Journal of Studies in Science and Engineering

EngiScience Publisher

Review Article

Scheduling and Synchronization Algorithms in Operating System: A Survey

Mohammed Y. Shakor [1,*]

[1] English Department, College of Education, University of Garmian, Kalar, 46021, Iraq

*Corresponding Author: Mohammed Y. Shakor, E-mail: mohammed.yousif@garmian.edu.krd

Article Info

Article History

Received Sep 20, 2021

Revised Oct 23, 2021

Accepted Nov 08, 2021

Keywords

CPU Scheduling

Multiprocessor

Round-Robin Scheduling

Priority Scheduling

Real-time

Abstract

An operating system is software that is designed to manage computer hardware and software resources. However, this management requires applying an ample number of techniques and algorithms which are called synchronization and scheduling. The scheduling algorithms are used to arrange the way that the CPU is assigned to the processes, while synchronization is utilized to indicate how to work with multi-processes at the same time. Therefore, they are related to each other. CPU scheduling is a vital phenomenon of an operating system. At present, numerous CPU scheduling algorithms exist as First Come First Serve **(**FCFS**)**, Shortest Job First (SJF), Shortest Remaining Time First (SRTF), Priority Scheduling, and Round Robin (RR). In this paper, a survey of the current synchronization and scheduling algorithms have been presented. An overview of each technique with the main algorithms have been described in detail with the advantages and the issues of each algorithm. Furthermore, this paper has dug deep into the real-time operating system scheduling issues, which is the current trend in operating system researches.

1. Introduction

Back in the old days of batch systems with input in the form of card images on a magnetic tape, the scheduling algorithm was simple: just run the next job on the tape. With timesharing systems, the scheduling algorithm became more complex, because there were generally multiple users waiting for service. There may be one or more batch streams as well (e.g., at an insurance company, for processing claims). On a personal computer, you might think there would be only one active process. After all, a user entering a document on a word processor is unlikely to be simultaneously compiling a program in the background. However, there are often background jobs, such as electronic mail daemons sending or receiving e-mail. You might also think that computers have gotten so much faster over the years that the CPU is rarely a scarce resource anymore. However, new applications tend to demand more resources. Processing digital photographs or watching real-time videos are examples.

The problem of scheduling is concerned with searching for optimal (or near-optimal) schedules subject to several constraints. A variety of approaches have been developed to solve the problem of scheduling.

However, many of these approaches are often impractical in dynamic real-world environments where there are complex constraints and a variety of unexpected disruptions. In most real-world environments, scheduling is an ongoing reactive process where the presence of real-time information continually forces reconsideration and revision of pre-established schedules. Scheduling research has largely ignored this problem, focusing instead on the optimization of static schedules. This paper outlines the limitations of static approaches to scheduling in the presence of real-time information and presents many issues that have come up in recent years on dynamic scheduling.

In this paper, an overview of both synchronization and scheduling techniques in OS are going to be introduced, addressing the advantages and disadvantages of using algorithms in both techniques and the issues that call to use synchronization and scheduling.

2. Scheduling Criteria

Various CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another [1]. For selection of an algorithm for a particular situation, we must consider the properties of various algorithms. The scheduling criteria include the following [2]:

- Context Switch: A context switch is a process of storing and restoring context (state) of a preempted process, so that execution can be resumed from the same point at a later time. Context switching is usually computationally intensive, leading to wastage of time and memory, which in turn increases the overhead of the scheduler, so the design of the operating system is to optimize only these switches.
- Throughput: Throughput is defined as a number of processes completed per unit time. Throughput is slow in round-robin scheduling implementation. Context switching and throughput are inversely proportional to each other.
- CPU Utilization: This is a measure of how much busy the CPU is. Usually, the goal is to maximize CPU utilization.
- Turnaround Time: Turnaround time refers to the total time which is spent to complete the process and is how long it takes the time to execute that process. The time interval from the time of submission of a process to the time of completion is the turnaround time. Total turnaround time is the sum of the periods spent waiting to get into memory, waiting time in the ready queue, execution time on the CPU and doing I/O.
- Waiting Time: Waiting time is the total time a process has been waiting in the ready queue. The CPU scheduling algorithm does not affect the amount of time during which a process executes or does input-output; it affects only the amount of time that a process spends waiting in a ready queue.
- Response Time: In an interactive system, turnaround time may not be the best measure. Often, a process can produce some output fairly early and can continue computing new results while previous

results are being produced to the user. Thus, response time is the time from the submission of a request until the first response is produced that means time when the task is submitted until the first response is received. So, the response time should be low for best scheduling.

3. Scheduling Algorithm

Scheduling is characterized as the portion of assets to employments after some time. It is a basic leadership process to improve at least one destination. The goals can be the minimization of the finishing time of occupations, mean stream time, the delay of employments, preparing cost, and so on [1].

Scheduling assumes a vital part in numerous assembling and creation frameworks. Planning issues, which are worried about hunting down ideal (or close ideal) prescient timetables subject to various limitations, are for the most part NP-hard. Up until this point, studies have principally been centred around finding ideal (or close ideal) answers for static models as for different measures, i.e. most brief aggregate preparing time, negligible creation cost, and so on. These methodologies for the most part have utilized the verifiable supposition of static conditions with no sort of disappointments. CPU scheduling is a key concept in computer multitasking, multiprocessing operating systems and real-time operating system designs. Scheduling refers to the way processes are assigned to run on the available CPUs, since there are typically many more processes running than there are available CPUs [1]. CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU [2]. By switching the CPU among processes, the operating system can make the computer more productive. A multiprogramming operating system allows more than one process to be loaded into the executable memory at a time and for the loaded processes to share the CPU using time multiplexing. Figure 1 shows the main principles of CPU scheduling with the states in the CPU.

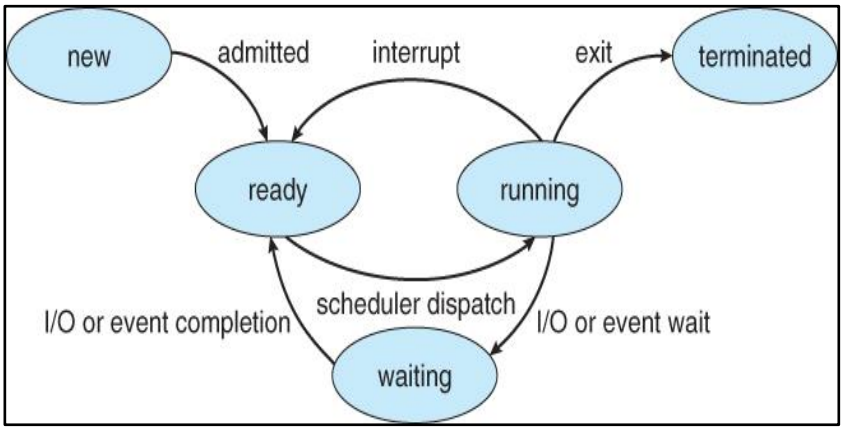

Figure 1. Basic CPU scheduler

The method by which threads, processes or data flows are given access to system resources (e.g. processor time, communications bandwidth) is called a scheduling algorithm. The need for a scheduling

algorithm arises from the requirement for most modern systems to perform multitasking (execute more than one process at a time) and multiplexing (transmit multiple flows simultaneously).

A system designer must consider a variety of factors in designing a scheduling algorithm, such as the type of systems used and what are the user's needs. Depending on the system, the user and designer might expect the scheduler to:

1. Maximize throughput: A scheduling algorithm should be capable of servicing the maximum number of processes per unit of time.
2. Degrade gracefully under heavy load.
3. Avoid indefinite blocking or starvation: A process should not wait for an unbounded time before or while processing service.
4. Enforcement of priorities: if the system assigns priorities to processes, the scheduling mechanism should favor the higher-priority processes.
5. Minimize overhead: Overhead causes wastage of resources. But when we use system resources effectively, then overall system performance improves greatly.
6. Achieve balance between response and utilization: The scheduling mechanism should keep the resources of the system busy.
7. Favor processes exhibit desirable behavior.

3.1 First-Come, First-Served Scheduling

First-Come-First-Served (FCFS) is the simplest scheduling algorithm, it simply queues processes in the order that they arrive in the ready queue. Processes are dispatched according to their arrival time on the ready queue. Being a non-preemptive discipline, once a process has a CPU, it runs to completion. The FCFS scheduling is fair in the formal sense or human sense of fairness but it is unfair in the sense that long jobs make short jobs wait and unimportant jobs make important jobs wait.

3.2 Shortest-Job-First Scheduling

Shortest Job First (SJF) is the strategy of arranging processes with the least estimated processing time remaining to be next in the queue. It works under the two schemes (preemptive and non-preemptive). It’s probably optimal since it minimizes the average turnaround time and the average waiting time. The main problem with this discipline is the necessity of previous knowledge about the time required for a process to

complete. Also, it undergoes a starvation issue especially in a busy system with many small processes being run.

3.3 Priority Scheduling

The operating system assigns a fixed priority to every process, and the scheduler arranges the processes in the ready queue in order of their priority. Lower priority processes get interrupted by incoming higher priority processes.

Overhead is not minimal, nor is it significant in this case. Waiting time and response time depend on the priority of the process. Higher priority processes have smaller waiting and response times. Deadlines can be easily met by giving higher priority to the earlier deadline processes. Disadvantage: Starvation of lower priority processes is possible if a large no of higher priority processes keeps arriving continuously[3].

3.4 Round-Robin Scheduling

Round Robin (RR) is one of the oldest, simplest and fairest and most widely used scheduling algorithms, designed especially for time-sharing systems [4]. It's designed to give a better response but the worst turnaround and waiting time due to the fixed time quantum concept. The scheduler assigns a fixed time unit (quantum) per process usually 10-100 milliseconds, and cycles through them. RR is similar to FCFS except that preemption is added to switch between processes.

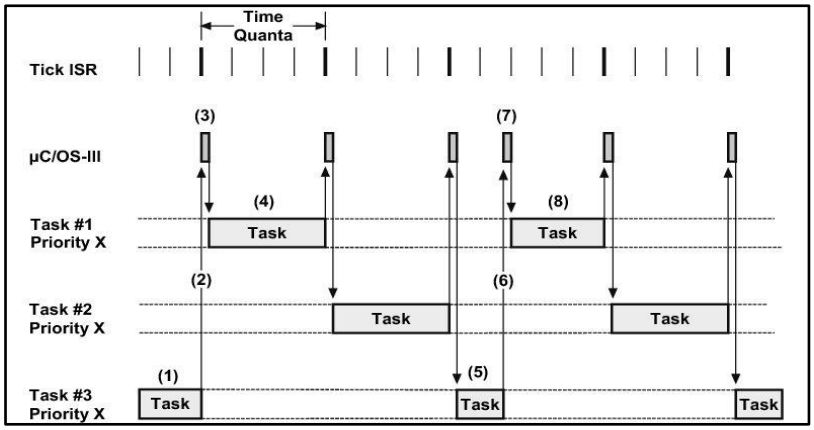

Figure 2. Round-Robin algorithm example

The RR planning calculation is given by the following advances [4]:

1. The scheduler keeps up a line of prepared procedures and a rundown of shut and swapped out procedures.

2. The Process Control Block of a recently made process is added to the end of a prepared line. The Process Control Block of the ending process is expelled from the booking information structures.
3. The scheduler dependably chooses the Process Control Block from the leader of the prepared line. This is a drawback since all procedures are fundamentally given a similar need. Round robin additionally supports the procedure with short CPU bursts and punishes long ones.
4. When a running procedure completes its opportunity cut, it is moved to the end of a prepared line. A period cut is a measure of time that each procedure spends on the processor per emphasis of the Round Robin calculation. All procedures are executed in first start things out serve way however are seized after a period cut. The procedure will either complete in the time cut given or the procedure will come back to the tail of the prepared line and come back to the processor at a later time.
5. The occasion handler plays out the accompanying activities:
   a. When a procedure influences a contribution to yield ask for or swapped out, its Process Control Block is expelled from prepared line to blocked/swapped out rundown.
   b. When input-yield activity anticipated by a procedure completes or process is swapped in its Process Control Block is expelled from blocked/swapped rundown to the finish of the prepared line.

There are some disadvantages of the Round Robin CPU scheduling algorithm for the operating system which are as follows:

- Larger waiting time and response time: In the Round Robin architecture the time which a process spends in the ready queue waiting for the processor to get executed is known as waiting time and the time when the process takes to complete its job and exit from the task is called as turnaround time. Larger waiting and response time are a drawback in the Round Robin architecture as it leads to degradation of system performance.
- Context Switches: When the time slice of the task ends and the task is still executing on the processor the scheduler forcibly preempts the tasks on the processor and stores the task context in stack or registers and allocates the processor to the next task in the ready queue. This action which is performed by the scheduler is called a context switch. Context switch leads to the wastage of time, memory and leads to scheduler overhead.

- Low throughput: Throughput is defined as the number of processes completed per time unit. If round robin is implemented in soft real-time systems throughput will be low which leads to severe degradation of system performance. If the number of context switches is low then the throughput will be high. Context switch and throughput are inversely proportional to each other.

3.5 Multilevel Queue Scheduling

In the circumstance in which the procedure is partitioned into various gatherings, multi-level line planning is utilized. The attributes of multi-level planning are as per the following: Based on the sorts, the procedures are isolated into various lines. Procedures are doled out to one line for all time. The booking calculation in each line is extraordinary. For instance, as appeared in the figure beneath, intelligent process and clump employment may use the Round Robin planning technique and FCFS strategy separately. Additionally, the line must be planned and for the most part, it has settled need. A typical division between frontal area procedures and foundation forms is made. These two kinds of procedures may contrast in their reaction times; thusly, they may require distinctive booking. The frontal area procedures may have need over foundation procedures and this need might be characterized remotely. The bunch line must be executed when the line for framework executes and the intelligent procedures are sat without moving or exhausting. If the bunch procedure is still in process and the intelligent procedure enters the prepared line, the cluster will be avoided.

3.6 Multilevel Feedback Queue Scheduling

MLFQ utilizing Three lines: In MLFQ (Multi-level criticism line) planning, the line is separated into three sections where two lines have Round Robin booking method and the staying one has FCFS booking system [5]. Every one of the procedures is arranged in the first line as indicated by their burst time and after that, they are permitted to execute for a particular time. At the point when the procedures finish their underlying execution, they are arranged in the second line as per their outstanding burst time. After the execution, the procedures are moved to the third line where they are arranged to keep running with the FCFS booking method. This calculation limits the holding up time and turnaround time however CPU needs to hold up to assemble a line of the considerable number of procedures. It is the fundamental driver that restricts the best usage of assets.

MLFQ utilizing Five lines: In this procedure, the procedures are planned for five distinct lines. Starting need isn't relegated to the procedure yet they are booked utilizing Round Robin planning with a reasonable time quantum esteem when the procedure is started. The procedures are booked and permitted to keep running in a line as per their burst time [6]. At last, either the procedures are finished or they are arranged to keep running into the second line to run again with the rest of the burst time and holding-up time. This booking is finished utilizing the Round Robin CPU planning system which likewise doles out an appropriate time quantum esteem. Holding up time and remaining CPU burst time and turnaround time are the principal parameters that are ascertained and refreshed in each progression. Toward the finish of the second line, either the procedures get finished or they are additionally arranged to the next line till every one of the procedures gets executed. A few procedures need to wait for a long time for execution. This planning calculation executes every one of the procedures in parallel. Along these lines, this calculation expels the starvation issue of various procedures. For this situation, the quantity of switches is more a direct result of the capacity and estimations of burst time and holding-up time of each line.

3.7 Non-preemptive scheduling algorithm

Non-preemptive or also known as cooperative scheduling is the first scheme where once a process has control of the CPU no other processes can preemptively take the CPU away. The process retains the CPU until either it terminates or enters the waiting state. Two algorithms can be used for non-preemptive scheduling.

4. Multiple-Processor Scheduling

When a computer system contains multiple processors, a few new issues arise. Multiprocessor systems can be categorized into the following: loosely coupled or distributed. The latter consists of a collection of relatively autonomous systems connected with an interconnection network. Each of them has its memory and I/O Channels [7]. A system with functionality specialized processors, or servers, such as an I/O, network, graphics, or a math coprocessor, works in an environment controlled by a general-purpose, master processor, to provide specific services. A tightly coupled multiprocessing system consists of processors that share a common memory and are under the control of an operating system. The presently popular multicore architecture falls into this category.

One way to characterize and compare multiprocessor systems is to consider their synchronization granularity, namely, the frequency of synchronization between processors in a system. We can thus categorize parallelism in terms of their granularity degree; with independent parallelism, there is no explicit synchronization among processes. Each process in the system represents a separate, independent application, or job. For example, in a time-sharing system such as turning, each user performs a particular application, such as c programming, system services, database related stuff, etc. The multiprocessor system thus provides the same service as a multi-programmed uniprocessor system, but with less response time from the perspective of a user.

With coarse and very coarse-grained parallelism, there is minimum synchronization among processes. This kind of situation can be easily handled as a set of concurrent processes running on a multi-programmed uniprocessor system and can be supported on a multiprocessor with little chance of the associated software. As an example, a program has been developed which takes in specifications of files that need recompilation and decides which of these compilations can be done simultaneously. It is reported that the actual speedup is more than expected since some of the compiled codes can be shared. In these situations, the linear speedup is certainly what we can expect the most.

Because the threads coming from a single process interact among themselves so frequently, scheduling decisions concerning one thread may have some impact on other threads belonging to the same process. When dealing with a multiprocessor system, besides the dispatching policies, we have to talk about a few other issues, including how to assign processors to processes, now that we have more to give away, how to make use of multiprogramming on individual processors. Assume that we have a fair and uniform environment, then the simplest approach of processor assignment is to treat all the processors as a pool. If a processor is permanently assigned to a process throughout its life, we should associate a short-term queue with each processor. This will lead to smaller overhead, but an uneven workload for processors. An alternative is to use a common queue to serve all the processes. Thus, a process can run on different processors at different times.

Regarding the actual assignment, at least two approaches can be followed: With a master/slave approach, key kernel functions, including the scheduler of the operating system, always run on a “master” processor, while the rest of the processors are used to run user processes. When a slave process needs some

service, it simply sends a request to the master processor and waits for its response. This approach is very simple and does not need a conflicting resolution mechanism. But the master can become a bottleneck, and it can even bring down the whole system when it fails.

In a peer structure, the OS can execute on any processor, and each of them does its scheduling among the available processes. This certainly makes the situation messier, since the OS must ensure that two processors will not choose the same process and no process is starved, i.e., never gets chosen. Also, competition among processes for various resources must be resolved. There is plenty of room between these two extremes. For example, a subset of processors, instead of just one processor, can be selected to run the kernel functions, including the dispatcher policies.

4.1 Multicore Processors

Driven by a performance-hungry market, microprocessors have always been designed keeping performance and cost in mind. Gordon Moore, the founder of Intel Corporation, predicted that the number of transistors on a chip will double once every 18 months to meet this ever-growing demand which is popularly known as Moore's Law in the semiconductor industry. Advanced chip fabrication technology alongside integrated circuit processing technology offers increasing integration density which has made it possible to integrate one billion transistors on a chip to improve performance. However, the performance increase by micro-architecture governed by Pollack's rule is roughly proportional to the square root of the increase in complexity. This would mean that doubling the logic on a processor core would only improve the performance by 40%. With advanced chip fabrication techniques comes along another major bottleneck: power dissipation issue. Studies have shown that transistor leakage current increases as the chip size shrinks further and further which increases static power dissipation to large values. One alternative means of improving performance is to increase the frequency of operation which enables faster execution of programs. However, the frequency is again limited to 4GHz currently as any increase beyond this frequency increases power dissipation again. "Battery life and system cost constraints drive the design team to consider power over performance in such a scenario" [8]. Power consumption has increased to such high levels that traditional air-cooled microprocessor server boxes may require budgets for liquid cooling or refrigeration hardware. Designers eventually hit what is referred to as the power wall, the limit on the amount of power a microprocessor could dissipate.

"A Multi-core processor is typically a single processor which contains several cores on a chip" [9]. The cores as shown in Figure 3 are functional units made up of computation units and caches. These multiple cores on a single chip combine to replicate the performance of a single faster processor. The individual scores on a multi-core processor don't necessarily run as fast as the highest performing single-core processors, but they improve overall performance by handling more tasks in parallel. The performance boost can be seen by understanding how single-core and multi-core processors execute programs. Single-core processors running multiple programs would assign time slices to work on one program and then assign different time slices for the remaining programs. If one of the processes is taking a long time to complete then all the rest of the processes start lagging. However, In the case of multi-core processors, if you have multiple tasks that can be run in parallel at the same time, each of them will be executed by a separate core in parallel thus boosting the performance.

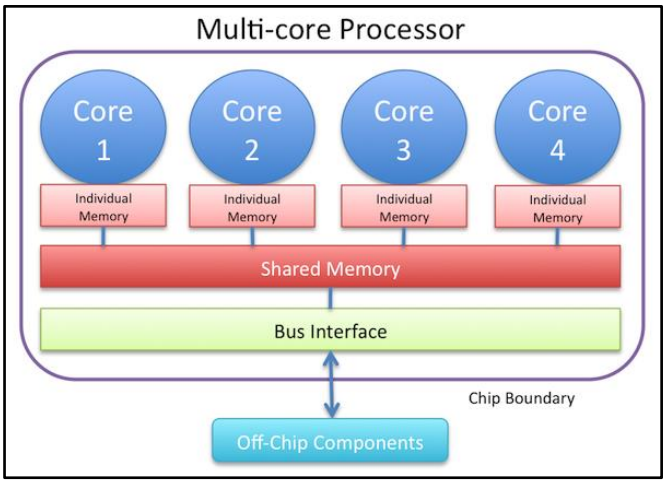

Figure 3. Multi-core processor

5. Real-Time CPU Scheduling

Real-time systems are characterized as those systems in which the accuracy of the system depends on the coherent consequence of calculation, as well as on the time at which the outcomes are created. Cases of this kind of real-time system are the charge and control system, process control system, flight real-time system, the Space Shuttle aeronautics system, future system, for example, the space station, space-based safeguard system, for example, SDI, and expansive order and control system. A dominant part of the current systems expects that quite a bit of this learning is accessible from earlier and subsequently depend on static plans which add to their high cost and resoluteness. The cutting-edge hard ongoing system must be intended to be dynamic, unsurprising, and adaptable.

A real-time operating system must be able to perform integrated CPU scheduling and resource allocation so that collections of cooperating tasks can obtain the resources they need, at the right time, to meet timing constraints. In addition to proper scheduling algorithms, predictability requires bounded operating system primitives.

Scheduling involves the allocation of resources and time to tasks in such a way that certain performance requirements are met [10]. Scheduling has been perhaps the most widely researched topic within real-time systems. This is due to the belief that the basic problem in real-time systems is to make sure that tasks meet their time constraints.

Consistency is one of the essential issues of a real-time system. Schedulability investigation or practicality checking of the errands of a real-time system must be done to anticipate whether the assignments will meet their planning imperatives. A few planning ideal models develop, contingent upon a) regardless of whether a system performs schedulability examination, b) on the off chance that it does, whether it is done statically or powerfully, and c) whether the consequence of the investigation itself creates a calendar or plan as indicated by which errands are dispatched at run-time. Because of this, we can distinguish the accompanying classes of calculations:

- Static table-driven approaches: These perform static schedulable investigation and the subsequent calendar (or table, as it is typically called) is utilized at runtime to choose when an errand must start execution [11].
- Static priority-driven preemptive approaches: These perform static schedulable investigation yet not at all like in the past approach, no express calendar is built. At runtime, errands are executed "most elevated need first." [12].
- Dynamic planning-based approaches: Unlike the past two methodologies, attainability is checked at runtime, i.e., a powerfully arriving assignment is acknowledged for execution only if it is found feasible. (Such a task is said to be guaranteed to meet its time constraints.) One of the results of the feasibility analysis is a schedule or plan that is used to decide when a task can begin execution.
- Dynamic best effort approaches: Here no feasibility checking is done. The system tries to do its best to meet deadlines. But since no guarantees are provided, a task may be aborted during its execution [13].

5.1 Real-time Operating System

Real-time operating systems are an integral part of real-time systems. Not surprisingly, the four main functional areas that they support are process management and synchronization, memory management, inter-process communication, and I/O. However, how they support these areas differs from conventional operating systems as will be discussed in this section. In particular, real-time operating systems stress predictability and include features to support real-time constraints. Three general categories of real-time operating systems exist: small, proprietary kernels (commercially available as well as homegrown kernels), real-time extensions to commercial timesharing operating systems such as UNIX, and research kernels. these three main categories of real-time operating systems [13].

5.1.1 Small, Fast, Proprietary Kernels

The little, quick, exclusive portions come in two assortments: homegrown and business offerings' [14]. The two assortments are frequently utilized for little inserted frameworks when quick and exceptionally unsurprising execution must be ensured. The homegrown bits are typically very specific to the application. The cost of interestingly creating and keeping up a homegrown piece, and in addition the expanding nature of the business offerings is fundamentally decreasing the act of producing homegrown portions. For the two assortments of exclusive portions, to accomplish speed and consistency, the pieces are stripped down and advanced variants of time-sharing working frameworks.

5.1.2 Real-Time Extensions to Commercial Operating Systems

A second way to deal with a real-time operating system is the augmentation of business items, e.g., stretching out UNIX to RT-UNIX, or POSIX to RT-POSIX, or MACH to RT-MACH, or CHORUS to a constant form [14]. The real-time version of a commercial operating system is, for the most part, slower and less unsurprising than the restrictive pieces, yet have more prominent usefulness and better programming improvement situations imperative contemplations in numerous applications. Another critical favorable position is that they depend on an arrangement of well-known interfaces (norms) that encourage transportability. For UNIX, since numerous varieties of UNIX have developed another model's exertion, called POSIX which has characterized a typical arrangement of client level interfaces for operating systems. Specifically, the POSIX P. 1003.4 subcommittee is characterizing measures for a real-

time operating system. To date, the exertion has concentrated on eleven essential continuous related capacities: clocks, need planning, shared memory, ongoing documents, semaphores, inter-process correspondence, offbeat occasion warning, process memory locking, nonconcurrent I/O, synchronous I/O, and strings.

5.1.3 Research Operating System

While many real-time applications will continue to be constructed with proprietary real-time kernels and with extensions to commercial time-sharing operating systems, as discussed above, significant problems still exist [15]. In particular, the proprietary kernels have difficulty when scaling to large applications, and the time-sharing extensions emphasize speed rather than predictability, thereby perpetuating the myth that real-time computing is fast computing.

6. Synchronization

In some operating systems, processes that are working together may share some common storage that each one can read and write. The shared storage may be in the main memory (possibly in a kernel data structure) or it may be a shared file; the location of the shared memory does not change the nature of the communication or the problems that arise [16]. To see how interprocess communication works in practice, let us consider a simple but common example: a print spooler. When a process wants to print a file, it enters the file name in a special spooler directory. Another procedure, the printer daemon, intermittently verifies whether there are any documents to be printed, and if there are, it prints them and after that expels their names from the index. Envision that our spooler index has countless, numbered 0, 1, 2 every one fit for holding a record name. Additionally, envision that there are two shared factors: out, which focuses on the following document to be printed, and in, which focuses on the following free space in the catalogue. These two factors may well be kept on a two-word record accessible to all procedures. At a specific moment, spaces 0 to 3 are purged (the records have just been printed) and spaces 4 to 6 are full (with the names of documents lined for printing). Pretty much at the same time, forms An and B choose they need to line a document for printing.

The way to avert the inconvenience of race condition here and in numerous different circumstances including shared memory shared documents, and shared everything else is to discover some approach to

preclude more than one process from perusing and composing the mutual information in the meantime. Put as such, what we require is common rejection, that is, some method for ensuring that on the off chance that one procedure is utilizing a shared variable or record, alternate procedures will be prohibited from doing likewise thing. The trouble above happened because procedure B began utilizing one of the shared factors previously process A was done with it. The decision of proper crude activities for accomplishing shared rejection is a noteworthy outline issue in any working framework and a subject that we will inspect in extraordinary detail in the accompanying segments.

The issue of maintaining a strategic distance from race conditions can likewise be planned conceptually. Some portion of the time, a procedure is occupied with doing inside calculations and other things that don't prompt race conditions. In any case, some of the time a procedure needs to get to shared memory or documents, or do other basic things that can prompt races. That piece of the program where the mutual memory is gotten to is known as the basic locale or basic segment. On the off chance that we could orchestrate matters to such an extent that no two procedures were ever in their basic areas in the meantime, we could keep away from races.

6.1 Critical Section Problem

Mutual exclusion is a fundamental process synchronization problem in concurrent systems, and it has been studied extensively in the last half-century [17]. Generalizations of mutual exclusion have been studied extensively as well. Such generalizations include k-mutual exclusion, mutual inclusion, and -mutual inclusion. The number of processes that are in the critical section simultaneously is at most k by k-mutual exclusion, while that is at least by -mutual inclusion. Unfortunately, these generalized problems have been studied individually [18].

7. Conclusion

Many functions are provided by an operating system like process management, memory management, file management, input/output management, networking, protection system and command interpreter system. An operating system is a very promising field for researchers because many issues need to be solved. Synchronization and scheduling contain most issues in OS; therefore, this paper focuses on the main algorithms in synchronization and scheduling with a focus on the issues and advantages in each algorithm.

There are many algorithms are available for CPU scheduling. But all algorithms have their deficiency and limitations. Each algorithm has some advantages or disadvantages. Furthermore, an overview of the new multiprocessor hardware and its effects on the operating system, scheduling and synchronization have been introduced. This paper, it is aimed to present a startup point for the researcher who wants to work on the synchronization and scheduling algorithms issues.

Declaration of Competing Interest: The author declare no conflict of interest.

References

[1] M. L. Pinedo, Scheduling: theory, algorithms, and systems. Springer, 2016.

[2] K. Ramamritham and J. A. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," Proc. IEEE, vol. 82, no. 1, pp. 55–67, 1994.

[3] B. Andersson, S. Baruah, and J. Jonsson, "Static-priority scheduling on multiprocessors," in Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE, 2001, pp. 193–202.

[4] A. Singh, P. Goyal, and S. Batra, "An optimized round robin scheduling algorithm for CPU scheduling," Int. J. Comput. Sci. Eng., vol. 2, no. 7, pp. 2383–2385, 2010.

[5] R. K. Yadav and A. Upadhayay, "A fresh loom for multilevel feedback queue scheduling algorithm," Int. J. Adv. Eng. Sci., vol. 2, no. 3, pp. 21–23, 2012.

[6] H. S. Behera, R. K. Naik, and S. Parida, "Improved multilevel feedback queue scheduling using dynamic time quantum and its performance analysis," Int. J. Comput. Sci. Inf. Technol., vol. 3, pp. 3801–3807, 2012.

[7] L. M. Ni and K. Hwang, "Optimal load balancing in a multiple processor system with many job classes," IEEE Trans. Softw. Eng., no. 5, pp. 491–496, 1985.

[8] D. M. Brooks et al., "Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors," IEEE Micro, vol. 20, no. 6, pp. 26–44, 2000.

[9] L. Wang, J. Tao, G. von Laszewski, and H. Marten, "Multicores in Cloud Computing: Research Challenges for Applications.," JCP, vol. 5, no. 6, pp. 958–964, 2010.

[10] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," J. ACM, vol. 20, no. 1, pp. 46–61, 1973.

[11] A. Burns, Preemptive priority based scheduling: An appropriate engineering approach. University of York, Department of Computer Science, 1993.

[12] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings, "Fixed priority pre-emptive scheduling: An historical perspective," Real-Time Syst., vol. 8, no. 2–3, pp. 173–198, 1995.

[13] S. B. N. Meghanathan, "A survey of contemporary real-time operating systems," Informatica, vol. 29, no. 2, 2005.

[14] J. A. Stankovic and R. Rajkumar, "Real-time operating systems," Real-Time Syst., vol. 28, no. 2–3, pp. 237–253, 2004.

[15] M. Quigley et al., "ROS: an open-source Robot Operating System," in ICRA workshop on open source software, 2009, vol. 3, no. 3.2, p. 5.

[16] A. Silberschatz, P. B. Galvin, and G. Gagne, Operating system concepts essentials. John Wiley & Sons, Inc., 2014.

[17] E. W. Dijkstra, "Solution of a problem in concurrent programming control," in Pioneers and Their Contributions to Software Engineering, Springer, 2001, pp. 289–294.

[18] A. S. Tanenbaum, "Processes and Threads," Mod. Oper. Syst., 2008.