NATIONAL OPEN UNIVERSITY OF NIGERIA

FACULTY OF SCIENCES

DEPARTMENT OF COMPUTER SCIENCE

COURSE CODE: CIT315

COURSE TITLE: OPERATING SYSTEM

National Open University of Nigeria
University Village, Plot 91
Jabi Cadastral Zone
Nnamdi Azikiwe Expressway
Jabi, Abuja

Lagos Office
14/16 Ahmadu Bello Way
Victoria Island, Lagos
Departmental email: computersciencedepartment@noun.edu.ng
NOUN e-mail: centralinfo@noun.edu.ng
URL: www.nou.edu.ng
First Printed 2022

ISBN: 978-058-557-5

Printed by: NOUN PRESS

January 2022

COURSE GUIDE

CIT315

OPERATING SYSTEM

Course Team

Prof. Olumide Babatope Longe (Developer/Writer)

Prof. Virginia Ejiofor - Content Editor

Dr. Francis B. Osang – HOD/Internal Quality Control Expert

CONTENT PAGE

Table of Contents

Introduction

An operating system serves as a liaison between the computer's user and the computer's hardware. The objective of an operating system is to offer a comfortable and efficient environment in which a user may run applications.

An operating system is a piece of software that controls the hardware on a computer. The hardware must have adequate measures to guarantee the computer system's proper operation and to prevent user applications from interfering with the system's proper functioning.

Internally, operating systems vary significantly in their composition, since they are arranged in a variety of ways. The creation of a new operating system is a significant undertaking. Prior to beginning the design process, it is critical to have a firm grasp of the system's objectives. These objectives serve as the foundation for selecting amongst different algorithms and tactics.

Due to the size and complexity of an operating system, it must be constructed piecemeal. Each of these components should be a distinct part of the system with well-defined inputs, outputs, and functions. An operating system is a software program that handles the hardware of a computer. Additionally, it serves as a foundation for application programs and serves as a liaison between the computer user and the computer hardware. A remarkable feature of operating systems is their diversity in doing these functions.

Before delving into the minutiae of computer system functioning, it's necessary to understand the system's structure. We begin by reviewing the fundamental operations of the operating system, including initialization, I/O, and storage. Additionally, we present the fundamental computer architecture that enables the development of a working operating system.

Due to the size and complexity of an operating system, it must be constructed piecemeal. Each of these components should be a distinct part of the system with well-defined inputs, outputs, and functions. We will present an overview of the key components of an operating system in this course.

What You Will Be Learning in this Course

This course consists of units and a course guide. This course guide tells you briefly what the course is about, what course material you will be using and how you can work through these materials. In addition, it advocates some general guidelines for the amount of time you are likely to spend on each unit of the course in order to complete it successfully.

It gives you guidance in respect of your Tutor-Marked Assignments which will be made available in the assignment file. There will be regular tutorial classes that are related to the course. It is advisable for you to attend these tutorial sessions.

The course will prepare you for the technical concepts of operating system.

Course Aim

Operating System CIT315 aims to furnish you with enough knowledge onprinciple of operating system.

Course Objectives

To achieve the aim set out, the course has a set of objectives. Each unit has specific objectives which are included at the beginning of the unit.

You may wish to refer to them during your study to check on your progress. You should always look at the unit objectives after completion of each unit. By doing so, you would know whether you have followed the instruction in the unit.

Below are the comprehensive objectives of the course as a whole. By meeting these objectives, you should have achieved the aims of the course as a whole. In addition to the aim earlier stated, this course sets to achieve some objectives. Thus, after going through the course, you should be able to:

- Explain what process is
- Identify the states of process
- Identify the importance of context switching

- Know how to interrupt a running process
- Explain the concepts of context switching
- Explain how a running process is interrupted via system call
- Narrate switches from procedure to procedure
- Explain the concept of interrupt
- Demonstrate masking and unmasking of interrupt request
- Demonstrate handling of interrupts
- Explain the concepts of thread
- Differentiate between thread and process
- Write simple thread creation code/instructions
- Explain the types of threads
- Implement a thread at user, kernel and hybrid level
- Demonstrate the Thread Control Block diagram
- Explain the thread life cycle
- Show thread states and transition
- Define race condition
- Explain how race condition occur
- Resolve race condition
- Describe the concept of deadlock
- Write a program code that avoid deadlock
- Explain deadlock and starvation resolution using semaphores and monitors
- Solving synchronization problems
- Describing the techniques of memory swapping
- Demonstrate how resources are allocated
- Describe memory allocation to processes
- Use the best fit method to fix memory allocation issues
- Explain the concept of memory segment and paging
- Solving memory allocation issues into non-contiguous memory spaces
- Understand the importance of cache
- How companies have succeeded as a result of cache memory

- Find out the cause of thrashing in operating system
- Demonstrate policies to address thrashing issues

Working through this course

To complete this course, you are required to read each study unit, read the textbooks and read other materials which may be provided by the National Open University of Nigeria.

Each unit contains self-assessment exercises and at certain point in the course you would be required to submit assignments for assessment purposes. At the end of the course there is a final examination. The course should take you about a total of 17 weeks to complete. Below you will find listed all the components of the course, what you have to do and how you should allocate your time to each unit in order to complete the course on time and successfully.

This course entails that you spend lots of time reading. I recommend that you take advantage of knowledge sharing among other students

Course Material

The major components of the course are:

1. Course Guide
2. Study Units
3. Presentation Schedule
4. Tutor-Marked Assignments
5. References/Further Reading

Study Units

The study units in this course are as follows:

Module 1	**Process Management**
Unit 1	Processes and State
Unit 2	Context Switching
Unit 3	Interrupts and Interrupts Handler
Module 2	**Concurrency – Multithreading**
Unit 1	Threads & Multithreading
Unit 2	Types of Threads
Unit 3	Threads Data Structure and Lifecycle
Module 3	**Process Synchronization**
Unit 1	Race condition, Critical Region and Mutual Exclusion
Unit 2	Deadlocks
Unit 3	Synchronization
Unit 4	Synchronization Problems
Module 4	**Memory Management**
Unit 1	Memory Swapping
Unit 2	Memory Partition
Unit 3	Virtual Memory
Unit 4	Caching and Thrashing
Unit 5	Replacement Policies

Modules 1 and 2 introduce the notion of processes and concurrency, which are at the core of contemporary operating systems. In a system, a process is the unit of work. A system of this kind is composed of a collection of simultaneously running processes, some of which are operating-system processes (those that execute system code) and the remainder of which are user processes.

Module 3 described process synchronization, and deadlock. Deadlock is the state of permanent blocking of a set of processes each waiting for event to occur. Part of the discussion in the module is a race condition and synchronization problems. Monitor and semaphores are few among many synchronization problem solutions. All of which are discussed extensively in this module

Finally, module 4 is concerned with the management of main memory throughout a process's operation. To maximize the CPU's use and the speed with which it responds to its users, the computer must maintain several processes in memory. There are several memory management algorithms, and the success of any one technique is situation-dependent.

Each unit consists of one or two weeks' work and include an introduction, objectives, reading materials, exercises, conclusion, summary, tutor-marked assignments (TMAs), references and other resources. The units direct you to work on exercises related to the required reading. In general, these exercises test you on the materials you have just covered or require you to apply it in some way and thereby assist you to evaluate your progress and to reinforce your comprehension of the material. Together with TMAs, these exercises will help you in achieving the stated learning objectives of the individual units and of the course as a whole.

Presentation Schedule

Your course materials have important dates for the early and timely completion and submission of your TMAs and attending tutorials. You should remember that you are required to submit all your assignments by the stipulated time and date. You should guide against falling behind in your work.

Assessment

There are three aspects to the assessment of the course. First is made up of self-assessment exercises. Second, consists of the tutor-marked assignments and third is the written examination/end of course examination.

You are advised to do the exercises. In tackling the assignments, you are expected to apply information, knowledge and techniques you have gathered during the course. The assignments must be submitted to your facilitator for formal assessment in accordance with the deadline stated in the presentation schedule and the assessment file. The work you submit to your tutor for assessment will count for 30% of your total course mark. At the end of the course, you will need to sit for a final or end of course examination of about three hours duration. This examination will count for 70% of your total course mark.

Tutor-Marked Assignment (TMAs)

The TMA is a continuous assessment component of your course. It accounts for 30% of the total score. You will be given four TMAs to answer. Three of these must be answered before you are allowed to sit for end of course examination. The TMAs would be given to you by your facilitator and should be returned after you have done the assignment. Assignment questions for the units in this course are contained in the assignment file. You will be able to complete your assignments from the information and material contained in your reading, references and study units. However, it is desirable in all degree level of education to demonstrate that you have read and researched more into your references, which will give a wider view point and may provide you with a deeper understanding of the subject.

Make sure that each assignment reaches your facilitator on or before the deadline given in the presentation schedule and assignment file. If for any reason you cannot complete your work on time, contact your facilitator before the assignment is due to discuss the possibility of an extension. Extension will not be granted after the due date unless in exceptional circumstances.

Final Examination and Grading

The end of course examination for operating system (CIT315) will be for three (3) hours and it has a value of 70% of the total course score. The examination will consist of questions, which will reflect the type of self-testing, practice exercise and tutor-marked assignment problems you have previously encountered. All areas of the course will be assessed.

Use the time between finishing the last unit and sitting for the examination to revise the whole course. You might find it useful to review your self-test, TMAs and comments on them before the examination. The end of course examination covers information from all parts of the course.

Course Marking Scheme

Assignment	**Marks**
Assignment 1 – 4	For assignment, best three marks of the four counts at 10% each, i.e., 30% of Course Marks.
End of Course Examination	70% 0f the overall Course Marks.
Total	100% of Course Material.

Facilitators/Tutors and Tutorials

There are 16 hours of tutorials provided in support of this course. You will be notified of the dates, time, and location of these tutorials as well as the name and phone number of your facilitator, as soon as you are allocated to a tutorial group.

Your facilitator will mark and comment on your assignments, keep a close watch on your progress and any difficulties you might face and provide assistance to you during the course. You are expected to mail your Tutor-Marked Assignments to your facilitator before the schedule date (at least two working days are required). They will be marked by your tutor and returned to you as soon as possible.

Do not delay to contact your facilitator by telephone or e-mail if you need assistance.

The following might be circumstances in which you would find assistance necessary, hence you would have to contact your facilitator if:

- You do not understand any part of the study or assigned readings
- You have difficulty with self-tests
- You have question or problem with an assignment or with the grading of an assignment.

You should endeavour to attend the tutorials. This is the only chance to have face to face contact with your course facilitator and to ask questions which may be answered instantly. You can raise any problem encountered in the course of your study.

To have more benefits from course tutorials, you are advised to prepare a list of questions before attending them. You will learn a lot from participating actively in discussions.

Summary

Operating System is a course that intends to intimate the learner with the principles of the operating system. Upon completing this course, you will be equipped with the knowledge of process management consisting of processes and concurrency, process synchronization, issues occurring during process execution such as race conditions, deadlock, and solutions to free yourself from synchronization problems. You will be trained on memory management schemes, algorithms and techniques to solve resource allocation and memory addressing scenarios.

I wish you success in the course and I hope you find it very interesting.

CONTENTS **PAGE**

Module 1: Process Management

Introduction of Module

Early computer systems allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, current-day computer systems allow multiple programs to be loaded into memory and executed concurrently. This evolution required firmer control and more compartmentalization of the various programs; and these needs resulted in 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 concept of a process helps us understand how programs execute in an operating system. A process is the unit of work in most systems. Systems consist of a collection of processes: Operating-system processes execute system code, and user processes execute user code. All these processes may execute concurrently.

In this module, we are going to look into processes, state of processes and how processes can be switched to improve multitasking of processors.

Unit 1: Processes and State

Unit 2: Context Switching

Unit 3: Interrupts and Interrupts Handler

Unit 1 Processes and States

Contents

1.0 Introduction

All multiprogramming operating systems, from single-user systems such as Windows to mainframe systems such as IBM's mainframe operating system, which can support thousands of users, are built around the concept of the process.

In general, theoperating system must switch between the execution of multiple processes, to maximize processor utilization while providing reasonable response time.The operating system must allocate resources to processes in conformance with a specific policy (e.g., certain functions that are of higher priority) while at the same time avoiding deadlock. The Operating System may be required to support inter process communication and user creation of processes, both of which may aid in the structuring of applications.

2.0 Intended Learning Outcomes (ILOs)

At the end of this unit, the students will be able to

- Explain what process is
- Identify the states of process
- Identify the importanceof context switching
- Know how to interrupt a running process

3.0 Main Content

3.1 What is a process?

A process is a program in execution. A process is more than the program code, which is sometimes known as the **text section**. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's registers. A process generally also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables), and a **data section**, which contains global variables and a **heap**, which is memory that is dynamically allocated during process run time. A program is a passive entity that does not perform any actions by itself; it has to be executed if the actions it calls for are to take place. A process is an execution of a program. It actually performs the actions specified in a program. An operating system shares the CPU resources among processes. This is how it gets user programs to execute.

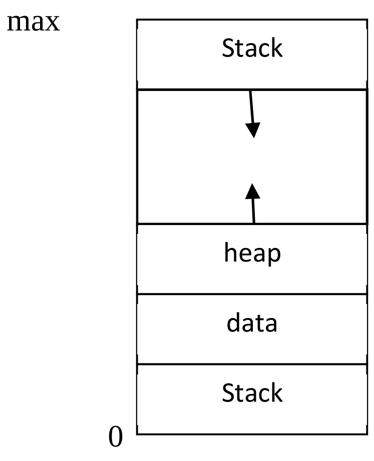


Figure 1.1: *Process in memory*

When a user initiates execution of a program, the OS creates a new process and assigns a unique id to it. It now allocates some resources to the process— sufficient memory to accommodate the address space of the program, and some devices such as a keyboard and a monitor to facilitate interaction with the user. The process may make system calls during its operation to request additional resources such as files. We refer to the address space of the program and resources allocated to it as the address space and resources of the process, respectively.

3.2 Process Control Block (PCB)

To represent a process, we need a data structure, which we simply call a process control block (PCB). It either contains or refers to all the per-process information mentioned above, including the address space. To achieve efficient sharing of these resources the O/S needs to keep track of all processes at all times.

Suppose that the processor begins to execute a given program code and we refer to executing entity as a process. At any given point of time, while the program is executing the O/S uniquely keeps track of this process state by maintaining a table known as Process Control Block (PCB) shown in Figure 1.2 below

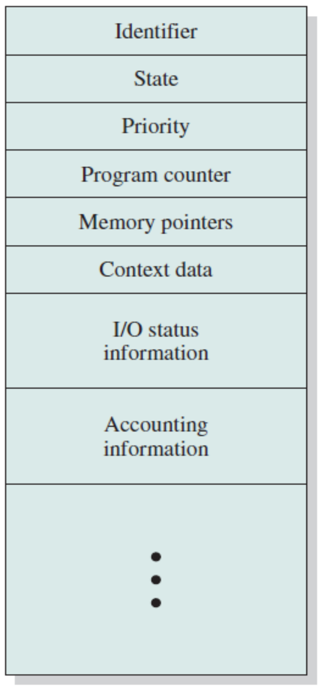


Figure 1.2: *Process Control Block diagram*

- **Identifier**: A unique identifier associated with this process, to distinguish it from all other processes.
- **State**: If the process is currently executing, it is in the running state.
- **Priority**: Priority level relative to other processes.
- **Program counter**: The address of the next instruction to be executed.

- **Memory pointers**: Includes pointers to the program code and data associated with this process, plus any memory blocks shared with other processes.
- **Context data**: These are data that are present in registers in the processor while the process is executing.
- **I/O status information**: Includes outstanding I/O requests, I/O devices assigned to this process, a list of files in use by the process, and so on.
- **Accounting information**: May include the amount of processor time and clock time used, time limits, account numbers, and so on

3.3 State Process Model and Diagrams

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

- **New**. The process is being created.
- **Running**. Instructions are being executed.
- **Ready**. The process is waiting to be assigned to a processor.
- **Blocked/Waiting:** A process that cannot execute until some event occurs, such as the completion of an I/O operation.
- **Terminated**. The process has finished execution.

3.3.1 A two State process model

The first step in designing an OS to control processes is to describe the behavior that we would like the processes to exhibit. We can construct the simplest possible model by observing that at any time, a process is either being executed by a processor or not. In this model, a process may be in one of two states: **Running** or **Not Running**, as shown in Figure 1.3.

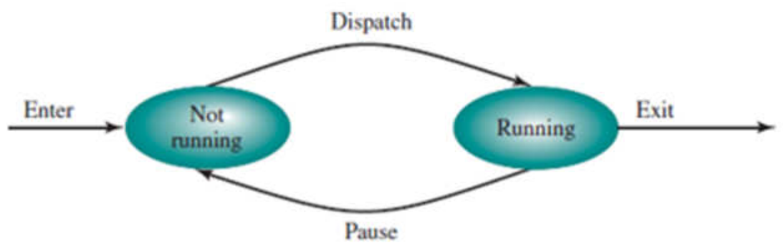


Figure 1.3: *A two state process model*

From time to time the currently running process will be interrupted and the dispatcher (O/S) selects some other process to run. The former process moves from the **Running** state to the **Not Running** state, and one of the other processes moves to the **Running** state

3.3.2 A Five State process model

In a five-state process model, implementation above is inadequate: Some processes in the Not Running state are ready to execute, while others are blocked, waiting for an I/O operation to complete. Thus, using a single queue, the dispatcher could not just select the process at the oldest end of the queue. Rather, the dispatcher would have to scan the list looking for the process that is not blocked and that has been in the queue the longest.

A more natural way to handle this situation is to split the Not Running state into two states: Ready and Blocked. This is shown in Figure 1.4

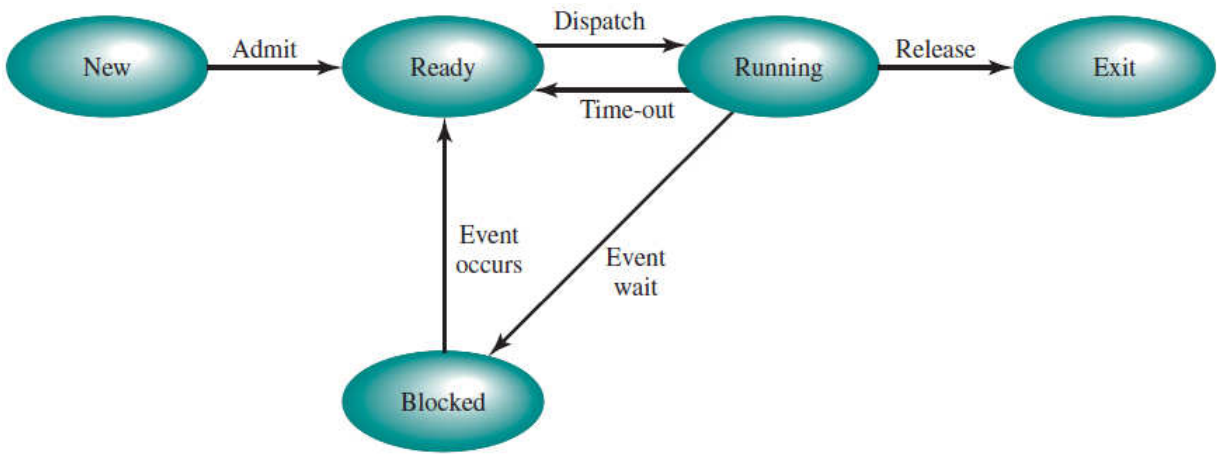


Figure 1.4: A five state process model

In fig. 1.4, *the possible transitions are as follows:*

• **Null⟶ New:** A new process is created to execute a program.

• **New⟶ Ready:** The OS will move a process from the New state to the Ready state when it is prepared to take on an additional process.

Ready$\rightarrow$ Running: When it is time to select a process to run, the OS chooses one of the processes in the Ready state. This is the job of the scheduler or dispatcher.

• **Running $\rightarrow$Exit:** The currently running process is terminated by the OS if the process indicates that it has completed, or if it aborts.

• **Running $\rightarrow$Ready:** The most common reason for this transition is that the running process has reached the maximum allowable time for uninterrupted execution.

Running $\rightarrow$ Blocked: A process is put in the Blocked state if it requests something for which it must wait. For example, a process may request a service from the OS that the OS is not prepared to perform immediately. Or the process may initiate an action, such as an I/O operation, that must be completed before the process can continue. A process may be blocked when it is waiting for another process to provide data or waiting for a message from another process.

3.4 Scheduling Algorithms

Scheduling deals with the problem of deciding which of the process in the ready queue is to be allocated to the CPU. There are three characteristics comparison used to make a substantial difference in the determination of the best algorithm. The criteria include the following:

- CPU Utilization
- Throughput
- Turnaround time
- Waiting time and;
- Response time

Two scheduling schemes exist. The Preemptive and Non-preemptive Schemes

Preemptive scheduling occurs in the event that a process transitions from running to ready or waiting to ready, During this period, the resources (mostly CPU cycles) are assigned to the process and subsequently removed. If the process still has CPU burst time left, it is then put back in the ready queue. This procedure will wait in the ready queue for another opportunity to run. While in non-preemptive scheduling, when a process quits or transitions from the running to the waiting state, scheduling occurs. Once the resources (CPU cycles) are allotted to a process, the process retains control of the CPU until it is terminated or enters a waiting state. Non-preemptive scheduling does not interrupt a running CPU process in the midst of its execution. Instead, it waits until the process's CPU burst duration is over before allocating the CPU to another process.

Now we describe several of the many CPU scheduling algorithms that exist:

1. **First-In, First-Out (FIFO) Scheduling Algorithm**

   FCFS is the simplest of all scheduling algorithms. The key concept of this algorithm is that it allocates the CPU in the order in which the processes arrive. That is the process that requests the CPU first is allocated the CPU first. The implementation of FCFS policy is managed with a FIFO (First-in-first out) queue. When the CPU is assigned to a process, that process retains it until it releases it, either by terminating or by requesting I/O devices. The FIFO algorithm is appropriate for the Batch operating system. Under the FIFO approach, the average waiting time is often fairly lengthy, particularly if requests for brief CPU burst processes wait behind long ones. FIFO is an example of non-preemptive scheduling.

2. **Shortest-Job First (SJF) Scheduling Algorithm**

   When the CPU is available, this algorithm assigned it to the process that has the smallest next CPU burst. If the two processes have the same length of CPU burst then FCFS scheduling algorithms are followed. The key concept of this algorithm is “CPU is allocated to the process with least CPU-burst time”. This algorithm is considered to be an optimal algorithm, as it gives the minimum average waiting time. The disadvantage of this algorithm is the problem of knowing ahead of time the length of time for which CPU is needed by a process. A prediction formula may be used to predict the amount of time for which CPU may be required by a process.

   The SJF algorithm may be either preemptive or non-preemptive

   The choice arises when a new process arrives in the ready queue while a previous process is executing.

   - A preemptive SJF will preempt a currently executing process and starts the execution of newly entered process
   - While non-preemptive SJF will allow the currently executing process to complete its burst time without any interruption in its execution.

3. **Round-Robin Scheduling Algorithm**

The Round-Robin (RR) scheduling algorithm is-designed especially for time-sharing systems. It is similar to FCFS scheduling algorithm, but preemption is added to switch between processes. A small unit of time called a **time quantum** (or time-slice) is defined where range is kept between 10 to 100ms. In this algorithm Ready Queue is assumed to be a circular Queue. The CPU scheduler (short-term) goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

To implement RR scheduling, the ready queue is kept as a FIFO Queue of the processes. New processes are added to the-tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt it after 1 time quantum, and dispatches the process.

In RR one of the two situations may arise.

- The process may have a **CPU burst of less than 1 time quantum**. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue.
- Otherwise, if the CPU burst of the currently **running process is longer than 1 time quantum,** the timer will go off and will cause an interrupt to the O/S. A context switch will be executed, and the process will be put at the tail of the ready queue.

3.5 Cases/Example

Consider the time-sharing system below which uses a time slice of 10 ms. It contains two processes P1 and P2. P1 has a CPU burst of 15 ms followed by an I/O operation that lasts for 100 ms, while P2 has a CPU burst of 30 ms followed by an I/O operation that lasts for 60 ms. Execution of P1 and P2 are described in Figure 1.5. Actual execution of programs proceeds as follows: System operation starts with both processes in the ready state at time 0. The scheduler selects process P1 for execution and changes its state to running. At 10 ms, P1 is preempted and P2 is dispatched. Hence P1's state is changed to ready and P2's state is changed to running. At 20 ms, P2 is preempted and P1 is dispatched. P1 enters the blocked state at 25 ms because of an I/O operation. P2 is dispatched because it is in the ready state. At 35 ms, P2 is preempted because its time slice elapses; however, it is dispatched again since no other process is in the ready state. P2 initiates an I/O operation at 45 ms. Now both processes are in the blocked state

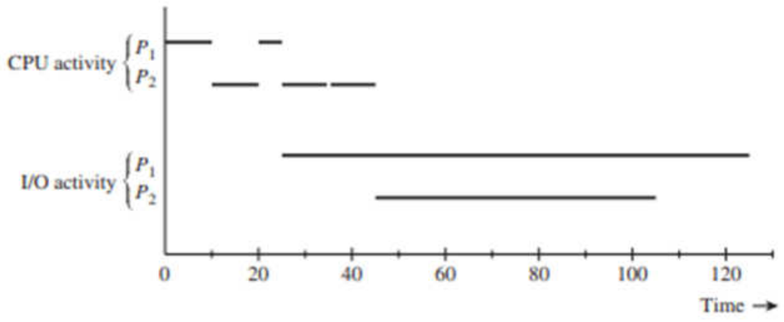


Figure 1.5: Real-time share system with two processes scenario

Discussion

Is it possible for two processors to execute three (3) processes?

4.0 Self-Assessment Exercise

What is a process creation and principal events that can create a process?

Answer

The triggering of an event raising to new non-exist process is referred to process creation. When a new process is to be added to those currently being managed, the operating system builds the data structures that are used to manage the process and allocates address space in main memory to the process.

There are four principal events that can trigger the creation of a process

1. System initialization.
2. Execution of a process-creation system call by a running process.
3. A user request to create a new process.
4. Initiation of a batch job.

What is the possible occurrence of a child process being terminated by parent process?

Answer

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

5.0 Conclusion

Using the process model, it becomes much easier to think about what is going on inside the system. Some of the processes run programs that carry out commands typed in by a user. Other processes are part of the system and handle tasks such as carrying out requests for file services or managing the details of running a disk or a tape drive. When a disk interrupt occurs, the system makes a decision to stop running the current process and run the disk process, which was blocked waiting for that interrupt. Thus, instead of thinking about interrupts, we can think about user processes, disk processes, terminal processes, and so on, which block when they are waiting for something to happen. When the disk has been read or the character typed, the process waiting for it is unblocked and is eligible to run again.

6.0 Summary

A computer user and the operating system have different views of execution of programs. The user is concerned with achieving execution of a program in a sequential or concurrent manner as desired, whereas the OS is concerned with allocation of resources to programs and servicing of several programs simultaneously, so that a suitable combination of efficient use and user service may be obtained. In this chapter, we discussed various aspects of these two views of execution of programs. A process is a model of execution of a program. When the user issues a command to execute a program, the OS creates the primary process for it.

The operating system allocates resources to a process and stores information about them in the process context of the process. To control operation of the process, it uses the notion of a process state. The process state is a description of the current activity within the process; the process state changes as the process operates. The fundamental process states are: ready, running, blocked, terminated, and suspended. The OS keeps information concerning each process in a process control block (PCB). The PCB of a process contains the process state, and the CPU state associated with the process if the CPU is not currently executing its instructions. The scheduling function of the kernel selects one of the ready processes and the dispatching function switches the CPU to the selected process through information found in its process context and the PCB.

7.0 References/Further Reading

Abraham, S., Peter, B. G., & Gagne, G. (2009). *Operating System Concepts* (C. Weisman (ed.); 8th ed.). John Wiley & Sons Inc.

Andrew S, T., & Bos, H. (2015). *Modern Operating Systems* (H. Marcia & J. Tracy (eds.); Fourth). Pearson Education.

Dhananjay, M. D. (2009). *Operating Systems A Concept-Based Approach* (B. Melinda (ed.)). McGraw-Hill Higher Education.

OS Process States - javatpoint. (n.d.). Retrieved May 1, 2022, from https://www.javatpoint.com/os-process-states

Process Table and Process Control Block (PCB) - GeeksforGeeks. (2020, June 28). https://www.geeksforgeeks.org/process-table-and-process-control-block-pcb/

Thomas, A., & Dahlin, M. (2015). *Operating Systems Principles & Practice Volume III : Memory Management* (S. Kaplan & S. Whitney (eds.); Second). Recursive Books, Ltd.

Williams, L. (n.d.). *CPU Scheduling Algorithms in Operating Systems*. Retrieved May 1, 2022, from https://www.guru99.com/cpu-scheduling-algorithms.html

Unit 2 Context Switching

Contents

1.0 Introduction

We use the term "**context**" to mean the setting in which execution is currently taking place. This setting determines what information, in the form of storage contents, is available. When a thread makes a system call, the processor switches from user mode to privileged mode and the thread switches from user context to system context; information within the operating system is now available to it. **Note** that these contexts overlap: for instance, a thread in a user context or the system context enters and exits numerous procedure contexts.

2.0 Intended Learning Outcomes (ILOs)

At the end of this unit, students will be able to

- Explain the concepts of context switching
- Explain how a running process is interrupted via system call
- Narrate switches from procedure to procedure

3.0 Main Content

3.1 Context/Process Switching

Changing the execution environment of one process to that of another is called process switching, which is the basic mechanism of multitasking.

Switching the CPU to another process requires saving the state of the old process and loading the saved state of the new process. The context of a process is represented in the PCB of the process; it includes the value of the CPU registers, the process state, and memory management information. When a context switch occurs, the operating system saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). Typical speeds are less than 10 milliseconds.

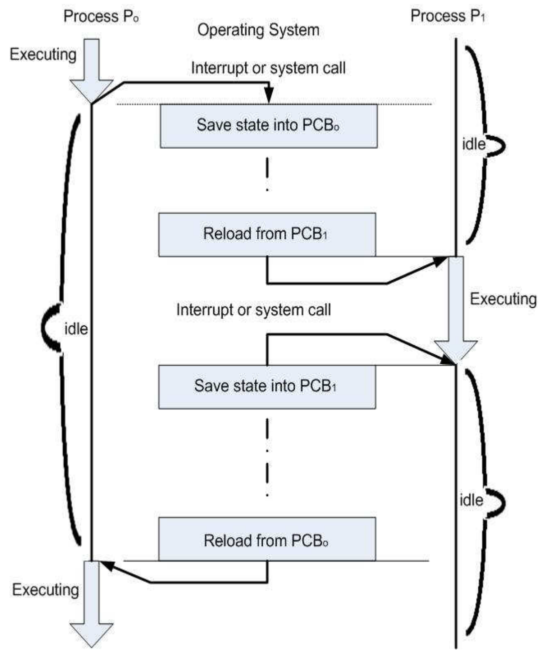


Figure 2.1: CPU Switching from one process to another

The switching time between two processesare pure overhead, because the system does no useful work.

Process switching overhead depends on the size of the state information of a process. Some computer systems provide special instructions to reduce the process switching overhead, e.g., instructions that save or load the PSW and all general-purpose registers, or flush the address translation buffers used by the memory management unit (MMU).

Switching can take place when the O/S has control of the system. An O/S can acquire control by:

- ***Interrupt***: an external event which is independent on the instructions.
- ***Trap:*** that is associated with current instruction execution.
- ***Supervisor call/system call***: that is explicit call to the O/S

Context Switching Steps

The steps involved in context switching are as follows −

- Save the context of the process that is currently running on the CPU. Update the process control block and other important fields.
- Move the process control block of the above process into the relevant queue such as the ready queue, I/O queue etc.
- Select a new process for execution.
- Update the process control block of the selected process. This includes updating the process state to running.
- Update the memory management data structures as required.
- Restore the context of the process that was previously running when it is loaded again on the processor. This is done by loading the previous values of the process control block and registers.

3.2 Procedures

The following code illustrates a simple procedure call in C:

```
int main( ) {
     int i;
    int a;
   . . .
i = sub(a, 1);
   . . .
return(0);
   }
int sub(int x, int y) {
  int i;
  int result = 1;
   for (i=0; i<y; i++)
     result *=x;
   return (result);
  }
```

The purpose of the procedure sub is pretty straightforward: it computes x^y. How the context is represented and is switched from that of main to that of sub depends on the architecture. The context of main includes any global variables (none in this case) as well as its local variables, i and a. The context of sub also includes any global variables, its local variables, i and result, and its arguments, x and y. On most architectures, global variables are always found at a fixed location in the address space, while local variables and arguments are within the current stack frame.

3.3 System Calls

System calls involve the transfer of control from user code to system (or kernel) code and back again. Keep in mind that this does not involve a switch between different threads, the original thread executing in user mode merely changes its execution mode to kernel (privileged) mode. However, it is now executing operating-system code and is effectively part of the operating system. Most systems provide threads with two stacks, one for use in user mode and one for use in kernel mode. Thus, when a thread performs a system call and switches from user mode to kernel mode, it also switches from its user-mode stack to its kernel-mode stack. For an example, consider a C program running on a Unix system that calls write. From the programmer's perspective, write is a system call, but a bit more work needs to be done before we enter the kernel. Write is actually a routine supplied in a special library of (userlevel) programs, the C library. Write is probably written in assembly language; the heart of it is some instruction that causes a trap to occur, thereby making control enter the operating system. Prior to this point, the thread had been using the thread's user stack. After the trap, as part of entering kernel mode, the thread switches to using the thread's kernel stack. Within the kernel our thread enters a fault-handler routine that determines the nature of the fault and then calls the handler for the write system call.

3.4 Case/Example

The following diagram depicts the process of context switching between the two processes P1 and P2.

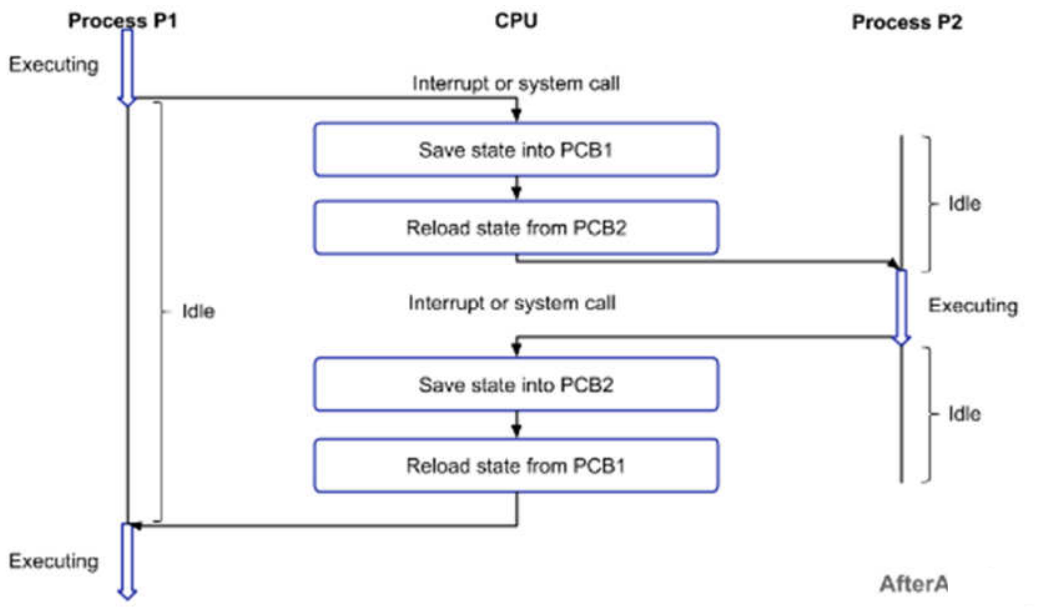


Figure 2.1b: Switching between two processes

In the above figure, you can see that initially, the process P1 is in the running state and the process P2 is in the ready state. Now, when some interruption occurs then you have to switch the process P1 from running to the ready state after saving the context and the process P2 from ready to running state. The following steps will be performed:

1. Firstly, the context of the process P1 i.e. the process present in the running state will be saved in the Process Control Block of process P1 i.e. PCB1.
2. Now, you have to move the PCB1 to the relevant queue i.e. ready queue, I/O queue, waiting queue, etc.
3. From the ready state, select the new process that is to be executed i.e. the process P2.

4. Now, update the Process Control Block of process P2 i.e. PCB2 by setting the process state to running. If the process P2 was earlier executed by the CPU, then you can get the position of last executed instruction so that you can resume the execution of P2.
5. Similarly, if you want to execute the process P1 again, then you have to follow the same steps as mentioned above(from step 1 to 4).

4.0 Self-Assessment Exercises

Context switch adds an overhead. Why?

Answer

1. In a multitasking computer environment, overhead is any time not spent executing tasks. It is overhead because it is always there, even if nothing productive is going on context switching is a part of the overhead but not the only part.
2. Because it takes time away from processing the task(s) at hand.
   For example, if there is only one task running, the processor can give it 100% of its processing time. There is no overhead of context switching

What is a context switch?

Answer

The term "context switch" refers to the act of saving the state of a process or thread in order to recover it and restart execution at a later time. This enables numerous processes to share a single central processing unit (CPU), which is a necessary characteristic of a multitasking operating system. When a process terminates; when the timer elapses indicating that the CPU should switch to another process, when the current process suspends itself, when the current process needs time consuming I/O, when an interrupt arises from some source aside from the timer

5.0 Conclusion

In the Operating System, there are cases when you have to bring back the process that is in the running state to some other state like ready state or wait/block state. If the running process wants to perform some I/O operation, then you have to remove the process from the running state and then put the process in the I/O queue. Sometimes, the process might be using a round-robin scheduling algorithm where after every fixed time quantum, the process has to come back to the ready state from the running state. So, these process switchings are done with the help of Context Switching. In this unit, we learned the concept of context switching in the operating system, and we will also learn about the advantages and disadvantages of context switching.

6.0 Summary

Context switching is a process that involves switching the CPU from one process or task to another. In this phenomenon, the execution of the process that is present in the running state is suspended by the kernel and another process that is present in the ready state is executed by the CPU. It is one of the essential features of the multitasking operating system. The processes are switched so fast that it gives the user an illusion that all the processes are being executed at the same time.System calls involve the transfer of control from user code to system (or kernel) code and back again. When a thread performs a system call and switches from user mode to kernel mode, it also switches from its user-mode stack to its kernel-mode stack.

7.0 Further Studies/References

Abraham, S., Peter, B. G., & Gagne, G. (2009). *Operating System Concepts* (C. Weisman (ed.); 8th ed.). John Wiley & Sons Inc.

Andrew S, T., & Bos, H. (2015). *Modern Operating Systems* (H. Marcia & J. Tracy (eds.); Fourth). Pearson Education.

Dhananjay, M. D. (2009). *Operating Systems A Concept-Based Approach* (B. Melinda (ed.)). McGraw-Hill Higher Education.

Thomas, A., & Dahlin, M. (2015). *Operating Systems Principles & Practice Volume III : Memory Management* (S. Kaplan & S. Whitney (eds.); Second). Recursive Books, Ltd.

Introduction of System Call - GeeksforGeeks. (2019, August 16). https://www.geeksforgeeks.org/introduction-of-system-call/

What is Context Switching. (2020, April 5). https://www.tutorialandexample.com/what-is-context-switching

Other Study sources

https://www.scribd.com/document/196387986/Context-Switch-Question-Answer

https://afteracademy.com/blog/what-is-context-switching-in-operating-system

Unit 3 Interrupts and Interrupts Handler

Contents

1.0 Introduction

In early years of computing processor has to wait for the signal for processing, so processor has to check each and every hardware and software program in the system if it has any signal to process. This method of checking the signal in the system for processing is called polling method. In this method, the problem processor has is to waste several clock cycles just to check the signal in the system. Because of this, the processor will become busy unnecessarily. If any signal came for the process, processor will take some time to process the signal due to the polling process in action. So, system performance also will be degraded and response time of the system will also decrease.

To overcome this problem engineers introduced a new mechanism, in this mechanism processor will not check for any signal from hardware or software but instead hardware/software will only send the signal to the processor for processing. The signal from hardware or software should have highest priority because processor should leave the current process and process the signal of hardware or software. This mechanism of processing the signal is called interrupt of the system.

2.0 Intended Learning Outcomes (ILOs)

At the end of this unit, students will be able to

- Explain the concept of interrupt
- Demonstrate masking and unmasking of interrupt request
- Demonstrate handling of interrupts

3.0 Main Content

3.1 Type of Interrupts

Interrupt is an event that requires the operating system's attention at any situation. The computer designer associates an interrupt with each event, whose sole purpose is to report the occurrence of the event to the operating system and enable it to perform appropriate event handling actions. When an I/O device has finished the work given to it, it causes an interrupt (assuming that interrupts have been enabled by the operating system). It does this by asserting a signal on a bus line that it has been assigned. This signal is detected by the interrupt controller chip on the parentboard, which then decides what to do. If no other interrupts are pending, the interrupt controller handles the interrupt immediately. However, if another interrupt is in progress, or another device has made a simultaneous request on a higher-priority interrupt request line on the bus, the device is just ignored for the moment. In this case it continues to assert an interrupt signal on the bus until it is serviced by the CPU.

Examples of interrupts

Here are some examples of the causes of interrupts. Note that not all need any intervention from the user.

- Hardware issue, such as a printer paper jam
- Key press by the user, e.g. CTRL ALT DEL
- Software error
- Phone call (mobile device)
- Disk drive indicating it is ready for more data

There are two types of interrupts: Hardware and Software interrupt

Hardware Interrupts

An electronic signal sent from an external device or hardware to communicate with the processor indicating that it requires immediate attention. For example, strokes from a keyboard or an action from a mouse invoke hardware interrupts causing the CPU to read and process it. So it arrives asynchronously and during any point of time while executing an instruction. Some key concepts such as Trap, Flag and watchdog timer need to be explained

The Interrupt flag (IF) is a bit in the CPU's FLAGS register that indicates whether the (CPU) will instantly react to maskable hardware interrupts. A trap is initiated by a user application, while an interrupt is initiated by a hardware device such as a keyboard, timer, or other similar device. A watchdog timer is a piece of software that monitors and recovers from computer problems. Watchdog timers are commonly employed in computers to automate the repair of transient hardware failures and to prevent malicious or errant software from interfering with system function.

Hardware interrupts are classified into two types

- **Maskable Interrupts –**those which can be disabled or ignored by the microprocessor. These interrupts are either edge-triggered or level-triggered, so they can be disabled. A level-triggered interrupt is requested by maintaining the interrupt signal at its specified active logic level (high or low). An edge-triggered interrupt is one that is triggered by a change in the level of the interrupt line, either a falling edge (from high to low) or a rising edge (from low to high). INTR, RST 7.5, RST 6.5, RST 5.5 are maskable interrupts in 8085 microprocessor. Processors have to interrupt mask register that allows enabling and disabling of hardware interrupts. Every signal has a bit placed in the mask register. If this bit is set, an interrupt is enabled & disabled when a bit is not set, or vice versa. Signals that interrupt the processors through these masks are referred to as masked interrupts.
- **Non-maskable Interrupts (NMI) –** Non-Maskable Interrupts are those which cannot be disabled or ignored by microprocessor. TRAP is a non-maskable interrupt. It consists of both level as well as edge triggering and is used in critical power failure conditions. The NMIs are the highest priority activities that need to be processed

immediately and under any situation, such as a timeout signal generated from a watchdog timer.

Software Interrupts

The processor itself requests a software interrupt after executing certain instructions or if particular conditions are met. These can be a specific instruction that triggers an interrupt such as subroutine calls and can be triggered unexpectedly because of program execution errors, known as exceptions or traps. They are – RST 0, RST 1, RST 2, RST 3, RST 4, RST 5, RST 6, RST 7.

3.3 Interrupt Handler

The job of the interrupt handler is to service the device and stop it from interrupting. Once the handler returns, the CPU resumes what it was doing before the interrupt occurred.When microprocessor receives multiple interrupt requests simultaneously, it will execute the interrupt service request (ISR) according to the priority of the interrupts.

Instruction for Interrupts –

1. **Enable Interrupt (EI) –** The interrupt enable flip-flop is set and all interrupts are enabled following the execution of next instruction followed by EI. No flags are affected. After a system reset, the interrupt enable flip-flop is reset, thus disabling the interrupts. This instruction is necessary to enable the interrupts again (except TRAP).
2. **Disable Interrupt (DI) –** This instruction is used to reset the value of enable flip-flop hence disabling all the interrupts. No flags are affected by this instruction.
3. **Set Interrupt Mask (SIM) –** It is used to implement the hardware interrupts (RST 7.5, RST 6.5, RST 5.5) by setting various bits to form masks or generate output data via the Serial Output Data (SOD) line. First the required value is loaded in accumulator then SIM will take the bit pattern from it
4. **Read Interrupt Mask (RIM) –** This instruction is used to read the status of the hardware interrupts (RST 7.5, RST 6.5, RST 5.5) by loading into the A register a byte

which defines the condition of the mask bits for the interrupts. It also reads the condition of SID (Serial Input Data) bit on the microprocessor.

Three main classes of interrupts:

I/O interrupts

An I/O device requires attention; the corresponding interrupt handler must query the device to determine the proper course of action. We cover this type of interrupt in the later section "I/O Interrupt Handling.”

Timer interrupts

Some timer, either a local APIC (Advanced Programmable Interrupt Controller) timer or an external timer, has issued an interrupt; this kind of interrupt tells the kernel that a fixed-time interval has elapsed. These interrupts are handled mostly as I/O interrupts.

Interprocessor interrupts

A CPU issued an interrupt to another CPU of a multiprocessor system.

I/O Interrupt Handling

In general, an I/O interrupt handler must be flexible enough to service several devices at the same time. In the PCI bus architecture, for instance, several devices may share the same IRQ (interrupt request) line. This means that the interrupt vector alone does not tell the whole story. In the example shown in Table 4-3, the same vector 43 is assigned to the USB port and to the sound card. However, some hardware devices found in older PC architectures (such as Instruction Set Architecture ISA) do not reliably operate if their IRQ line is shared with other devices.

Interrupt handler flexibility is achieved in two distinct ways, as discussed in the following list.

IRQ sharing

The interrupt handler executes several interrupt service routines (ISRs). Each ISR is a function related to a single device sharing the IRQ line. Because it is not possible to know in advance which particular device issued the IRQ, each ISR is executed to verify whether its device needs attention; if so, the ISR performs all the operations that need to be executed when the device raises an interrupt.

IRQ dynamic allocation

An IRQ line is associated with a device driver at the last possible moment; for instance, the IRQ line of the floppy device is allocated only when a user accesses the floppy disk device. In this way, the same IRQ vector may be used by several hardware devices even if they cannot share the IRQ line; of course, the hardware devices cannot be used at the same time.

Not all actions to be performed when an interrupt occurs have the same urgency. In fact, the interrupt handler itself is not a suitable place for all kind of actions. Long noncritical operations should be deferred, because while an interrupt handler is running, the signals on the corresponding IRQ line are temporarily ignored. Most important, the process on behalf of which an interrupt handler is executed must always stay in the TASK_RUNNING state, or a system freeze can occur. Therefore, interrupt handlers cannot perform any blocking procedure such as an I/O disk operation.

Regardless of the kind of circuit that caused the interrupt, all I/O interrupt handlers perform the same four basic actions:

i. Save the IRQ value and the register’s contents on the Kernel Mode stack.

ii. Send an acknowledgment to the PIC that is servicing the IRQ line, thus allowing it to issue further interrupts.

iii. Execute the interrupt service routines (ISRs) associated with all the devices that share the IRQ.

iv. Terminate by jumping to the ret_from_intr() address.

Interrupt Handler Responsibilities

The interrupt handler has a set of responsibilities to perform. Some are required by the framework, and some are required by the device. All interrupt handlers are required to do the following:

- Determine if the device is interrupting and possibly reject the interrupt.

  The interrupt handler must first examine the device and determine if it has issued the interrupt. If it has not, the handler must return DDI_INTR_UNCLAIMED. This step allows the implementation of device polling: it tells the system whether this device, among a number of devices at the given interrupt priority level, has issued the interrupt.

- Inform the device that it is being serviced.

  This is a device-specific operation, but it is required for the majority of devices. For example, SBus devices are required to interrupt until the driver tells them to stop. This guarantees that all SBus devices interrupting at the same priority level will be serviced.

- Perform any I/O request-related processing.

  Devices interrupt for different reasons, such as transfer done or transfer error. This step may involve using data access functions to read the device's data buffer, examine the device's error register, and set the status field in a data structure accordingly. Interrupt dispatching and processing are relatively time consuming.

- Do any additional processing that could prevent another interrupt.

  For example, read the next item of data from the device.

- Return DDI_INTR_CLAIMED.

Discussion

Explain interrupt received by the OS as a result of paper been empty from the printer tray

4.0 Self-Assessment/Exercises

1). Why interrupts are used?

These are used to get the attention of the CPU to perform services requested by either hardware or software.

2). What is NMI?

NMI is a non-maskable interrupt, that cannot be ignored or disabled by the processor

3). What is the function of interrupt acknowledge line?

The processor sends a signal to the devices indicating that it is ready to receive interrupts.

4). Describe hardware interrupt. Give examples

It is generated by an external device or hardware; such as keyboard keys or mouse movement invokes hardware interrupts

5). Describe software interrupt.

It is defined as a special instruction that invokes an interrupt such as subroutine calls. Software interrupts can be triggered unexpectedly because of program execution errors

6). Which interrupt has the highest priority?

- Non-maskable edge and level triggered
- TRAP has the highest priority

7). Give some uses of interrupt

- Respond quickly to time-sensitive or real-time events
- Data transfer to and from peripheral devices
- Responds to high-priority tasks such as power-down signals, traps, and watchdog timers
- Indicates abnormal events of CPU

5.0 Conclusion

An important property of interrupts is that they can be masked, i.e., temporarily blocked. If an interrupt occurs while it is masked, the interrupt indication remains pending; once it is unmasked, the processor is interrupted. How interrupts are masked is architecture-dependent, but two approaches are common. One approach is that a hardware register implements a bit vector where each bit represents a class of interrupts. If a particular bit is set, then the corresponding class of interrupts is masked. Thus the kernel masks interrupts by setting bits in the register. When an interrupt does occur, the corresponding mask bit is set in the register and cleared when the handler returns — further occurrences of that class of interrupts are masked while the handler is running.

6.0 Summary

When an interrupt occurs, the processor puts aside the current context (that of a thread or another interrupt) and switches to an interrupt context. When the interrupt handler is finished, the processor generally resumes the original context. Interrupt contexts require stacks; which stack is used? There are a number of possibilities: we could allocate a new stack each time an interrupt occurs, we could have one stack that is shared by all interrupt handlers, or the interrupt handler could borrow a stack from the thread it is interrupting.

7.0 Further Studies/References

Abraham, S., Peter, B. G., & Gagne, G. (2009). *Operating System Concepts* (C. Weisman (ed.); 8th ed.). John Wiley & Sons Inc.

Andrew S, T., & Bos, H. (2015). *Modern Operating Systems* (H. Marcia & J. Tracy (eds.); Fourth). Pearson Education.

Dhananjay, M. D. (2009). *Operating Systems A Concept-Based Approach* (B. Melinda (ed.)). McGraw-Hill Higher Education.

Thomas, A., & Dahlin, M. (2015). *Operating Systems Principles & Practice Volume III : Memory Management* (S. Kaplan & S. Whitney (eds.); Second). Recursive Books, Ltd.

Interrupts - GeeksforGeeks. (2022, January 13). https://www.geeksforgeeks.org/interrupts/

Types of Interrupts | How to Handle Interrupts? | Interrupt Latency. (2015, August 20). https://www.electronicshub.org/types-of-interrupts-and-how-to-handle-interrupts/

What is an Interrupt Handler? (n.d.). Retrieved May 1, 2022, from https://www3.nd.edu/~lemmon/courses/ee224/web-manual/web-manual/lab7/node5.html

Module 2 Concurrency – Multithreading

In computer systems many things must go on at the same time; that is, they must be **concurrent**. Even in systems with just one processor, execution is generally multiplexed, providing at least the illusion that many things are happening at once. At any particular moment there may be a number of running programs, a disk drive that has completed an operation and requires service, packets that have just arrived from the network and also require service, characters that have been typed at the keyboard, etc. The operating system must divide processor time among the programs and arrange so that they all make progress in their execution. And while all this is going on, it must also handle all the other input/output activities and other events requiring attention as well. Concurrent processing is a computing model in which multiple processors execute instructions simultaneously for better performance. Tasks are broken down into subtasks that are then assigned to separate processors to perform simultaneously, instead of sequentially as they would have to be carried out by a single processor. Concurrent processing is sometimes said to be synonymous with parallel processing.

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.

This chapter covers multithreaded programming. The discussion here not only goes through the basics of using concurrency in user-level programs, but also introduces a number of concepts that are important in the operating system.

Unit 1 Threads

Contents

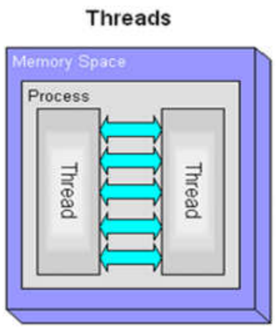
1.0 Introduction

In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by an operating system scheduler. A thread is a light-weight process. The implementation of threads and processes differs from one operating system to another, but in most cases, a thread is contained inside a process. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources.

Process switching overhead has two components that imposes challenges on multitasking of the processor

- Execution related overhead: The CPU state of the running process has to be saved and the CPU state of the new process has to be loaded in the CPU. This overhead is unavoidable.
- Resource-use related overhead: The process context also has to be switched. It involves switching of the information about resources allocated to the process, such as memory and files, and interaction of the process with other processes. The large size of this information adds to the process switching overhead.

This distinction has led to the development, in many operating systems, of a construct known as the thread. To distinguish the two characteristics, the unit of dispatching is usually referred to as a thread or lightweight process, while the unit of resource ownership is usually referred to as a process or task.



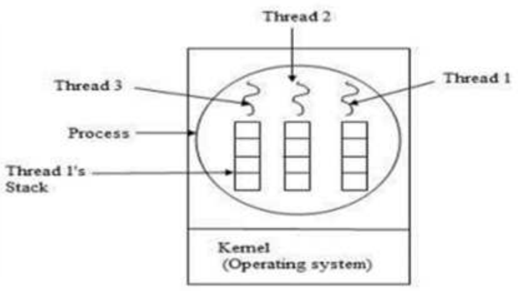


Figure 2.1(a): Threads intra-process Figure 2.1(b):AMultiple Thread Structure

Threads represent the software approach to improving performance of O/S by reducing the overhead of process switching. The main characteristics of threads comprise of thread ID, a program counter, a register set and a stuck. State reduction is achieved by having a group of related threads belonging to the same process and sharing its code section, data section and operating system resources such as memory and files to execute concurrently.

In a thread-based systems, thread takes over the role of process as the smallest individual unit of scheduling. In such a system, the process OR task serves as an environment for execution of threads.

2.0 Intended Learning Outcomes (ILOs)

At the end of this unit, the students will be able to

1. Explain the concepts of thread
2. Differentiate between thread and process
3. Write simple thread creation code/program instructions

3.0 Main Content

3.1 POSIX Thread

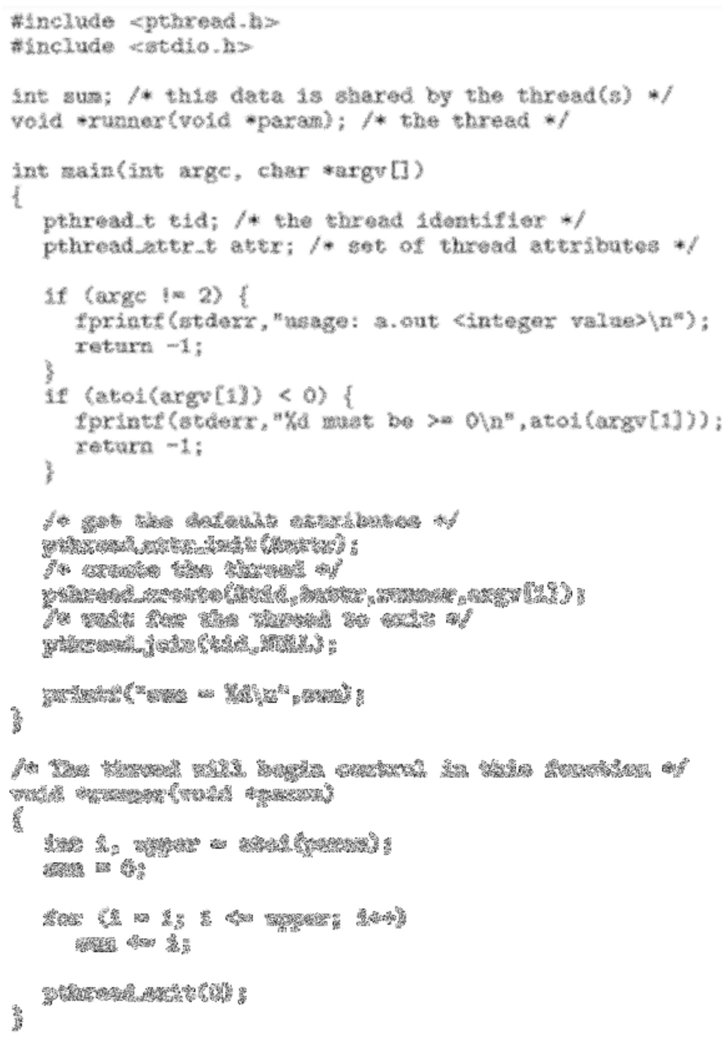
The ANSI/IEEE Portable Operating System Interface (POSIX) standard defines the pthreads application program interface for use by C language programs. Popularly called POSIX threads. The threads package it defines is called Pthreads. Most UNIX systems support it. The standard defines over 60 function calls.

All **Pthreads** have certain properties. Each one has an identifier, a set of registers (including the program counter), and a set of attributes, which are stored in a structure. The attributes include the stack size, scheduling parameters, and other items needed to use the thread. A new thread is created using the *pthread* create call. The thread identifier of the newly created thread is returned as the function value.

When a thread has finished the work it has been assigned, it can terminate by calling *pthread_exit*. This call stops the thread and releases its stack. Often a thread needs to wait for another thread to finish its work and exit before continuing. The thread that is waiting calls *pthread_join* to wait for a specific other thread to terminate. The thread identifier of the thread to wait for is given as a parameter.

Sometimes it happens that a thread is not logically blocked, but feels that it has run long enough and wants to give another thread a chance to run. It can accomplish this goal by calling *pthread_yield*. There is no such call for processes because the assumption there is that processes are fiercely competitive and each wants all the CPU time it can get. However, since the threads of a process are working together and their code is invariably written by the same programmer, sometimes the programmer wants them to give each other another chance.

Other two thread calls deal with attributes. *Pthread_attr_init* creates the attribute structure associated with a thread and initializes it to the default values. These values (such as the priority) can be changed by manipulating fields in the attribute structure. Finally, *pthread_attr_destroy* removes a thread's attribute structure, freeing up its memory. It does not affect threads using it; they continue to exist.



```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }

   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Figure 2.2: Multithreaded C Program using Pthread API

The C program shown in Figure 2.2 demonstrates the basic Pthreads API for constructing a multithreaded program that calculates the summation of a nonnegative integer in a separate thread. In a Pthreads program, separate threads begin execution in a specified function. In Figure 2.2, this is the runner() function. When this program begins, a single thread of control begins in main (). After some initialization, main() creates a second thread that begins control in the runner () function. Both threads share the global data sum. Let's look more closely at this program. All Pthreads programs must include the pthread.h header file. The statement *pthread_ttid* declares the identifier for the thread we will create. Each thread has a set of attributes, including stack size and scheduling information. The *pthread_attr_t*attr declaration represents the attributes for the thread. We set the attributes in the function call *pthread_attr _init(&att*r). Because we did not explicitly set any attributes, we use the default attributes provided. A separate thread is created with the *pthread_create()* function call. In addition to passing the thread identifier and the attributes for the thread, we also pass the name of the function where the new thread will begin execution-in this case, the *runner()* function. Last, we pass the integer parameter that was provided on the command line, argv. At this point, the program has two threads: the initial (or parent) thread in main() and the summation (or child) thread performing the summation.

3.2 Multithreading

Multithreading refers to the ability of an operating system to support multiple, concurrent paths of execution within a single process. The traditional approach of a single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach.

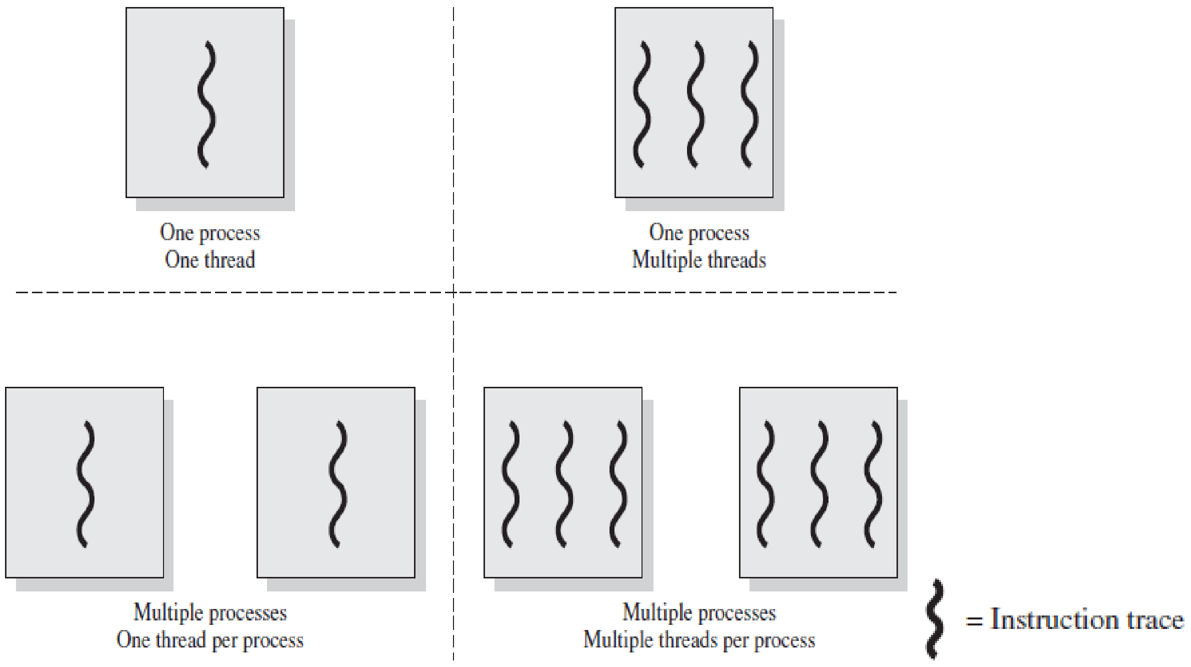


Figure 2.3: Threads and Processes

The two arrangements shown in the left half of Figure 2.3 are single-threaded approaches. MS-DOS is an example of an operating system that supports a single user process and a single thread.

Other operating systems, such as some variants of UNIX, support multiple user processes but only support one thread per process.

The right half of Figure 2.3 depicts multithreaded approaches. A Java run-time environment is an example of a system of one process with multiple threads.

A process is divided into a number of smaller tasks, each of which is called a thread. Multithreading describes the number of threads within a process that are executed at a time. Based on functionality, threads are divided into four categories:

1) One thread per process (One to one)
2) Many threads per process (One to Many)
3) Many single-threaded processes (Many to one)
4) Many kernel threads (Many to many)

1. **One thread per process**: A simple single-threaded application has one sequence of instructions, executing from beginning to end. The operating system kernel runs those instructions in user mode to restrict access to privileged operations or system memory. The process performs system calls to ask the kernel to perform privileged operations on its behalf.
2. **Many threads per process.** Alternately, a program may be structured as several concurrent threads, each executing within the restricted rights of the process. At any given time, a subset of the process's threads may be running, while the rest are suspended. Any thread running in a process can make system calls into the kernel, blocking that thread until the call returns but allowing other threads to continue to run. Likewise, when the processor gets an I/O interrupt, it preempts one of the running threads so the kernel can run the interrupt handler; when the handler finishes, the kernel resumes that thread.
3. **Manysingle-threaded processes**. As recently as twenty years ago, many operating systems supported multiple processes but only one thread per process. To the kernel, however, each process looks like a thread: a separate sequence of instructions, executing sometimes in the kernel and sometimes at user level. For example, on a multiprocessor, if multiple processes perform system calls at the same time, the kernel, in effect, has multiple threads executing concurrently in kernel mode.
4. **Many kernel threads**. To manage complexity, shift work to the background, exploit parallelism, and hide I/O latency, the operating system kernel itself can benefit from using multiple threads. In this case, each kernel thread runs with the privileges of the kernel: it can execute privileged instructions, access system memory, and issue commands directly to I/O

devices. The operating system kernel itself implements the thread abstraction for its own use. Because of the usefulness of threads, almost all modern operating systems support both multiple threads per process and multiple kernel threads.

Thus, all of the threads of a process share the state and resources of that process. They reside in the same address space and have access to the same data. When one thread alters an item of data in memory, other threads see the results when they access that item. If one thread opens a file with read privileges, other threads in the same process can also read from that file.

3.3 Threads Creation

A process is always created with one thread, called the *initial thread*. The initial thread provides compatibility with previous single-threaded processes. The initial thread's stack is the process stack.

Creating a thread should be a pretty straightforward operation: in response to some sort of directive, a new thread is created and proceeds to execute code independently of its creator. There are, of course, a few additional details. We may want to pass parameters to the thread. A stack of some size must be created to be the thread’s execution context.

Suppose we wish to create a bunch of threads, each of which will execute code to provide some service. In POSIX, we do the follows as shown below

```
voidstart_server( ) {
    pthread_tthread;
    int i;

    for (1=0; i<nr_of_server_threads; i++)
        pthread_create (
            &thread,        //thread ID
            0,              //default attributes
            server,         //start routine
```

```
                    argument);          //argument
}
void *server (void *arg) {
    //perform service
    return (0);
}
```

A thread is created by calling pthread_create. If it returns successfully (returns 0), a new thread has been created that is now executing independently of the caller. This thread's ID is returned via the first parameter (an output parameter that, in standard C programming style, is a pointer to where the result should be stored). The second parameter is a pointer to an attributes structure that defines various properties of the thread. Usually, we can get by with the default properties, which we specify by supplying a null pointer. The third parameter is the address of the routine in which our new thread should start its execution. The last argument is the argument that is actually passed to the first procedure of the thread. If *pthread_create* fails, it returns a positive value indicating the cause of the failure.

3.4 Threads Termination

Terminating threads has its share of subtle issues as well. Our threads return values: which threads receive these values and how do they do it? Clearly a thread that expects to receive another's return value must wait until that thread produces it, and this happens only when the other thread terminates. Thus, we need a way for one thread to wait until another terminates. Though a technique for one thread to wait for any other thread to terminate might seem useful, that threads return values makes it important to be particular about which thread (or threads) one is waiting for. Thus, it's important to identify a thread uniquely. Such identification can be a bit tricky if identifiers can be reused. POSIX provides a rather straightforward construct for waiting for a thread to terminate and retrieving its return value: pthread_join.

```
voidrlogind (intr_in, intr_out, intl_in, intl_out) {
        pthread_tin_thread, out_thread;
        two_ints_t in = {r_in, l_out}, out={l_in, r_out};

        pthread_create(&in_thread, 0, incoming, &in);
        pthread_create(&out_thread, 0, outgoing, &out);

        pthread_join(in_thread, 0);
        pthread_join(out_thread, 0);
}
```

Here our thread uses *pthread_join* to insure that the threads it creates terminate before it leaves the scope of their arguments. The first argument to *pthread_join* indicates which thread to wait for and the second argument, if nonzero, indicates where the return value should go. How exactly does a thread terminate? Or, more precisely, what does a thread do to cause its own termination?

One way for a thread to terminate is simply to return from its first procedure, returning a value of type void *. An alternative approach is that a thread call*pthread_exit*, supplying a void * argument that becomes the thread's return value.

3.5 Cases/Examples

Here is a simple complete multithreaded program that computes the product of two matrices. Our approach is to create one thread for each row of the product and have these threads compute the necessary inner products.

```
#include<stdio.h>
#include<pthread.h>          /* all POSIX threads declarations */
#include<string.h>           /* needed to use strerror below */
#define M 3
```

```
#define N 4
#define P 5
int A[M][N];            /* multiplier matrix */
int B[N][P];            /* multiplicand matrix */
int C[M][P];            /* product matrix */
void *matmult(void *);
intmain( ) {
inti, j;
pthread_tthr[M];
interror;
/* initialize the matrices ... */
...
for (i=0; i<M; i++) {  /* create the worker threads */
        if (error = pthread_create (
                &thr[i],
                0,
                Matmult,
                (void *)i)) {
                fprintf(stderr, "pthread_create: %s",  strerror(error) ):
                /*  This is how one convert error code to useful text */
                exit(1);
                }
        }
for (i=0; i<M; i++)    /* wait for workers to finish */
```

```
        pthread_join(&thr[i], 0)
/* Print the result */
…
return (0);
}
```

Discussion

Is a kernel interrupt handler a thread? Why?

4.0 Self-Assessment/Exercises

A MultiThreaded Hello World

```
#include <stdio.h>
 #include "thread.h"
static void go(int n);
#define NTHREADS 10
static thread_tthreads[NTHREADS];
int main(int argc, char **argv) {
        int i;
        long exitValue;
        for (i = 0; i< NTHREADS; i++){
                thread_create(&(threads[i]),
                &go, i);
        }
        for (i = 0; i< NTHREADS; i++) {
                exitValue = thread_join(threads[i]);
                printf("Thread %d returned with %ld\n",
                i, exitValue);
        }
        printf("Main thread done.\n");
        return 0;
}
void go (int n) {
printf("Hello from thread %d\n", n);
thread_exit(100 + n);                   // Not reached
}
```

Output

```
% ./threadHello
Hello from thread 0
Hello from thread 1
Thread 0 returned 100
Hello from thread 3
Hello from thread 4
 Thread 1 returned 101
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 8
Hello from thread 7
Hello from thread 9
Thread 2 returned 102
Thread 3 returned 103
Thread 4 returned 104
Thread 5 returned 105
Thread 6 returned 106
Thread 7 returned 107
Thread 8 returned 108
Thread 9 returned 109
Main thread done.
```

Multithreaded program using the simple threads API that prints "Hello" ten times. Also shown is the output of one possible run of this. Refer to the above program to answer the below questions.

1) **Why might the "Hello" message from thread 2 print after the "Hello" message for thread 5, even though thread 2 was created before thread 5?**

Answer

Creating and scheduling threads are separate operations.

Although threads are usually scheduled in the order that they are created, there is no guarantee. Further, even if thread 2 started running before thread 5, it might be preempted before it reaches the printf call. Rather, the only assumption the programmer can make is that each of the threads runs on its own virtual processor with unpredictable speed. Any interleaving is possible.

2) **Why must the "Thread returned" message from thread 2 print before the Thread returned message from thread 5?**

Answer

Since the threads run on virtual processors with unpredictable speeds, the order in which the threads finish is indeterminate. However, the main thread checks for thread completion in the order they were created. It calls thread_join for thread i +1 only after thread_join for thread i has returned.

3) **What is the minimum and maximum number of threads that could exist when thread 5 prints "Hello?"**

Answer

When the program starts, a main thread begins running main. That thread creates NTHREADS = 10 threads. All of those could run and complete before thread 5 prints "Hello." Thus, the minimum is two threads. The maximum and thread is 5. On the other hand, all 10 threads could have been created, while 5 was the first to run. Thus, **the maximum is 11 threads**.

5.0 Conclusion

The purpose of this unit was not just to teach you how to write a multithreaded program, but also to introduce various important operating-systems issues. The concerns include both how threads are implemented and how multithreaded programming is used within operating system.

6.0 Summary

Concurrency is ubiquitous not only do most smartphones, servers, desktops, laptops, and tablets have multiple cores, but users have come to expect a responsive interface at all times.

Although threads are not the only possible solution to these issues, they are a general-purpose technique that can be applied to a wide range of concurrency issues. In our view, multithreaded programming is a skill that every professional programmer must master.

In the unit discussed,

A process is divided into a number of smaller tasks, each of which is called a thread. A process is always created with one thread, called the *initial thread, which* is created by calling *pthread_create*. Each thread within a process is identified by a *thread ID*, whose datatype is *pthread_t*. Many threads within a process execute at a time and is called Multithreading.

A thread can be waited to terminate by calling *pthread_join*; or by explicitly calling *pthread_exit*.

7.0 Further Studies/References

Abraham, S., Peter, B. G., & Gagne, G. (2009). *Operating System Concepts* (C. Weisman (ed.); 8th ed.). John Wiley & Sons Inc.

Andrew S, T., & Bos, H. (2015). *Modern Operating Systems* (H. Marcia & J. Tracy (eds.); Fourth). Pearson Education.

Dhananjay, M. D. (2009). *Operating Systems A Concept-Based Approach* (B. Melinda (ed.)). McGraw-Hill Higher Education.

Thomas, A., & Dahlin, M. (2015). *Operating Systems Principles & Practice Volume III : Memory Management* (S. Kaplan & S. Whitney (eds.); Second). Recursive Books, Ltd.

CHAPTER 5 THREADS & MULTITHREADING 1. Single and Multithreaded Processes. (n.d.). Retrieved May 1, 2022, from https://slideplayer.com/slide/4402210/

Silberschatz, A., Gagne, G., & Galvin, P. (n.d.). *Operating Systems Concepts: Chapter 4 - Threads* (Ninth). Retrieved May 1, 2022, from https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

Threads in Operating System - javatpoint. (n.d.). Retrieved May 1, 2022, from https://www.javatpoint.com/threads-in-operating-system

Unit 2 Types of Threads

Contents

1.0 Introduction

There are two main places to implement threads: user space and the kernel. The choice is a bit controversial, and a hybrid implementation is also possible. We will now describe these methods, along with their advantages and disadvantages. Threads are implemented in different ways. The main difference is how much the kernel and the application program know about the threads. These differences lead to different implementations for overhead and concurrency in an application program.

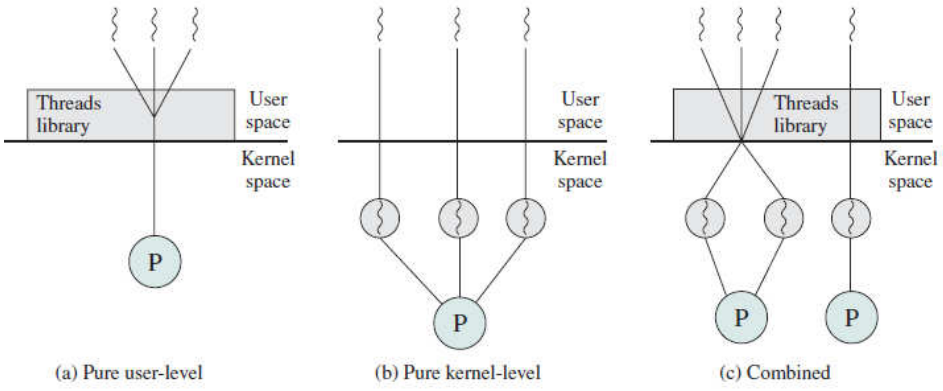


Figure 2.4:Threads level

2.0 Intended Learning Outcomes (ILOs)

At the end of this unit, students will be able to

- Explain the types of threads
- Implement a thread at user, kernel and hybrid level

3.0 Main Content

3.1 User-level Thread

User-level threads

User level threads are implemented by a thread library. The library set up the thread implementation arrangement without using the kernel, and interleaves operation of the threads in the process. Thus, the kernel is not aware of presence of user-level threads in a process, as it sees only the process.

Implementing Threads in User Space

The first method is to put the threads package entirely in user space. The kernel knows nothing about them. As far as the kernel is concerned, it is managing ordinary, single-threaded processes. The first, and most obvious, advantage is that a user-level threads package can be implemented on an operating system that does not support threads. The process will be unblocked when some activities occurred, thus resume execution of the thread library function, which will perform scheduling and switch to the execution of the newly activated thread.

The threads run on top of a run-time system, which is a collection of procedures that manage threads. We have seen four of these already: *pthread_create*, *pthread_exit*, *pthread_join*, and *pthread_yield*.

A process uses the library function **create-thread** to create a new thread. The library function creates a thread control block for the new thread and start considering the new thread for scheduling.

In this scheme, the library function performs 'scheduling' and switching from one thread to another. The kernel is not able to see the switching between two threads in a process, it believes that the process is continuously in operation. If the thread library cannot find a ready thread in the process, it makes a **block me** system call. Then the kernel now blocks the process. When a thread is moved to ready state or blocked state, the information needed to restart it is stored in the thread table, exactly the same way as the kernel stores information about processes in the process table.

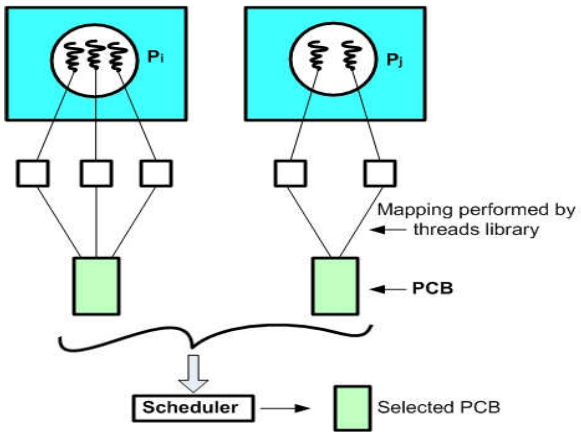


Figure 2.5: Scheduling of User-level thread

When a thread does something that may cause it to become blocked locally, for example, waiting for another thread in its process to complete some work, it calls a run-time system procedure. This procedure checks to see if the thread must be put into blocked state. If so, it stores the thread's registers (i.e., its own) in the thread table, looks in the table for a ready thread to run, and reloads the machine registers with the new thread's saved values. As soon as the stack pointer and program counter have been switched, the new thread comes to life again automatically. If the machine happens to have an instruction to store all the registers and another one to load them all, the entire thread switch can be done in just a handful of instructions

Advantages and Disadvantages of User-level Threads

Advantages:

Since thread scheduling is implemented by thread library, so thread switching overhead is smaller than the kernel-level thread. This arrangement enables each process to use a scheduling policy that is best suited to it. A process implementing a real multi-threaded server may perform R-R scheduling of its threads.

Disadvantages:

Managing threads without the involvement of kernel has few drawbacks such as:

1. The kernel does know the distinction between a thread and a process, so if a thread wants to be blocked, the kernel will block its parent. As a result, all the threads in the process get blocked until the cause of the blocking is removed.
2. Since kernel schedules a process, and thread library schedule the thread within a process, so at most one of the thread of a process is in operation at any time. Thus, process ULT cannot provide parallelism and concurrency provided by KLTs. Thus, a serious impairment if a thread makes a system call that leads to blocking.

3.2 Kernel-level Thread

A KLT is implemented by kernel. Hence, creation and termination of KLTs, and checking their status is performed by system calls.

When a process makes a *create-thread* system call, the kernel assigns an ID to it, and allocates a thread control block (TCB), which contain the pointer to the PCB of the process. When an event occurs, the kernel saves the CPU state of the interrupted thread in its TCB.

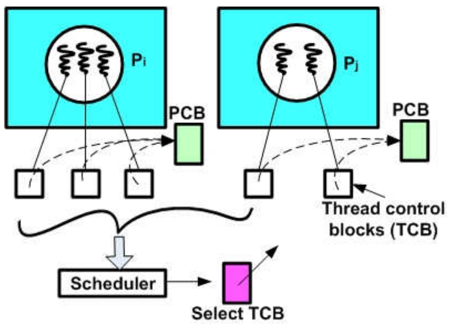


Figure 2.6: Scheduling of Kernel-level Threads

After event handling, the scheduler considers TCB of all threads and selects a ready thread. The dispatcher uses the PCB pointer in its TCB to check if the selected thread belongs to a different process than the interrupted thread. If so, it saves the context of the process to which the selected thread belongs. It then dispatches the selected thread.The actions to save and load the process context are not necessary, if both threads belong to the same process. This feature reduces switching overhead.

The specifics of the implementation vary depending on the context:

Kernel threads: The simplest case is implementing threads inside the operating system kernel, sharing one or more physical processors. A kernel thread executes kernel code and modifies kernel data structures. Almost all commercial operating systems today support kernel threads.

Kernel threads and single-threaded processes. An operating system with kernel threads might also run some single-threaded user processes. These processes can invoke system calls that run concurrently with kernel threads inside the kernel.

Multithreaded processes using kernel threads: Most operating systems provide a set of library routines and system calls to allow applications to use multiple threads within a single user-level process. These threads execute user code and access user-level data structures. They also make system calls into the operating system kernel. For that, they need a kernel interrupt stack just like a normal single threaded process.

Due to the relatively greater cost of creating and destroying threads in the kernel, some systems take an environmentally correct approach and recycle their threads. When a thread is destroyed, it is marked as not runnable, but its kernel data structures are not otherwise affected. Later, when a new thread must be created, an old thread is reactivated, saving some overhead. Thread recycling is also possible for user-level threads, but since the thread-management overhead is much smaller, there is less incentive to do this. Kernel threads do not require any new, nonblocking system calls. In addition, if one thread in a process causes a page fault, the kernel can easily check to see if the process has any other runnable threads, and if so, run one of them while waiting for the required page to be brought in from the disk.

Advantages and Disadvantages of Kernel-level threads

Advantages:

A KLT is like a process except that it has smaller amount of state information. The similarity between process and thread is convenient for programmers, as programming for thread is same as programming for processes.

In a multiprocessor system, KLTs provides parallelism i.e several threads belonging to same process can be scheduled simultaneously, which is not possible using the ULTs.

Disadvantages:

However, handling the threads like processes has its disadvantages too. Switching between threads is perform by the kernel, as a result of event handling. Hence, it incurs the overhead of event handling, even if the interrupted thread and the selected thread belong to the same process. This feature limits the saving in the switching overhead.

Another possible issue is signal delegation. As signals are frequently sent to processes which are handled by non-dedicated threads. When signals are sent, conflict might arise in the course of more than one thread attempting to handle signal at the same time. Which thread should handle it?

3.3 Hybrid Thread

A hybrid thread model has both ULT and KLT and method of associating ULTs with KLTs. Different methods of associating user and kernel-level threads provided different combination of the low switching overhead of ULT, and high concurrency and parallelism of KLT.

Following are three search methods in which thread library creates ULTs in a process, and associates a TCB with each ULT, and kernel creates KLT and associates kernel thread control block (KTCB) with each KLT.

Many-to-one association method: In this method a single KLT is created in each process by the kernel and all ULTs created by thread library are associated with this single KLT. In this method, ULT can be current without being parallel, thread switching results in low overhead and blocking of a ULT leads to a blocking of all threads in the process.

One-to-one method of association: In this method each ULT is permanently mapped into a KLT. This method provides an effect similar to KLT: thread can operate in parallel on different CPUs of a multiprocessor system. However, switching between threads is performed at the kernel level and results in high overhead. Blocking a user level thread does not block other threads.

Many-to-many association method: In this method ULT can be mapped to any KLT. By this method parallelism is possible and switching is possible at kernel-level with low overhead. Also blocking of ULT does not block other threads as they are mapped with different KLTs. Overall this methods requires complex implementation eg, Sun Solari operating system.

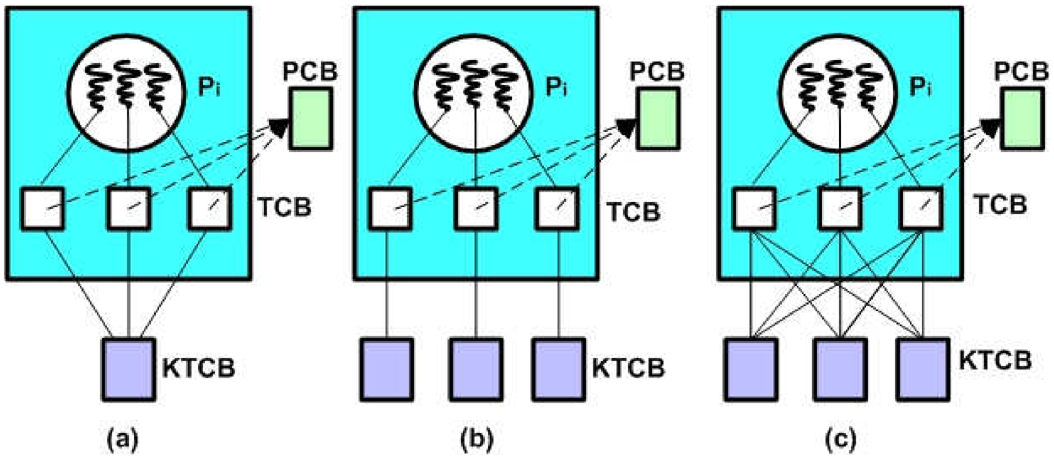


Figure 2.6: (a) Many to one (b) One to one (c)Many to many associations in hybrid threads

3.4 Cases/Examples

Figure 2.7 below demonstrates a uniprocessor multiprogramming using threads to interleave between the processes. Three threads in two processes are interleaved on the processor. Execution passes from one thread to another either when the currently running thread is blocked or when its time slice is exhausted.

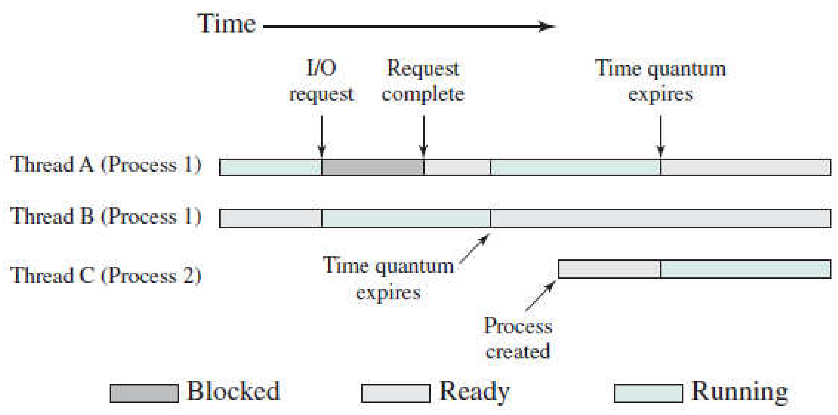


Figure 2.7: Multithreading on a uniprocessor

Discussion

Consider an unprocessed kernel that user programs trap into using system calls. The kernel receives and handles interrupt requests from I/O devices. Would there be any need for critical sections within the kernel?

4.0 Self-Assessment/Exercises

1. **Name three ways to switch between user mode and kernel mode in a general-purpose operating system.**

Answer

The three ways to switch between user-mode and kernel-mode in a general-purpose operating system are in response to a system call, an interrupt, or a signal. A system call occurs when a user program in user-space explicitly calls a kernel-defined "function" so the CPU must switch into kernel-mode. An interrupt occurs when an I/O device on a machine raises an interrupt to notify the CPU of an event. In this case kernel-mode is necessary to allow the OS to handle the interrupt. Finally, a signal occurs when one process wants to notify another process that some event has happened, such as that a segmentation fault has occurred or to kill a child process. When this happens the OS executes the default signal handler for this type of signal.

5.0 Conclusion

Technology trends suggest that concurrent programming will only increase in importance over time. After several decades in which computer architects were able to make individual processor cores run faster and faster, we have reached a point where the performance of individual cores is leveling off and where further speedups will have to come from parallel processing.

Experiments have shown that switching between KLTs of a process is over 10 times faster than switching between processes, and switching between ULTs is 100 times faster than between processes. Kernel-level provides better parallelism and speed up in multiprocessor system.

6.0 Summary

In these operating systems, a process consists of an address space and one or more threads of control. Each thread of a process has its own program counter, its own register states, and its own stack. But all the threads of a process share the same address space. Hence, they also share the same global variables. In addition, all threads of a process also share the same set of operating system resources, such as open files, child processes, semaphores, signals, accounting information, and so on.

7.0 Further Studies/References

Abraham, S., Peter, B. G., & Gagne, G. (2009). *Operating System Concepts* (C. Weisman (ed.); 8th ed.). John Wiley & Sons Inc.

Andrew S, T., & Bos, H. (2015). *Modern Operating Systems* (H. Marcia & J. Tracy (eds.); Fourth). Pearson Education.

Dhananjay, M. D. (2009). *Operating Systems A Concept-Based Approach* (B. Melinda (ed.)). McGraw-Hill Higher Education.

Thomas, A., & Dahlin, M. (2015). *Operating Systems Principles & Practice Volume III : Memory Management* (S. Kaplan & S. Whitney (eds.); Second). Recursive Books, Ltd.

CHAPTER 5 THREADS & MULTITHREADING 1. Single and Multithreaded Processes. (n.d.). Retrieved May 1, 2022, from https://slideplayer.com/slide/4402210/

Silberschatz, A., Gagne, G., & Galvin, P. (n.d.). *Operating Systems Concepts: Chapter 4 - Threads* (Ninth). Retrieved May 1, 2022, from https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

Threads in Operating System - javatpoint. (n.d.). Retrieved May 1, 2022, from https://www.javatpoint.com/threads-in-operating-system

Unit 3 Threads Data Structure and Lifecycle

Contents

1.0 Introduction

A process is divided into several light-weight processes, each light-weight process is said to be a **thread.** The thread has a program counter that keeps track of which instruction to execute next if the process registers hold their current working variables. It has a stack which contains the executing thread history. The data structure of thread is fully represented by the Thread Control Block (TCB) while the life cycle of thread is **Initial state**, **Ready state**, **Running state**, **Blocked State**, **Sleep**, **Dead**. All these will be discussed extensively in this unit.

2.0 Intended Learning Outcomes (ILOs)

At the end of this unit, students will be able to

1. Demonstrate the Thread Control Block diagram
2. Explain the thread life cycle
3. Show thread states and transition

3.0 Main Content

3.1 Thread Control Block

Thread Control Block (**TCB**) is a data structure in the operating system kernel which contains thread-specific information needed to manage it. The TCB is "the manifestation of a thread in an operating system". The data structure of thread contains information such as:Thread ID, Stack pointer, Program counter: CPU information, Thread priority and the Pointers

Thread ID

Thread state
CPU information: Program counter Register contents
Thread priority
Pointer to process that created this thread
Pointer(s) to other thread(s) that were created by this thread

Figure 2.8: Thread Control Block

- **Thread ID:** It is a unique identifier assigned by the Operating System to the thread when it is being created.
- **Thread states:** These are the states of the thread which changes as the thread progresses through the system
- **CPU information:** It includes everything that the OS needs to know about, such as how far the thread has progressed and what data is being used.
- **Thread Priority:** It indicates the weight (or priority) of the thread over other threads which helps the thread scheduler to determine which thread should be selected next from the READY queue.
- A **pointer** which points to the process which triggered the creation of this thread.
- A **pointer** which points to the thread(s) created by this thread.

3.2 Thread States

As with processes, the key states for a thread are Running, Ready, and Blocked. Generally, it does not make sense to associate suspend states with threads because such states are process-level concepts. There are four basic thread operations associated with a change in thread state:

• **Spawn:** Typically, when a new process is spawned, a thread for that process is also spawned. Subsequently, a thread within a process may spawn another thread within the same process, providing an instruction pointer and arguments for the new thread. The new thread is provided with its own register context and stack space and placed on the ready queue.

• **Block:** When a thread needs to wait for an event, it will block (saving its user registers, program counter, and stack pointers). The processor may now turn to the execution of another ready thread in the same or a different process.

• **Unblock:** When the event for which a thread is blocked occurs, the thread is moved to the Ready queue.

• **Finish**: When a thread completes, its register context and stacks are deallocated.

3.3 Thread lifecycle

It is useful to consider the progression of states as a thread goes from being created, to being scheduled and de-scheduled onto and off of a processor, and then to exiting. Figure 2.9 shows the states of a thread during its lifetime

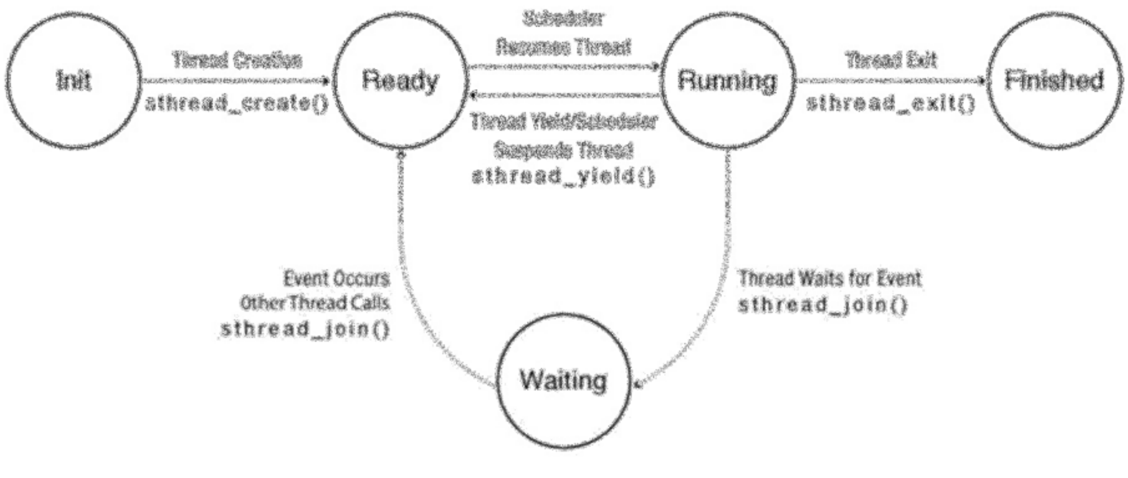


Figure 2.9: The states of thread during lifetime

INIT: Thread creation puts a thread into its INIT state and allocates and initializes perthread data structures. Once that is done, thread creation code puts the thread into the READY state by adding the thread to the ready list. The ready list is the set of runnable threads that are waiting their turn to use a processor. In practice, the ready list is not in fact a "list"; the operating system typically uses a more sophisticated data structure to keep track of runnable threads, such as a priority queue.

READY: A thread in the READY state is available to be run but is not currently running. Its TCB is on the ready list, and the values of its registers are stored in its TCB. At any time, the scheduler can cause a thread to transition from READY to RUNNING by copying its register values from its TCB to a processor's registers.

RUNNING: A thread in the RUNNING state is running on a processor. At this time, its register values are stored on the processor rather than in the TCB. A RUNNING thread can transition to the READY state in two ways: The scheduler can preempt a running thread and move it to the READY state by:

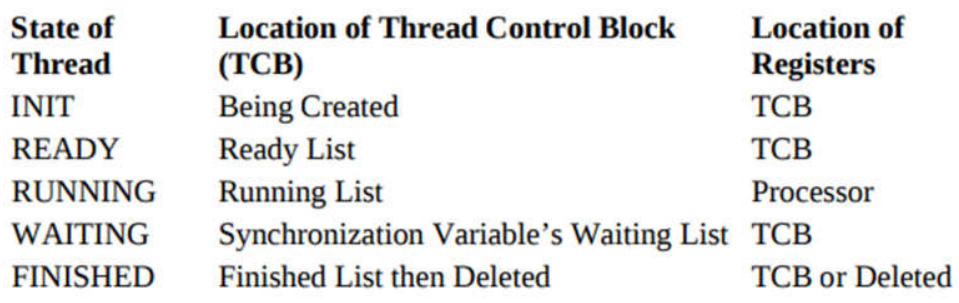
1. saving the thread's registers to its TCB and
2. switching the processor to run the next thread on the ready list.

A running thread can voluntarily relinquish the processor and go from RUNNING to READY by calling yield (e.g., thread_yield in the thread library).

Notice that a thread can transition from READY to RUNNING and back many times. Since the operating system saves and restores the thread's registers exactly, only the speed of the thread's execution is affected by these transitions.

WAITING: A thread in the WAITING state is waiting for some event. Whereas the scheduler can move a thread in the READY state to the RUNNING state, a thread in the WAITING state cannot run until some action by another thread moves it from WAITING to READY.

FINISHED: A thread in the FINISHED state never runs again. The system can free some or all of its state for other uses, though it may keep some remnants of the thread in the FINISHED state for a time by putting the TCB on a finished list. For example, the thread_exit call lets a thread pass its exit value to its parent thread via thread_join. Eventually, when a thread's state is no longer needed (e.g., after its exit value has been read by the join call), the system can delete and reclaim the thread's state.



State of Thread	**Location of Thread Control Block (TCB)**	**Location of Registers**
INIT	Being Created	TCB
READY	Ready List	TCB
RUNNING	Running List	Processor
WAITING	Synchronization Variable's Waiting List	TCB
FINISHED	Finished List then Deleted	TCB or Deleted

Figure 3.0: State of a thread

Discussion

An airline reservation system using a centralized database services user requests concurrently. Is it preferable to use threads rather than processes in this system? Give reasons for your answer.

3.4 Cases/Examples

The **threadHello** program of **4.0 Self-Assessment/Exercise** in **Unit 2** is example of a WAITING thread. After creating its children's threads, the main thread must wait for them to complete, by calling thread_join once for each child. If the specific child thread is not yet done at the time of the join, the main thread goes from RUNNING to WAITING until the child thread exits.

While a thread waits for an event, it cannot make progress; therefore, it is not useful to run it. Rather than continuing to run the thread or storing the TCB on the scheduler's ready list, the TCB is stored on the waiting list of some synchronization variable associated with the event. When the required event occurs, the operating system moves the TCB from the synchronization variable's waiting list to the scheduler's ready list, transitioning the thread from WAITING to READY.

4.0 Self-Assessment/Exercises

1. **What is the primary difference between a kernel-level context switch and (address spaces) and a user-level context switch?**

   Answer

The primary difference is that kernel-level context switches involve execution of OS code. As such it requires crossing the boundary between user- and kernel-level two times. When the kernel is switching between two different address spaces it must store the registers as well as the address space. Saving the address space involves saving pointers to the page tables, segment tables, and whatever other data structures are used by the CPU to describe an address space. When switching between two user-level threads only the user-visible registers need to be saved and the kernel need not be entered. The overhead observed on a kernel-level context switch is much higher than that of a user-level context switch.

2. **Does spawning two user-level threads in the same address space guarantee that the threads will run in parallel on a 2-CPU multiprocessor? If not, why?**

   Answer

No, the two user-level threads may run on top of the same kernel thread. There are, in fact, many reasons why two user-level threads may not run in parallel on a 2-CPU MP. First is that there may be many other processes running on the MP, so there is no other CPU available to execute the threads in parallel. Second is that both threads may be executed on the same CPU because the OS does not provide an efficient load balancer to move either thread to a vacant CPU. Third is that the programmer may limit the CPUs on which each thread may execute.

5.0 Conclusion

A computer user and the operating system have different views of execution of programs. The user is concerned with achieving execution of a program in a sequential or concurrent manner as desired, whereas the OS is concerned with allocation of resources to programs and servicing of several programs simultaneously, so that a suitable combination of efficient use and user service may be obtained. In this chapter, we discussed various aspects of these two views of execution of programs. Execution of a program can be speeded up through either parallelism or concurrency. Parallelism implies that several activities are in progress within the program at the same time. Concurrency is an illusion of parallelism, activities that appear to be parallel, but may not be actually so. We have now understood fully the concepts of threads, multiprogramming, how threads are created and terminated, types of threads and the lifecycle of threads.

6.0 Summary

Although threads are not the only possible solution to these issues, they are a general-purpose technique that can be applied to a wide range of concurrency issues. In our view, multithreaded programming is a skill that every professional programmer must master.

In this unit, we have described that;

TCB is the data structure of a thread containing information like thread ID, Computer information, thread priority and pointer. States of a thread range from Init, ready, running, waiting, block, and finished.

7.0 Further Studies/References

Thomas, A., & Dahlin, M. (2015). *Operating Systems Principles & Practice Volume III : Memory Management* (S. Kaplan & S. Whitney (eds.); Second). Recursive Books, Ltd.

Abraham, S., Peter, B. G., & Gagne, G. (2009). *Operating System Concepts* (C. Weisman (ed.); 8th ed.). John Wiley & Sons Inc.

Andrew S, T., & Bos, H. (2015). *Modern Operating Systems* (H. Marcia & J. Tracy (eds.); Fourth). Pearson Education.

Dhananjay, M. D. (2009). *Operating Systems A Concept-Based Approach* (B. Melinda (ed.)). McGraw-Hill Higher Education.

CHAPTER 5 THREADS & MULTITHREADING 1. Single and Multithreaded Processes. (n.d.). Retrieved May 1, 2022, from https://slideplayer.com/slide/4402210/

Silberschatz, A., Gagne, G., & Galvin, P. (n.d.). *Operating Systems Concepts: Chapter 4 - Threads* (Ninth). Retrieved May 1, 2022, from https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

Threads in Operating System - javatpoint. (n.d.). Retrieved May 1, 2022, from https://www.javatpoint.com/threads-in-operating-system

Module Process Synchronization

Introduction of Module

Interacting processes are concurrent processes that share data or coordinate their activities with respect to one another. **Data access** synchronization ensures that shared data do not lose consistency when they are updated by interacting processes. It is implemented by ensuring that processes access shared data only in a mutually exclusive manner. **Control** synchronization ensures that interacting processes perform their actions in a desired order. Together, these two kinds of synchronization make up what we refer to as **process** synchronization.

In this module, we will introduceprocess synchronization issues and their solutions. Issues such as Race condition in processes, deadlocks; and solutions such as mutual exclusion in critical regions, event handling, semaphores and monitors.

Unit 1: Race condition, Critical Region and Mutual Exclusion

Unit 2: Deadlocks

Unit 3: Synchronization

Unit 4:Synchronization Problems

Unit 1 Race Condition and Critical Region

Contents

1.0 Introduction

The nature of interaction between processes when the *write_set* of one overlaps the **read_set** of another is obvious. The first process may set the value of a variable which the other process may read. The situation when the *write_sets* of two processes overlap is refered to as **interacting processes.** The manner in which the processes perform their write operations can lead to incorrect results, so the processes must cooperate to avoid such situations. Processes that do not interact are said to be independent processes; they can execute freely in parallel.

2.0 Intended Learning Outcomes (ILOs)

At the end of this unit, students will able to

- Define race condition
- Explain how race condition occur
- Resolve race condition

3.0 Main Content

3.1 Race Condition

An application may consist of a set of processes sharing some data say **'ds'.** Data access synchronization involves blocking and activation of these processes such that they can share the data **'ds'** in a correct manner.

The need for data access synchronization arises because accesses to shared data in an arbitrary manner may lead to wrong results in the processes and may also affect consistency of data.

To see this problem, consider processes Pi and Pj that update the value of a shared data item ds through operations ai and aj, respectively

Operation ai : ds := ds + 10;

Operation aj : ds := ds + 5;

Let $(ds)_{initial}$ be the initial value of ds, and let process Pi be the first one to perform its operation. The value of ds after operation ai will be $(ds)_{initial}$ + 10. If process Pj performs operation aj now, the resulting value of ds will be $(ds)_{new}$ = ($(ds)_{initial}$ + 10) + 5, i.e., (ds)initial + 15. If the processes perform their operations in the reverse order, the new value of ds would be identical. If processesPi andPj perform their operations concurrently, we would expect the result to be (ds)initial+15; however, it is not guaranteed to be so. This situation is called a **race condition**

A race condition on a share data ds is a situation in which the value of ds resulting from the execution of two operations may be different, i.e

$$f_i(f_j(d_s)) \neq f_j(f_i(ds))$$

the result of the two operations will be correct, if one of them operates on the value, resulting from the other operation. But will be wrong, if both ai and aj operates on old value of ds. This can happen if one process is engaged in performing the load-add-store sequence but the other process has performed the load instruction before this sequence is completed.

A program containing a race condition may produce correct or incorrect results depending on the order in which instructions of its processes are executed. This feature complicates both testing and debugging of concurrent programs, so race conditions should be prevented.

Data Access Synchronization

Race conditions are prevented if we ensure that operations ai and aj do not execute concurrently, that is, only one of the operations can access shared data ds at any time. This requirement is called **mutual exclusion.** When mutual exclusion is ensured, we can be sure that the result of executing operations ai and aj would be either fi(fj(ds)) or fj(fi(ds)). Data access synchronization is coordination of processes to implement mutual exclusion over shared data. A technique of data access synchronization is used to delay a process that wishes to access ds if another process is currently accessing ds, and to resume its operation when the other process finishes using ds. This will be discussed further in unit 3.

We identify this set of processes by following the definition stated below for each pair of processes. We use the notation **update-set i** for this purpose

Update-seti is a set of data items updated by process Pi , i.e., set of data items whose values are read, modified and written back by process Pi.

Definition of processes containing Race Condition

A race condition exist in processes Pi and Pj of an application if

$$Update\_set_i \cap Update\_set_j \neq \phi$$

The following method can be used to prevent race conditions in an application program:

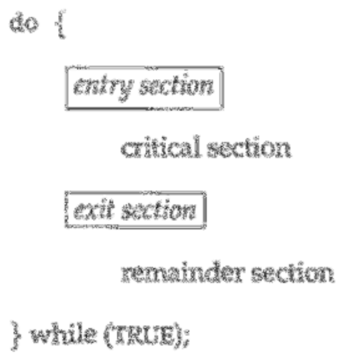
1) For every pair of processes that share some data, check whether a race condition exists.
2) If so, ensure that processes access shared data in a mutually exclusive manner.

3.2 Critical Regions

The key to preventing trouble here and in many other situations involving shared memory, shared files, and shared everything else, is to find some way to prohibit more than one process from reading and writing the shared data at the same time. In other words, what we need is **mutual exclusion,** some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing. A **critical section** for a data item ds is a section of code that is designed so that it cannot be executed concurrently either with itself or with other critical section(s) for ds

If some process Pi is executing a critical section for ds, another process wishing to execute a critical section for ds will have to wait until Pi finishes executing its critical section. Thus, a critical section for a data item ds is a mutual exclusion region with respect to accesses to ds. We mark a critical section in a segment of code by a dashed rectangular box usually given notation as CS

The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the ***entry section***. The critical section may be followed by an ***exit section***. The remaining code is the ***remainder section***.The general structure of a typical process Pi is shown in figure 3.0 below.



```
do {
      entry section
            critical section
      exit section
            remainder section
} while (TRUE);
```

Figure 3.1: Typical process Pi running in critical section

A solution to the critical-section problem must satisfy the following three properties:

1. **Mutual exclusion**. If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress**. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

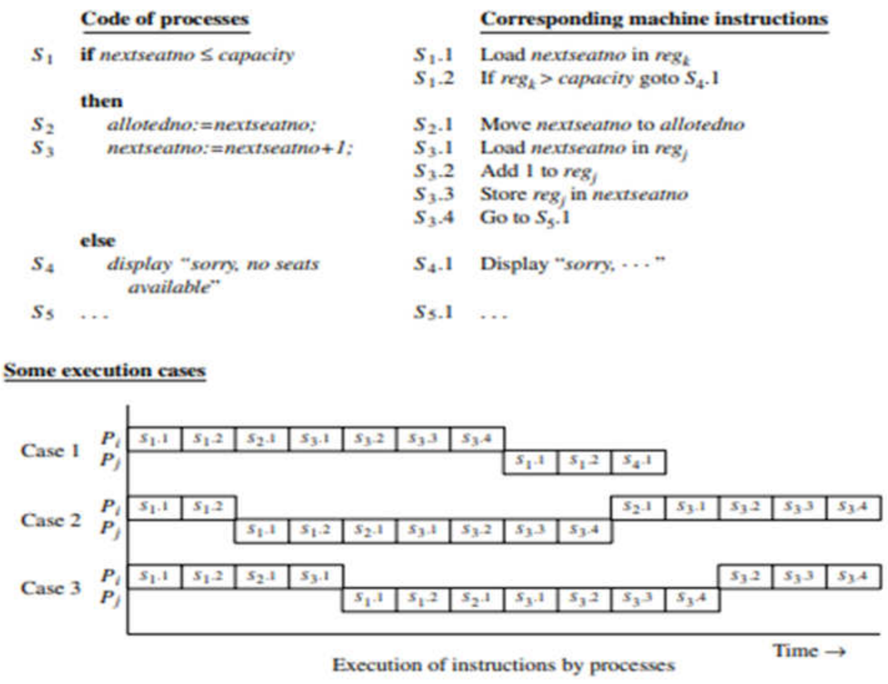
3. **Bounded waiting.** There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Properties of a Critical Section Implementation

Refer to the above three properties. When several processes wish to use critical sections for a data item ds, a critical section implementation must ensure that it grants entry into a critical section in accordance with the notions of correctness and fairness to all processes. There are three essential properties that a critical section implementation must possess to satisfy these requirements. The **mutual exclusion** property guarantees that two or more processes will not be in critical sections for ds simultaneously. It ensures correctness of the implementation. The second and third properties together guarantee that no process wishing to enter a critical section will be delayed indefinitely, i.e., starvation will not occur.The **progress** property ensures that if some processes are interested in entering critical sections for a data item ds, one of them will be granted entry if no process is currently inside any critical section for ds—that is, use of a CS cannot be “reserved” for a process that is not interested in entering a critical section at present. However, this property alone cannot prevent starvation because a process might never gain entry to a CS if the critical section implementation always favors other processes for entry to the CS. The **bounded wait** property ensures that this does not happen by limiting the number of times other processes can gain entry to a critical section ahead of a requesting process Pi. Thus, the progress and bounded wait properties ensure that every requesting process will gain entry to a critical section in finite time; however, these properties do not guarantee a specific limit to the delay in gaining entry to a CS.

3.3 Cases/Examples

Race Condition in an Airline Reservation Application



```
    Code of processes                        Corresponding machine instructions
S1  if nextseatno ≤ capacity           S1.1  Load nextseatno in reg_k
                                       S1.2  If reg_k > capacity goto S4.1
    then
S2     allotedno:=nextseatno;          S2.1  Move nextseatno to allotedno
S3     nextseatno:=nextseatno+1;       S3.1  Load nextseatno in reg_j
                                       S3.2  Add 1 to reg_j
                                       S3.3  Store reg_j in nextseatno
                                       S3.4  Go to S5.1
    else
S4     display "sorry, no seats        S4.1  Display "sorry, . . . "
          available"
S5  . . .                              S5.1  . . .
```

Some execution cases
Case 1
Case 2
Case 3
P_i
P_j
$S_1.1$ $S_1.2$ $S_2.1$ $S_3.1$ $S_3.2$ $S_3.3$ $S_3.4$ $S_4.1$
Execution of instructions by processes
Time →

Figure 3.2: Airline reservation deadlock processes

The left column in the upper half of Figure above shows the code used by processes in an airline reservation application. The processes use identical code, hence ai and aj, the operations performed by processes Pi and Pj, are identical. Each of these operations examines the value of nextseatno and updates it by 1 if a seat is available. The right column of Figure above show the machine instructions corresponding to the code. Statement S3 corresponds to three instructions S3.1, S3.2 and S3.3 that form a load-add-store sequence of instructions for updating the value of nextseatno. The lower half of the Figure is a timing diagram for the applications. It shows three possible sequences in which processesPi andPj could execute their instructions when nextseatno = 200 and capacity = 200. In case 1, process Pi executes the if statement that compares values of nextseatno with capacity and proceeds to execute instructions S2.1, S3.1, S3.2 and S3.3 that allocate a seat and increment nextseatno. When process Pj executes the if statement, it finds that no seats are available so it does not allocate a seat. In case 2, process Pi executes the if statement and finds that a seat can be allocated. However, it gets preempted before it can execute instruction S2.1. Process Pj now executes the if statement and finds that a seat is available. It allocates a seat by executing instructions S2.1, S3.1, S3.2 and S3.3 and exits. nextseatno is now 201. When process Pi is resumed, it proceeds to execute instruction S2.1, which allocates a seat. Thus, seats are allocated to both requests. This is a race condition because when nextseatno = 200, only one seat should be allocated. In case 3, process Pi gets preempted after it loads 200 in regj through instruction S3.1. Now, again both Pi and Pj allocate a seat each, which is a race condition.

Discussion

How could the race condition obtained in the airline reservation system in the above Cases/Example be resolved? Any Idea

4.0 Self-Assessment/Exercises

1. Can a mutual exclusion algorithm be based on assumptions on the relative speed of processes, i.e. that some processes may be "faster" than the others in executing the same section of code?

Answer

No, mutual exclusion algorithms cannot be based on assumptions about the relative speed of processes. There are MANY factors that determine the execution time of a given section of code, all of which would affect the relative speed of processes. A process that is 10x faster through a section of code one time, may be 10x slower the next time.

5.0 Conclusion

The problem of avoiding race conditions can also be formulated in an abstract way. Part of the time, a process is busy doing internal computations and other things that do not lead to race conditions. However, sometimes a process has to access shared memory or files, or do other critical things that can lead to races. That part of the program where the shared memory is accessed is called the critical region or critical section. If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid races.

6.0 Summary

Processes frequently need to communicate with other processes. When a user process wants to read from a file, it must tell the file process what it wants. Then the file process has to tell the disk process to read the required block. We have seen that a race condition, which is referred to as two processes trying to write to the file at the same time, can occur if proper measures are not put in place. Critical regions using the mutual exclusion property have come to the rescue, making sure no two processes access shared memory at the same time by putting one of the processes in a waiting state until CPU resources are completely released by the other process.This had solved a lot of problem in the airline reservation system.

7.0 References/Further Reading

Abraham, S., Peter, B. G., & Gagne, G. (2009). *Operating System Concepts* (C. Weisman (ed.); 8th ed.). John Wiley & Sons Inc.

Andrew S, T., & Bos, H. (2015). *Modern Operating Systems* (H. Marcia & J. Tracy (eds.); Fourth). Pearson Education.

Dhananjay, M. D. (2009). *Operating Systems A Concept-Based Approach* (B. Melinda (ed.)). McGraw-Hill Higher Education.

Thomas, A., & Dahlin, M. (2015). *Operating Systems Principles & Practice Volume III : Memory Management* (S. Kaplan & S. Whitney (eds.); Second). Recursive Books, Ltd.

kartik. (n.d.). *Race Condition in OS*. Retrieved May 1, 2022, from https://practice.geeksforgeeks.org/problems/race-condition-in-os

Silberschatz, A., Galvin, P. B., & Gagne, G. (2012). Operating System Concepts. In *Information and Software Technology* (Vol. 32, Issue 8). Wiley. http://linkinghub.elsevier.com/retrieve/pii/095058499090158N%0Ahttp://os-book.com/%0Ahttps://drive.google.com/open?id=1mMQ_sLWoW7mDHEQzTjBtQgvU_kAR-PcQ

What Is A Critical Region In Computer Science? – TheSassWay.com. (n.d.). Retrieved May 1, 2022, from https://thesassway.com/what-is-a-critical-region-in-computer-science/

Unit 2 Deadlock

Contents

1.0 Introduction

In real life, a deadlock arises when two persons wait for phone calls from one another, or when persons walking a narrow staircase in opposite directions meet face to face. A deadlock is characterized by the fact that persons wait indefinitely for one another to perform specific actions; these actions cannot occur.

Deadlocks arise in process synchronization when processes wait for each other's signals, or in resource sharing when they wait for other processes to release resources that they need. Deadlocked processes remain blocked indefinitely, which adversely affects user service, throughput and resource efficiency.

The objectives of this unit are

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks.
- To present a number of different methods for preventing or avoiding deadlocks in a computer system.

2.0 Intended Learning Outcomes (ILOs)

At the end of this unit, the student will be able to

- Describe the concept of deadlock
- Write a program code that avoid deadlock

3.0 Main Content

3.1 Deadlock Concept

Deadlock is the state of permanent blocking of a set of processes each of which is waiting for an event that only another process in the set can cause. All the processes in the set block permanently because all the processes are waiting and hence none of them will ever cause any of the events that could wake up any of the other members of the set.

A deadlock situation can be best explained with the help of an example. Suppose that a system has two tape drives TI and T2 and the resource allocation strategy is such that a requested resource is immediately allocated to the requester if the resource is free. Also suppose that two concurrent processes PI and P2 make requests for the tape drives in the following order:

1. P1 requests for one tape drive and the system allocates T1 to it.
2. P2 requests for one tape drive and the system allocates T2 to it.
3. P1 requests for one more tape drive and enters a waiting state because no tape drive is presently available.
4. P2 requests for one more tape drive and it also enters a waiting state because no tape drive is presently available.

From now on, P1 and P2 will wait for each other indefinitely, since PI will not release T1 until it gets T2 to carry out its designated task, that is, not until P2 has released T2, whereas P2 will not release T2 until it gets T1. Therefore, the two processes are in a state of deadlock. Note that the requests made by the two processes are totally legal because each is requesting for only two tape drives, which is the total number of tapes drives available in the system. However, the deadlock problem occurs because the total requests of both processes exceed the total number of units for the tape drive and the resource allocation policy is such that it immediately allocates a resource on request if the resource is free.

The deadlock illustrated above is called a resource deadlock. Other kinds of deadlock can also arise in an OS. A synchronization deadlock occurs when the awaited events take the form of signals between processes. For example, if a process Pi decides to perform an action ai only after process Pj performs action aj, and process Pj decides to perform action aj only after Pi performs ai, both processes get blocked until the other process sends it a signal. An OS is primarily concerned with resource deadlocks because allocation of resources is an OS responsibility. The other two forms of deadlock are seldom handled by an OS; it expects user processes to handle such deadlocks themselves.

Conditions for a Resource Deadlock

This condition is essential because waits may not be indefinite if a blocked process is permitted to withdraw a resource request and continue its operation. However, it is not stated explicitly in the literature, because many operating systems typically impose the no-withdrawal condition on resource requests.

1. Non-shareable resources cannot be shared; a process needs exclusive access to a resource.
2. No Preemption: A resource cannot be preempted from one process and allocated to another process.
3. A Hold-and-wait process continues to hold the resources allocated to it while waiting for other resources.
4. A circular wait chain of hold-and-wait conditions exists in the system; e.g., process Pi waits for Pj, Pj waits for Pk, and Pk waits for Pi.

Table 1: Conditions for Resource Deadlock

Condition	**Explanation**
Non-shareable resources	Resources cannot be shared; a process needs exclusive access to a resource
No preemption	A resource cannot be preempted from one process and allocated to another process.
Hold-and-wait	A process continues to hold the resources allocated to it while waiting for other resources.
Circular waits	A circular chain of hold-and-wait conditions exists in the system; e.g., process Pi waits for Pj, Pj waits for Pk, and Pk waits for Pi.

3.2 Deadlock Detection

In this approach for deadlock handling, the system does not make any attempt to prevent deadlocks and allows processes to request resources and to wait for each other in an uncontrolled manner. Rather, it uses an algorithm that keeps examining the state of the system to determine whether a deadlock has occurred. When a deadlock is detected, the system takes some action to recover from the deadlock. If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred.
- An algorithm to recover from the deadlock

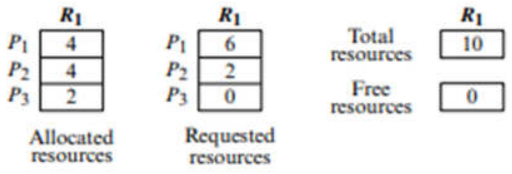
It should be noted that a detection-and-recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm but also the potential losses inherent in recovering from a deadlock.

A process in the blocked state is not involved in a deadlock at the current moment if the request on which it is blocked can be satisfied through a sequence of process completion, resource release, and resource allocation events. If each resource class in the system contains a single resource unit, this check can be made by checking for the presence of a cycle in an Resource Request Allocation Graph (RRAG) or Wait For Graph(WFG). However, more complex graph-based algorithms have to be used if resource classes may contain more than one resource, so we instead discuss a deadlock detection approach using the matrix model. We check for the presence of a deadlock in a system by actually trying to construct fictitious but feasible sequences of events whereby all blocked processes can get the resources they have requested. Success in constructing such a sequence implies the absence of a deadlock at the current moment, and a failure to construct it implies presence of a deadlock.

We perform the above check by simulating the operation of a system starting with its current state. We refer to any process that is not blocked on a resource request as a running process, i.e., we do not differentiate between the ready and running states. In the simulation we consider only two events— completion of a process that is not blocked on a resource request, and allocation of resource(s) to a process that is blocked on a resource request. It is assumed that a running process would complete without making additional resource requests, and that some of the resources freed on its completion would be allocated to a blocked process only if the allocation would put that process in the running state. The simulation ends when all running processes complete. The processes that are in the blocked state at the end of the simulation are those that could not obtain the requested resources when other processes were completed, hence these processes are deadlocked in the current state. There is no deadlock in the current state if no blocked processes exist when the simulation ends.

Consider the following

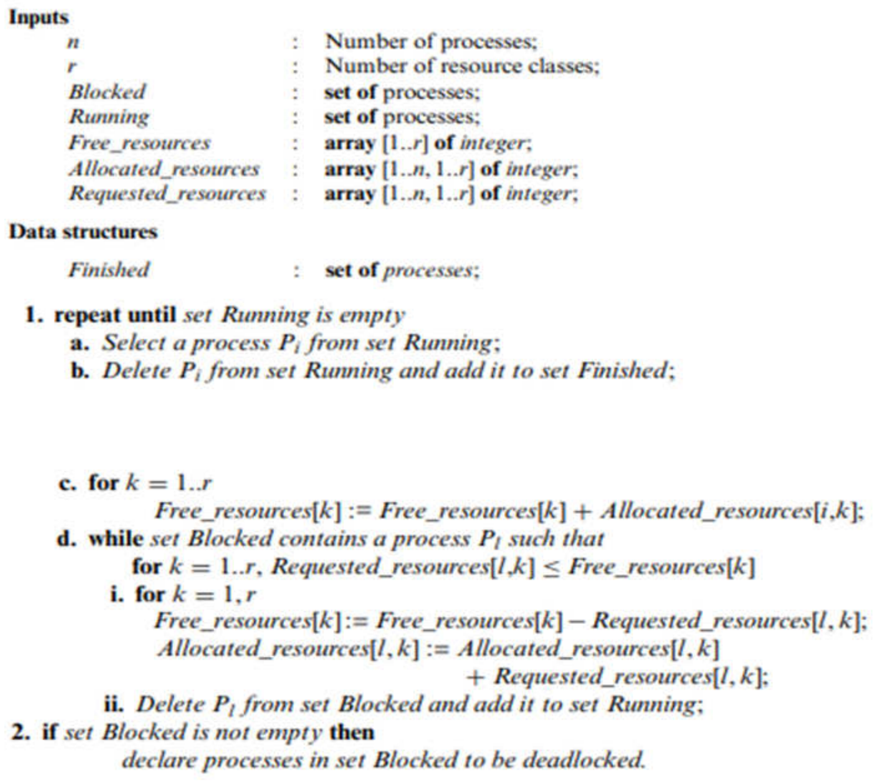
The allocation state of a system containing 10 units of a resource class R1 and three processes P1–P3 is as follows:



Process P3 is in the running state because it is not blocked on a resource request. All processes in the system can complete as follows: Process P3 completes and releases 2 units of the resource allocated to it. These units can be allocated to P2. When it completes, 6 units of the resource can be allocated to P1. Thus, no blocked processes exist when the simulation ends, so a deadlock does not exist in the system. If the requests by processes P1 and P2 were for 6 and 3 units, respectively, none of them could complete even after process P3 released 2 resource units. These processes would be in the blocked state when the simulation ended, and so they are deadlocked in the current state of the system.

Deadlock Detection Algorithm

Algorithm below performs deadlock detection. The inputs to the algorithm are two sets of processes Blocked and Running, and a matrix model of the allocation state comprising the matrices Allocated_resources, Requested_resources, and Free_resources. The algorithm simulates completion of a running process Pi by transferring it from the set Running to the set Finished [Steps 1(a), 1(b)]. Resources allocated to Pi are added to Free_resources [Step 1(c)]. The algorithm now selects a blocked process whose resource request can be satisfied from the free resources [Step 1(d)], and transfers it from the set Blocked to the set Running. Sometime later the algorithm simulates its completion and transfers it from Running to Finished. The algorithm terminates when no processes are left in the Running set. Processes remaining in the set Blocked, if any, are deadlocked. The complexity of the algorithm can be analyzed as follows: The sets Running and Blocked can contain up to n processes, where n is the total number of processes in the system. The loop of Step 1 iterates ≤ n times and Step 1(d) performs an order of n × r work in each iteration. Hence the algorithm requires an order of n2 × r work.



```
Inputs
    n                     :  Number of processes;
    r                     :  Number of resource classes;
    Blocked               :  set of processes;
    Running               :  set of processes;
    Free_resources        :  array [1..r] of integer;
    Allocated_resources   :  array [1..n, 1..r] of integer;
    Requested_resources   :  array [1..n, 1..r] of integer;
Data structures
    Finished              :  set of processes;

 1. repeat until set Running is empty
     a. Select a process $P_i$ from set Running;
     b. Delete $P_i$ from set Running and add it to set Finished;
     c. for $k = 1..r$
            Free_resources[k] := Free_resources[k] + Allocated_resources[i,k];
     d. while set Blocked contains a process $P_l$ such that
          for $k = 1..r$, Requested_resources[l,k] $\leq$ Free_resources[k]
        i. for $k = 1, r$
            Free_resources[k] := Free_resources[k] − Requested_resources[l, k];
            Allocated_resources[l, k] := Allocated_resources[l, k]
                                         + Requested_resources[l, k];
        ii. Delete $P_l$ from set Blocked and add it to set Running;
 2. if set Blocked is not empty then
        declare processes in set Blocked to be deadlocked.
```

Figure 3.3: Deadlock detection algorithm

3.3 Deadlock Avoidance

In deadlock avoidance, for each new request, the system consider the resources currently available, the resources currently allocated to the processes and further request and releases of each process, to decide whether the current request can be satisfied or must wait to avoid a possible future deadlock.

` The various algorithm differ in the amount and type of information required. The simplest and most useful model requires that each process declare the maximum number of resource classes (resource types) and their resource units (instances) that it may need in the system.

Safe State

A state is safe, if the system can allocate resources to each process (up to its maximum) in some order and avoid deadlock. Formally, a system is said to be in safe state only if there exist a safe sequence. A sequence of processes < P1, P2, ..., Pn> is a safe sequence for the current allocation state iff for each process Pi, the resources that Pi can still request can be satisfied by the currently available resources plus the resources held by all the processes Pj with Pj< Pi.

Given a prior information about the maximum number of resources of each type that may be requested for each process, it is possible to construct an algorithm that ensures that the system will never enter a deadlock state. This approach is known as Deadlock Avoidance.

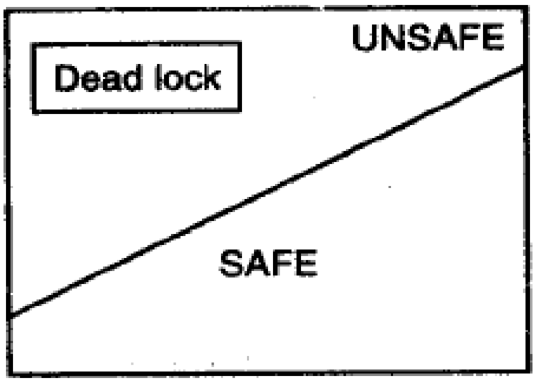


Figure 3.4: Safe,unsafe and deadlock of state

In this situation, if the resources that process Pi needs are not immediately available then Pi can wait until all Pj have finished. A safe state is not a deadlock state. Conversely, a deadlock state is an unsafe state. However, not all unsafe states are deadlock. An unsafe state may lead to a deadlock state. As long as the state is safe, the O/S can avoid deadlocks while in unsafe state the O/S cannot prevent processes from requesting resources such that a deadlock occurs.

Banker's Algorithm

The resource allocation graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. The deadlock avoidance scheme that we describe next is applicable to such a system and is commonly known as Bankers' Algorithm.

The name was chosen because this algorithm could be used in a banking system to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all its customers. When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need.

This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in safe state. If it will, the resources are allocated, otherwise, the process must wait until some other process releases enough resources.

Several data structures are required to implement the Banker's Algorithm. These data structures encode the state of the resource-allocation system. Let 'n' be the number of processes in the system and 'm' be the number of resource types. The following explain the data structures:

Available

A vector of length 'm' indicates the number of available resources of each type. If available [j] = k, then there are 'k' instances of resource type Rj is available. It is a 1-dimensional array of length 'm'.

Max

It is a 2-dimensional array (matrix) of length n x m that defines the maximum demand of each process in a system. If Max [i, j] = k, then process Pi may request at most 'k' instances of resource type Rj.

Allocation

It is also a 2-d array of length n x m that defines the number of resources of each type currently allocated to each process. If allocation [i, j] = k, then process Pi is currently allocated 'k' instances of resource type Rj.

Need

It is also a 2-d array of length n x m that indicates the remaining resource need of each process. If need [i, j] = k, then process Pi may need 'k' more instances of resource type to complete its task.

Note that: need [i, j] = Max [i, j] — Allocation [i, j].

These data structures vary over time in both size and value. We can treat each row in the matrices allocation and need as vectors and refer them **Allocation-i** and **Need-i** respectively. **Allocation-i** specifies the resources currently allocated to process Pi and vector **Need-i** specifies the additional resources that process Pi may still request to complete its task.

3.4 Deadlock Prevention

This approach is based on the idea of designing the system in such a way that deadlocks become impossible. It differs from avoidance and detection in that no runtime testing of potential allocations need be performed.It has been seen that mutual-exclusion, hold-and-wait, no-preemption, and circular-wait are the four necessary conditions for a deadlock to occur in a system. Therefore ensuring that none of these conditions is never satisfied, deadlocks will be impossible.

Mutual Exclusion - The mutual-exclusion condition must hold for non-sharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

Hold and Wait - To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. One protocol that can be used requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls. An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.

No Preemption - The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Circular Wait - A circular wait can result from the hold-and-wait condition, which is a consequence of the non-shareability and non-preemptibility conditions, so it does not arise if either of these conditions does not arise. Circular waits can be separately prevented by not allowing some processes to wait for some resource. It can be achieved by applying a validity constraint to each resource request. The validity constraint is a Boolean function of the allocation state. It takes the value false if the request may lead to a circular wait in the system, so such a request is rejected right away. If the validity constraint has the value true, the resource is allocated if it is available; otherwise, the process is blocked for the resource.

Deadlock Prevention Policy

All Resources Together

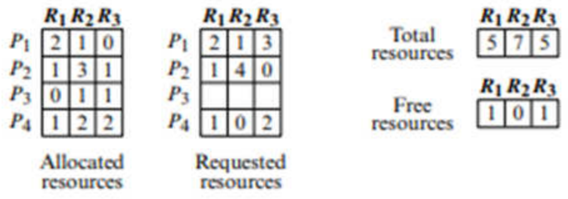
This is the simplest of all deadlock prevention policies. A process must ask for all resources it needs in a single request; the kernel allocates all of them together. This way a blocked process does not hold any resources, so the hold-and-wait condition is never satisfied. Consequently, circular waits and deadlocks cannot arise. Under this policy, both processes for instance must request a tape drive and a printer together. Now a process will either hold both resources or hold none of them, and the hold-and-wait condition will not be satisfied

Simplicity of implementation makes “all resources together” an attractive policy for small operating systems. However, it has one practical drawback; it adversely influences resource efficiency. For example, if a process Pi requires a tape drive at the start of its execution and a printer only toward the end of its execution, it will be forced to request both a tape drive and a printer at the start. The printer will remain idle until the latter part of Pi’s execution and any process requiring a printer will be delayed until Pi completes its execution. This situation also reduces the effective degree of multiprogramming and, therefore, reduces CPU efficiency

3.5 Cases/Examples

Deadlock operation detection

A system has four processes P1–P4, and 5, 7, and 5 units of resource classes R1, R2, and R3, respectively. It is in the following state just before process P3 makes a request for 1 unit of resource class R1:



Allocated resources

	R_1	R_2	R_3
P_1	2	1	0
P_2	1	3	1
P_3	0	1	1
P_4	1	2	2

Requested resources

	R_1	R_2	R_3
P_1	2	1	3
P_2	1	4	0
P_3			
P_4	1	0	2

	R_1	R_2	R_3
Total resources	5	7	5
Free resources	1	0	1

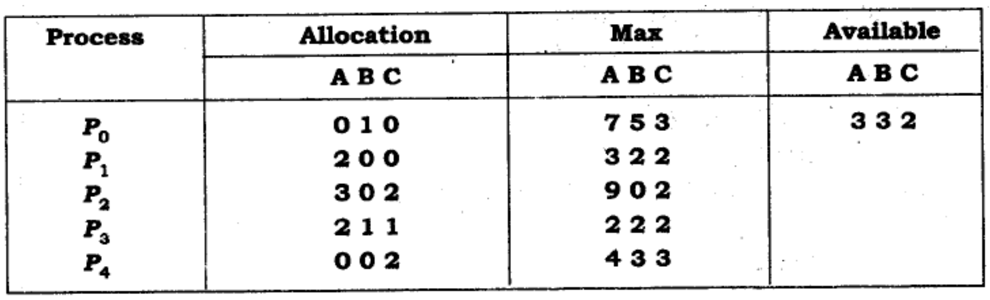
One resource unit of resource class R1 is allocated to process P3 and figure 3.2 is invoked to check whether the system is in a deadlock. The figure above shows steps in operation of the algorithm. Inputs to it are the sets Blocked and Running initialized to {P1, P2, P4} and {P3}, respectively, and matrices Allocated_resources, Requested_resources, and Free_resources. The algorithm transfers process P3 to the set Finished and frees the resources allocated to it. The number of free units of the resource classes is now 1, 1 and 2, respectively. The algorithm finds that process P4’s pending request can now be satisfied, so it allocates the resources requested by P4 and transfers P4 to the set Running. Since P4 is the only process in Running, it is transferred to the set Finished. After freeing P4’s resources, the algorithm finds that P1’s resource request can be satisfied and, after P1 completes, P2's resource request can be satisfied. The set Running is now empty so the algorithm completes. A deadlock does not exist in the system because the set Blocked is empty

Discussion

1. Discuss the resource ranking policy in deadlock prevention
2. Discuss how deadlock can be avoid using the resource-request algorithm

4.0 Self-Assessment/Exercises

Considering a system with five processes P0 through P4 and three resources types A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t0 following snapshot of the system has been taken:



Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

(i) What will be the content of need matrix?

(ii) Is the system in safe state? If yes, then what is the safe sequence?

(iii) What will happen if process P1 requests one additional instance of resource type A and two instances of resource type C?

(iv) If a request [3, 3, 0] by process P4 arrives in the state defined by (iii), can it be granted immediately?

(v) If resource request [0, 2, 0] by process Po arrives then check whether it is granted or not?

Solution:

Need

	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

(i) As we show that, Need [i, j] = Max [i,j] — Allocation [i, j]

So, the content of Need matrix is :

(ii) Applying safety algorithm on the given system.

For Pi , if Need-i ≤ Available, then Pi is in safe sequence.

Available = Available + Allocation-i

So, for P0, Need0 = [7, 4, 3]

(i = 0)

Available = [3, 3, 2]

⇒ Condition is false, So P0 must wait.

for P1,(i=1)

Need-1 = [1,2,2]

Available = [3, 3, 2]

Need-1 < Available

So, P1 will be kept in safe sequence.

Now, Available will be updated as:

Available = Available + Allocation i

Available = [3,3,2] + [2,0,0] = [5,3,2]

for P2(i = 2),

Need, = [6, 0, 0]

Available = [5, 3, 2]

=> Condition is again false so P2 must wait.

for P3(i = 3),

Need3 = [0, 1, 1]

Available = [5, 3, 2]

$\Rightarrow$ Condition is true; P3 will be in safe sequence

Available = [5,3,2] + [2, 1, 1] = [7,4,3]

for P4(i = 4),

Need, = [4, 3, 1]

Available = [7, 4, 3]

=> Condition Need4 < Available is true

So,P4 is in safe sequence and Available = [7, 4,3] + [0,0,2] = [7, 4, 5].

Now, we have two process P0 and P2 in waiting state.

With current available either P0 or P2 can move to safe sequence.

Firstly, we take P2 whose Need2 = [6, 0, 0]

Available = [7, 4, 5]

Need2 < Available

So, P2 now comes in safe state leaving the Available = [7, 4, 5] + [3,0,2] = [10,4, 7]

Next, take P0 whose Need = [7, 4, 3]

Available = [10, 4, 7]

Need0 < Available

So, P0 now comes in safe state

Available = [10,4, 7] + [0,1,0] = [10,5, 7]

So the safe sequence is < P1, P3, P4, P2, P0 > and the system is in safe **state.**

(iii) Since P1 requests some additional instances of resources such that:

Request-1 = [1, O, 2]

To decide that whether this request is immediately granted we first check that

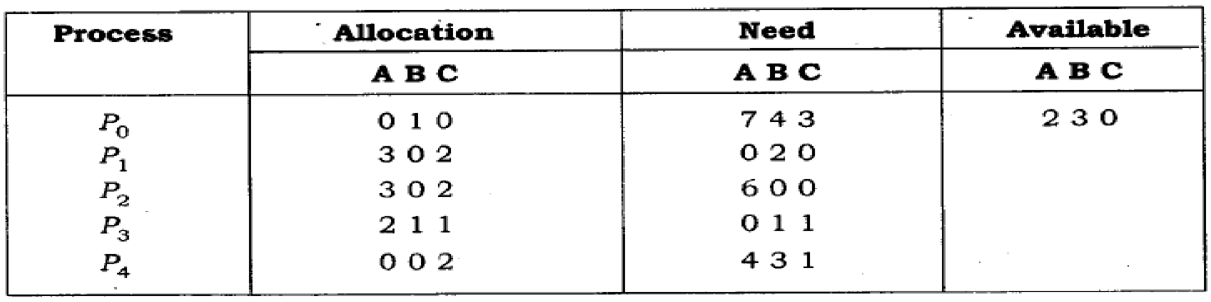
Request1 < Available

i.e., [1, 0, 2] $\leq$ [3, 3, 2] which holds true.

So, the request may be granted.

To confirm that this request is granted we check the new state by applying safety algorithm that our system is in safe state or not. If the new state is in safe state then only this request is granted otherwise not.

To define the new state of the system because of the arrival of request of P1 we follow the Resource-Request algorithm which results as:



Process	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we again execute our safety algorithm and find the safe sequence as < P1, P3, P4, Po, P2 > which satisfies our safety requirements. Hence, we can immediately grant the request for process P1.

(iv) The request for [3, 3, 0] by P4 cannot be granted because -

Request4 = [3, 3, 0]

Available = [2, 3, 0]

In this situation the condition

Request4 < Available is false

So, it is not granted since resources are not available.

(v) The request for [0, 2, 0] by P0

request = [0, 2, 0] and Available = [2, 3,0]

Condition Request0 < Available is true

So, it may be granted. If it is granted then the new state of the system is defined as

Available = Available — Request0

= [2, 3, 0] — [0,2,0]

= [2, 1, 0]

Allocation-0 = Allocation‘+ Request0

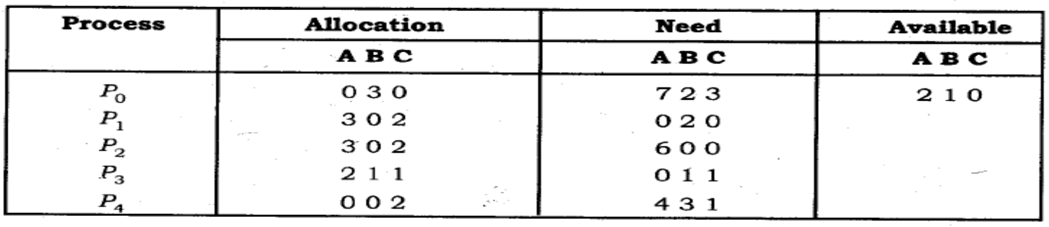
=[0,1,0] + [0,2,0]

= [0, 3, 0]

Need0 = Need0 — Request0

= [7, 4, 3] — [0, 2, 0]

= [7, 2, 3]



Process	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 3 0	7 2 3	2 1 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Applying safety algorithm on this new state—

All five processes are in waiting state as none is able to satisfy the condition,

Need-i $\leq$ Available

So the state represents an unsafe state. Thus, request for P0 is not granted though the resources are available, but the resulting state is unsafe.

5.0 Conclusion

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two.

A resource deadlock arises when four conditions hold simultaneously: Resources are nonshareable and nonpreemptible, a process holds some resources while it waits for resources that are in use by other processes, which is called the hold-and-wait condition; and circular waits exist among processes. An OS can discover a deadlock by analyzing the allocation state of a system, which consists of information concerning allocated resources and resource requests on which processes are blocked.

6.0 Summary

Deadlocks arise in resource sharing when a set of conditions concerning resource requests and resource allocations hold simultaneously. Operating systems use several approaches to handle deadlocks. In the deadlock detection and resolution approach, the kernel checks whether the conditions contributing to a deadlock hold simultaneously, and eliminates a deadlock by judiciously aborting some processes so that the remaining processes are no longer in a deadlock. In the deadlock prevention approach, the kernel employs resource allocation policies that ensure that the conditions for deadlocks do not hold simultaneously; it makes deadlocks impossible. In the deadlock avoidance approach, the kernel does not make resource allocations that may lead to deadlocks, so deadlocks do not arise.

7.0 References/Further Reading

Abraham, S., Peter, B. G., & Gagne, G. (2009). *Operating System Concepts* (C. Weisman (ed.); 8th ed.). John Wiley & Sons Inc.

Andrew S, T., & Bos, H. (2015). *Modern Operating Systems* (H. Marcia & J. Tracy (eds.); Fourth). Pearson Education.

Dhananjay, M. D. (2009). *Operating Systems A Concept-Based Approach* (B. Melinda (ed.)). McGraw-Hill Higher Education.

Thomas, A., & Dahlin, M. (2015). *Operating Systems Principles & Practice Volume III : Memory Management* (S. Kaplan & S. Whitney (eds.); Second). Recursive Books, Ltd.

Deadlock Detection and Prevention - Engineering LibreTexts. (2021, March 22). https://eng.libretexts.org/Courses/Delta_College/Operating_System%3A_The_Basics/06%3A_Deadlock/6.2%3A_Deadlock_Detection_and_Prevention

Deadlock Detection Algorithm in Operating System - GeeksforGeeks. (n.d.). Retrieved May 1, 2022, from https://www.geeksforgeeks.org/deadlock-detection-algorithm-in-operating-system/

Deadlock Prevention in Operating System (OS) - Scaler Topics. (2022, February 16). https://www.scaler.com/topics/operating-system/deadlock-prevention-in-operating-system/

OS Deadlock Prevention - javatpoint. (n.d.). Retrieved May 1, 2022, from https://www.javatpoint.com/os-deadlock-prevention

Silberschatz, A., Gagne, G., & Galvin, P. B. (n.d.). Operating Systems Concepts: Deadlocks: Vol. Chapter 7 (Ninth). Retrieved May 1, 2022, from https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/7_Deadlocks.html

Unit 3 Synchronization

Contents

1.0 Introduction

Synchronization primitives were developed to overcome the limitations of algorithmic implementations. The primitives are simple operations that can be used to implement both mutual exclusion and control synchronization. A semaphore is a special kind of synchronization data that can be used only through specific synchronization primitives.

2.0 Intended Learning Outcomes (ILOs)

At the end of this unit, students will be able to

Explain deadlock and starvation resolution using

- Semaphores
- Monitors

3.0 Main Content

3.1 Semaphores

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

When a process performs a wait operation on a semaphore, the operation checks whether the value of the semaphore is > 0. If so, it decrements the value of the semaphore and lets the process continue its execution; otherwise, it blocks the process on the semaphore. A signal operation on a semaphore activates a process blocked on the semaphore, if any, or increments the value of the semaphore by 1. Due to these semantics, semaphores are also called counting semaphores. Indivisibility of the wait and signal operations is ensured by the programming language or the operating system that implements it. It ensures that race conditions cannot arise over a semaphore. Processes use wait and signal operations to synchronize their execution with respect to one another.

The initial value of a semaphore determines how many processes can get past the wait operation. A process that does not get past a wait operation is blocked on the semaphore.

The below program describe both wait and signal operations

Wait Operation

```
wait (S)
    {
        while (S <= 0);
        S--,
    }
```

Signal Operation

```
    Signal (S)
    {
        S++;
    }
```

Counting semaphores can be used to control access to a given resource consisting of a finite number instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0. We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: P1 with a statement 51 and P2 with a statement 52. Suppose we require that 52 be executed only after 51 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0, and by inserting the statements.

Types of Semaphores

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows −

- **Counting Semaphores**

  These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count is automatically incremented and if the resources are removed, the count is decremented.

- **Binary Semaphores**

  The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores

Advantages of Semaphores

Some of the advantages of semaphores are as follows −

- Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
- Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

Disadvantages of Semaphores

Some of the disadvantages of semaphores are as follows −

- Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.

- Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
- Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

3.2 Monitors

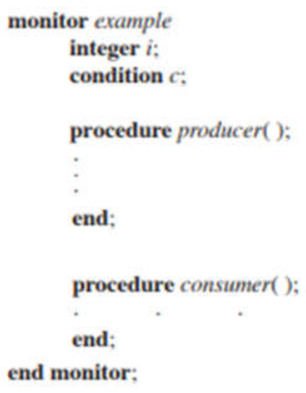
A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package. Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.

Monitors have an important property that makes them useful for achieving mutual exclusion: only one process can be active in a monitor at any instant. Monitors are a programming-language construct, so the compiler knows they are special and can handle calls to monitor procedures differently from other procedure calls. Typically, when a process calls a monitor procedure, the first few instructions of the procedure will check to see if any other process is currently active within the monitor. If so, the calling process will be suspended until the other process has left the monitor. If no other process is using the monitor, the calling process may enter. It is up to the compiler to implement mutual exclusion on monitor entries, but a common way is to use a mutex or a binary semaphore. Because the compiler, not the programmer, is arranging for the mutual exclusion, it is much less likely that something will go wrong. In any event, the person writing the monitor does not have to be aware of how the compiler arranges for mutual exclusion. It is sufficient to know that by turning all the critical regions into monitor procedures, no two processes will ever execute their critical regions at the same time.

The solution lies in the introduction of condition variables, along with two operations on them, wait and signal. When a monitor procedure discovers that it cannot continue (e.g., the producer finds the buffer full), it does a wait on some condition variable, say, full. This action causes the calling process to block. It also allows another process that had been previously prohibited from entering the monitor to enter now. We saw condition variables and these operations in the context of Pthreads earlier. This other process, for example, the consumer, can wake up its sleeping partner by doing a signal on the condition variable that its partner is waiting on. To avoid having two active processes in the monitor at the same time, we need a rule telling what happens after a signal. Three propose solutions were postulated by Hoare and Brinch Hansen which are:

- letting the newly awakened process run, suspending the other one.
- finessing the problem by requiring that a process doing a signal must exit the monitor immediately

Condition variables are not counters. They do not accumulate signals for later use the way semaphores do. Thus, if a condition variable is signaled with no one waiting on it, the signal is lost forever.

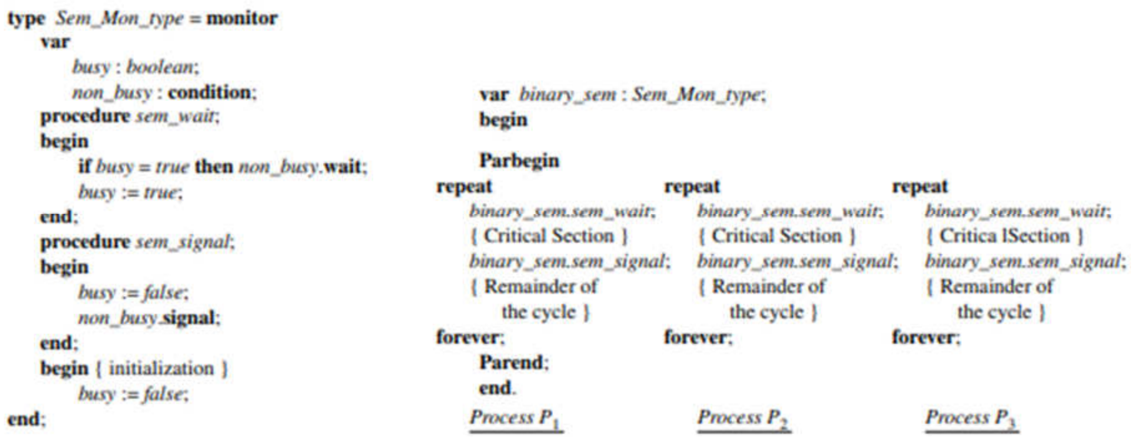


```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .       .       .
        end;
end monitor;
```

Figure 3.5: A monitor

3.3 Cases/Examples

Figure 3.5 shows a monitor type Sem_Mon_type that implements a binary semaphore, and the lower half shows three processes that use a monitor variable binary_sem.



```
type Sem_Mon_type = monitor
    var
        busy : boolean;
        non_busy : condition;
    procedure sem_wait;
    begin
        if busy = true then non_busy.wait;
        busy := true;
    end;
    procedure sem_signal;
    begin
        busy := false;
        non_busy.signal;
    end;
    begin { initialization }
        busy := false;
end;
```

```
    var binary_sem : Sem_Mon_type;
    begin
    Parbegin
repeat                           repeat                           repeat
  binary_sem.sem_wait;             binary_sem.sem_wait;             binary_sem.sem_wait;
  { Critical Section }             { Critical Section }             { Critica lSection }
  binary_sem.sem_signal;           binary_sem.sem_signal;           binary_sem.sem_signal;
  { Remainder of                   { Remainder of                   { Remainder of
      the cycle }                      the cycle }                      the cycle }
forever;                         forever;                         forever;
    Parend;
    end.
  Process P1                       Process P2                       Process P3
```

(a) (b)

Figure 3.6: Monitor implementation of binary semaphores

Recall from that, binary semaphore takes only values 0 and 1, and is used to implement a critical section. The boolean variable busy is used to indicate whether any process is currently using the critical section. Thus, its values true and false correspond to the values 0 and 1 of the binary semaphore, respectively. The condition variable non_busy corresponds to the condition that the critical section is not busy; it is used to block processes that try to enter a critical section while busy = true. The proceduressem_wait and sem_signal implement the wait and signal operations on the binary semaphore. Binary_sem is a monitor variable. The initialization part of the monitor type, which contains the statement busy :=false; is invoked when binary_sem is created. Hence variable busy of binary_sem is initialized to false.

Discussion

Explain the relationship between Monitor and semaphore

4.0 Self-Assessment/Exercises

1. **Can condition variables be implemented with semaphores?**

Answer

Semaphores can be implemented with condition variables, provided that there is also a primitive to protect a critical section (lock/unlock) so that both the semaphore value and the condition are checked atomically. In Nachos for example, this is done with disabling interrupts.

2. **Define briefly the lost wakeup problem.**
   Answer

The lost wakeup problem occurs when two threads are using critical section to synchronize their execution. If thread 1 reaches the case where the necessary condition will be false, such as when a consumer sees that the buffer is empty, it will go to sleep. It is possible that the OS will interrupt thread 1 just before it goes to sleep and schedule thread 2 which could make the condition for thread 1 true again, for example by adding something to the buffer. If this happens thread 2 will signal thread 1 to wake up, but since thread 1 is not asleep yet, the wakeup signal is lost. At some point thread 1 will continue executing and immediately go back to sleep. Eventually, thread 2 will find its condition to be false, for example if the buffer becomes full, and it will go to sleep. Now both threads are asleep and neither can be woken up.

5.0 Conclusion

Semaphores are used to implement synchronization in critical region solving a lot of processor issue. However, error arises when programmer fail to construct sequential processes. This issue that may arise have been dealt with the implementation of monitors

6.0 Summary

We have seen how processes and synchronization issue were solved with help of semaphores and monitors. Semaphores solve the critical section problem by using wait and signal operations. Monitors uses procedures, variable and data structures to ensure that only one process can be active in a monitor at any instant.

7.0 References/Further Reading

Abraham, S., Peter, B. G., & Gagne, G. (2009). *Operating System Concepts* (C. Weisman (ed.); 8th ed.). John Wiley & Sons Inc.

Andrew S, T., & Bos, H. (2015). *Modern Operating Systems* (H. Marcia & J. Tracy (eds.); Fourth). Pearson Education.

Dhananjay, M. D. (2009). *Operating Systems A Concept-Based Approach* (B. Melinda (ed.)). McGraw-Hill Higher Education.

Monitors in Operating System. (2020, September 26). https://www.tutorialandexample.com/monitors-in-operating-system

Monitors in Process Synchronization - GeeksforGeeks. (2019, September 30). https://www.geeksforgeeks.org/monitors-in-process-synchronization/

Semaphores in Operating System. (n.d.). Retrieved May 1, 2022, from https://www.tutorialspoint.com/semaphores-in-operating-system

Semaphores in Process Synchronization - GeeksforGeeks. (2021, November 22). https://www.geeksforgeeks.org/semaphores-in-process-synchronization/

Williams, L. (2022, April 16). *What is Semaphore? Counting, Binary Types with Example*. https://www.guru99.com/semaphore-in-operating-system.html

Unit 4 Synchronization Problems

Contents

1.0 Introduction

The operating systems literature is full of interesting problems that have been widely discussed and analyzed.

A solution to a process synchronization problem should meet three important criteria:

- Correctness: Data access synchronization and control synchronization should be performed in accordance with synchronization requirements of the problem.
- Maximum concurrency: A process should be able to operate freely except when it needs to wait for other processes to perform synchronization actions.
- No busy waits: To avoid performance degradation, synchronization should be performed through blocking rather than through busy waits.

Critical sections and signaling are the key elements of process synchronization, so a solution to a process synchronization problem should incorporate a suitable combination of these elements. In this unit, we analyze some classic problems in process synchronization, which are representative of synchronization problems in various application domains, and discuss issues (and common mistakes) in designing their solutions. We will also be implementing their solutions using various synchronization features provided in programming languages.

2.0 Intended Learning Outcomes (ILOs)

At the end of this unit, students will be able to

- Solving synchronization problems

3.0 Main Content

3.1 Producer – Consumer Problem

A producers–consumers system with bounded buffers consists of an unspecified number of producer and consumer processes and a finite pool of buffers. Each buffer is capable of holding an item of information. It is said to become full when a producer writes a new item into it, and become empty when a consumer copies out an item contained in it; it is empty when the producers–consumers system starts its operation. A producer process produces one item of information at a time and writes it into an empty buffer. A consumer process consumes information one item at a time from a full buffer. A producers–consumers system with bounded buffers is a useful abstraction for many practical synchronization problems.

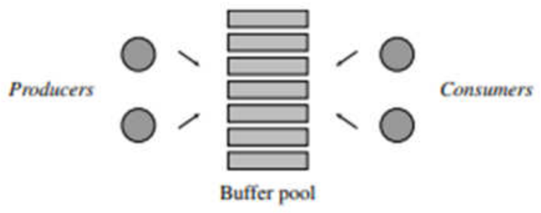


Figure 3.4: Producer – Consumer with bounded buffer

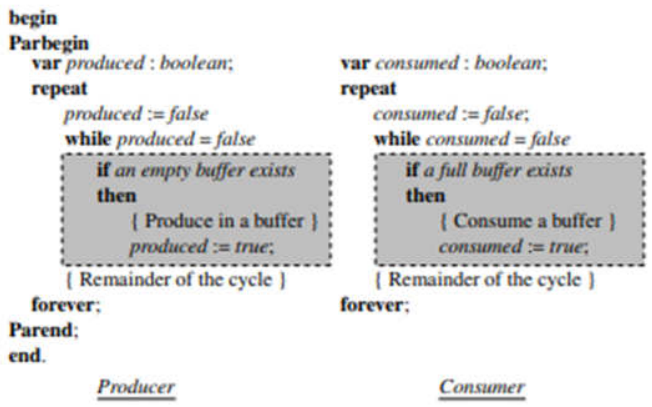
A print service is a good example. A fixed-size queue of print requests is the bounded buffer. A process that adds a print request to the queue is a producer process, and a print daemon is a consumer process.

A solution to the producers – consumers problem must satisfy the following conditions:

1. A producer must not overwrite a full buffer.
2. A consumer must not consume an empty buffer.
3. Producers and consumers must access buffers in a mutually exclusive manner.
4. Information must be consumed in the same order in which it is put into the buffers, i.e., in FIFO order.

Figure 3.5 shows an outline for the producers–consumers problem. Producer and consumer processes access a buffer inside a critical section. A producer enters its critical section and checks whether an empty buffer exists. If so, it produces into that buffer; otherwise, it merely exits from its critical section. This sequence is repeated until it finds an empty buffer. The boolean variable produced is used to break out of the **while** loop after the producer produces into an empty buffer. Analogously, a consumer makes repeated checks until it finds a full buffer to consume from.

This outline suffers from two problems—poor concurrency and busy waits. The pool contains many buffers, and so it should be possible for producers and consumers to concurrently access empty and full buffers, respectively.



```
begin
Parbegin
  var produced : boolean;                var consumed : boolean;
  repeat                                 repeat
     produced := false                      consumed := false;
     while produced = false                 while consumed = false
        if an empty buffer exists              if a full buffer exists
        then                                   then
           { Produce in a buffer }                { Consume a buffer }
           produced := true;                      consumed := true;
     { Remainder of the cycle }             { Remainder of the cycle }
  forever;                               forever;
Parend;
end.
          Producer                               Consumer
```

Figure 3.5: An outline of producer – consumer using critical sections

However, both produce and consume actions take place in critical sections for the entire buffer pool, and so only one process, whether producer or consumer, can access a buffer at any time. Busy waits exist in both producers and consumers. A producer repeatedly checks for an empty buffer and a consumer repeatedly checks for a full buffer. To avoid busy waits, a producer process should be blocked if an empty buffer is not available. When a consumer consumes from a buffer, it should activate a producer that is waiting for an empty buffer. Similarly, a consumer should be blocked if a full buffer is not available. A producer should activate such a consumer after producing in a buffer.

3.2 Reader-Writer Problem

Another problem is the readers and writers problem which models access to a data base. Imagine a big data base, such as banking system, with many transaction processes wishing to read and write it. It is acceptable to have multiple processes reading the data base at the same time, but if one process is writing (i.e., changing) the data base, no other processes may have access to the data base, not even readers. The question is how do you program the readers and the writers?

A solution to the readers–writers problem must satisfy the following conditions:

1. Many readers can perform reading concurrently.
2. Reading is prohibited while a writer is writing.
3. Only one writer can perform writing at any time.
4. A reader has a nonpreemptive priority over writers; i.e., it gets access to the shared data ahead of a waiting writer, but it does not preempt an active writer.

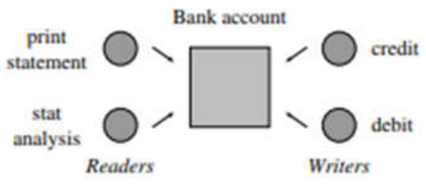
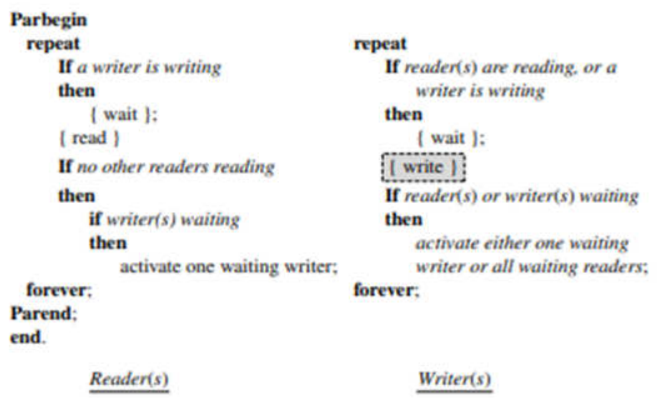


Figure 3.6: Reader-Writer problem in banking system

Figure 3.6 illustrates an example of a readers–writers system. The readers and writers share a bank account. The reader processes print statement and stat analysis merely read the data from the bank account; hence they can execute concurrently. Credit and debit modify the balance in the account. Clearly only one of them should be active at any moment and none of the readers should be concurrent with it.

Figure 3.7 is an outline for a readers–writers system. Writing is performed in a critical section. A critical section is not used in a reader, because that would prevent concurrency between readers. A signaling arrangement is used to handle blocking and activation of readers and writers.



```
Parbegin
  repeat                                  repeat
      If a writer is writing                  If reader(s) are reading, or a
      then                                        writer is writing
          { wait };                           then
      { read }                                    { wait };
      If no other readers reading             { write }
      then                                    If reader(s) or writer(s) waiting
          if writer(s) waiting                then
          then                                    activate either one waiting
              activate one waiting writer;        writer or all waiting readers;
  forever;                                forever;
Parend;
end.

          Reader(s)                               Writer(s)
```

Figure 3.7: An outline for a readers-writers system

3.3 Dining Philosophers

Five philosophers sit around a table pondering philosophical issues. A plate of spaghetti is kept in front of each philosopher, and a fork is placed between each pair of philosophers. To eat, a philosopher must pick up the two forks placed between him and the neighbors on either side, one at a time. The problem is to design processes to represent the philosophers such that each philosopher can eat when hungry and none dies of hunger.

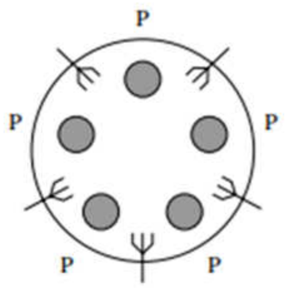


Figure 3.8: Dining Philosophers

The correctness condition in the dining philosophers system is that a hungry philosopher should not face indefinite waits when he decides to eat. The challenge is to design a solution that does not suffer from either deadlocks, where processes become blocked waiting for each other), or livelocks, where processes are not blocked but defer to each other indefinitely.

Consider the outline of a philosopher process Pi, where details of process synchronization have been omitted. A philosopher picks up the forks one at a time, say, first the left fork and then the right fork. This solution is prone to deadlock, because if all philosophers simultaneously lift their left forks, none will be able to lift the right fork! It also contains race conditions because neighbors might fight over a shared fork.

```
repeat
    if left fork is not available
    then
        block ($P_i$);
    lift left fork;
    if right fork is not available
    then
        block ($P_i$);
    lift right fork;
    { eat }
    put down both forks
    if left neighbor is waiting for his right fork
    then
        activate (left neighbor);
    if right neighbor is waiting for his left fork
    then
        activate (right neighbor);
    { think }
forever
```

Figure 3.9: An outline of dining philosopher

Discussion

Explain Dekker's Algorithm

4.0 Self-Assessment/Exercise

The dining philosopher online solution is prone to deadlock, because if all philosophers simultaneously lift their left forks, none will be able to lift the right fork! It also contains race conditions because neighbors might fight over a shared fork. How can it be improved to avoid deadlock?

Answer

We can avoid deadlocks by modifying the philosopher process so that if the right fork is not available, the philosopher would defer to his left neighbor by putting down the left fork and repeating the attempt to take the forks sometime later. The below is an improved outline for the dining philosopher

```
var     successful : boolean;
repeat
    successful := false;
    while (not successful)
        if both forks are available then
            lift the forks one at a time;
            successful := true;
        if successful = false
        then
            block ($P_i$);
    { eat }
  put down both forks;
    if left neighbor is waiting for his right fork
    then
        activate (left neighbor);
    if right neighbor is waiting for his left fork
    then
        activate (right neighbor);
    { think }
forever
```

5.0 Conclusion

We have seen few synchronization problems such as producer-consumer, readers and writers and dining philosophers. We have also provided outlines that poised solutions to the problem. These solutions solved synchronization problems to an extent. However, the solutions cannot solve the problems completely.

6.0 Summary

We present a solution to the dining-philosophers problem that ensures freedom from deadlocks. Note, however, that any satisfactory solution to the dining-philosophers problem must guard against the possibility that one of the philosophers will starve to death. A deadlock-free solution does not necessarily eliminate the possibility of starvation.

7.0 References/Further Reading

Abraham, S., Peter, B. G., & Gagne, G. (2009). *Operating System Concepts* (C. Weisman (ed.); 8th ed.). John Wiley & Sons Inc.

Andrew S, T., & Bos, H. (2015). *Modern Operating Systems* (H. Marcia & J. Tracy (eds.); Fourth). Pearson Education.

Dhananjay, M. D. (2009). *Operating Systems A Concept-Based Approach* (B. Melinda (ed.)). McGraw-Hill Higher Education.

OS - Classical Problems of Synchronization | i2tutorials. (n.d.). Retrieved May 1, 2022, from https://www.i2tutorials.com/os-introduction/os-classical-problems-of-synchronization/

Readers-Writers Problem | Set 1 (Introduction and Readers Preference Solution) - GeeksforGeeks. (2022, February 28). https://www.geeksforgeeks.org/readers-writers-problem-set-1-introduction-and-readers-preference-solution/

The Dining Philosophers Problem - javatpoint. (n.d.). Retrieved May 1, 2022, from https://www.javatpoint.com/os-dining-philosophers-problem

Module 4 Memory Management

Introduction of Module

Gaining an understanding of memory management takes us deeply into the shared realm of hardware architecture and software architecture. To make sense of the hardware architecture, we need to understand the software requirements. Memory management is much more than dynamic storage allocation. It involves providing an appropriate memory abstraction and making it work on the available physical resources, usually considered to consist of a high-speed cache, moderate-speed primary storage, and low-speed secondary storage. We begin with a brief history of memory management, introducing some of the more important memory management issues and how they were handled early on. We then focus on the concept of memory, swapping and partitions, memory paging and segmentation, Thrashing, caching and finally the replacement policies.

Unit 1: Memory Swapping

Unit 2: Memory Partition

Unit 3: Virtual Memory

Unit 4: Caching and Thrashing

Unit 5: Replacement Policies

Unit 1 Memory Swapping and Addresses

Contents

1.0 Introduction

Memory swapping is a computer technology that enables an operating system to provide more memory to a running application or process than is available in physical random access memory (RAM). When the physical system memory is exhausted, the operating system can opt to make use of memory swapping techniques to get additional memory.

Memory swapping is among the multiple techniques for memory management in modern systems. Physical memory alone is sometimes not sufficient, which is why there are different ways of augmenting RAM in a system with these additional options.

2.0 Intended Learning Outcomes (ILOs)

At the end of this unit, students will be able to

- Describe the techniques of memory swapping
- Demonstrate how resources are allocated

3.0 Main Content

3.1 Logical and Physical Addresses

Logical Address is generated by CPU while a program is running. The logical address is virtual address as it does not exist physically, therefore, it is also known as Virtual Address. This address is used as a reference to access the physical memory location by CPU. The term Logical Address Space is used for the set of all logical addresses generated by a program's perspective. The hardware device called Memory-Management Unit is used for mapping logical address to its corresponding physical address. **Physical Address** identifies a physical location of required data in a memory. The user never directly deals with the physical address but can be access by its corresponding logical address. The user program generates the logical address and thinks that the program is running in this logical address but the program needs physical memory for its execution, therefore, the logical address must be mapped to the physical address by MMU before they are used. The term Physical Address Space is used for all physical addresses corresponding to the logical addresses in a Logical address space.

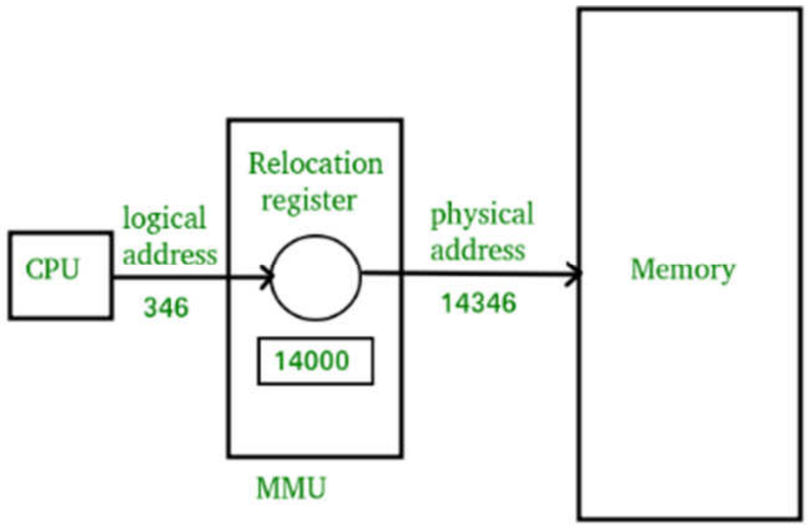


Figure 4.0: Memory addresses

Relocation is a method of shifting a user program from one memory location to another. There are two types of relocation: static and dynamic. Static relocation takes place at load time and remain fixed once the program is relocated, while the dynamic relocation takes place at run time.

Differences Between Logical and Physical Address in Operating System

1. The basic difference between Logical and physical address is that Logical address is generated by CPU in perspective of a program whereas the physical address is a location that exists in the memory unit.
2. Logical Address Space is the set of all logical addresses generated by CPU for a program whereas the set of all physical address mapped to corresponding logical addresses is called Physical Address Space.

3. The logical address does not exist physically in the memory whereas physical address is a location in the memory that can be accessed physically.
4. Identical logical addresses are generated by Compile-time and Load time address binding methods whereas they differ from each other in run-time address binding method.
5. The logical address is generated by the CPU while the program is running whereas the physical address is computed by the Memory Management Unit (MMU).

3.2 Address Binding Mechanism

Memory consists of large array of words or bytes, each with its own address. Usually, a program resides on a disk as binary executable file. For the program to execute, it must be brought into the main memory using its address. When the process completes its execution, it set the memory space free for next process to use.

Address binding of instructions and data to memory addresses can happen at three different stages.

- **Compile time**: Binding at compile time generates absolute addresses where a prior knowledge is required in the main memory. Must recompile code if starting location changes
- **Load time**: If it is not known at compile time where a process will reside in memory then the complier must generate re-locatable address. In this case final binding is delayed until load time.
- **Execution time**: This method permits moving of a process from one memory segment to another during run time. In this case final binding is delayed until run time. Need hardware support for address maps (e.g., base and limitregisters)

3.3 Memory Swapping

Swapping is a memory management technique and is used to temporarily remove the inactive programs from the main memory of the computer system. Any process must be in the memory for its execution, but can be swapped temporarily out of memory to a backing store and then again brought back into the memory to complete its execution. Swapping is done so that other processes get memory for their execution.

Due to the swapping technique performance usually gets affected, but it also helps in running multiple and big processes in parallel. **The swapping** process is also known as a technique for **memory compaction.** Basically, low priority processes may be swapped out so that processes with a higher priority may be loaded and executed.

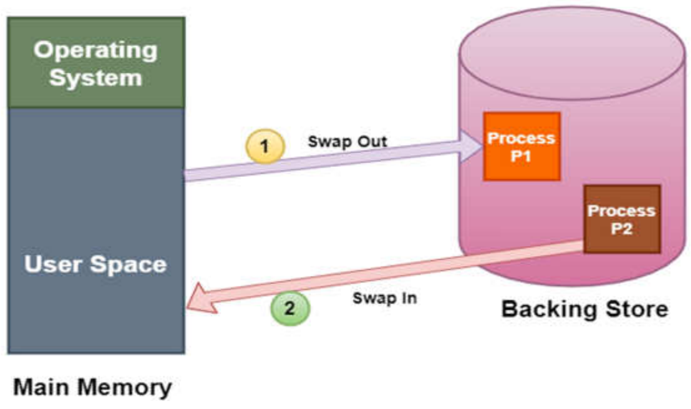


Figure 4.1: Memory Swapping In & Out

The above diagram shows swapping of two processes where the disk is used as a Backing store.

In the above diagram, suppose there is a multiprogramming environment with a round-robin scheduling algorithm; whenever the time quantum expires then the memory manager starts to swap out those processes that are just finished and swap another process into the memory that has been freed. And in the meantime, the CPU scheduler allocates the time slice to some other processes in the memory. The swapping of processes by the memory manager is fast enough that some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU.

A variant of the swapping technique is the priority-based scheduling algorithm. If any higher-priority process arrives and wants service, then the memory manager swaps out lower priority processes and then load the higher priority processes and then execute them. When the process with higher priority finishes, then the process with lower priority swapped back in and continues its execution. This variant is sometimes known as roll in and roll out.

There are two more concepts that come in the swapping technique and these are: **swap in** and **swap out.**

Swap In and Swap Out in OS

The procedure by which any process gets removed from the hard disk and placed in the main memory or RAM commonly known as Swap In**.**

On the other hand, Swap Out is the method of removing a process from the main memory or RAM and then adding it to the Hard Disk**.**

Advantages of Swapping

The advantages/benefits of the Swapping technique are as follows:

1. The swapping technique mainly helps the CPU to manage multiple processes within a single main memory.
2. This technique helps to create and use virtual memory.
3. With the help of this technique, the CPU can perform several tasks simultaneously. Thus, processes need not wait too long before their execution.
4. This technique is economical.
5. This technique can be easily applied to priority-based scheduling in order to improve its performance.

Disadvantages of Swapping

The drawbacks of the swapping technique are as follows:

1. There may be inefficiency in the case if a resource or variable is commonly used by those processes that are participating in the swapping process.
2. If the algorithm used for swapping is not good then the overall method can increase the number of page faults and thus decline the overall performance of processing.

3. If the computer system loses power at the time of high swapping activity, then the user might lose all the information related to the program.

Discussion

Explain the term Fragmentation

5.0 Conclusion

The process of memory swapping is managed by an operating system or by a virtual machine hypervisor. Swapping is often enabled by default, though users can choose to disable the capability.

The actual memory swapping process and the creation of a swap file is automatically managed by the operating system. It is initiated when needed as physical RAM is used and additional capacity is required by processes and applications. As additional RAM is required, the state of the physical memory page is mapped to the swap space, enabling a form of virtual (non-physical RAM) memory capacity. In other words, the main purpose of swapping in memory management is to enable more usable memory than held by the computer hardware.

6.0 Summary

The process of memory swapping is managed by an operating system or by a virtual machine hypervisor. Main purpose of swapping in memory management is to enable more usable memory than held by the computer hardware. Compile time and load time address binding generates logical and physical addresses which are identical, whereas address binding generated at execution time, results in different logical and physical address.

7.0 References/Further Reading

Abraham, S., Peter, B. G., & Gagne, G. (2009). *Operating System Concepts* (C. Weisman (ed.); 8th ed.). John Wiley & Sons Inc.

Andrew S, T., & Bos, H. (2015). *Modern Operating Systems* (H. Marcia & J. Tracy (eds.); Fourth). Pearson Education.

Dhananjay, M. D. (2009). *Operating Systems A Concept-Based Approach* (B. Melinda (ed.)). McGraw-Hill Higher Education.

Thomas, A., & Dahlin, M. (2015). *Operating Systems Principles & Practice Volume III : Memory Management* (S. Kaplan & S. Whitney (eds.); Second). Recursive Books, Ltd.

Address Binding and its Types - GeeksforGeeks. (2020, October 23). https://www.geeksforgeeks.org/address-binding-and-its-types/

Kerner, M. (2019, August 8). What is Memory Swapping? | Enterprise Storage Forum. https://www.enterprisestorageforum.com/hardware/what-is-memory-swapping/

Logical and Physical Address in Operating System - GeeksforGeeks. (2022, March 25). https://www.geeksforgeeks.org/logical-and-physical-address-in-operating-system/

Unit 2 Memory Partition

Contents

1.0 Introduction

Memory is allocated to processes until finally, the memory requirements of the next process cannot be satisfied i.e, no available block of memory (or hole) is large enough to hold that process. The operating system can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met. The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. This unit explains one common methods, partitioning and memory allocation.

2.0 Intended Learning Outcomes (ILOs)

At the end of unit, students will be able to

- Describe memory allocation to processes
- Use the best fit memory to fix allocation issues

3.0 Main Content

3.1 Contiguous and non-contiguous Memory Allocation

Contiguous Memory Allocation

In Contiguous Memory Allocation whenever any user process request for the memory then a single section of the contiguous memory block is allocated to that process according to the requirements of the process. Contiguous memory allocation is achieved just by dividing the memory into the fixed-sized partition**.**

In this, all the available memory space remains together at one place and freely available memory partitions are not distributed here and there across the whole memory space.

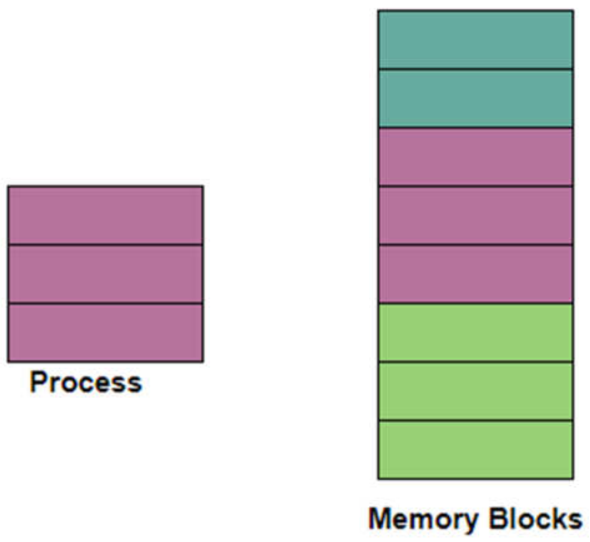


Figure 4.2: Contiguous allocation

Non-Contiguous Memory Allocation

With the help of Non-contiguous memory allocation, a process is allowed to acquire several memory blocks at different locations in the memory according to its need. In the non-contiguous memory allocation**,** the available free memory space is distributed here and there which means that all the free memory space is not in one place.

In this technique, memory space acquired by a process is not at one place but it is at different locations according to the requirements of the process.

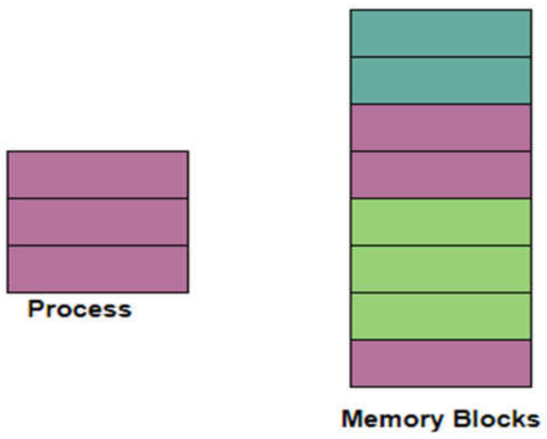


Figure 4.3: Non-contiguous memory allocation

3.2 Memory Partitioning

Memory partitioning is the system by which the memory of a computer system is divided into sections for use by the resident programs. These memory divisions are known as partitions. There are different ways in which memory can be partitioned (i.e fixed, variable, and dynamic partitioning).

Fixed partitioning

It is defined as the system of dividing memory into non-overlapping sizes that are fixed, unmovable, static. A process may be loaded into a partition of equal or greater size and is confined to its allocated partition.

If we have comparatively small processes with respect to the fixed partition sizes, this poses a big problem. This results in occupying all partitions with lots of unoccupied space left. This unoccupied space is known as **fragmentation**. Within the fixed partition context, this is known as **internal fragmentation (IF)**. This is because of unused space created by a process within its allocated partition (internal).

In fixed partition method, the partitions could be of different sizes, but once decided at the time of system generation, they remain fixed.

An example of partition memory is shown here. Out of the six partitions one is occupied by the OS and 3 partitions are allocated to processes P_i, $P_j$and P_k.The remaining 2 shaded partitions are free and available for allocation of next processes.

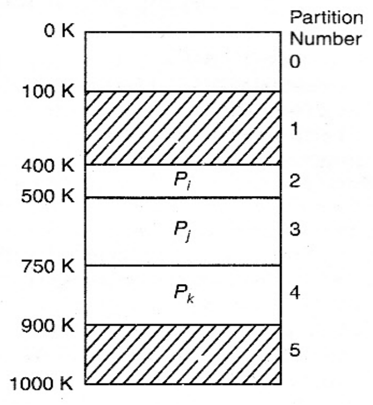


Figure 4.4: Blocks of memory

Variable partitioning

Variable partitioning is the system of dividing memory into non-overlapping but variable sizes. This system of partitioning is more flexible than the fixed partitioning configuration, but it's still not the most ideal solution. Small processes are fit into small partitions (item 1) and large processes fit into larger partitions (items 2 and 3). These processes do not necessarily fit exactly, even though there are other unoccupied partitions. Items 3 and 4 are larger processes of the same size, but memory has only one available partition that can fit either of them.

The flexibility offered in variable partitioning still does not completely solve our problems.

Most **disadvantages of static partition** are directly attributed to its **inflexibility** and **inability** to adapt changing system needs. One of the primary problems is the internal fragmentation***.Internal fragmentation*** is the difference in size between the process and its allocated partition.

To remove such problems attributed to static partitioning of memory, dynamic (variable) partitioning is used in which partitions are defined dynamically rather than static.

Dynamic partitioning

In dynamic partitioning, the partitions used are of variable length and number. When a process is brought into the memory, it is allocated exactly to that amount of memory that it needs, not more than that it requires.

An example shown below uses 1 MB memory for dynamic partition. The first three processes $P_{1,}P_{2,}P_3$ are loaded starting where OS ends and occupy just enough space for each process (Figure 4.5: dynamic partitioning a-h). This leaves a small hole at the end of the memory, which is too small for the fourth process P_4. Thus, the OS must swap out any of the three process which is not currently required according to short term scheduler. Say P_2 is swapped out (f, this leaves sufficient space to load P_4 (fig 4.5(f)). Since P_4 is smaller than P_2 so another hole is created.

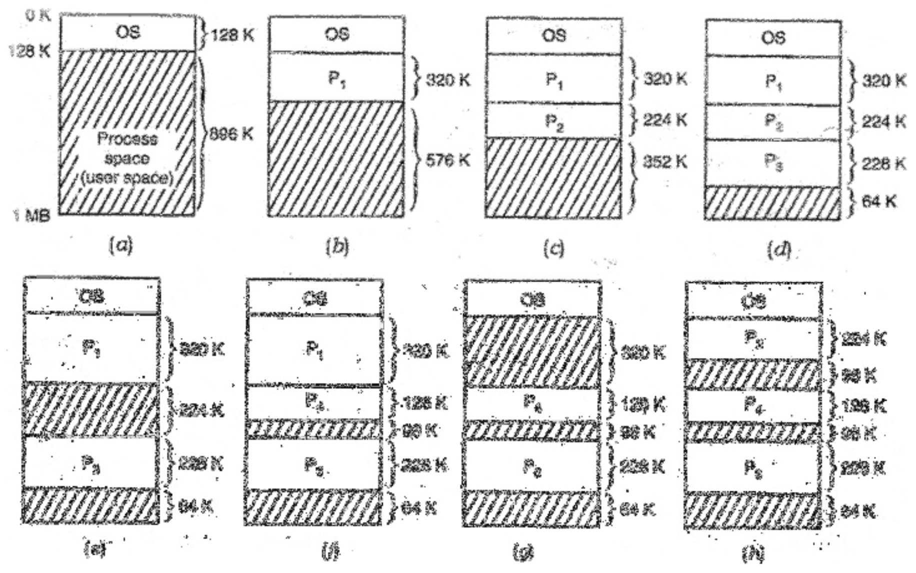
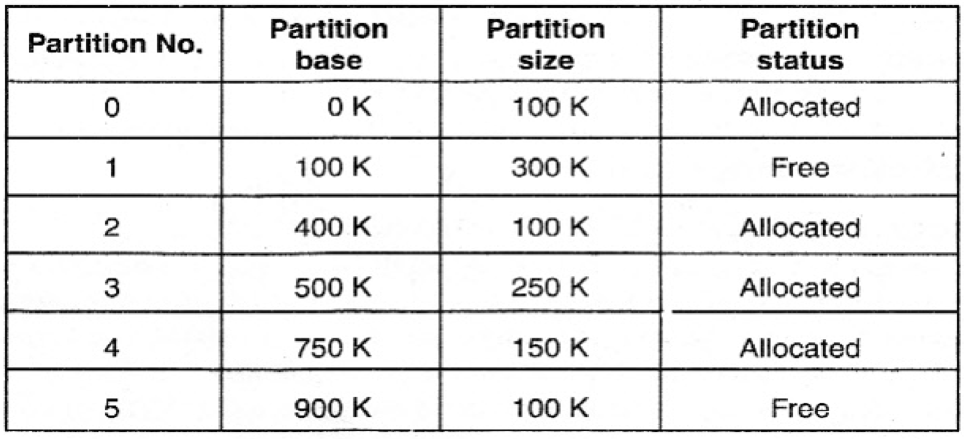


Figure 4.5:dynamic memory partitioning

Partitioning Descriptive Table (PDT)

Once partitions are defined, an OS needs to keep track of their status, such as free or in use, for allocation purposes. Current partitions status and attributes are collected in a data structure called PDT. As shown below



Partition No.	Partition base	Partition size	Partition status
0	0 K	100 K	Allocated
1	100 K	300 K	Free
2	400 K	100 K	Allocated
3	500 K	250 K	Allocated
4	750 K	150 K	Allocated
5	900 K	100 K	Free

Figure 4.6: Partition Table description

In PDT, partition is defined by its base address, size and status. When static partition is used, only status field of each entry varies either free or allocated, all other fields are static and contained the values defined at partition definition time. Two major problems arise in static partition (we will propose solution in the partition allocation methods).

1. How to select a specific partition for a given process?
2. What is done when no suitable partition is available for allocation

3.3 Partition Allocation Methods

The following are four common memory management techniques.

Single contiguous allocation: Simplest allocation method used by MS-DOS. All memory (except some reserved for OS) is available to a process.

Partition allocation: Memory is divided into different blocks or partitions. Each process is allocated according to the requirement.

In **Partition Allocation**, when there is more than one partition freely available to accommodate a process's request, a partition must be selected. To choose a particular partition, a partition allocation method is needed. A partition allocation method is considered better if it avoids internal fragmentation.

When it is time to load a process into the main memory and if there is more than one free block of memory of sufficient size then the OS decides which free block to allocate.

There are different Placement Algorithm:

A. First Fit

B. Best Fit

C. Worst Fit

D. Next Fit

1. **First Fit**: In the first fit, the partition is allocated which is the first sufficient block from the top of Main Memory. It scans memory from the beginning and chooses the first available block that is large enough. Thus, it allocates the first hole that is large enough.

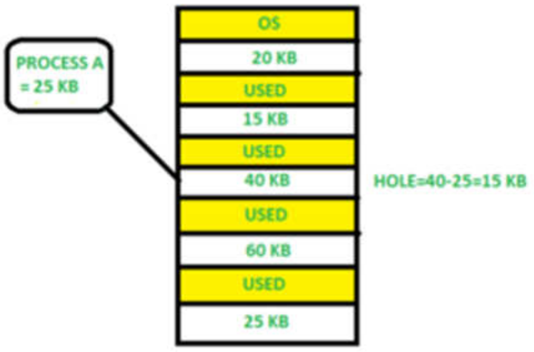


Figure 4.6: First fit technique

2. **Best Fit** Allocate the process to the partition which is the first smallest sufficient partition among the free available partition. It searches the entire list of holes to find the smallest hole whose size is greater than or equal to the size of the process.

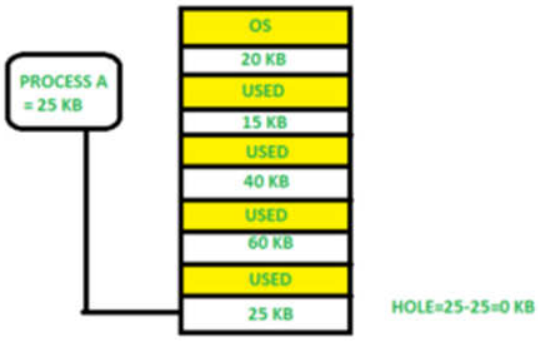


Figure 4.7: Best fit technique

3. **Worst Fit** Allocate the process to the partition which is the largest sufficient among the freely available partitions available in the main memory. It is opposite to the best-fit

algorithm. It searches the entire list of holes to find the largest hole and allocate it to process.

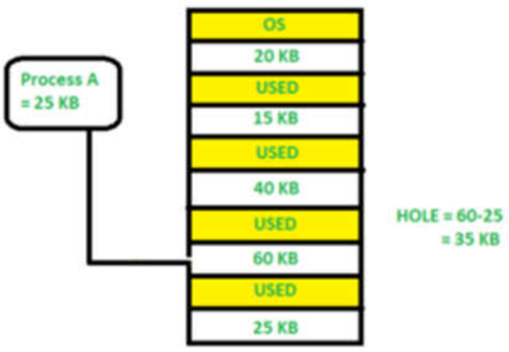


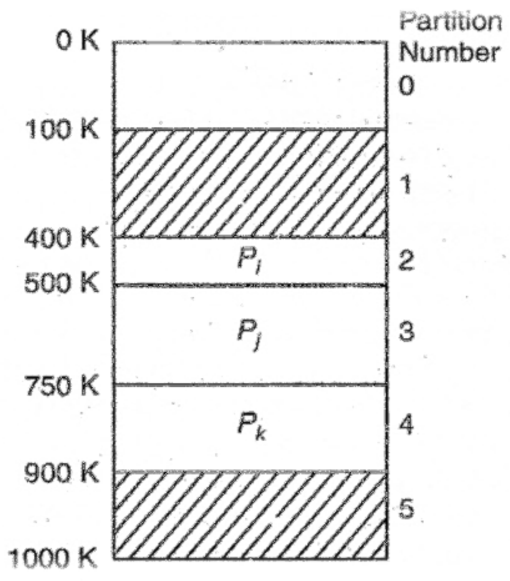
Figure 4.8: Worst fit technique

4. **Next Fit:** Next fit is similar to the first fit but it will search for the first sufficient partition from the last allocation point.

The first problem stated in the DPT can be solved in several ways, for which first fit and best fit are probably the most common strategies. When selecting between the two approaches, a trade-off between execution speed and memory utilization is considered. While first fit terminates upon finding the 1st partition of adequate size (faster), the best fit searches all PDT entries to identify the tightest fit, as such best fit may achieve higher utilization of memory.

3.4 Cases/Example

Considering the example of the working of the two partition allocation strategies., considering a process *P* of size 70KB to be created for its allocation in the example shown below.



Applying first fit method, the process *P* will be allocated in partition 1 which will result 300 – 70 = 230 gap of unusable memory until process *P* terminates or swapped out of the memory. Applying best fit algorithm, process *P* will be allocated in partition 5 which will result 100 KB – 70KB = 30 KB of unusable memory units.

Comparing the two methods, best fit gives us better memory management. However, if we are interested in time, the best fit requires more time than first fit.

Discussion

Explain what you understand by Resident Monitor

4.0 Self-Assessment/Exercises

1. Define external and internal fragmentation and identify the differences between them.

   Answer

   Internal fragmentation is where the memory manager allocates more for each allocation than is actually requested. Internal fragmentation is the wasted (unused) space within a page. For example if I need 1K of memory, but the page size is 4K, then there is 3K of wasted space due to internal fragmentation. External fragmentation is the inability to use memory because free memory is divided into many small blocks. If live objects are scattered, the free blocks cannot be coalesced, and hence no large blocks can be allocated. External fragmentation is the wasted space outside of any group of allocated pages that is too small to be used to satisfy another request. For example if best-fit memory management is used, then very small areas of memory are likely to remain, which may not be usable by any future request. Both types of fragmentation result in free memory that is unusable by the system.

2. Given memory partitions of 100 KB, 500 KB, 200 KB, 300 KB and 600 KB (in order), how would each of the first-fit, best-fit and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB and 426 KB (in that order) ? Which algorithm makes the most efficient use of memory?

   Answer
   First-Fit:
   212K is put in 500K partition.
   417K is put in 600K partition.
   112K is put in 288K partition (new partition 288K = 500K - 212K).
   426K must wait.

   Best-Fit:
   212K is put in 300K partition.
   417K is put in 500K partition.
   112K is put in 200K partition.
   426K is put in 600K partition.

   Worst-Fit:
   212K is put in 600K partition.
   417K is put in 500K partition.
   112K is put in 388K partition.
   426K must wait.

In this example, Best-Fit turns out to be the best.

5.0 Conclusion

Operating systems choose static and dynamic memory allocation under different circumstances to obtain the best combination of execution efficiency and memory efficiency. When sufficient information about memory requirements is available a priori, the kernel or the run-time library makes memory allocation decisions statically, which provides execution efficiency. When little information is available a priori, the memory allocation decisions are made dynamically, which incurs higher overhead but ensures efficient use of memory.

6.0 Summary

Memory binding is the association of memory addresses with instructions and data of a program. To provide convenience and flexibility, memory binding is performed several times to a program, the compiler and linker perform it statically, i.e., before program execution begins, whereas the OS performs it dynamically, i.e., during execution of the program. The kernel uses a model of memory allocation to a process that provides for both static and dynamic memory binding

7.0 References/Further Reading

Abraham, S., Peter, B. G., & Gagne, G. (2009). *Operating System Concepts* (C. Weisman (ed.); 8th ed.). John Wiley & Sons Inc.

Andrew S, T., & Bos, H. (2015). *Modern Operating Systems* (H. Marcia & J. Tracy (eds.); Fourth). Pearson Education.

Dhananjay, M. D. (2009). *Operating Systems A Concept-Based Approach* (B. Melinda (ed.)). McGraw-Hill Higher Education.

7.4: Memory Partitioning - Engineering LibreTexts. (2021, March 31). https://eng.libretexts.org/Courses/Delta_College/Operating_System%3A_The_Basics/07%3A_Memory/7.4%3A_Memory_Partitioning

Difference between Contiguous and Noncontiguous Memory Allocation - GeeksforGeeks. (2021, June 1). https://www.geeksforgeeks.org/difference-between-contiguous-and-noncontiguous-memory-allocation/

Unit 3 Virtual Memory

Contents

1.0 Introduction

Virtual memory is a part of the memory hierarchy that consists of memory and a disk. Virtual memory is implemented through the noncontiguous memory allocation model and comprises both hardware components and a software component called a virtual memory manager. The hardware components speed up address translation and help the virtual memory manager perform its tasks more effectively. The virtual memory manager decides which portions of a process address space should be in memory at any time. Memory paging and segmentation are the two techniques that will be discussed in this unit.

2.0 Intended Learning Outcomes (ILOs)

At the end of this unit, students will able to

- Explain the concept of memory segmentation and paging
- Solving memory allocation issues into non-contiguous memory spaces

3.0 Main Content

3.1 Paging

Paging is a memory management scheme that removes the requirement of contiguous allocation of physical memory. It permits the physical address space of the process to be non-contiguous.

The physical memory is conceptually divided into a number of fixed –size blocks called **frames** and the logical address space is also split into fixed size blocks, called **pages**. For a process to be executed, its pages are loaded into any available frames from the backing store. The backing store is also divided into fixed size blocks that are of the same size of the frames, i.e the size of a frame is same as the size of a page for particular hardware.

Allocation of memory consists of finding sufficient number of unused page frames for loading the pages of the requesting process for its execution. An address translation scheme is used to map the logical pages to their physical counterparts. Since each page is mapped separately, different page frame allocated to a single process need not occupy contiguous areas of the physical memory.

The figure below illustrates the basic principle of paging, using a 16 MB system in which logical and physical addresses are assume to be 24 bits each in size. The logical address is divided into four pages numbered from 0 to 3. The mapping of logical address to physical address in paging system is performed at page level. Particularly, each virtual address is divided into two parts: the page number (p) and the offset (d) within that page. Since page (p) and frame (f) have identical sizes, offsets within each are identical and need not be mapped.

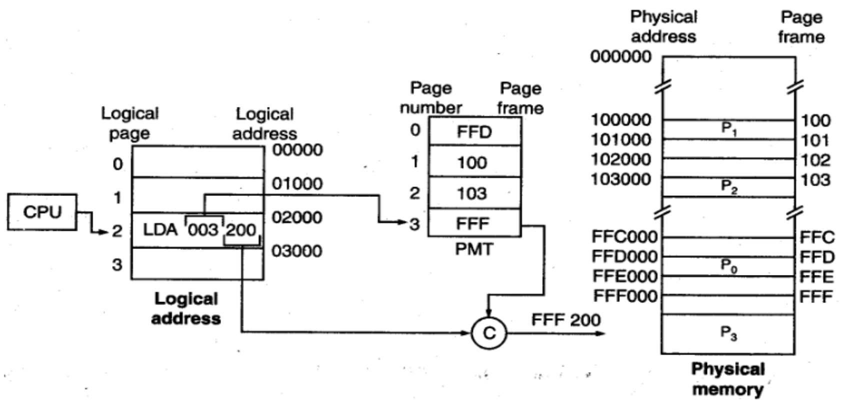


Figure 4.9: memory paging

In the above example, each 24 bit logical address may be regarded as a 12-bit page numbers (higher order bits) and a 12 bit offset within the page. Address translation is performed with help of a mapping table called ***page-map table.*** PMT is constructed at process loading time in order to establish the correspondence between the virtual and physical address.

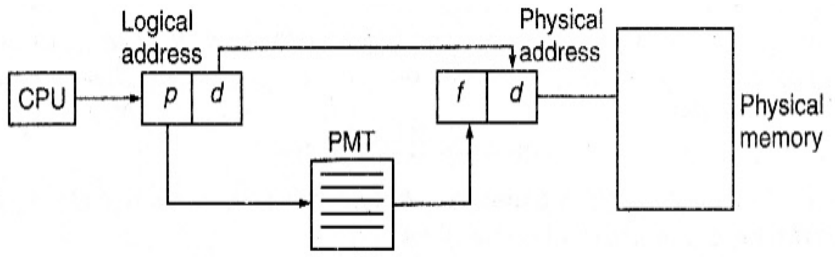
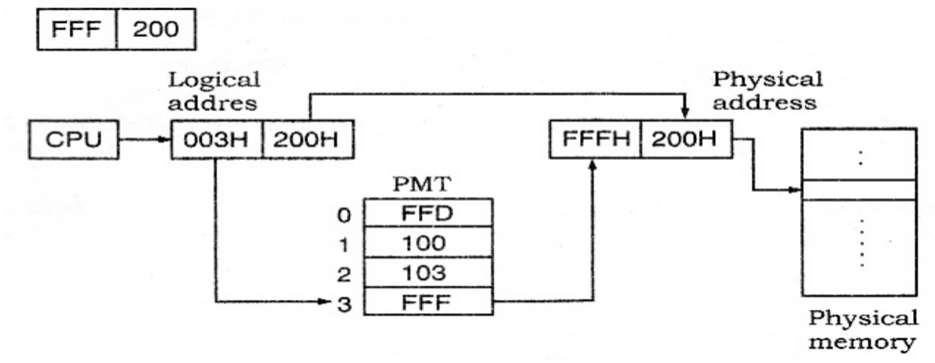


Figure 4.10: Memory allocation with PMT

For example, page 0 is placed in the frame whose starting address is FFD000H (Hex) with each frame being 1000 H in size, the corresponding frame number is FFDH as indicated on the right side of the physical memory.

The logic address translation in paged system is illustrated below, on the example of virtual addresses 3200 H. such address is split into two parts:

003 H indicates the page number and 200 H indicates the offset within that page. Page number is used to index PMT and to obtain the corresponding physical frame number (FFFH in our example) this value is then concatenated with the offset (200 H) to produce the physical address (FFF 200 H) which is used as a reference address in physical memory.

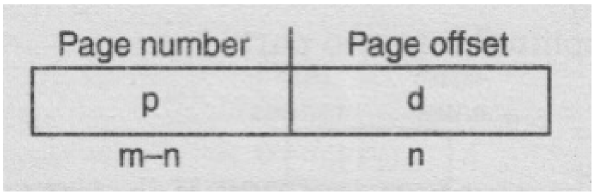


If the capacity of physical memory is ***m*** and the size of each page is ***p*** then the number of frames in the physical memory will be

$f = \frac{m}{p}(2)$

Both ***m*** and ***p*** are usually integers power of 2, thus resulting in ***f*** being also an integer.

The page size is similar to the frame size which is defined by the hardware and it varies according to architecture of computer system. For convenience of mapping, page sizes are usually an integer power of 2. If the size of a logical address space is $2^m$and page size is 2^n units, then the higher order $m — n$ units of logical address designate the page number and the n low order units designate the page offset. Thus, the logical address is defined as:



Where ***p*** is the index number in the page table and ***d*** is the offset value (displacement)

Note: If the page size is not the power of **2** the separation of ***p***and ***d*** is not possible

3.2 Memory Segmentation

Segmentation is another technique for the ***non-contiguous*** allocation. It is different from ***paging*** as pages are physical in nature and hence are of ***fixed size***, whereas segmentation are logical division of a program and hence are of ***variablesize***.

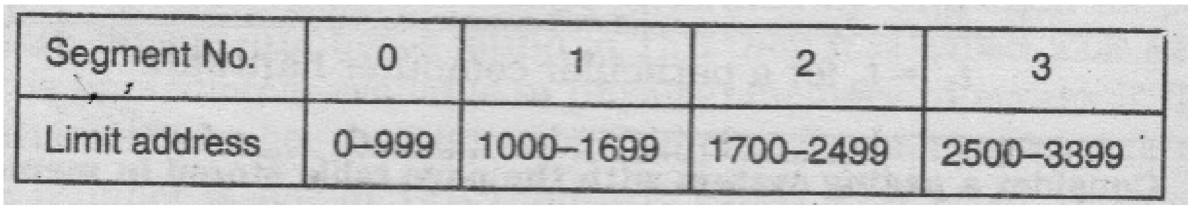
It is a memory management scheme that supports the ***user view*** of memory rather than ***system view*** of memory as paging. In this case, we divide the logical address space into different segments. For simplicity, the segments are referred by a segment number, rather than the segment name.

Thus, a logical address consist of two parts: <segment_name, offset>

The size of the segment varies according to the data stored in it or the nature of operation performed on that segment.

So, segmentation is a method of dividing memory into independent address space called ***segments.*** Each segment consist of linear sequence (0,1,2,3….) of address starting from 0 to maximum value depending upon the size of segment.

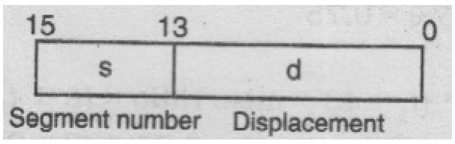
Like in paging system, we face the same problem in segmentation that how the compiler will generate 2-d array and how it is used in address translation? When we put all segments in a conceptual manner one after the other as shown below, then it is clear that the compiler itself has to generate the 2-d address (s, d) ie (segment no, offset)



Segment No.	0	1	2	3
Limit address	0–999	1000–1699	1700–2499	2500–3399

Figure 4.11: Virtual Address segmentation

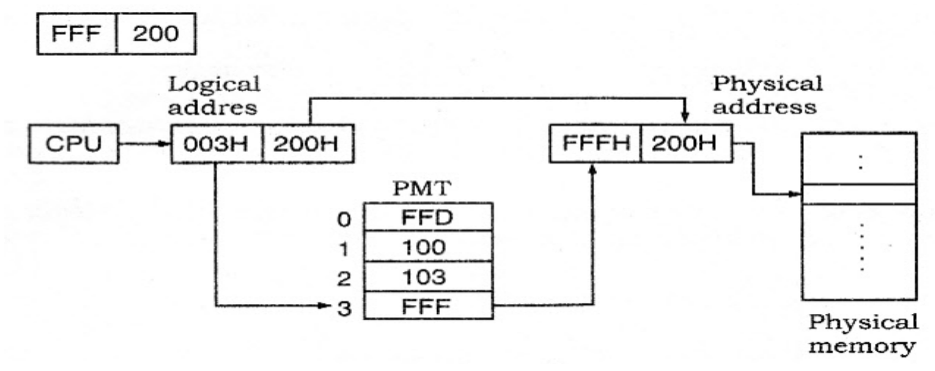
This is different from paging system in which a single dimensional virtual address and two-dimensional address would be exactly same in binary form as the page size is an integer power of 2. In segmentation it is not possible as the segmentation size is unpredictable. So we need to express the address in ***2d*** form. So, the virtual address space is divided into two parts in which higher order units refers to **s** ie, segment number and lower order units refer to ***d***ie displacement (limit value).



3.3 Cases/Examples

The logic address translation in paged system is illustrated below, on the example of virtual addresses 3200 H, such address is split into two parts:

003 H indicates the page number and 200 H indicates the offset within that page. Page number is used to index PMT and to obtain the corresponding physical frame number (FFFH in our example) this value is then concatenated with the offset (200 H) to produce the physical address (FFF 200 H) which is used as a reference address in physical memory.



Discussion

What technique best solve the memory allocation issues?

4.0 Self-Assessment/Exercises

1. Consider a logical address space of eight pages of 1024 words, each mapped onto a physical memory of 32 frames .

1) ***How many bits are in the logical address?***
2) ***How many bits are in the physical address?***

Answer

let No of bits in the physical memory be m,

Then the size of physical memory $=2^{n}$

Given: No of pages = 8 = 2^{3} and No of frames = 32 = 2^{5}

Size of each frame = size of each page = 1024 $=2^{10}$

So $f = \frac{m}{p}$ $2^{5} = \frac{2^{m}}{2^{10}}$

or $2^{m} = 2^{5} \times 2^{10} = 2^{15}$

Applying log to the base 2 0n both sides results: m = 15

So, required No of bits in physical memory = 15 bits.

Similarly, assuming No bits in logical memory be n then the size of logical memory = 2^n

So $\text{No. of pages} = \frac{\text{size of logical memory}}{\text{size of each page}}$

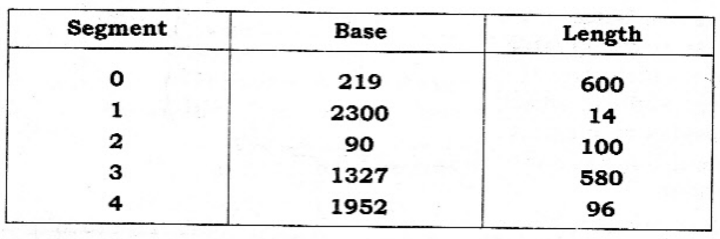
or $2^3 = \frac{2^n}{2^{10}}$

or $2^n = 2^3 \text{ x } 2^{10} = 2^{13}$

Applying log to the base 2 on both sides results: n=13

So No of bits required in logical memory = 13 bits

2. Consider the following segment table: what are the physical address for the following logical addresses? (i) 0430, (ii) 110, (iii) 2500, (iv) 3400 (v) 4112.



Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

Answer

For the given logical address the first digits refers to the segment no. S while remaining digits refers to the offset value die in (i) 0430 the 1st digit 0 refer to segment and 430 refers to offset value for the logical address (0430). Also the size of segment 0 is 600 as shown in the given table.

Physical address for logical address 0430 will be = base + offset(0430) = 219 + 430 = 649

Similarly, (ii) physical address for 110 = 2300 +10 =2310

(iii) Physical address for 2500 = 90 + 500 = 590 (but it is impossible since the segment size is 100 so it is illegal address)

(iv) physical address for 3400 = 1327 + 400 = 1727.

(v) physical address for 4112 = illegal address as the size of segment four (96) < offset value (112)

5.0 Conclusion

When all page frames are in use, the operating system must select a page frame to reuse for the page the program now needs. If the evicted page frame was dynamically allocated by a program to hold data, or if a program modified it since it was read into RAM, it must be written out to disk before being freed. If a program later references the evicted page, another page fault occurs and the page must be read back into RAM.

6.0 Summary

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. This scheme permits the physical address space of a process to be non – contiguous. A process is divided into Segments. The chunks that a program is divided into which are not necessarily all of the same sizes are called segments.

7.0 References/Further Reading

Abraham, S., Peter, B. G., & Gagne, G. (2009). *Operating System Concepts* (C. Weisman (ed.); 8th ed.). John Wiley & Sons Inc.

Andrew, S. T. (1987). *Operating Systems: Design and Implementation*. Prentice-Hall International, Inc.

Andrew S, T., & Bos, H. (2015). *Modern Operating Systems* (H. Marcia & J. Tracy (eds.); Fourth).

Pearson Education.

Dhananjay, M. D. (2009). *Operating Systems A Concept-Based Approach* (B. Melinda (ed.)). McGraw-Hill Higher Education.

Thomas, A., & Dahlin, M. (2015). *Operating Systems Principles & Practice Volume III : Memory Management* (S. Kaplan & S. Whitney (eds.); Second). Recursive Books, Ltd.

Difference between Paging and Segmentation. (n.d.). Retrieved May 1, 2022, from https://www.tutorialspoint.com/difference-between-paging-and-segmentation

Difference Between Paging and Segmentation | Difference Between. (n.d.). Retrieved May 1, 2022, from http://www.differencebetween.net/technology/difference-between-paging-and-segmentation/

Unit 4 Memory Caching

Contents

1.0 Introduction

Caches are central to the design of a huge number of hardware and software systems, including operating systems, Internet naming, web clients, and web servers. In particular, smartphone operating systems are often memory constrained and must manage memory carefully. Server operating systems make extensive use of remote memory and remote disk across the data center, using the local server memory as a cache. Even desktop operating systems use caching extensively in the implementation of the file system. Most importantly, understanding when caches work and when they do not is essential to every computer systems designer.

2.0 Intended Learning Outcomes (ILOs)

At the end of this unit, students will able to

- Understand the importance of cache
- How companies have succeeded as a result of cache memory

3.0 Main Content

3.1 Cache Concept

A cache is a copy of a computation or data that can be accessed more quickly than the original. While any object on my page might change from moment to moment, it seldom does.

The simplest kind of a cache is a memory cache. It stores (address, value) pairs. When we need to read value of a certain memory location, we first consult the cache, and it either replies with the value (if the cache knows it) and otherwise it forwards the request onward. If the cache has the value, that is called a cache **hit**. If the cache does not, that is called a cache **miss.**

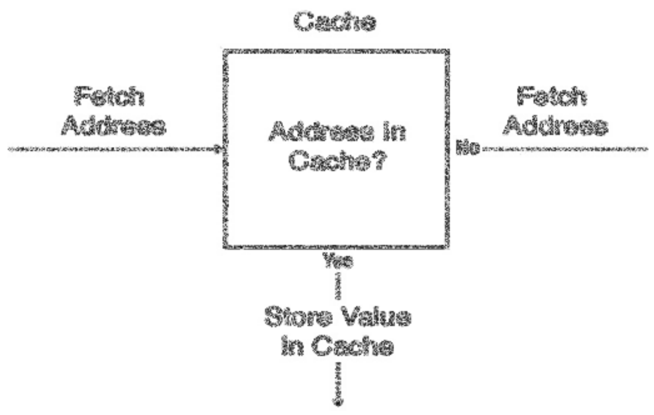


Figure 4.12: Cache

For a memory cache to be useful, two properties need to hold. First, the cost of retrieving data out of the cache must be significantly less than fetching the data from memory. In other words, the cost of a cache hit must be less than a cache miss, or we would just skip using the cache. Second, the likelihood of a cache hit must be high enough to make it worth the effort. Predictability sources help cache activities to be very accessible. Temporal locality, spatial locality and Prefetch are all techniques of cache.

In Temporal locality, programs tend to reference the same instructions and data that they had recently accessed; Spatial locality, Programs tend to reference data near other data that has been recently referenced. Prefetch fetch data ahead of time before needed. For example, if the file system observes the application reading a sequence of blocks into memory, it will read the subsequent blocks ahead of time, without waiting to be asked.

Putting these together, the latency of a read request is as follows:

Latency(read request) = Prob(cache hit) × Latency(cache hit) + Prob(cache miss) × Latency(cache miss)

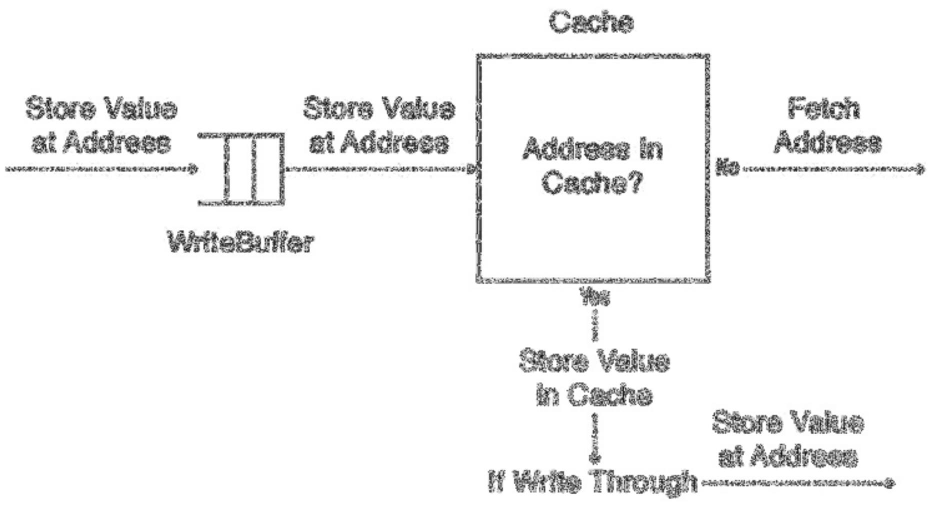


Figure 4.13: Operation of a memory cache write

The behavior of a cache on a write operation is shown in Figure 4.13. The operation is a bit more complex, but the latency of a write operation is easier to understand. Most systems buffer writes. As long as there is room in the buffer, the computation can continue immediately while the data are transferred into the cache and to memory in the background. Subsequent read requests must check both the write buffer and the cache — returning data from the write buffer if it is the latest copy. In the background, the system checks if the address is in the cache. If not, the rest of the cache block must be fetched from memory and then updated with the changed value. Finally, if the cache is write-through, all updates are sent immediately onward to memory. If the cache is write-back, updates can be stored in the cache, and only sent to memory when the cache runs out of space and needs to evict a block to make room for a new memory block.

3.2 Memory Hierarchy

When we are deciding whether to use a cache in the operating system or some new application, it is helpful to start with an understanding of the cost and performance of various levels of memory and disk storage.

Cache	**Hit**	**Cost Size**
1st level cache/ 1st level TLB	1 ns	64KB
2nd level cache/TLB	4 ns	256KB

3rd level cache	12 ns	2 MB
Memory (DRAM)	100 ns	10 GB
Data Center Memory (DRAM)	100 µs	100 TB
Local non-volatile memory	100 µs	100 GB
Local Disk	10 ms	1 TB
Data center disk	10 ms	100 PB
Remote data center disk	200 ms	1 XB

There is a fundamental tradeoff between the speed, size, and cost of storage. The smaller memory is, the faster it can be; the slower memory is, the cheaper it can be.

First-level cache. Most modern processor architectures contain a small first-level, virtually addressed, cache very close to the processor, designed to keep the processor fed with instructions and data at the clock rate of the processor.

Second-level cache. Because it is impossible to build a large cache as fast as a small one, the processor will often contain a second-level, physically addressed cache to handle cache misses from the first-level cache.

Third-level cache. Likewise, many processors include an even larger, slower third-level cache to catch second-level cache misses. This cache is often shared across all of the on-chip processor cores.

First-and second-level TLB. The translation lookaside buffer (TLB) will also be organized with multiple levels: a small, fast first-level TLB designed to keep up with the processor, backed up by a larger, slightly slower, second-level TLB to catch first-level TLB misses.

Main memory (DRAM). From a hardware perspective, the first-, second-, and third-level caches provide faster access to main memory; from a software perspective, however, main memory itself can be viewed as a cache.

Data center memory (DRAM). With a high-speed local area network such as a data center, the latency to fetch a page of data from the memory of a nearby computer is much faster than fetching it from disk. In aggregate, the memory of nearby nodes will often be larger than that of the local disk. Using the memory of nearby nodes to avoid the latency of going to disk is called cooperative caching, as it requires the cooperative management of the nodes in the data center. Many large scale data center services, such as Google and Facebook, make extensive use of cooperative caching.

Local disk or non-volatile memory. For client machines, local disk or non-volatile flash memory can serve as backing store when the system runs out of memory. In turn, the local disk serves as a cache for remote disk storage. For example, web browsers store recently fetched web pages in the client file system to avoid the cost of transferring the data again the next time it is used; once cached, the browser only needs to validate with the server whether the page has changed before rendering the web page for the user.

Data center disk. The aggregate disks inside a data center provide enormous storage capacity compared to a computer's local disk, and even relative to the aggregate memory of the data center.

Remote data center disk. Geographically remote disks in a data center are much slower because of wide-area network latencies, but they provide access to even larger storage capacity in aggregate. Many data centers also store a copy of their data on a remote robotic tape system, but since these systems have very high latency (measured in the tens of seconds), they are typically accessed only in the event of a failure.

3.3 Memory Cache Lookup

A memory cache maps a sparse set of addresses to the data values stored at those addresses. You can think of a cache as a giant table with two columns: one for the address and one for the data stored at that address. To exploit spatial locality, each entry in the table will store the values for a block of memory, not just the value for a single memory word.

We need to be able to rapidly convert an address to find the corresponding data, while minimizing storage overhead. Three common mechanisms for cache lookup are:

Fully associative

With a fully associative cache, the address can be stored anywhere in the table, and so on a lookup, the system must check the address against all of the entries in the table as illustrated in Figure below. There is a cache hit if any of the table entries match. Because any address can be stored anywhere, this provides the system maximal flexibility when it needs to choose an entry to discard when it runs out of space. Physical memory is a fully associative cache. Any page frame can hold any virtual page, and we can find where each virtual page is stored using a multi-level tree lookup

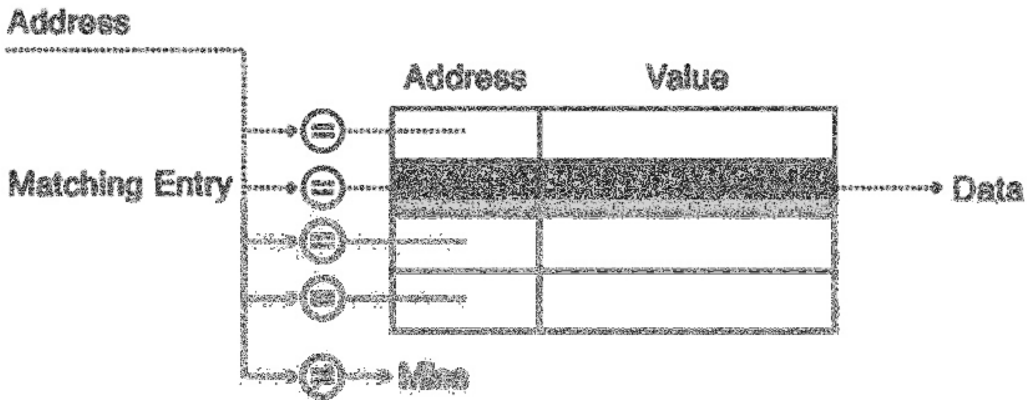


Figure 4.14: Full Associate lookup

Direct mapped

With a direct mapped cache, each address can only be stored in one location in the table. Lookup is easy: we hash the address to its entry, as shown in Figure 4.14. There is a cache hit if the address matches that entry and a cache miss otherwise.

A direct mapped cache allows efficient lookup, but it loses much of that advantage in decreased flexibility. If a program happens to need two different addresses that both hash to the same entry, such as the program counter and the stack pointer, the system will thrash.

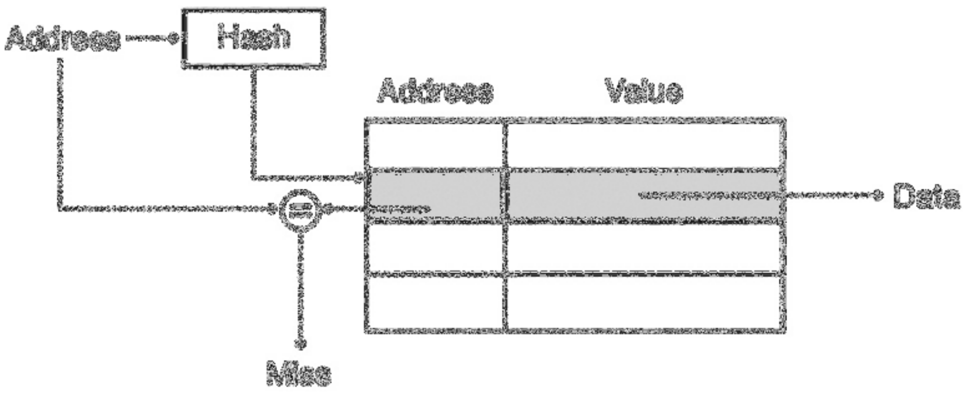


Figure 4.15: Direct mapped cache lookup

Set associative

A set associative cache melds the two approaches, allowing a tradeoff of slightly slower lookup than a direct mapped cache in exchange for most of the flexibility of a fully associative cache. With a set associative cache, we replicate the direct mapped table and lookup in each replica in parallel. A k set associative cache has k replicas; a particular address block can be in any of the k replicas. (This is equivalent to a hash table with a bucket size of k.) There is a cache hit if the address matches any of the replicas.

A set associative cache avoids the problem of thrashing with a direct mapped cache, provided the working set for a given bucket is larger than k. Almost all hardware caches and TLBs today use set associative matching; an 8-way set associative cache structure is common.

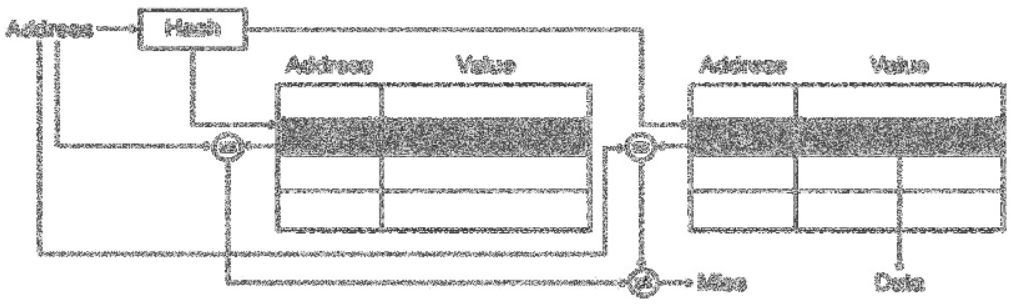


Figure 4.16: set associative cache

Discussion

Explain the Cache technique in web server prefetching

5.0 Conclusion

To make cache behavior more predictable and more effective, operating systems use a concept called page coloring. With page coloring, physical page frames are partitioned into sets based on which cache buckets they will use. For example, with a 2 MB 8-way set associative cache and 4

KB pages, there will be 64 separate sets, or colors. The operating system can then assign page frames to spread each application's data across the various colors.

6.0 Summary

Regardless of the program, a sufficiently large cache will have a high cache hit rate. In the limit, if the cache can fit all of the program's memory and data, the miss rate will be zero once the data are loaded into the cache. At the other extreme, a sufficiently small cache will have a very low cache hit rate.

7.0 References/Further Reading

Abraham, S., Peter, B. G., & Gagne, G. (2009). *Operating System Concepts* (C. Weisman (ed.); 8th ed.). John Wiley & Sons Inc.

Andrew S, T., & Bos, H. (2015). *Modern Operating Systems* (H. Marcia & J. Tracy (eds.); Fourth). Pearson Education.

Dhananjay, M. D. (2009). *Operating Systems A Concept-Based Approach* (B. Melinda (ed.)). McGraw-Hill Higher Education.

Thomas, A., & Dahlin, M. (2015). *Operating Systems Principles & Practice Volume III : Memory Management* (S. Kaplan & S. Whitney (eds.); Second). Recursive Books, Ltd.

Lookup Caches Overview. (2022, January 24). https://docs.informatica.com/data-integration/powercenter/10-2/transformation-guide/lookup-caches/lookup-caches-overview.html

Lutkevich, B. (n.d.). *What is Cache Memory? Cache Memory in Computers, Explained*. Retrieved May 1, 2022, from https://www.techtarget.com/searchstorage/definition/cache-memory

Memory Hierarchy Design and its Characteristics - GeeksforGeeks. (2018, December 17).

https://www.geeksforgeeks.org/memory-hierarchy-design-and-its-characteristics/

What is Memory Hierarchy: Definition, Diagram, Architecture and Advantages. (n.d.). Retrieved May 1, 2022, from https://www.elprocus.com/memory-hierarchy-in-computer-architecture/

Unit 4 Thrashing and Replacement Policies

Contents

1.0 Introduction

In case, if the page fault and swapping happens very frequently at a higher rate, then the operating system has to spend more time swapping these pages. This state in the operating system is termed thrashing. Because of thrashing the CPU utilization is going to be reduced. Replacement policies introduce solutions to thrashing problem.

2.0 Intended Learning Outcomes (ILOs)

At the end of this unit, student will able to

- Find out the cause of thrashing in operating system
- Demonstrate policies to address thrashing issues

3.0 Main Content

3.1 Thrashing

Thrash is the poor performance of a virtual memory (or paging) system when the same pages are being loaded repeatedly due to a lack of main memory to keep them in memory. Depending on the configuration and algorithm, the actual throughput of a system can degrade by multiple orders of magnitude. **Thrashing** occurs when a computer's virtual memory resources are overused, leading to a constant state of paging and page faults, inhibiting most application-level processing. It causes the performance of the computer to degrade or collapse. The situation can continue indefinitely until the user closes some running applications or the active processes free up additional virtual memory resources.

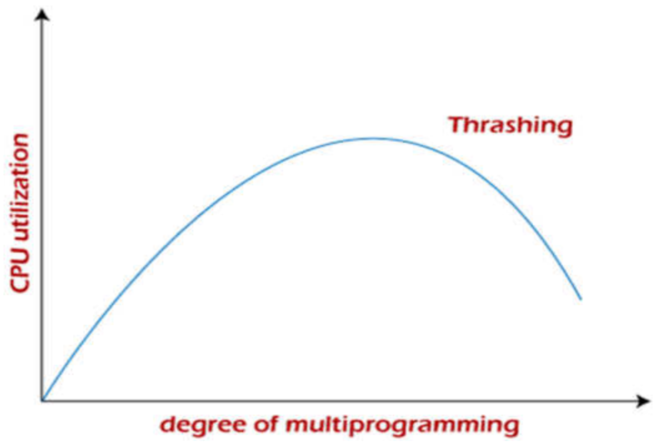


Figure 4.17: CPU utilization against degree of multiprogramming

Whenever thrashing starts, the operating system tries to apply either the Global page replacement Algorithm or the Local page replacement algorithm.

1. Global Page Replacement

Since global page replacement can bring any page, it tries to bring more pages whenever thrashing is found. But what actually will happen is that no process gets enough frames, and as a result, the thrashing will increase more and more. Therefore, the global page replacement algorithm is not suitable when thrashing happens.

2. Local Page Replacement

Unlike the global page replacement algorithm, local page replacement will select pages which only belong to that process. So there is a chance to reduce the thrashing. But it is proven that there are many disadvantages if we use local page replacement. Therefore, local page replacement is just an alternative to global page replacement in a thrashing scenario.

Causes of Thrashing

Programs or workloads may cause thrashing, and it results in severe performance problems, such as:

- If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new system. A global page replacement algorithm is used. The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming.
- CPU utilization is plotted against the degree of multiprogramming.
- As the degree of multiprogramming increases, CPU utilization also increases.
- If the degree of multiprogramming is increased further, thrashing sets in, and CPU utilization drops sharply.
- So, at this point, to increase CPU utilization and to stop thrashing, we must decrease the degree of multiprogramming.

How to Eliminate Thrashing

Thrashing has some negative impacts on hard drive health and system performance. Therefore, it is necessary to take some actions to avoid it. To resolve the problem of thrashing, here are the following methods, such as:

- **Adjust the swap file size:**If the system swap file is not configured correctly, disk thrashing can also happen to you.

- **Increase the amount of RAM:** As insufficient memory can cause disk thrashing, one solution is to add more RAM to the laptop. With more memory, your computer can handle tasks easily and don't have to work excessively. Generally, it is the best long-term solution.
- **Decrease the number of applications running on the computer:** If there are too many applications running in the background, your system resource will consume a lot. And the remaining system resource is slow that can result in thrashing. So while closing, some applications will release some resources so that you can avoid thrashing to some extent.
- **Replace programs:** Replace those programs that are heavy memory occupied with equivalents that use less memory.

3.2 Replacement Policies

Policies also vary depending on the setting: hardware caches use a different replacement policy than the operating system does in managing main memory as a cache for disk. A hardware cache will often have a limited number of replacement choices, constrained by the set associativity of the cache, and it must make its decisions very rapidly. Even within the operating system, the replacement policy for the file buffer cache is often different than the one used for demand paged virtual memory, depending on what information is easily available about the access pattern.

Page Fault – A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.

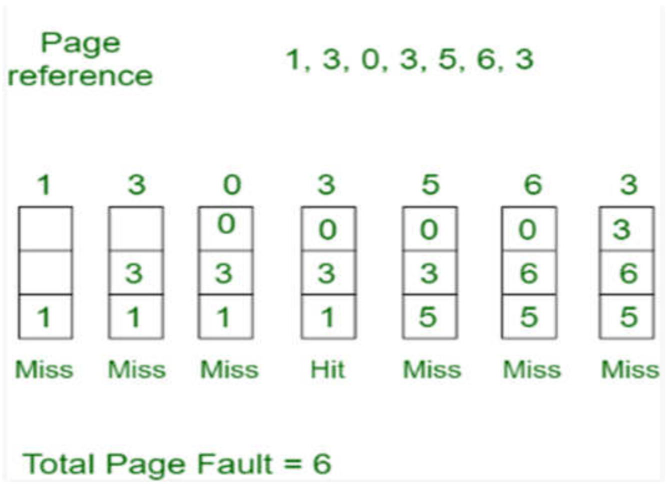
Since actual physical memory is much smaller than virtual memory, page faults happen. In case of page fault, Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

Page Replacement Algorithms

1. First In First Out(FIFO)

This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced, the page in the front of the queue is selected for removal.

Example-1Consider page reference string 1, 3, 0, 3, 5, 6, 3 with 3 page frames. Find number of page faults.



Initially all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> 3 Page Faults.

when 3 comes, it is already in memory so —> 0 Page Faults.

Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —>1 Page Fault.

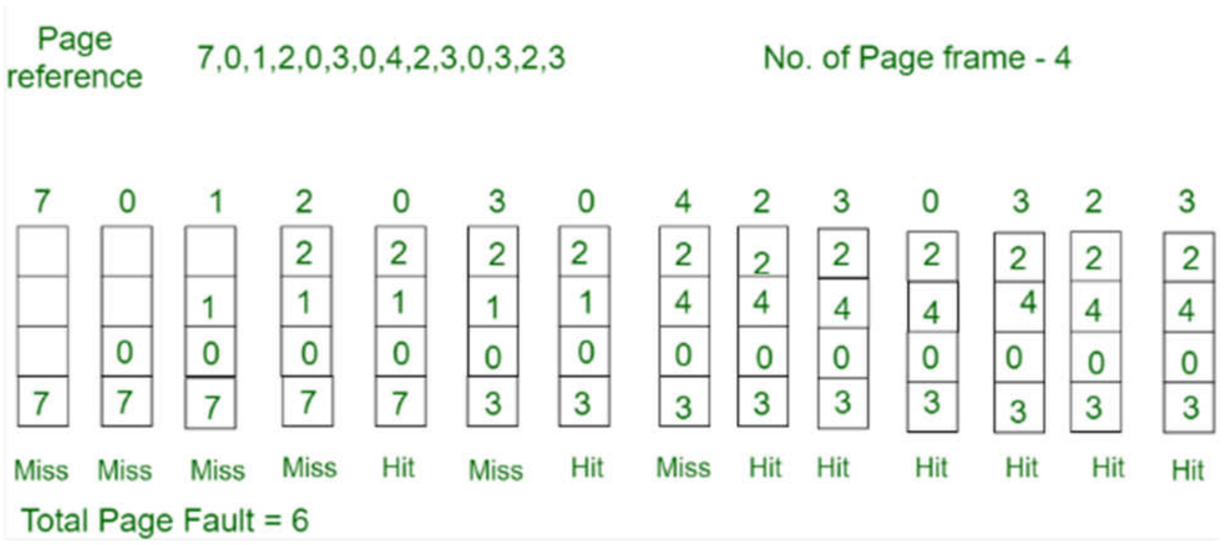
6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —>1 Page Fault.

Finally when 3 come it is not available so it replaces 0 1 page fault

2. Optimal Page replacement

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.

Example-2:Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, with 4 page frame. Find number of page fault.



Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> 4 Page faults

0 is already there so —> 0 Page fault.

when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future.—>1 Page fault.

0 is already there so —> 0 Page fault..

4 will takes place of 1 —> 1 Page Fault.

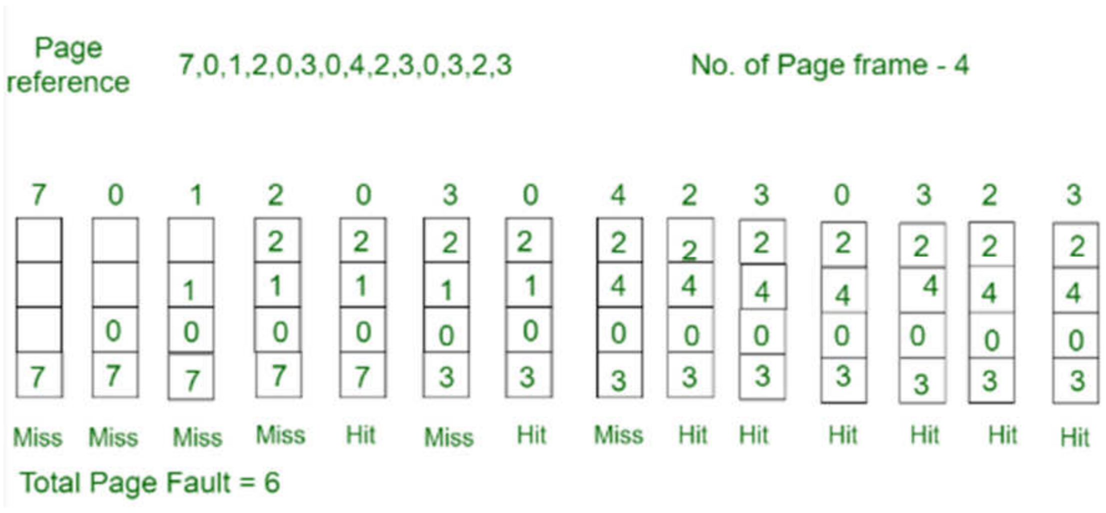
Now for the further page reference string —> 0 Page fault because they are already available in the memory.

Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

3. Least Recently Used

In this algorithm page will be replaced which is least recently used.

Example-3Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 with 4 page frames. Find number of page faults.



Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> 4 Page faults

0 is already their so —> 0 Page fault.

when 3 came it will take the place of 7 because it is least recently used —>1 Page fault

0 is already in memory so —> 0 Page fault.

4 will takes place of 1 —> 1 Page Fault

Now for the further page reference string —> 0 Page fault because they are already available in the memory.

Discussion

Discuss any other replacement algorithms you know

4.0 Self-Assessment/Exercises

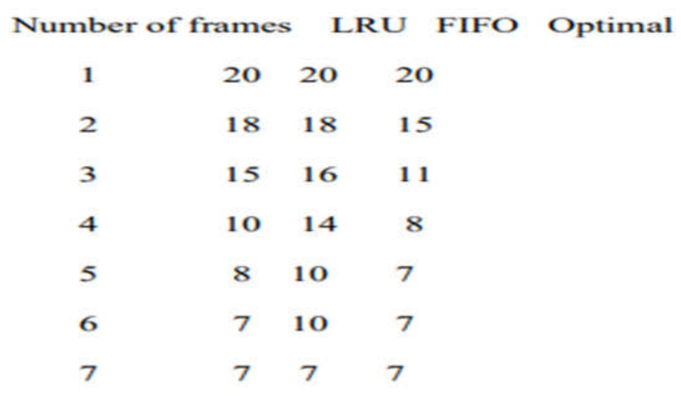
Consider the following page-reference string:

1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6

How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, or seven frames? Remember that all frames are initially empty, so your first unique pages will all cost one fault each.

- LRU replacement.
- FIFO replacement.
- Optimal replacement.

Answer



Number of frames	LRU	FIFO	Optimal
1	20	20	20
2	18	18	15
3	15	16	11
4	10	14	8
5	8	10	7
6	7	10	7
7	7	7	7

a. Define a page-replacement algorithm using this basic idea. Specifically address the problems of (1) what the initial value of the counters is, (2) when counters are increased, (3) when counters are decreased, and (4) how the page to be replaced is selected.

b. How many page faults occur for your algorithm for the following reference string, for four page frames? 1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.

c. What is the minimum number of page faults for an optimal page-replacement strategy for the reference string in part b with four page frames?

Answer

a. Define a page-replacement algorithm addressing the problems of:

i. Initial value of the counters: 0.

ii. Counters are increased whenever a new page is associated with that frame.

iii. Counters are decreased whenever one of the pages associated with that frame is no longer required. iv. How the page to be replaced is selected: find a frame with the smallest counter. Use FIFO for breaking ties.

b. 14 page faults

c. 11 page faults

5.0 Conclusion

Page replacement becomes necessary when a page fault occurs and there are no free page frames in memory. However, another page fault would arise if the replaced page is referenced again. Hence it is important to replace a page that is not likely to be referenced in the immediate future.

6.0 Summary

Policies also vary depending on the setting: hardware caches use a different replacement policy than the operating system does in managing main memory as a cache for disk. A hardware cache will often have a limited number of replacement choices, constrained by the set associativity of the cache, and it must make its decisions very rapidly. In the operating system, there is often both more time to make a choice and a much larger number cached items to consider; e.g., with 4 GB of memory, a system will have a million separate 4 KB pages to choose from when deciding which to replace.

7.0 References/Further Reading

Abraham, S., Peter, B. G., & Gagne, G. (2009). *Operating System Concepts* (C. Weisman (ed.); 8th ed.). John Wiley & Sons Inc.

Andrew S, T., & Bos, H. (2015). *Modern Operating Systems* (H. Marcia & J. Tracy (eds.); Fourth). Pearson Education.

Dhananjay, M. D. (2009). *Operating Systems A Concept-Based Approach* (B. Melinda (ed.)). McGraw-Hill Higher Education.

Thomas, A., & Dahlin, M. (2015). *Operating Systems Principles & Practice Volume III : Memory Management* (S. Kaplan & S. Whitney (eds.); Second). Recursive Books, Ltd.

Cache Replacement Policy - an overview | ScienceDirect Topics. (n.d.). Retrieved May 1, 2022, from https://www.sciencedirect.com/topics/computer-science/cache-replacement-policy

Kumari Riddhi. (2021). *Thrashing in OS (Operating System) - Scaler Topics*. https://www.scaler.com/topics/thrashing-in-os/

Techniques to handle Thrashing - GeeksforGeeks. (2022, March 11). https://www.geeksforgeeks.org/techniques-to-handle-thrashing/