BT 0073

OS and DBMS Lab Manual

Contents

Edition: Spring 2009

BKID – B0957 3rd Jan. 2009

Prof. V. B. Nanda Gopal
Director & Dean
Directorate of Distance Education
Sikkim Manipal University of Health, Medical & Technological Sciences (SMU DDE)

Board of Studies

Content Preparation Team

Content Writing
Mr. Vinayak G. Pai
Assistant Professor (IT)
Sikkim Manipal University – DDE
Manipal

Content Editing
Mr. Balasubramani R
Assistant Professor (IT)
Sikkim Manipal University – DDE
Manipal.

Edition: Spring 2009

GM, Manipal Universal Learning Pvt. Ltd., Manipal – 576 104. Printed at Manipal Press Limited, Manipal.

1. Introduction

Without its software, a computer is basically a lump of metal. With its software, a computer can store, process and retrieve information, find spelling errors in manuscripts, play games, and engage in many valuable activities to earn its keep computer software can be roughly divided into two kinds: the system programs, which manage the operation of the computer itself, and the application programs, which solves problem for their users. The most fundamental of all the system programs is the operating systems. Which controls all the computer resources and provides the base upon which the application programs can be written.

This book makes an attempt to give practical exposure in the implementation of Operating System and Database management system concepts. In addition, an effort has been made to brief some of the Operating System concepts. This lab Manual consists of 4 Operating System Experiments and 6 Database Management System Experiments.

The Operating System experiments Covers the concepts of how all computer resources are scheduled before use, what is critical section problem and how to solve the problem , concepts of Deadlock, Deadlock prevention, Deadlock avoidance , Deadlock detection and about Process synchronization. These experiments may be implemented using ‘C” programming Language.

The Database management system experiments should be implemented using MY-SQL. SQL is the most commonly used database definition and manipulation language in the world. MY-SQL only works on Windows-based platforms, including Windows 9x, Windows NT, Windows 2000 and Windows CE.

In comparison with SQL Server 2000, Oracle 9i Database supports all known platforms, including Windows-based platforms, AIX-Based Systems, Compaq Tru64 UNIX, HP 9000 Series HP-UX, Linux Intel, Sun Solaris and so on.

The DBMS Experiments Covers the concepts of Creation of Table, inserting data into the table, Report generation, Writing SQL Queries and Cursor Management with PL SQL.

This Manual covers the portion of the subject concerning database design as well as implementation from the user's viewpoint and, thus gives Practical exposure to develop large project in the area of Database Management System.

2. Objectives

The objective of this manual is to make the student to get hands on experience regarding the application of theoretical concepts learnt in this semester. It covers Operating system algorithms and SQL Experiments.

The set of experiments included as part of this lab manual intend to make the students more practical oriented and also to pave the way for using SQL programming language for real time business applications.

The manual provides the user with the following details regarding the experiments to be completed:

1. Problem Statement
2. Algorithms

The time required by the students to complete each and every experiment is also mentioned against the experiment.

The format for preparation and submission of the experiments at the Learning Centers would be as follows and it should be placed as the '**Table of Contents**' page in the Record Note Book.

3. Software and Hardware Requirements

– Software Requirements: (Minimum)

- Turbo C
- MSSQL 2000
- MySQL
- MS – Access

– Hardware Requirements: (Minimum)

- PIV Computer
- 512 MB RAM
- 40GB Hard Disk

4. Methodology of Conduct of Practical Exercises

The course titled "OS & DBMS Lab" bearing the subject code BT0073 has 2 credit weight and the practical exercises have to be performed for duration of 60 Hours. It contains 10 Exercises as given in the Table 1.

Guided and Unguided Exercises

The time allocated for various components is given below. The practical component of "BT0073 OS & DBMS Lab" has to be completed within 7 days of 14 sessions; each session has to be for a minimum of 3 hours. You are advised to consult the LC Faculty for the detailed practical schedule.

Type of Activity	Day(s)	Hours	Sessions
General Practice	1	6	2
Guided Exercises	5.5	33	11
Unguided	0.5	3	1
Self Study and Preparation of Record	–	18	–
Total	**7**	**60**	**14**

In the course of these Exercises, the Learning Centre Faculty will provide you guidance, arrange for demonstration (wherever necessary), and clarify your doubts. The following courses deal with the theoretical aspects and sample exercises for reference:

1. BT0070 – Operating Systems
2. BT0066 – DBMS

You are required to follow the instruction provided by the LC Faculty and observe the demonstration exercises, carry out the exercises, prepare the Practical record book in accordance with the guidelines mentioned and submit the Practical record book to your Learning Centre Faculty. Each Exercise has assessment component, which will be carried out by your LC Faculty. Your attendance is compulsory for these Exercises.

The assessment during the Guided sessions accounts for 70 marks (out of a total of 100 marks for 2 credit Practical subject). In short, during these Guided Exercises, your Learning Centre faculty will offer you guidance and also perform assessment of your work.

The end semester examination will be conducted by the CoE for three hours for 30 marks, which forms the **Unguided Exercise(s) (UGE),** where one or more problem(s) will be assigned to you. The exercise prescribed as unguided could be based on the concepts underlying guided exercises, but need not be exactly the same.

To complete the lab course successfully, you have to score a combined average of 40% (i.e., 40 marks) in both Guided and Unguided parts together and a minimum of 35% in each part (i.e., 25 in marks in Guided part and 11 marks in Unguided part). In case of failure in either or both the parts, you have to redo the concerned part(s).

5. Instructions

- Each experiment should be performed based on the algorithm outlined for each experiment.
- The faculty or the instructor at the LC should demonstrate the execution process of the software involved.
- The student should be motivated to produce different outputs from the program by providing varying inputs to each experiment and these outputs should be recorded in the individual student report.

All the student records should contain the following details with respect to each and every experiment.

- **Aim:** It should contain a one or two line description of the problem being solved.
- **Objective:** Should contain the step-by-step procedure or methods followed for completing the experiment.
- **Algorithm:** A pseudo-code should be written presenting the steps involved in solving the problem.
- The detailed executable version of the program should be written by hand and a print out of these executable versions along with sample outputs should be taken; which should be signed by the concerned authority on the same date of experiment and other details should be recorded as per the table mentioned above.
- A record of work should be submitted at the end semester examination.

6. Exercises

The following is the list of experiments to be completed by the students within the timeline specified by the Learning Center. The detailed algorithm is outlined in the exercises given below.

S.No.	Name of the Experiment	Date	Time	Status	Remarks
1.	First Come First Served scheduling algorithm				
2.	The critical-section problem				
3.	Banker's Algorithm				
4.	The Dining Philosopher's Problem				
5.	Creation of tables				
6.	Report Generation				
7.	A PL/SQL program involving simple SQL Queries				
8.	A PL/SQL program using aggregate functions				
9.	Cursor Management				
10.	Library Management System				

Faculty Name: **Faculty Signature:**

Lab Administrator Name: **Lab Administrator Signature:**

***External Examiner Name:** ***External Examiner Signature:**

(* To be done at the time of final external examination)

1. First Come First Served scheduling

(Refer to Unit No. 3 titled “CPU Scheduling Algorithms”)

Problem Statement: Implement the FCFS Problem using C language constructs.

Algorithm:

This is one of the very *brute force algorithms*. A process that requests for the CPU first is allocated the CPU first. Hence, the name first come first serve.

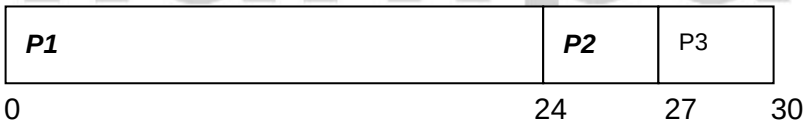
The FCFS algorithm is implemented by using a first-in-first-out (FIFO) queue structure for the ready queue. This queue has a head and a tail. When a process joins the ready queue its PCB is linked to the tail of the FIFO queue. When the CPU is idle, the process at the head of the FIFO queue is allocated the CPU and deleted from the queue.

Even though the algorithm is simple, the average waiting is often quite long and varies substantially if the CPU burst times vary greatly, as seen in the following example.

Numerical Example: Consider a set of three processes P1, P2 and P3 arriving at time instant 0 and having CPU burst times as shown below:

Process	Burst time (msecs)
P1	24
P2	3
P3	3

The Gantt chart below shows the result.



Average waiting time and average turnaround time are calculated as follows:

The waiting time for process P1 = 0 msecs
P2 = 24 msecs
P3 = 27 msecs

Average waiting time = (0 + 24 + 27) / 3 = 51 / 3 = 17 msecs.

P1 completes at the end of 24 msecs, P2 at the end of 27 msecs and P3 at the end of 30 msecs. Average turnaround time = (24 + 27 + 30) / 3 = 81 / 3 = 27 msecs.

2. The Critical – Section problem

(Refer to Unit No. 4 titled “Process Synchronization”)

Problem Statement: Implement the Critical Section problem using C Language as per the algorithm given below.

Algorithm

A **critical-section** is a part of program that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one process of execution. Consider a system consisting of n processes { p0,p1,……pn-1}. Each process has a segment of code called a critical-section. The important feature of the system is that, when one process is executing in its critical-section, no other process is to be allowed to execute in its critical-section. Thus the execution of critical-sections by the processes is mutually exclusive in time. The critical-section problem is to design a protocol that the processes can use to co-operate. Each process must request permission to enter its critical-section. The section of the 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.

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

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 there exist some processes that are not executing in their remainder section can participate in the decision of which will enter in its critical-section next, and this selection cannot be postponed indefinitely.

3. Bounded Waiting: There exist a bound 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.

When presenting an algorithm, we define only the variables used for synchronization purposes, and describe only a typical process pi whose general structure is shown in figure below: The entry section and exit section are enclosed in boxes to highlight these important segments of code.

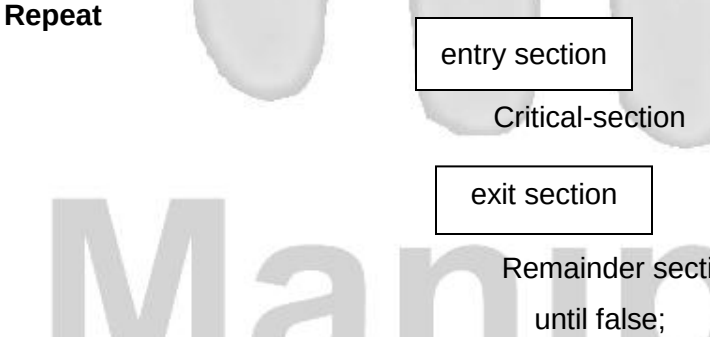


Figure: General structure of a typical process pi

The processes are numbered p0 and p1. For convenience, when presenting pi, we use pj to denote the other process; that is j=1-i.

Algorithm Implementation

Let the processes share a common variable turn initialized to 0 (or 1). If turn =i, then process pi is allowed to execute in its critical-section. The structure of process pi is shown in figure below.

This solution ensures that only one process at a time can be in its critical-section. But it does not satisfy the progress requirement, since it requires strict alternation of processes in the execution of the critical-section. For example, if turn=0 and p1 is ready to enter its critical-section, p1 cannot do so, even though p0 may be in its remainder section.

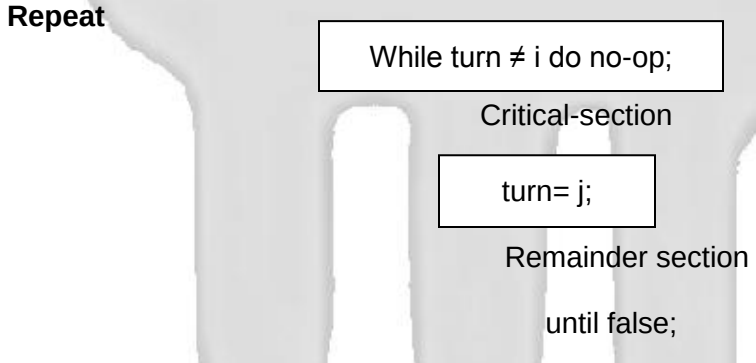


Figure: The structure of process pi in the algorithm

3. Banker's Algorithm

(Refer to Unit No. 5 titled " Introduction to Deadlocks ")

Problem Statement: Implement Banker's algorithm using C Language as per the algorithm given below.

A deadlock avoidance algorithm requires each process to make known in advance the maximum number of resources of each type that it may need.

Also known is the maximum number of resources of each type available. Using both the above a priori knowledge, deadlock avoidance algorithm ensures that a circular wait condition never occurs.

Safe State: A system is said to be in a safe state if it can allocate resources upto the maximum available and is not in a state of deadlock. A safe sequence of processes always ensures a safe state. A sequence of processes $< P_1, P_2, ..., P_n >$ is safe for the current allocation of resources to processes if resource requests from each P_i can be satisfied from the currently available resources and the resources held by all P_j where $j < i$. If the state is safe then P_i requesting for resources can wait till P_j's have completed. If such a safe sequence does not exist, then the system is in an unsafe state.

The resource allocation graph algorithm is not applicable where resources have multiple instances. In such a case Banker's algorithm is used.

A new process entering the system must make known a priori the maximum instances of each resource that it needs subject to the maximum available for each type. As execution proceeds and requests are made, the system checks to see if the allocation of the requested resources ensures a safe state. If so only are the allocations made, else processes must wait for resources.

The following are the data structures maintained to implement the Banker's algorithm:

1. **n:** Number of processes in the system.
2. **m:** Number of resource types in the system.
3. **Available:** is a vector of length m. Each entry in this vector gives maximum instances of a resource type that are available at the instant. Available[j] = k means to say there are k instances of the jth resource type R_j.

4. **Max:** is a demand vector of size n x m. It defines the maximum needs of each resource by the process. Max[i][j] = k says the ith process P_i can request for atmost k instances of the jth resource type R_j.
5. **Allocation:** is a n x m vector which at any instant defines the number of resources of each type currently allocated to each of the m processes. If Allocation[i][j] = k then ith process P_i is currently holding k instances of the jth resource type R_j.
6. **Need:** is also a n x m vector which gives the remaining needs of the processes. Need[i][j] = k means the ith process P_i still needs k more instances of the jth resource type R_j. Thus Need[i][j] = Max[i][j] – Allocation[i][j].

Safety Algorithm

Using the above defined data structures, the Banker's algorithm to find out if a system is in a safe state or not is described below:

1. Define a vector Work of length m and a vector Finish of length n.
2. Initialize Work = Available and Finish[i] = false for i = 1, 2, ., n.
3. Find an i such that
   a. Finish[i] = false and
   b. $Need_i$ <= Work ($Need_i$ represents the ith row of the vector Need).
      If such an i does not exist , go to step 5.
4. Work = Work + $Allocation_i$
   Go to step 3.
5. If finish[i] = true for all i, then the system is in a safe state.

Resource-Request Algorithm

Let $Request_i$ be the vector representing the requests from a process P_i. Requesti[j] = k shows that process P_i wants k instances of the resource type R_j. The following is the algorithm to find out if a request by a process can immediately be granted:

1. If $Request_i$ <= $Need_i$, go to step 2.
   else Error "request of P_i exceeds Max_i".
2. If $Request_i$ <= $Available_i$, go to step 3.
   else P_i must wait for resources to be released.
3. An assumed allocation is made as follows:
   Available = Available – $Request_i$
   $Allocation_i$ = $Allocation_i$ + $Request_i$
   $Need_i$ = $Need_i$ – $Request_i$

If the resulting state is safe, then process P_i is allocated the resources and the above changes are made permanent. If the new state is unsafe, then P_i must wait and the old status of the data structures is restored.

4. The Dining Philosopher's Problem

(Refer to Unit No. 4 titled "Process Synchronization ")

Problem Statement: Implement the Dining Philosopher's problem using C Language as per the algorithm given below.

Algorithm

The dining philosophers problem is summarized as five philosophers sitting at a table doing one of two things: eating or thinking. While eating, they are not thinking, and while thinking, they are not eating. The five philosophers sit at a circular table with a large bowl of spaghetti in the center. A fork is placed in between each philosopher, and as such, each philosopher has one fork to his or her left and one fork to his or her right. As spaghetti is difficult to serve and eat with a single fork, it is assumed that a philosopher must eat with two forks. The philosopher can only use the fork on his or her immediate left or right.

Illustration :

The dining philosophers problem is sometimes explained using rice and chopsticks rather than spaghetti and forks, as it is more intuitively obvious that two chopsticks are required to begin eating.

The philosophers never speak to each other, which creates a dangerous possibility of deadlock when every philosopher holds a left fork and waits perpetually for a right fork (or vice versa).

Originally used as a means of illustrating the problem of deadlock, this system reaches deadlock when there is a 'cycle of unwarranted requests'. In this case philosopher P_1 waits for the fork grabbed by philosopher P_2 who is waiting for the fork of philosopher P_3 and so forth, making a circular chain.

Starvation: (and the pun was intended in the original problem description) might also occur independently of deadlock if a philosopher is unable to acquire both forks due to a timing issue. For example there might be a rule that the philosophers put down a fork after waiting five minutes for the other fork to become available and wait a further five minutes before making their next attempt. This scheme eliminates the possibility of deadlock (the system can always advance to a different state) but still suffers from the problem of livelock. If all five philosophers appear in the dining room at *exactly* the same time and each picks up their left fork at the same time the philosophers will wait five minutes until they all put their forks down and then wait a further five minutes before they all pick them up again.

The lack of available forks is an analogy to the lacking of shared resources in real computer programming, a situation known as concurrency. Locking a resource is a common technique to ensure the resource is accessed by only one program or chunk of code at a time. When the resource a program is interested in is already locked by another one, the program waits until it is

unlocked. When several programs are involved in locking resources, deadlock might happen, depending on the circumstances. For example, one program needs two files to process. When two such programs lock one file each, both programs wait for the other one to unlock the other file, which will never happen.

In general the dining philosophers problem is a generic and abstract problem used for explaining various issues which arise in problems which hold mutual exclusion as a core idea. For example, as in the above case deadlock/livelock is well explained with the dining philosophers problem.

Waiter solution

A simple solution is achieved by introducing a waiter at the table. Philosophers must ask his permission before taking up any forks. Because the waiter is aware of which forks are in use, he is able to arbitrate and prevent deadlock. When four of the forks are in use, the next philosopher to request one has to wait for the waiter's permission, which is not given until a fork has been released. The logic is kept simple by specifying that philosophers always seek to pick up their left hand fork before their right hand fork (or vice versa).

To illustrate how this works, consider the philosophers are labeled clockwise from A to E. If A and C are eating, four forks are in use. B sits between A and C so has neither fork available, whereas D and E have one unused fork between them. Suppose D wants to eat. Were he to take up the fifth fork, deadlock becomes likely. If instead he asks the waiter and is told to wait, we can be sure that next time two forks are released there will certainly be at least one philosopher who could successfully request a pair of forks. Therefore deadlock cannot happen.

Database Management Systems Lab

Note: Any database software can be used. For example, Oracle, SQL Server, or MS-Access.

5. Creation of tables

(Refer to unit no 6 titled “An Introduction to SQL”)

Problem Statement:

Given the following relations:

Emp Table (Emp#, Empname, City, Acc#)

Acc Table (Acc#, OpenDate, Bal-Amt)

Write corresponding create table statements, insert data into the relations and show the output.

Syntax:

```
CREATE TABLE TableName
{(colName dataType [NOT NULL] [UNIQUE]
[DEFAULT defaultOption]
[CHECK searchCondition] [,...]}
[PRIMARY KEY (listOfColumns),]
{[UNIQUE (listOfColumns),] […,]}
{[FOREIGN KEY (listOfFKColumns)
REFERENCES ParentTableName [(listOfCKColumns)],
[ON UPDATE referentialAction]
[ON DELETE referentialAction ]] [,…]}
{[CHECK (searchCondition)] [,…] })
```

Insert Command Syntax

1. Insert into<table name>values (value1, value2, value3….);
2. Insert into <table name> [(<column1> [, column2>…])] values (<values1> [, <value2>…]);
3. Insert into <table name> values (<&col1> [&col2>,<&col3>,….);

6. Report Generation

(Refer to unit no 7 titled “Relational Algebra”)

Generate the yearly report (from first January of an year to 31st December of the same year) on the total amounts of individual accounts of each employee in the following format:

Emp Table (Emp#, Empname, City, Acc#)

Acc Table (Acc#, OpenDate, Bal-Amt)

Yearly Report

Emp#	Empname	Acc#	Bal-Amt

Explanation:

1. Use Join conditions
2. Use Date Functions for range date options

Function	Description
DateDiff("d", [OrderDate], [ShippedDate])	Displays the variance in days between the values of the OrderDate and ShippedDate fields.
DateDiff("yyyy", [Birthday1], [Birthday2])	Displays the variance in years between the values of the Birthday1 and Birthday2 fields.
DateDiff("m", #12/24/2000#, #11/26/2000#)	Displays the variance in months between the two dates. The expression evaluates to -1, since the first date falls after the second date.
DateDiff("yyyy", #12/31/2000#, #1/1/2001#)	Displays the variance in years between the two dates. The expression evaluates to 1, even though only a day has elapsed.

Note: Above syntax corresponds to the SQL functions of Dates used in MS-Access 2003. For others refer to the corresponding help files or manuals of the respective software.

7. A PL/SQL program involving simple SQL Queries

(Refer to unit no 6 titled “An Introduction to SQL”)

Problem Statement:

Write PL/SQL program to create a Employee table with the fields EmpNo, Empname, Designation, Salary, and Dateofjoin . Insert 10 records and display the records of only those employee whose salary is more than 15,000.

Algorithm

1. start
2. Create a table with a name Employee with the fields EmpNo, Empname, Designation, Salary, and Dateofjoin.
3. Insert 10 different records
4. Write a query with where clause to get the desired output.
5. Display the result.
6. Stop

8. Aggregate Functions

(Refer to unit no 6 titled “An Introduction to SQL”)

Problem Statement:

Write a PL /SQL program to calculate the total ,average and grade of given student.

Algorithm

1. Start
2. Create a table student with the fields name,rollno,mark1,mark2,mark3.
3. Insert the values into the table using insert command.
4. Write a PL /SQL query to find the total , average and grade of the given student.
5. Display the result.
6. Stop.

9. Cursor Management

(Refer to unit no 6 titled “An Introduction to SQL”)

Problem Statement:

Write a PL /SQL program to calculate the total ,average and grade of students using cursors.

Algorithm

1. Start
2. Create a table student with the fields name,rollno,mark1,mark2,mark3.
3. Insert the values into the table using insert command.
4. Create a table report with the fields name , rollno, total, average, grade.
5. Write a PL /SQL Query to find the total , average and grade using cursors.
6. Insert that in the second table.
7. Display the result.
8. Stop.

10. Library Management System

(Refer to unit no 5 titled “An Introduction to RDBMS ”)

Problem Statement:

Design and implement library management system in RDBMS

Algorithm

1. Collect the essential requirements for library management system such as student details, book details, issue.
2. Define the entity sets and the attributes for library management system
   Student details – studname, studno,
   Book details – bookno, title, authorname, booktype.
3. Define the Relationship sets such as lender, borrower, issue.
4. Represent the strong and weak entity sets
5. Design E-R diagram for library management system

Reduce the E-R schema of library management system into tables using generalization and aggregation.
