together in a circular queue and at each Time Quantum step, another process in the queue is executed. Until every process in the queue is terminated, the algorithm repeats.

As explained by Farooq, Shakoor, and Siddique (2017) in a conference: “This algorithm allocates CPU to all processes for an equal time interval. A process is blocked and put at the end of the ready queue after a constant time slice, known as Time Quantum. This process is assigned CPU time again once the execution of all other processes in their respective time quanta.” (p.244).

Priority scheduling is implemented using a priority queue, and it may be preemptive or non-preemptive. Priority scheduling is based on priority criteria. The priority queue is created by processes, ordered in decreasing order of priority (from head to rear).  The CPU scheduler picks the highest priority first and continues from the queue in decreasing order (Derbala, 2005). In this paper, we will combine these algorithms and simulate a process scheduling system via C++ code. In the methodology part, the environments that were used to create the process scheduling system will be explained. In the Main Findings part, the details of the project code will be explained in detail. In the discussion part, how the combination of these algorithms improves the effectiveness of the system and what constraints we have faced will be discussed. Last but not least, the conclusion part will sum up the project and its results.

**2 METHODOLOGY**

**2.1 System Design**

In this project, we applied our coding knowledge to be able to control the CPU usage effectively. We used Visual Studio to monitor CPU operation. Thanks to Visual Studio, we were able to create a clean and comfortable workspace and compile our code. In addition, we chose the language we used in our project as C++ because it is one of several languages that enable general-purpose programming and is widely used around the world. The reason we used C++ was that it is an object-oriented language and includes OOP concepts such as classes, objects, inheritance, and multiple inheritance. Since we used struct and vector structures in our project, it was convenient for us to use C++.

While creating the project's architecture, we ensured that the operating system simulation was shaped around three main topics. The first is that each process contains duration and resource units. Thus, the process remains in the queue until the duration finishes and its resources are preserved. The second topic is the user entering the first three processes’ duration time. We want the user to determine the duration of the first three initial processes, but we want all other required processes to be created randomly. The last one is, executing the processes sequentially using the Round Robin algorithm and priority queue together.

We used the Round Robin Algorithm because it is considered one of the most efficient ways for process scheduling. By choosing an optimal quantum time, slicing each process’ running time with this chosen value will increase the efficiency of CPU utilization (Putra & Purnomo, 2022). However, choosing an extremely large quantum time will lead to too much response time by the processes. On the other hand, choosing an extremely small quantum time will lead to unnecessarily frequent context switching. That will result in less throughput (Alam & Hoja & Biswas, 2008). Therefore, understanding why quantum time matter played an important role in our project. Understanding how Round Robin works allowed us to control the process scheduling system better. Combining the priority queue with this algorithm made the scheduling system more efficient by decreasing the turnaround time and waiting time of each process. Overall, it allowed us to observe the changes when Round Robin is used with a priority queue.

For our purpose, we provide a simulation framework that uses Round Robin and priority queue to simulate an operating system. Figure 1 shows the flow of the simulation process in an UML activity diagram. This process is included in two main headings: 'beginning before iteration' and 'iteration' in the main. The simulation starts with defining the process vector, resources, and counter. While the counter is less than 3, the process is created and added to the vector. This vector is printed and sorted so that the sorted vector creates the priority queue. It is the first time the unit is printed, and 'i' and 'deadlock' are defined for iteration. Afterwards, the iteration part divided into 'deadlock', 'executed process', 'finished process', and 'generate process' situations. Since this iteration will be done 30 times, the time unit is defined, and this iteration continues as long as the time unit is less than 30. The deadlock section checks if the process has not been used before and has insufficient resources. If the deadlock value becomes 3, the system terminates, or the deadlock is reset, and the flow continues with the executed process part. In the Executed Process section, two different situations are considered depending on whether the process has been used before or not. If this process has not been used before, it is changed to be used, its duration is decreased, and the resources are preserved. On the other hand, if this process has been used before, only its duration decreases. Afterward, the flow continues with the 'finished process' part. The important thing here is to understand whether the process has finished or not. If the duration is not finished, the vector is printed. If it is finished, the ID of the next process is kept in the ‘nextPID’ variable, the time unit is increased, and the current process is deleted. Since ‘TimeUnit’ is increased, the program checks if the time unit is still less than 30 and accordingly. The flow continues by going to the ‘Generate Process’ section. If the time unit is less than 30, a new process is generated and added to the vector. Additionally, the ‘i’ variable is set to ‘index-1' of ‘nextPID’ to initiate execution in the new time unit. Finally, ‘i’ is increased again and connected to the for loop that provides time unit control.

Figure 1 UML Activity Diagram of simulation

**2.2 Vector in C++**

For the purpose of creating a flexible working environment, the vector is utilized instead of the priority queue. Vectors can automatically resize themselves in response to the addition or deletion of a process, and the container takes care of their storage. Furthermore, vectors enable one-to-one navigation for the execution of the process.

For this purpose, as indicated in Figure 3, using the push\_back() function, each generated process was appended to the end of the vector; therefore, the vector used could be simulated as a priority queue. Then, to create prioritized sorting after each push\_back() function, the sort() function is used based on the duration times, involving the condition represented in Figure 2.

Figure 2 The code block, which enables sorting according to the duration times.

Figure 3 The code block allows that each created process can be added to the vector in the initial.

Ultimately, the next PID variable is defined, so if a new process is added, the system does not forget the previous priority process. This variable keeps the ID of the process next to the process that will be deleted. Fundamentally, as demonstrated in Figure 4, upon the addition of a new process, the 'search' value (the index of the nextPID) is assigned to the 'i' value. In the system, the 'i' value ensures navigation within the process. In other words, the 'i' value navigates each process individually when a process with the nextPID value is encountered. This control loop allows us to prioritize the properties of the Round-Robin algorithm when a new process is added to the queue.

Figure 4 This code block specifies which processing will be executed after every recently generated process.

In Figure 5, in the case that duration values of 4, 2, and 1 are entered by the user for the initial processes, these processes are appended to the vector and ordered according to their priority. On the other hand, Figures 6 and 7 represent that when a new process is appended after the initializations, the new execution process begins with the nextPID value, which is specified before the erase() function.

As a result, the vector, which ensures prioritizing through the sort() function, acts as a priority queue and simplifies the execution process since it allows navigation within the vector.

Figure 5 Adding process to the vector and sorting.

Figure 6 Choosing which process to execute when a different new process had entered the queue.

Figure 7 Choosing which process to execute when a different new process had entered the queue.

**2.3 System Simulation**

The libraries we use in the C++ code we write when creating our simulation are of great importance. We first used 4 libraries to write this code. These libraries and their roles are as follows:

<iostream>: It contains the basic objects required to receive duration time input from the user and to print the output on the screen.

<vector>: It contains the vector class to create the vector, which is a structure that we use as the priority queue

<algorithm>: It contains sort, which is the prioritization algorithm we provide when comparing the vector to a priority queue.

<ctime>: It includes time, which allows process resources to be randomly assigned.

After adding the libraries, since all processes contain the same variables, the common variables that all processes contain in the structure were defined as seen in Figure 8. Since the type of the 'usedBefore' variable is bool, it can only take the value true or false. Every process is initially defined as false. Represents if this process has been used before.

Figure 8 Structure of processes

In Figure 1, the code of the initial part of the main before the iteration is shown in Figure 9. Here, vector creation and processes are added to the vector. In Figure 10, the first processes are printed on the screen with the ‘PrintInitial’ function, and the processes are prioritized in the vector with the ‘comparingDurations’ function, which allows sorting according to the duration of the sorting process. 'deadlock' and 'i' variable are defined in this part of the code. 'i' is the variable that allows us to navigate between processes in the vector and is always used as 'i %3' in the code. 'Deadlock' is the variable that will cause the system to terminate if no process in the vector can be started.

Figure 9 Initial part before iteration

Figure 10 Comparison and print functions

Since the processes to be carried out after this point will continue until the time unit reaches 30, they will continue in a for-loop. Figure 11 shows the code block that controls the deadlock status. As long as a process that cannot be started is encountered in the while loop, the value of the 'i' and 'deadlock' variables increases. If no process can be started, the system shuts down when the deadlock reaches 3 with the if-statement so that the while loop does not last forever. The first if-statement causes the while loop to stop, and the second break causes the for loop containing the ‘TimeUnit’ to stop.

Figure 11 Code block that provides deadlock status.

If the process can exit the while loop in Figure 11, it means its resources are sufficient, even if they have not been used before. Therefore, as shown in Figure 12, if the ‘usedBefore’ variable indicates that it has not been used before, it preserves the resources of the process, replaces ‘usedBefore’ with true, which means used, and reduces the duration time. If it has already been used, then it just reduces the duration of the process.

Figure 12 Code block that provides the executing process part.

After these executions, it is checked whether the current process has finished or not. If the process is finished, the actions shown in Figure 13 take place. If the process is not finished, it prints the vector, increments 'i' to move on to the next process, and reaches the end of the for loop shown in Figure 9. However, if the process is finished, it releases the resources of the process. In addition, before the process is deleted, the ID of the process that will be executed in the next timeUnit is assigned to the nextPID variable. It increases the timeUnit again because the generated process must be printed to the screen in the new time unit.

Figure 13 Code block that provides the finishing process part.

If the ‘timeUnit’ is not 30, the process shown in Figure 14 is generated and printed in the new time unit. Afterward, it is determined which process will be executed in the next ‘timeUnit’, which is represented in Figure 6 and Figure 7. For this, each process in the vector is controlled in order with the for loop in line 163, shown in Figure 14. Search is initially defined as 0 because it is a search variable used to represent the index of ‘nextPID’. When the process ID with the value ‘nextPID’ is found in the for loop, the variable 'i' is changed to 'search-1'. Because 'i' will be incremented by one at the end of the code, the 'i' variable will have the value of 'search' at the end of the for loop.

Figure 14Code block that provides the generating process part

As a result, the first ‘TimeUnit’ is created at the beginning of the main function. The remaining ‘TimeUnit’ values were implemented in a for loop. Increasing the 'i' variable at the end of each for loop enabled the next process to be examined in the next ‘TimeUnit’. At the end of this examination, the 'i' value increased again, either because it had not been started before and its resources were not sufficient, and the next process began to be examined, or it could be executed because its resources were sufficient. If the process finished as a result of the execution, the process was deleted from the vector and if the ‘TimeUnit’ was available, a new process was created.

**3 MAIN FINDINGS**

In Figure 15, it is seen that the simulation prints 30 units of time according to the duration times received from the user. The user entered a duration of 4 for P1, 2 for P2, and 6 for P3.

Figure 15 View of the code in the terminal after running

**3.1 Duration Time**

When the simulation runs, Figure 16 shows the processes listed in order of priority in Time Unit 1. No processes have been run yet in this Time Unit. In Time Unit 2, the highest priority process runs and its units of resources are preserved while its duration decreases by one.

Figure 16 Changing the duration time of the process

**3.2 Process Generation**

While the system is running, the process with duration 0 finishes and exits the system, and a new process is created in its place. In Time Unit 3, P2 exits the queue because its duration has expired. In Time Unit 4, Figure 17 shows that P4 is created instead of P2, which exits the system. While the system is running, the process with a duration of 0 finishes and is deleted from the queue. In addition, a new process is generated and added to the queue. In Time Unit 3, P2 exits the queue because it has finished. In Time Unit 4, Figure 17 shows that P4 is randomly generated instead of P2.

Figure 17 Generating a new process

**3.3 Inter-Process Priority**

As seen in Figure 18, after a process is finished and deleted from the queue, a new process is generated. Since the P8 had the least duration, it came to the front of the queue. However, as shown at time unit 29 of Figure 18, the first process to run was P7 because P7 was the next priority before P5 was deleted.

Figure 18 Prioritization between processes

**3.4 Failure to Start the Process**

After a new process is added to the queue (seen in Time Unit 15), its execution may not be started in its turn (seen in Time Unit 17). If the system's resources cannot provide the resources the process needs, the process cannot be started. As seen in Figure 19, it will not be able to start until the system's resources are enough. If the newly added process, P5, needed a resource that is being used by a previous process and the second added process, P6, depends on P5’s resources, a deadlock would be encountered. However, in this process scheduling system, the limited number of resources one process can use minimizes the risk of deadlock.

Figure 19 Failure to start the process

**4 DISCUSSION**

As it is explained thoroughly in the introduction section of our report, there are various ways for process scheduling, and still, new algorithms were being developed. In the project, we combined priority scheduling and Round Robin algorithms. Round Robin is considered one of the most effective algorithms for process scheduling (Vayadande, K. et al. 2023). Allowing each process to have a fair share of the CPU resources and an equal time of execution at the CPU makes Round Robin more efficient than other algorithms. Also, reducing the starvation by a fair share of CPU is one of the major advantages of Round Robin. Combining the Priority queue for the Shortest Job First and the Round Robin Algorithm, we were able to dynamically adjust the priority of processes based on their burst times. Because of our priority criteria, the shortest job has the highest priority. The priority queue is formed by this criteria. The waiting time and turnaround time are minimized by executing the shortest job first. The Round Robin Algorithm’s fair time distribution for each process allows longer jobs to eventually get their chance to execute. There were some constraints we came across while writing the code. Both algorithms have their priorities. It was a problem to decide which algorithm’s operation would take place when a new process is entered into the queue. In our project, when a new process was entered into the queue the priority queue is reordered according to the shortest job criteria. The next process to be processed in the CPU should be the process at the header of the queue because it is the shortest job. However, if a process is expected to be executed before the added process is in the queue, the Round Robin algorithm prioritizes the execution of that process in the next time step (Karapici, et al. 2015). This is the most powerful advantage of combining these algorithms. Even if the duration time of the added process is smaller, the starvation of other processes is prevented with the help of Round the Robin Algorithm.

**5 CONCLUSION**

In conclusion, how an operating system can achieve CPU utilization by using two different process scheduling algorithms together was examined, which are the priority algorithm and the Round Robin Algorithm. It was observed that using these two algorithms together in the project had reduced effects on waiting time and turnaround time. Using the PQ and RR algorithms together also reduced the possibility of starvation. The C++ language was used in the Visual Studio Code environment to create the process scheduling system. Furthermore, by using the vector structure, the priority order was determined for each PID value with random numbers that were obtained from the user; therefore, the necessary operations were executed sequentially in every step. In the simulation, for every 30 units of time row, ordered values in the priority queue, the currently executed process, and the rest of the accessible resources after the execution are displayed successfully. For the future advancement of the project, distinct algorithms could be utilized and combined to decrease waiting time and promote the efficiency of the system.

**REFERENCES**

Alam, B., Doja, M. N., & Biswas, R. (2008). Finding time quantum of round robin CPU scheduling algorithm using fuzzy logic. 2008 International Conference on Computer and Electrical Engineering. https://doi.org/10.1109/iccee.2008.89

Derbala, A. (2005, February). Priority queuing in an operating system. *Computers &*  *Operations Research*, *32*(2), 229–238.

Farooq, M.U., Shakoor, A. and Siddique, A.B. (2017) ‘An efficient dynamic round robin

algorithm for CPU scheduling’, 2017 International Conference on Communication,

Computing and Digital Systems (C-CODE) [Preprint]. doi:10.1109/c-code.2017.7918936.

Karapici, A. et al. (2015) ‘The simulation of Round Robin and priority scheduling algorithm’, 2015 12th International Conference on Information Technology - New Generations [Preprint]. doi:10.1109/itng.2015.131.

Putra, T.D. and Purnomo, R. (2022) ‘Simulation of priority round robin scheduling algorithm’, Sinkron, 7(4), pp. 2170–2181. doi:10.33395/sinkron.v7i4.11665.

Rafi, U. *et al.* (2017) ‘Multi-queue priority based algorithm for CPU process scheduling’,

*Proceedings of the Eleventh International Conference on Management Science and*

*Engineering Management*, pp. 47–62. doi:10.1007/978-3-319-59280-0\_4.

Vayadande, K. et al. (2023) ‘An improved way to implement round robin scheduling algorithm’, Lecture Notes in Electrical Engineering, pp. 403–414. doi:10.1007/978- 981-99-6690-5\_30.