Operating Systems – Project 1

- Simple Scheduling -

ChangYoon Lee, ChanHo Park

Dankook University, Korea Republic
32183641@gmail.com, chanho.park@dankook.ac.kr

Abstract. Most modern operating systems have extended the concept of the process to allow having multiple threads of execution and thus to perform more than one task at a time (*Silberschatz et al., 2014*). While allowing this multi-threaded concept, there are some issues to consider in designing it: Synchronization (Concurrency Control). In this paper, we will discuss the concept of the multi-thread, synchronization examples, and their solutions, especially on producer and consumer problems. Also, we will implement the example case of producer and consumer problems with a multi-threaded word count program and evaluate it.

Keywords: Concurrency Control, Multi-Thread, Mutex, Producer and Consumer, Semaphore, Synchronization, Reader and Write

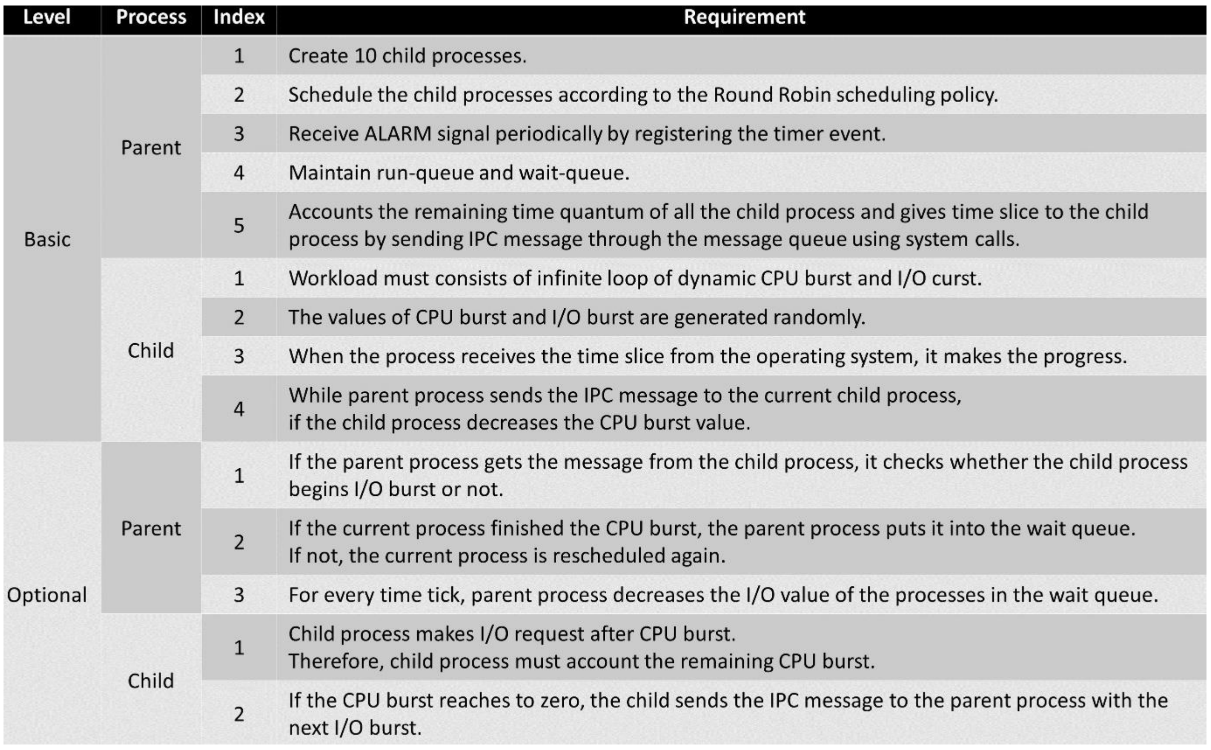
1 Introduction

In a single-processor computer system, only one process can run at a single time. Other processes must wait until the CPU resources are free and can be rescheduled. A process is executed until it must wait, typically for the completion of the I/O request. However, in multiprogramming, some process runs at all time, to maximize CPU utilization. Multiprogramming tries to use the waiting time productively. Some process is loaded into the memory at one time, and when one process has to wait, the operating system takes the CPU resources away from the process and gives the CPU resources to another process. The following progress continues, every time one process has to wait, while another process takes over the use of the CPU resources.

Scheduling the following progress is a fundamental operating system function. Almost all the computer resources are scheduled before use. Since the CPU is one of the primary computer resources, CPU scheduling is central to the operating system design.

In this paper, we will first explain the concepts of process, process scheduling, Inter-process communication (IPC), and CPU scheduling. By applying these concepts, we will explain how we implemented the simple scheduling program and its results for different algorithms, such as first-come-first-served (FCFS), shortest job first (SJF), round-robin (RR), and completely fair safe (CFS). Also, we will show the performance of each algorithm based on the features discussed in a later section. At the end of the paper, we will present the result of the execution of the different scheduling algorithms and compare them.

2 Requirements

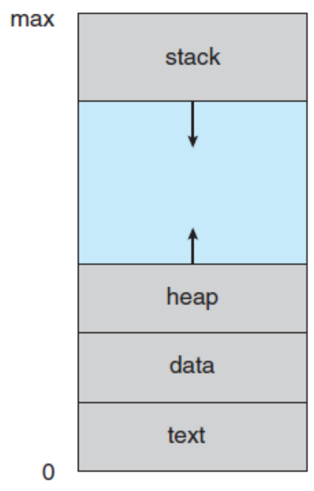


Level	Process	Index	Requirement
Basic	Parent	1	Create 10 child processes.
		2	Schedule the child processes according to the Round Robin scheduling policy.
		3	Receive ALARM signal periodically by registering the timer event.
		4	Maintain run-queue and wait-queue.
		5	Accounts the remaining time quantum of all the child process and gives time slice to the child process by sending IPC message through the message queue using system calls.
	Child	1	Workload must consists of infinite loop of dynamic CPU burst and I/O curst.
		2	The values of CPU burst and I/O burst are generated randomly.
		3	When the process receives the time slice from the operating system, it makes the progress.
		4	While parent process sends the IPC message to the current child process, if the child process decreases the CPU burst value.
Optional	Parent	1	If the parent process gets the message from the child process, it checks whether the child process begins I/O burst or not.
		2	If the current process finished the CPU burst, the parent process puts it into the wait queue. If not, the current process is rescheduled again.
		3	For every time tick, parent process decreases the I/O value of the processes in the wait queue.
	Child	1	Child process makes I/O request after CPU burst. Therefore, child process must account the remaining CPU burst.
		2	If the CPU burst reaches to zero, the child sends the IPC message to the parent process with the next I/O burst.

Figure 1 shows the requirements for a simple scheduling program. The implementations for these requirements will be described in detail afterwards.

3 Concepts

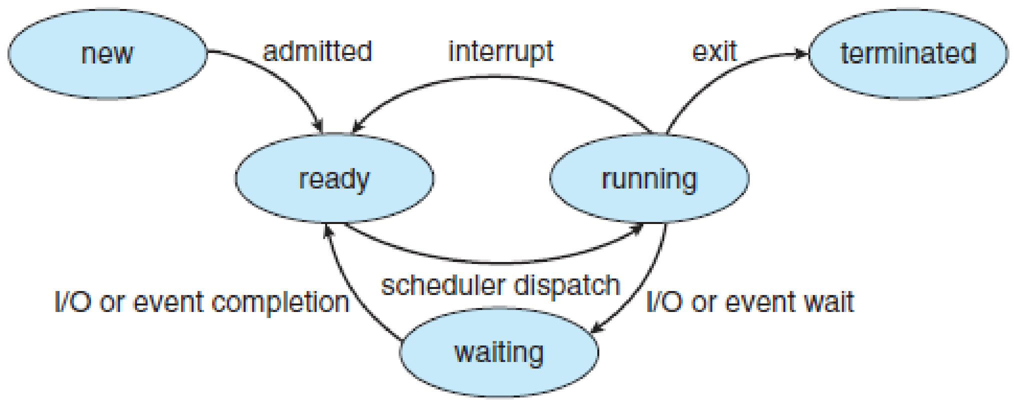
3.1 Process



Modern computers allow multiple programs to be loaded into the memory and executed concurrently. This operation requires firmer control and more compartmentalization of the various programs. To fulfill this requirement, the operating systems need the notion of a process, which is a program in execution. A process is the unit of work in a modern time-sharing system.

The process is an instance of a program in execution. A process is more than the program code, which is known as a text section. It is also consisting of a program counter (PC) and the contents of the registers that present the current state of the process. A process also includes the stack, which contains the temporary data, the data section, which contains global variables, and the heap, which is the memory that is dynamically allocated during the process runtime. The structure of the process is presented in Figure 2.

3.1.2 Process State

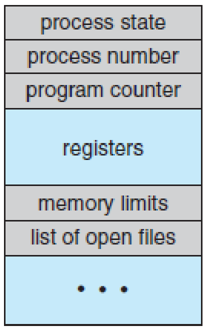


As a process is in execution, it changes its state. The state of a process can be defined in the part by the current activity of the process. The process may be in one of the following states:

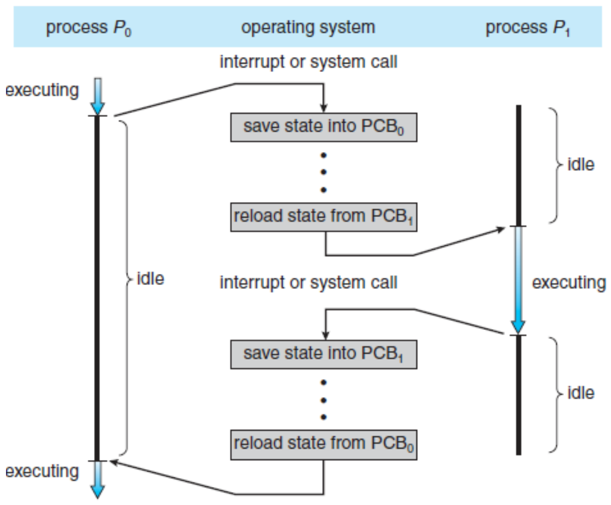
- **New**: The process is being created.
- **Running**: Instructions are being executed.
- **Waiting**: The process is waiting for some event to occur.
- **Ready**: The process is waiting to be assigned to a process.
- **Terminated**: The process has finished the execution.

It is important to realize that only one process can be running on any processor at any instant. However, many processes may be ready and waiting. The state diagram that presents these states is presented in Figure 3.

3.1.3 Process Control Block



Each process in the operating system is presented in a process control block (PCB), which is also called a task control block. The PCB is presented in Figure 4. It contains much information associated with a specific process, including the following information



- **Process state**: The state may be new, ready, running, waiting, or terminated.
- **Program counter**: The program counter indicates the address of the next instruction to be executed for this process.
- **CPU registers**: Depending on the computer architecture, the registers vary in number and type. They include accumulators, index registers, stack pointers, and general-purpose registers. Along with the program counter, the following state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward. The following progress is presented in Figure 5.
- **CPU scheduling information**: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory management information**: This information may include such items as the value of the base and limit registers and the page table, or the segment tables, depending on the memory system used by the operating system.

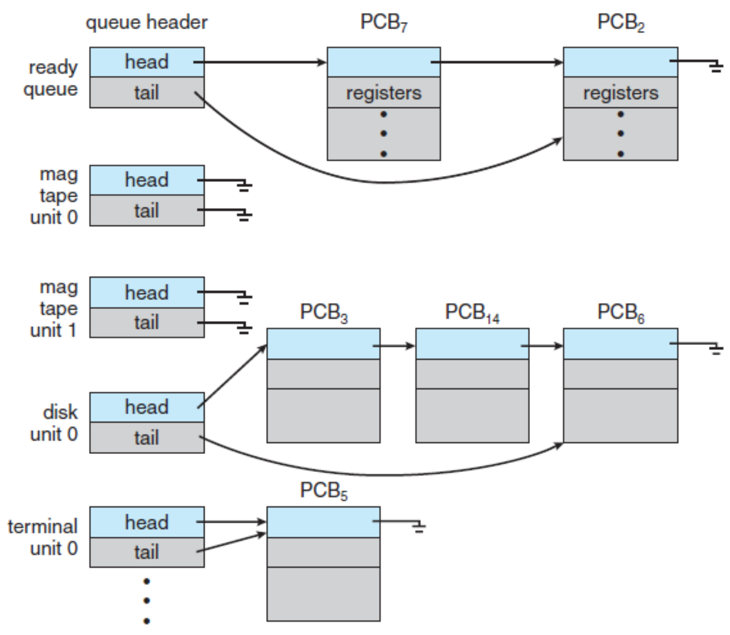
- **Accounting information**: This information includes the amount of CPU and real-time used, time limits, account numbers, and job or process numbers.
- **I/O status information**: This information includes the list of I/O devices allocated to the process and a list of open files.

In short, the PCB simply used as the repository for any information that may vary from process to process.

3.2 Process Scheduling

In multiprogramming, it has some process running at all times, to maximize CPU utilization. Also, time-sharing switches the CPU among processes so that the users can interact with each program frequently while it is running. To implement the following operations, the process scheduler selects an available process for program execution on the CPU. In the case of the single-processor system, there will never be more than one running process. If there are more processes, the rest of the processes will have to wait until the CPU is free and can be rescheduled.

3.2.1 Scheduling Queues



As a process enter the system, it is put into the job queue, which consists of all processes in the system. The process that is residing in the main memory and are ready and waiting to execute are kept on a list called the ready queue. The queue is generally stored as a linked list. A ready queue header contains the pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.

The system also includes the other queues. When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. Therefore, the process may have to wait for the disk. The list of processes waiting for a particular I/O device is called a device queue. The following queue is presented in Figure 6.

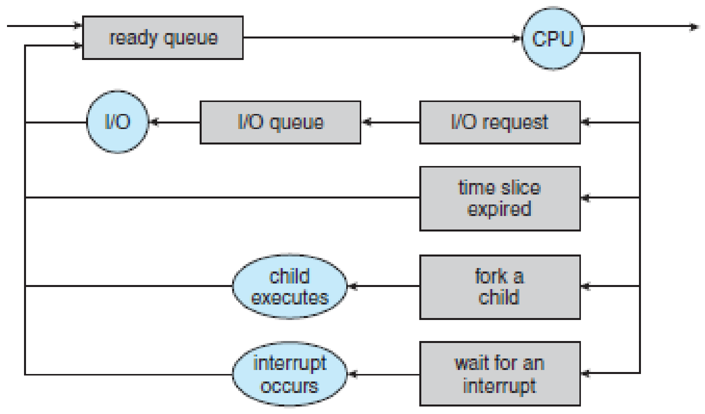


Figure 7 shows the representation of the queuing diagram. Each box in Figure 7 represents a queue. There are two types of queues, which are a ready queue and a set of device queues. The circles represented in Figure 7 are the resources that serve the queues, and the arrow indicates the flow of processes in the system.

A new process is initially put in the ready queue. It waits in the ready queue until it is selected for execution, or dispatched. Once the process is allocated the CPU and is in execution, the following events can occur:

- The process can issue an I/O request and then be placed in an I/O queue.
- The process can create a new child process and wait for the child's termination.
- The process can be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

In the first two cases, the process switches from the waiting state to the ready state and is then put back in the ready queue. A process continues this process until it terminates, at the time it is removed from all queues and has its PCB and resources deallocated.

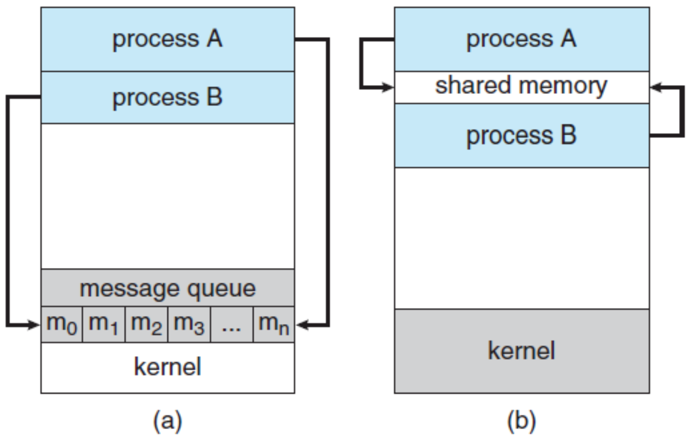
3.2.2 Scheduler

A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some method. The selection of the process is carried out by the appropriate scheduler. There are two schedulers, which are long-term scheduler (job scheduler), and the short-term scheduler (CPU scheduler), The long-term scheduler selects the processes to form the process pool and loads them into the memory for execution. The short-term scheduler selects the process that is ready to execute and allocates the CPU to one of them. The primary distinction between these two schedulers is in the frequency of the execution. The short-term scheduler must select a new process for the CPU frequently. However, the long-term scheduler executes with much less frequency, to create a new process. Therefore, it controls the degree of multiprogramming. If the degree of multiprogramming is stable, the average rate of the process creation must be equal to the average departure rate of the processes leaving the system. Also, the long-term scheduler may need to be invoked only when a process leaves the system. Due to the long interval between the executions, the long-term scheduler can afford to take more time to decide which process should be selected for the execution.

In short, the long-term scheduler must make a careful selection. In most cases, most processes can be described as either I/O bound or CPU bound. An I/O bound process spends more of its time doing I/O operation than it spends doing computation. In contrast, a CPU-bound process generates I/O requests infrequently, using more of its time doing computations. The long-term scheduler must select a good process mix of I/O-bound and CPU-bound processes. If all of the processes are I/O bound, the ready queue will be always empty, and the short-term scheduler will have nothing to do. If all of the processes are CPU bound, the I/O waiting queue will almost always be empty, hardware resources are not used, and again the system will be unbalanced. The system with the best performance will have a combination of CPU-bound and I/O-bound processes.

3.3 Inter-Process Communication (IPC)

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. An independent process cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. However, a process is cooperating if it can affect or be affected by the other processes executing in the system, which means that any process that shared data with other processes is a cooperating process.

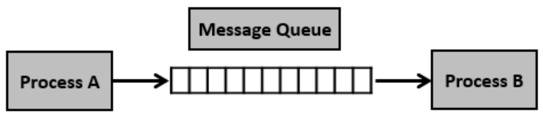


Cooperating processes require an inter-process communication (IPC) operation that will allow them to exchange data and information. There are two fundamental models of inter-process communication, which are shared memory and message passing. In the shared memory model, a region of the memory that is shared by cooperating processes is established. The process can then exchange information by reading and writing data to the shared region. In the message-passing model, communication takes place to be means of messages exchanged between the cooperating processes. The following models are presented in Figure 8.

Both of the models are commonly used in operating systems. Message passing is useful for exchanging smaller amounts of data because no conflicts need to be avoided. Shared memory can be faster than message passing since message-passing systems are typically implemented using system calls and require the more time-consuming task of kernel intervention. In shared memory systems, system calls are required only to establish shared memory regions. Once the shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

Latest researches indicate that message passing provides better performance than the shared memory on such systems. Shared memory suffers from cache coherency issues, which arise because shared data migrate among several caches. As the number of processing cores on the systems increases, it is possible that message passing is the preferred method for the IPC.

3.3.1 Message Passing



Message passing provides an operation that allows processes to communicate and synchronize their action without sharing the same address space. If processes A and B want to communicate, they must send messages to and receive messages from each other. A communication link must exist between them. This link can be implemented in a variety of ways: direct or indirect communication, synchronous or asynchronous communication, and automatic or explicit buffering.

3.3.2 Message Queue Naming

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication. Under direct communication, each process that wants to communicate must explicitly name the receiver or sender of the communication. This means that both the sender process and receiver process must name the other to communicate.

A variant of this concept is indirect communication. In this case, only the sender names the receiver, while the receiver does not require the name of the sender. With indirect communication, the messages are sent to and received from the mailboxes, also known as ports. A mailbox can be viewed abstractly as an object into which messages can be placed by the processes and from which messages can be removed. Each mailbox has a unique identification. A process can communicate with another process with several different mailboxes, but two processes can communicate only if they have a shared mailbox.

The process that creates a new mailbox becomes the mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receiving privilege may be passed to other processes through appropriate system calls. The following provision can result in multiple receivers for each mailbox.

3.3.3 Message Queue Synchronization

There are several solutions for the following situation that satisfies the requirements above. In this paper, we will present two main solutions for resolving the critical-section problem, which is mutex and semaphore. The details of the mutex and semaphore will be presented in the later sections.

3.3.4 Message Queue Buffering

There are several solutions for the following situation that satisfies the requirements above. In this paper, we will present two main solutions for resolving the critical-section problem, which is mutex and semaphore. The details of the mutex and semaphore will be presented in the later sections.

3.4 CPU Scheduling

In this paper, we will focus on applying the concept of multi-threaded programming, and the solutions for resolving the presented challenges in the word count program. These challenges will also be described in detail, in a section on the optimization of the implemented program.

3.4.1 Scheduling Criteria

In this paper, we will focus on applying the concept of multi-threaded programming, and the solutions for resolving the presented challenges in the word count program. These challenges will also be described in detail, in a section on the optimization of the implemented program.

3.4.2 Scheduling Algorithms

In this paper, we will focus on applying the concept of multi-threaded programming, and the solutions for resolving the presented challenges in the word count program. These challenges will also be described in detail, in a section on the optimization of the implemented program.

3.4.3 Algorithm Evaluation

In this paper, we will focus on applying the concept of multi-threaded programming, and the solutions for resolving the presented challenges in the word count program.

4 Simple Scheduling

4.1 Signal and Handler

Figure 4 shows the solution code that uses mutex lock to resolve the critical section problem. The acquire function acquires the lock, and the release function releases the lock. A mutex has a Boolean variable whose value indicates whether the lock is available or not. If the lock is available, the acquire function succeeds, and the lock is then considered to be unavailable. A thread that attempts to acquire an unavailable lock is blocked until the lock is released.
The main disadvantage of the implementation of the mutex lock is that it acquires busy waiting. While a process or thread is in its critical section, any other process that tries to enter its critical section must loop continuously in the call of acquiring function. This continual looping becomes a problem in the multi-programming system because it wastes the CPU cycle that some other processes and threads might be able to use productively. In the aspect of continuously looping for the busy wait, the mutex lock is also called a spinlock because the thread spins while waiting for the lock to become available.

4.2 Message Passing in POSIX

Figure 4 shows the solution code that uses mutex lock to resolve the critical section problem. The acquire function acquires the lock, and the release function releases the lock. A mutex has a Boolean variable whose value indicates whether the lock is available or not. If the lock is available, the acquire function succeeds, and the lock is then considered to be unavailable. A thread that attempts to acquire an unavailable lock is blocked until the lock is released.
The main disadvantage of the implementation of the mutex lock is that it acquires busy waiting. While a process or thread is in its critical section, any other process that tries to enter its critical section must loop continuously in the call of acquiring function. This

continual looping becomes a problem in the multi-programming system because it wastes the CPU cycle that some other processes and threads might be able to use productively. In the aspect of continuously looping for the busy wait, the mutex lock is also called a spinlock because the thread spins while waiting for the lock to become available.

4.3 First-Come First-Served Algorithm (FCFS)

Figure 4 shows the solution code that uses mutex lock to resolve the critical section problem. The acquire function acquires the lock, and the release function releases the lock. A mutex has a Boolean variable whose value indicates whether the lock is available or not. If the lock is available, the acquire function succeeds, and the lock is then considered to be unavailable. A thread that attempts to acquire an unavailable lock is blocked until the lock is released.
The main disadvantage of the implementation of the mutex lock is that it acquires busy waiting. While a process or thread is in its critical section, any other process that tries to enter its critical section must loop continuously in the call of acquiring function. This continual looping becomes a problem in the multi-programming system because it wastes the CPU cycle that some other processes and threads might be able to use productively. In the aspect of continuously looping for the busy wait, the mutex lock is also called a spinlock because the thread spins while waiting for the lock to become available.

4.4 Shortest Job First Algorithm (SJF)

Figure 4 shows the solution code that uses mutex lock to resolve the critical section problem. The acquire function acquires the lock, and the release function releases the lock. A mutex has a Boolean variable whose value indicates whether the lock is available or not. If the lock is available, the acquire function succeeds, and the lock is then considered to be unavailable. A thread that attempts to acquire an unavailable lock is blocked until the lock is released.
The main disadvantage of the implementation of the mutex lock is that it acquires busy waiting. While a process or thread is in its critical section, any other process that tries to enter its critical section must loop continuously in the call of acquiring function. This continual looping becomes a problem in the multi-programming system because it wastes the CPU cycle that some other processes and threads might be able to use productively. In the aspect of continuously looping for the busy wait, the mutex lock is also called a spinlock because the thread spins while waiting for the lock to become available.

4.5 Round Robin Algorithm (RR)

Figure 4 shows the solution code that uses mutex lock to resolve the critical section problem. The acquire function acquires the lock, and the release function releases the lock. A mutex has a Boolean variable whose value indicates whether the lock is available or not. If the lock

is available, the acquire function succeeds, and the lock is then considered to be unavailable. A thread that attempts to acquire an unavailable lock is blocked until the lock is released.
The main disadvantage of the implementation of the mutex lock is that it acquires busy waiting. While a process or thread is in its critical section, any other process that tries to enter its critical section must loop continuously in the call of acquiring function. This continual looping becomes a problem in the multi-programming system because it wastes the CPU cycle that some other processes and threads might be able to use productively. In the aspect of continuously looping for the busy wait, the mutex lock is also called a spinlock because the thread spins while waiting for the lock to become available.

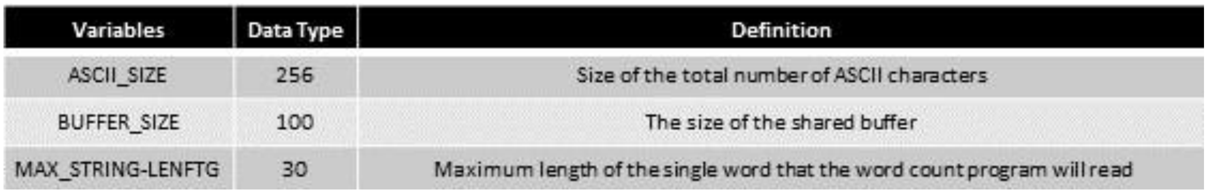
4.6 Completely Fair Safe Algorithm (CFS)

Figure 4 shows the solution code that uses mutex lock to resolve the critical section problem. The acquire function acquires the lock, and the release function releases the lock. A mutex has a Boolean variable whose value indicates whether the lock is available or not. If the lock is available, the acquire function succeeds, and the lock is then considered to be unavailable. A thread that attempts to acquire an unavailable lock is blocked until the lock is released.
The main disadvantage of the implementation of the mutex lock is that it acquires busy waiting. While a process or thread is in its critical section, any other process that tries to enter its critical section must loop continuously in the call of acquiring function. This continual looping becomes a problem in the multi-programming system because it wastes the CPU cycle that some other processes and threads might be able to use productively. In the aspect of continuously looping for the busy wait, the mutex lock is also called a spinlock because the thread spins while waiting for the lock to become available.

4.7 Program Definition

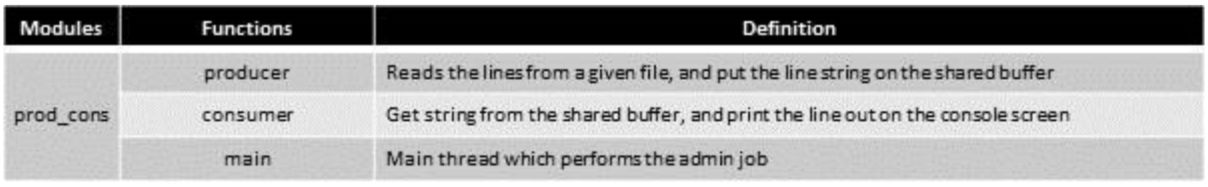
Before implementing the multi-threaded word count program, we will state the additional program definition that will be used in the real implementation.

- Global Variables



Variables	Data Type	Definition
ASCII_SIZE	256	Size of the total number of ASCII characters
BUFFER_SIZE	100	The size of the shared buffer
MAX_STRING-LENFTG	30	Maximum length of the single word that the word count program will read

- Modules and Functions



Modules	Functions	Definition
prod_cons	producer	Reads the lines from a given file, and put the line string on the shared buffer
	consumer	Get string from the shared buffer, and print the line out on the console screen
	main	Main thread which performs the admin job

5 Implementation

5.1 Queue Header

Figure 4 shows the solution code that uses mutex lock to resolve the critical section problem. The acquire function acquires the lock, and the release function releases the lock. A mutex has a Boolean variable whose value indicates whether the lock is available or not. If the lock is available, the acquire function succeeds, and the lock is then considered to be unavailable. A thread that attempts to acquire an unavailable lock is blocked until the lock is released.
The main disadvantage of the implementation of the mutex lock is that it acquires busy waiting. While a process or thread is in its critical section, any other process that tries to enter its critical section must loop continuously in the call of acquiring function. This continual looping becomes a problem in the multi-programming system because it wastes the CPU cycle that some other processes and threads might be able to use productively. In the aspect of continuously looping for the busy wait, the mutex lock is also called a spinlock because the thread spins while waiting for the lock to become available.

5.2 Heap Header

Figure 4 shows the solution code that uses mutex lock to resolve the critical section problem. The acquire function acquires the lock, and the release function releases the lock. A mutex has a Boolean variable whose value indicates whether the lock is available or not. If the lock is available, the acquire function succeeds, and the lock is then considered to be unavailable. A thread that attempts to acquire an unavailable lock is blocked until the lock is released.
The main disadvantage of the implementation of the mutex lock is that it acquires busy waiting. While a process or thread is in its critical section, any other process that tries to enter its critical section must loop continuously in the call of acquiring function. This continual looping becomes a problem in the multi-programming system because it wastes the CPU cycle that some other processes and threads might be able to use productively. In the aspect of continuously looping for the busy wait, the mutex lock is also called a spinlock because the thread spins while waiting for the lock to become available.

5.3 Inter-Process Control (IPC) Message Passing Header

Figure 4 shows the solution code that uses mutex lock to resolve the critical section problem. The acquire function acquires the lock, and the release function releases the lock. A mutex has a Boolean variable whose value indicates whether the lock is available or not. If the lock is available, the acquire function succeeds, and the lock is then considered to be unavailable. A thread that attempts to acquire an unavailable lock is blocked until the lock is released.
The main disadvantage of the implementation of the mutex lock is that it acquires busy waiting. While a process or thread is in its critical section, any other process that tries to

enter its critical section must loop continuously in the call of acquiring function. This continual looping becomes a problem in the multi-programming system because it wastes the CPU cycle that some other processes and threads might be able to use productively. In the aspect of continuously looping for the busy wait, the mutex lock is also called a spinlock because the thread spins while waiting for the lock to become available.

5.4 Main

Figure 4 shows the solution code that uses mutex lock to resolve the critical section problem. The acquire function acquires the lock, and the release function releases the lock. A mutex has a Boolean variable whose value indicates whether the lock is available or not. If the lock is available, the acquire function succeeds, and the lock is then considered to be unavailable. A thread that attempts to acquire an unavailable lock is blocked until the lock is released.
The main disadvantage of the implementation of the mutex lock is that it acquires busy waiting. While a process or thread is in its critical section, any other process that tries to enter its critical section must loop continuously in the call of acquiring function. This continual looping becomes a problem in the multi-programming system because it wastes the CPU cycle that some other processes and threads might be able to use productively.

6 Build Environment

- Build Environment:
  1. Linux Environment -> Vi editor, GCC Complier
  2. Program is built by using the Makefile.

- Build Command:
  1. $make prod_cons -> build the execution program for prod_cons from version 1 to 4.
  2. $make clean -> clean all the object files that consists of the prod_cons programs.
- Execution Command:
  1. . /Prod_cons_v1 {$readfile}
     -> Execute the producer and consumer version 1 program.
  2. . /Prod_cons_v2.1 {$readfile} #Producer #Consumer
     -> Execute the producer and consumer version 2.1 program.
  3. . /Prod_cons_v2.2 {$readfile} #Producer #Consumer
     -> Execute the producer and consumer version 2.2 program.
  4. . /Prod_cons_v2.3 {$readfile} #Producer #Consumer
     -> Execute the producer and consumer version 2.3 program.
  5. . /Prod_cons_v3 {$readfile} #Producer #Consumer
     -> Execute the producer and consumer version 3 program.
  6. . /Prod_cons_v4 {$readfile} #Producer #Consumer
     -> Execute the producer and consumer version 4 program.

7 Results

- Producer and Consumer Version 1 reading LICENSE file.

8 Evaluation

Figures 35 and 36 show the graph result of execution time per number of the threads for the producer and consumer program versions 2.3 that reads the file of FeeBSD9-Orig.tar and the android.tar. The following program is implemented by applying the methods that are presented previously. As we can see from the figures, the execution time decreases as the number of threads increases. In short, the producer and consumer program version 2.3 show the ideal and faster result of execution time among the other programs.

9 Conclusion

By understanding this paper, we can understand the basic concepts of thread and multi-threaded programming. Also, we can understand the problems and the solutions that occur by applying the following concept, which is about data dependency and synchronization.

Citations

[1] Kirvan, P. (2022, May 26). *What is multithreading?* WhatIs.com. Retrieved September 11, 2022, from https://www.techtarget.com/whatis/definition/multithreading

[2] Silberschatz, A., Galvin, P. B., &amp; Gagne, G. (2014). 2.2.1 Command Interpreters. In Operating Systems Concepts. essay, Wiley.

[3] Yan, D. (2020, December 22). *Producer-consumer problem using mutex in C++*. Medium. Retrieved September 9, 2022, from https://levelup.gitconnected.com/producer-consumer-problem-using-mutex-in-c-764865c47483

[4] Yoo, S. H. (n.d.). Mobile-os-DKU-cis-MSE. GitHub. Retrieved September 8, 2022, from https://github.com/mobile-os-dku-cis-mse/