Savitribai Phule Pune University

T. Y. B. C. A. (Science)

Semester-V (2019 Pattern)

Lab Course–III

(BCA-358)DSEIII Laboratory

(Operating System)

WorkBook

Name: _____

College Name: _____

Roll No.: _____ **Division:** _____

Academic Year _____

From the Chairman's Desk

It gives me a great pleasure to present this workbook prepared by the Board of studies in Computer Applications.

The workbook has been prepared with the objectives of bringing uniformity in implementation of lab assignments across all affiliated colleges, act as a ready reference for both fast and slow learners and facilitate continuous assessment using clearly defined rubrics.

The workbook provides, for each of the assignments, the aims, pre-requisites, related theoretical concepts with suitable examples wherever necessary, guidelines for the faculty/lab administrator, instructions for the students to perform assignments and a set of exercises divided into three sets.

I am thankful to the Chief Editor and the entire team of editors appointed. I am also thankful to the members of BOS. I thank everyone who have contributed directly or indirectly for the preparation of the workbook.

Constructive criticism is welcome and to be communicated to the Chief Editor. Affiliated colleges are requested to collect feedbacks from the students for the further improvements.

I am thankful to Hon. Vice Chancellor of Savitribai Phule Pune University Prof. Dr. Nitin Karmalkar and the Dean of Faculty of Science and Technology Prof. Dr. M G Chaskar for their support and guidance.

Prof. Dr. S S Sane
Chairman, BOS in Computer Applications
SPPU,Pune

Chairperson and Editor:

Dr. Manisha Bharambe - MES Abasaheb Garware College, Pune.

Prepared and Compiled by:

Sr.No.	Assignment Name	Teacher Name	College Name
1.	Assignment 1: Operations on processes	Prof.Rohini Kapse	KRT Arts, BH Commerce and AM Science (KTHM) College, Nashik
2.	Assignment 2: CPU Scheduling	Prof. Sampada Vaishampayan	MES Abasaheb Garware college of arts and science, Pune
3.	Assignment 3: Deadlock detection and avoidance-	Prof.Mayuri Desari	Abeda Inamdar College.Pune
4.	Assignment 4: Page Replacement Algorithms: FIFO, Optimal, LRU	Prof. Niket Tajne	PES Modern College Ganeshkhind
5.	Assignment 5: A* and AO* Algorithm	Prof. Suvarna S. Patil & Prof.Meenal Jabde	B.J.S. Arts, Science & Commerce College ,Wagholi, PES Modern College Ganeshkhind

Reviewed By:

1. Dr. Madhukar Shelar — KTHM College, Nashik02
2. Dr. Pallavi Bulakh — PES Modern CollegeOf Art Science and Commerce Ganeshkhind
3. Dr. Sayyed Razak — Ahmednagar College , Ahmednagar

Introduction

About the workbook:

This workbook is intended to be used by T.Y.B.C.A. (Science) students for the BCA358 –Operating system & AI Laboratory Assignments in Semester–V. This workbook is designed by considering all the practical concepts topics mentioned in syllabus.

1. **The objectives of this workbook are:**

1) Defining the scope of the course.
2) To bring the uniformity in the practical conduction and implementation in all colleges affiliated to SPPU.
3) To have continuous assessment of the course and students.
4) Providing ready reference for the students during practical implementation.
5) Provide more options to students so that they can have good practice before facing the examination.
6) Catering to the demand of slow and fast learners and accordingly providing the practice assignments to them.

2. **How to use this workbook:**
   The workbook is divided into5assignments.Each assignment has three SETs .It is mandatory for students to complete SETA ,SETB and SET C in given slot.

Instructions to the students

Please read the following instructions carefully and follow them.

Students are expected to carry this book every time when they come to the lab for computer science practical.

1. Students should prepare themselves before hand for the Assignment by reading these material.
2. Instructor will specify which problems to solve in the lab during the allotted slot and student should complete the verified by the instructor .How every student should spend additional hours in Lab and at home to cover as many problems as possible given in this workbook.
3. Students will be assessed for each exercise on as cale from0 to5.

Notdone	0
Incomplete	1
LateComplete	2
Needsimprovement	3
Complete	4
WellDone	5

Guide lines for Instructors

1. Explain the assignment and related concepts in around ten minute so using whiteboard if required or by demonstrating the software.
2. Youshouldevaluateeachassignmentcarriedoutbyastudentonascaleof5asspecifiedabove by ticking appropriate box.
3. Thevalueshouldalsobeenteredonassignmentcompletionpageoftherespective Labcourse.

Guide lines for Lab Administrator

You have to ensure appropriate hardware and software is available to each student in the Lab.
The operating system and software requirement so server side and also client side areas given below:

1) Server and ClientSide-(OperatingSystem)Linux (Ubuntu/RedHat/Fedora/ Centos)
2) Jupyter notebook or any IDLE of python on any platform.

INDEX

Assignment Completion Sheet

Subject:BCA-358 DSEIII Laboratory(Operating System)			
Sr.No.	**Assignment Name**	**Marks (Out of 5)**	**Teacher's Sign**
1	Operations on processes		
2	CPU Scheduling		
3	Deadlock detection and avoidance		
4	Page Replacement Algorithms:FIFO, Optimal, LRU		
5	A* and AO* Algorithm		
6	QUIZ/ Viva		
Total Out of 30 Total Out of 15			

CERTIFICATE

This is to certify that

Mr./Ms.__ _____ _____

Has successfully completed ***BCA-358DSEIII Laboratory(Operating System)*** *course in the year_____and his/her seat number is_____ .*

He/She has scored mark__out of 15.

Instructor ***H.O.D./Coordinator***

Internal Examiner ***External Examiner***

Assignment 1: Operations on Processes

Process: A process is a program in execution. For execution of the program, it is loaded in computer's memory and it becomes a Process.A program is a passive entity and process is an active entity. A process performs all the tasks mentioned in the program. A process can be divided into foursections ─ stack, heap, text and data.

Stack: The process Stack contains the temporary data such as method/function parameters, return address and local variables.

Heap: This is dynamically allocated memory to a process during its run time.

Text: This includes all instructions specified in the program.

Data: This section contains the global and static variables.

In Linux operating system, new processes are created through fork() system call.

Fork() System Call:

To create a new process fork() system call is used. It takes no arguments and returns a process ID. The process that creates new process is called as **Parent Process** while newly created process is called as **Child Process**. Child process is exact copy of parent process. Syntax of this system call is

int fork()

This system call is available in library file <**unistd.h**>. This system call return following values.

1) returns the value 0 to the newly created child process

2) returns Process ID of child process to parent process.

3) returns -1 if fork fail to create child process or on error.

Both parent and child processes continue executing with the instruction that follows the call to fork. Fork system call is often followed by exec call.

The returned process ID is of type pid_t defined in sys/types.h. Normally, the process ID is an

integer. Moreover, a process can use function getpid() to retrieve the process ID assigned to thisprocess.

If the call to fork() is executed successfully, Linux will

- Make two identical copies of address spaces, one for the parent and the other for the child.
- Both processes will start their execution at the next statement following the fork() call.

Program 1 : Consider the following example:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        // make two process which run same
        // program after this instruction
        fork();
        printf("Hello \n");
        return 0;
}
```

Output of above program:

Hello

Hello

Here printf() statement after fork() system call executed by both parent and childprocess. Both processes start their execution right after the system call fork(). Since bothprocesses have identical but separate address spaces, those variables initialized before the fork()call have the same values in both address spaces. Since every process has its own address space,any modifications will be independent of the others. In other words, if the parent changes the

value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by fork() calls will not be affected even though theyhave identical variable names.

Program 2: Run the below given program and see the output(Use of getpid() function)

```
#include <stdio.h>
#include <unistd.h>
#include<stdlib.h>
int main()
{
intpid;
pid=fork();
if(pid<0)
```

```
    {
printf("\n Error in creation of Child Process ");
exit(1);
    }
else if(pid==0)
    {
printf("\n Hello, I am the child process ");
printf("\n My pid is %d ",getpid());
exit(0);
    }
else
    {
printf("\n Hello, I am the parent process ");
printf("\n My pid is %d \n ",getpid());
exit(1);
    }
}
```

exec() system call

The exec family of system calls replaces the program executed by a process. When a process calls exec, all code (text) and data in the process is replaced with the executable of the new program. It loads the program into the current process space and runs it from the entry point. Theprocess id PID is not changedbecause we are not creating a new process, we are just replacing a process with anotherprocess in exec.

The exec() family consists of following functions,

execl(): l is for the command line arguments passed a list to the function.

Syntax: intexecl(const char *path, const char *arg, ...);

execlp(): p is the path environment variable which helps to find the file passed as an argument

to be loaded into process.

Syntax: intexeclp(const char *file, const char *arg, ...);

execle(): It is an array of pointers that points to environment variables and is passed explicitly tothe newly loaded process.

Syntax: intexecle(const char *path, const char *arg, ..., char * constenvp[]);

execv(): v is for the command line arguments. These are passed as an array of pointers to the function.

Syntax: intexecv(const char *path, char *constargv[]);

execlp() System Call:

execlp() system call is used after a fork() call by one of the two processes to replace the processes memory space with a new program. This call loads a binary file into memory and starts its execution. So two processes can be easily communicates with each other.

Program 3:

```
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
int main()
{
intpid;
pid = fork(); /* fork a child process */
if (pid< 0) /* error occurred */
{
        fprintf(stderr, "Fork Failed");
        return 1;
}
else if (pid == 0) /* child process */
        execlp("/bin/wc","wc",NULL);
else /* parent process */
{
        wait(NULL); /* parent will wait for the child to complete    */
        printf("Child Complete");
}
return 0;
}
```

fork() vs exec()

- Fork() starts a new process which is a copy of the one that calls it, while exec() replacesthe current process image with another (different) one.
- Both parent and child processes are executed simultaneously in case of fork() while

Control never returns to the original program unless there is an exec() error.

Sleep() System Call:

This system call takes a time value as a parameter, specifying the minimum amount of time that the process is to sleep before resuming execution. The parameter typically specifies seconds, although some operating systems provide finer resolution, such as milliseconds or microseconds.

Wait() system call

The wait() system call suspends execution of the calling process until one of its children terminates.

wait(int&status);

waitpid() system call :

By using this system call it is possible for parent process to synchronize its execution with child

process. Kernel will block the execution of a Process that calls waitpid system call if a some child of that process is in execution. It returns immediately when child has terminated by return

termination status of a child.

Syntax :intwaitpid(intpid, int *status, int options);

nice() System Call:

Using nice() system call, we can change the priority of the process in multi-taskingsystem. The new priority number is added to the already existing value.

int nice(intinc);

nice() adds inc to the nice value. A higher nice value means a lower priority. The range of the nice value is +19 (low priority) to -20 (high priority).

Program 4:

```
#include<stdio.h>
main()
```

```
{
intpid, retnice;
printf("press DEL to stop process \n");
pid=fork();
for(;;)
{
if(pid == 0)
{
retnice = nice (−5);
print("child gets higher CPU priority %d \n", retnice);
sleep(1);
}
else
{
retnice=nice(4);
print("Parent gets lower CPU priority %d \n", retnice);
sleep(1);
}
}
}
```

Orphan process

Orphan processes are those processes that are still running even though their parent process has terminated or finished. A process can be orphaned intentionally or unintentionally. Usually, a parent process waits for its child to terminate or finish their joband report to it after execution but if parent fails to do so its child results in the Orphan process.

In most cases, the Orphan process is immediately adopted by the init process (a very first processof the system).

Program 5:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
```

```
{
intpid = fork();
if (pid> 0) {
      //getpid() returns process id and getppid() will return parent process id
printf("Parent process\n");
printf("ID : %d\n\n", getpid());
   }
else if (pid == 0) {
printf("Child process\n");
      // getpid() will return process id of child process
printf("ID: %d\n", getpid());
      // getppid() will return parent process id of child process
printf("Parent -ID: %d\n\n", getppid());
sleep(10);
      // As this time parent process has finished, it will show different parent process
id
printf("\nChild process \n");
printf("ID: %d\n", getpid());
printf("Parent -ID: %d\n", getppid());
   }
else {
printf("Failed to create child process");
   }

return 0;
}
```

Clock() function: The clock function is used to determine the processor time in executing a program or part of a program. The header file <time.h> should be included in the program for its application. The function prototype is given below.

clock_t clock(void);

The function returns a value of *type* c1ock_t which should be divided by the value of the macroCLOCK_PER_SEC to obtain the processor time in seconds.

Shell:

Shell is an interface between user and operating system. It is the command interpreter, which accept the command name (program name) from user and executes that command/program. Shell mostly accepts the commands given by user from keyboard. Shell gets started automatically when Operating system is successfully started.When shell is started successfully it generally display some prompt (such as #,$ etc) to accept and execute the command.Shell executes the commands either synchronously or asynchronously.

When shell accepts the command then it locates the program file for that command, start its execution, wait for the program associated with the command to complete its execution and thendisplay the prompt again to accept further command. This is called as Synchronous execution ofshell.

In asynchronous execution shell accept the command from user, start the execution of program

associated to the given command but does not wait for that program to finish its execution, display prompt to accept next command.

How Shell Execute the command?

1. Accept the command from user.
2. Tokenize the different parts of command.
3. First word on the given command line is always a command name.
4. Creates (Forks) a child process for executing the program associated with given command.
5. Once child process is created successfully then it loads (exec) the binary (executable) imageof given program in child process area.
6. Once the child process is loaded with given program it will start its execution while shell iswaiting (wait) for it (child) to complete the execution. Shell will wait until child finish itsexecution.
7. Once child finish the execution then Shell wakeup, display the command prompt again andaccept the command and continue.

Example

$ cat file1.dat file2.dat file3.dat

This command line has four tokens- cat, file1.dat, file2.dat and file3.dat. First token is the command name which is to be executed. In Linux operating system, some commands are internally coded and implemented by shell such as mkdir, rmdir, cd, pwd, ls, cat, grep,

who etc.

Objective of this practical assignment is to simulate the shell which interprets all internal or predefined Linux commands and additionally implement to interpret following extended commands.

(1) Count: To count and display the number of lines, words and characters in a given file.

(2) List: It will list the files in current directory with some details of files

Program Logic

1) Main function will execute and display the command prompt as $

2) Accept the command at $ prompt from user.

3) Separate or tokenize the different parts of command line.

4) Check that first part is one of the extended commands or not (count, typeline, list, search).

5) If the command is extended command, then call corresponding functions which is implementing that command

6) Otherwise fork a new process and then load (exec) a program in that newly created process

and execute it. Make the shell to wait until command finish its execution.

7) Display the prompt and continue until given command is “q” to Quit.

Practical Assignments:

Set A

1. Create a child process using fork(), display parent and child process id. Child process will display the message “Hello World” and the parent process should display “Hi”.

2. Creating a child process using the command exec(). Note down process ids of the parent and the child processes, check whether the control is given back to the parent after the child process terminates. Write a similar program using execv() and execvp() and observe the differences in behaviours of the commands.

3. Creating a child process without terminating the parent process Write a program to create a child process using fork().The parent should goto sleep state and child process should begin its execution. In the child process, use execl() to execute the “ls” command.

Set B

1. Write a program to illustrate the concept of orphan process (Using fork() and

sleep())(Refer Program 5).

2. Write a program that demonstrates the use of nice() system call. After a child process is started using fork(), assign higher priority to the child using nice() system call.

3. Write a program to find the execution time taken for execution of a given set of instructions (Hint: use clock() function. This function clock() is called at the beginning of program and again at the end of the program and the difference between the values returned gives the time spent by processor on the program.)

Set C(Shell Command)

1.Implement the shell program that accepts the command at $ prompt displayed by your shell (myshell$). Implement the 'count' command by creating child process which works as follows:

myshell$ count c filename: To display the number of characters in given file

myshell$ count w filename: To display the number of words in given file

myshell$ count l filename: To display the number of lines in given file

2. Extend the shell to implement the commands 'list' which works as follows:

myshell$ list f dirname: It will display filenames in a given directory.

myshell$ list n dirname: It will count the number of entries in a given directory.

myshell$ list i dirname: It will display filenames and their inode number for the files in a given directory.

Assignment Evaluation

0: Not Done [] 1: Incomplete [ ] 2: Late Complete [ ]

3: Needs Improvement [ ] 4: Complete [ ] 5: Well Done [ ]

Signature of the Instructor **Date of Completion ____/____/_____**

Assignment 2- CPU Scheduling

CPU Scheduling is one of the important tasks performed by the operating system. In multiprogramming operating systems many processes are loaded into memory for execution and these processes are sharing the CPU and other resources of computer system.

Scheduler is a program that decides which program will execute next at the CPU or which program will be loaded into memory for execution. CPU scheduler is a program module of an operating system that selects the process to execute next at CPU out of the processes that are in memory. This scheduler is also called as **Short term scheduler** or CPU **Scheduler**.

Itselects from among the processes in memory that are ready toexecute, and allocates the CPU to one of them.

The main objective is to increase system performance in accordance with the chosen set ofcriteria.

There are 3 types of schedulers as

1) Short Term Scheduler

2) Long Term Scheduler

3) Medium Term Schedule

CPU Scheduler is a short term scheduler. CPU scheduling has to be done when a process:

1. Switches from running to waiting state.
2. Switches from running to ready state.
3. Switches from waiting to ready.
4. Terminates.

- Scheduling under 1 and 4 is non preemptive.
- All other scheduling is preemptive

Scheduling Criteria

• **CPU utilization** – keep the CPU as busy as possible

• **Throughput** – Number of processes that complete their execution per time unit

• **Turnaround time** – amount of time to execute a particular process

• **Waiting time** – amount of time a process has been waiting in the ready queue

• **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Optimization Criteria

• Max CPU utilization

• Max throughput

• Min turnaround time

• Min waiting time

• Min response time

Various CPU Scheduling algorithms are:

1) First Come First Serve (FCFS)

2) Shortest Job First (SJF)

3) Priority Scheduling

4) Round Robin Scheduling (RR)

These scheduling algorithms are further classified into 2 types as Preemptive and Non-Preemptive. FCFS scheduling is always Non-Preemptive while Round Robin is alwaysPreemptive, while Shortest Job First and Priority Scheduling can be preemptive or non-preemptive.

Data Structures

To simulate the working of various CPU scheduling algorithms following data structures isrequired.

1) Ready Queue - It represents the queue of processes which ready to execute but waiting forCPU to become available. Scheduling algorithm will select appropriate process from the readyqueue and dispatch it for execution. For FCFS and Round Robin this queue is strictly operated asFirst In First out queue. But for Shortest Job First and Priority Scheduling it will operate asPriority Queue.

2) Process Control Block- It will maintain various details about each process as processID,CPU-Burst, Arrival time, waiting time, completion time, execution time, turnaround time, etc. Astructure can be used to define all these fields for processes and we can use array of structures ofsize n. (n is number of processes)

1) First Come First Serve Scheduling (FCFS):

In this algorithm the order in which process enters the ready queue, in same order they

willexecute on the CPU.

2) Shortest Job First (SJF) :

In this algorithm the jobs will execute at the CPU according their next CPU-Burst time. The job in a ready queue which has shortest next CPU burst will execute next at the CPU. This algorithm can be preemptive or non-preemptive.

3) Priority Scheduling (PS) :

In this algorithm the job will execute according to their priority order. The job which has highest priority will execute first and the job which has least priority will execute last at the CPU. Priority scheduling can be preemptive or non-preemptive.

4) Round Robin Scheduling (RR) :

This algorithm is mostly used in time-sharing operating systems like Unix/Linux. This algorithm gives the fair chance of execution to each process in system in rotation. In this algorithm each process is allowed to execute for some fixed time quantum. If process has the CPU-burst more than time quantum then it will execute for the given time quantum. But if it has CPU-burst less than the time quantum then it will execute for its CPU-burst time and then immediately release the CPU so that it can be assigned to other process. The advantage of this algorithm is that the average waiting time is less. It this algorithm ready queue is strictly operated as First-In First-Out queue. This algorithm is intrinsically preemptive scheduling algorithm.

SET-A

1. Write the simulation program using FCFS. The arrival time and first CPU bursts of different jobs should be input to the system. Assume the fixed I/O waiting time (2 units). The next CPU burstshould be generated using random function. The output should give the Gantt chart, Turnaround Time and Waiting time for each process and average times.

2. Write the simulation program using SJF(non-preemptive). The arrival time and first CPU bursts of different jobs should be input to the system. The Assume the fixed I/O

waiting time (2 units).The next CPU burst should be generated using random function. The output should give the Gantt chart, Turnaround Time and Waiting time for each process and average times.

Set B:

1. Write the program to simulate Preemptive Shortest Job First (SJF) -scheduling. Thearrival time and first CPU-burst for different n number of processes should be input to thealgorithm. Assume the fixed IO waiting time (2 units). The next CPU-burst should be generated randomly. The output should give Gantt chart, turnaround time and waitingtime for each process. Also find the average waiting time and turnaround time.

2. Write the program to simulate Non-preemptive Priority scheduling. The arrival time andfirst CPU-burst and priority for different n number of processes should be input to thealgorithm. Assume the fixed IO waiting time (2 units). The next CPU-burst should begenerated randomly. The output should give Gantt chart, turnaround time and waitingtime for each process. Also find the average waiting time and turnaround time.

Assignment Evaluation

0: Not Done [ ]	1: Incomplete	[ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete	[ ]	5: Well Done [ ]

Signature of the Instructor **Date of Completion : ___/____ /____**

Assignment No: 3 Deadlock detection and avoidance

Introduction:

Every process needs some resources to complete its execution. However, the resource is granted in a sequential order.

1. The process requests for some resource.
2. OS grant the resource if it is available otherwise let the process waits.
3. The process uses it and release on the completion.

A Deadlock is a situation where each of the computer process waits for a resource which is being assigned to some another process. In this situation, none of the process gets executed since the resource it needs, is held by some other process which is also waiting for some other resource to be released.

Let us assume that there are three processes P1, P2 and P3. There are three different resources R1, R2 and R3. R1 is assigned to P1, R2 is assigned to P2 and R3 is assigned to P3.After some time, P1 demands for R1 which is being used by P2. P1 halts its execution since it can't complete without R2. P2 also demands for R3 which is being used by P3. P2 also stops its execution because it can't continue without R3. P3 also demands for R1 which is being used by P1 therefore P3 also stops its execution. So, this condition is called as deadlock.

Deadlock Characterization – Necessary Conditions, Resource Allocation Graph
Necessary conditions for Deadlocks

- Mutual Exclusion: A resource can only be shared in mutually exclusive manner. It implies, if two process cannot use the same resource at the same time.
- Hold and Wait: A process waits for some resources while holding another resource at the same time.
- No preemption: The process which once scheduled will be executed till the completion. No other process can be scheduled by the scheduler meanwhile.
- Circular Wait: All the processes must be waiting for the resources in a cyclic manner so that the last process is waiting for the resource which is being held by the first process.

Deadlock Avoidance - Safe state, Resource-Allocation-Graph Algorithm, Banker's Algorithm

Deadlock Avoidance

Avoid actions that may lead to a deadlock. Think of it as a state machine moving from one state to another as each instruction is executed

Safe State

- A state is safe if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state.

- More formally, a state is safe if there exists a safe sequence of processes {P0, P1, P2, ..., PN} such that all of the resource requests for Pi can be granted using the resources currently allocated to Pi and all processes Pj where j <i. (I.e. if all the processes prior to Pi finish and free up their resources, then Pi will be able to finish also, using the resources that they have freed up.)
- If a safe sequence does not exist, then the system is in an unsafe state, which MAY lead to deadlock. (All safe states are deadlock free, but not all unsafe states lead to deadlocks.)

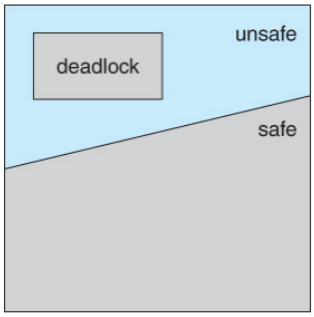


Figure: Safe, Unsafe and deadlocked state spaces.

- For example, consider a system with 12 tape drives, allocated as follows. Is this a safe state? What is the safe sequence?

	Maximum Needs	Current Allocation
P0	10	5
P1	4	2
P2	9	2

- What happens to the above table if process P2 requests and is granted one more tape drive?
- Key to the safe state approach is that when a request is made for resources, the request is granted only if the resulting allocation state is a safe one.

Deadlock Avoidance

Deadlock avoidance can be done with Banker's Algorithm.

Banker's Algorithm

Bankers' Algorithm is resource allocation and deadlock avoidance algorithm which test all the request made by processes for resources, it checks for the safe state, if after granting request system remains in the safe state it allows the request and if there is no safe state it doesn't allow the request made by the process.

Inputs to Banker's Algorithm:

1. Max need of resources by each process.
2. Currently, allocated resources by each process.
3. Max free available resources in the system.

The request will only be granted under the below condition:

1. If the request made by the process is less than equal to max need to that process.
2. If the request made by the process is less than equal to the freely available resource in the system.

Several data structures are used to implement Banker's algorithm. Let n be number of processes and m be number of resource types.

1. **Available**: A vector of length m indicating the number of available resources of each type. If Available [j] = k means there are k instances of resource type Rj available.

2. **Max**: A n x m matrix defining the maximum demand of each process. If Max [i, j] = k, then process Pi may request at most k instances of resource type Rj.

3. **Allocation:**A n × m matrix defining 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.

4. **Need:**A n × m matrix indicating the remaining resource need of each process. if Need [i, j] = k, then process Pi may need k more instances of resource type Rj, in order to complete its task.

Safety Algorithm

Let Work and Finish be vectors of length m and n, respectively

1. Initialize Work = Available
2. Finish [i] = false for i = 0, 1, …, n − 1.
3. Find i such that both:
    a) Finish [i] = = false
    b) $Need_i$<=Work
4. If no such i exists, go to step 4.
5. Work = Work + $Allocation_i$, Finish[i] = true, go to step 2.
6. If Finish [i] == true for all i, then the system is in a safe state.

Resource – Request Algorithm

- This algorithm determines if requests can be safely granted.
- Let Requesti be request vector for process Pi. If Requesti [j] = k then process Pi wants k instances of resource type Rj.
    1. If Requesti ≤ Needi go to step 2. Otherwise, raise error condition, since Process has exceeded its maximum claim.
    2. If Requesti ≤ Available, go to step 3. Otherwise Pi must wait, since resources are not available.
    3. Pretend to allocate requested resources to Pi by modifying the state as follows:
        a) Available = Available – Requesti;
        b) Allocationi = Allocationi + Requesti;
        c) Needi = Needi – Requesti

        If safe state ⇒ the resources are allocated to P_i.

        If unsafe state ⇒ P_i must wait, and the old resource-allocation state is restored

ALGORITHM:

1. Start the program.
2. Get the values of resources and processes.

3. Get the avail value.
4. After allocation find the need value.
5. Check whether it's possible to allocate
6. If it is possible then the system is in safe state.
7. Else system is not in safety state.
8. If the new request comes then check that the system is in safety.
9. Or not if we allow the request.
10. Stop the program.

DEADLOCK DETECTION

- Deadlock detection detects a deadlock by checking whether all conditions necessary for a deadlock hold simultaneously.
- The systems that do not implement algorithms for deadlock prevention or avoidance must implement an algorithm for deadlock detection and recovery.
- The system must maintain information about:
- Process status: resource allocated and requested by a process.
- An algorithm to detect the deadlock.
- There are two algorithms i.e. Single instance resource type and several instances of a resource type.
- The deadlock detection algorithm uses same data structures as Banker's algorithm:
    a) Available = A vector of length m indicating number of available resources of each type.
    b) Allocation = An n × m matrix indicating the number of resources of each type currently allocated to each process.
    c) Request = A n × m matrix indicating the current request of each process. If Request[i][j] = k then process Pi is requesting k more instances of resource type Rj.

ALGORITHM:

Step 1: Let Work and Finish be vectors of length m and n, respectively. Initialize Work = Available. For i = 1, 2, …, n, if Allocation$_i \neq 0$, then Finish[i] = false; otherwise, Finish[i] = true.

Step 2: Find an index i such that both:

a) Finish[i] == false
b) Request$_i \leq$ Work

If no such i exists, go to step 4.

Step 3: Work = Work + Allocation$_i$, Finish[i] = true go to step 2.

Step 4: If Finish[i] == false, for some i, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if Finish[i] == false, then Pi is deadlocked.

Algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in deadlocked state.

SET A

1. Write the program to calculate minimum number of resources needed to avoid deadlock.
2. Write a C program to accept the number of process and resources and find the need matrix content and display it.

SET B

1. Partially implement the Menu driven Banker's algorithm for accepting Allocation, Max from user.

a) Accept Available

b) Display Allocation, Max

c) Find Need and Display It,

d) Display Available

2. Consider the system with 3 resources types A,B, and C with 7,2,6 instancesrespectively. Consider the following snapshot:

	Allocation		
	A	**B**	**C**
P_0	0	1	0
P_1	2	0	0
P_2	3	0	3
P_3	2	1	1
P_4	0	0	2

	Request		
	A	**B**	**C**
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

Total Resources		
A	**B**	**C**
7	2	6

Answer the following questions:

a) Display the contents of Available array?

b) Is there any deadlock? Print the message

SET C

1. Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

- Consider the following snapshot of system, A, B, C and D are the resource type

	Allocation			
	A	**B**	**C**	**D**
P_0	0	0	1	2
P_1	1	0	0	0
P_2	1	3	5	4
P_3	0	6	3	2
P_4	0	0	1	4

	MAX			
	A	**B**	**C**	**D**
P_0	0	0	1	2
P_1	1	7	5	0
P_2	2	3	5	6
P_3	0	6	5	2
P_4	0	6	5	6

Available			
A	**B**	**C**	**D**
1	5	2	0

a) Calculate and display the content of need matrix?

b) Is the system in safe state? If display the safe sequence.

c) If a request from process P arrives for (0, 4, 2, 0) can it be granted immediately by keeping the system in safe state. Print a message

Assignment Evaluation

0: Not Done [ ] 1: Incomplete [ ] 2: Late Complete [ ]

3: Needs Improvement [ ] 4: Complete [ ] 5: Well Done [ ]

Set C:

1. Write the program to simulate Preemptive Priority scheduling. The arrival time and firstCPU-burst and priority for different n number of processes should be input to thealgorithm. Assume the fixed IO waiting time (2 units). The next CPU-burst should

begenerated randomly. The output should give Gantt chart, turnaround time and waitingtime for each process. Also find the average waiting time and turnaround time.

2. Write the program to simulate Round Robin (RR) scheduling. The arrival time and firstCPU-burst for different n number of processes should be input to the algorithm. Alsogive the time quantum as input. Assume the fixed IO waiting time (2 units). The nextCPU-burst should be generated randomly. The output should give Gantt chart, turnaroundtime and waiting time for each process. Also find the average waiting time andturnaround time.

Assignment Evaluation

0: Not Done [ ]	1: Incomplete	[ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete	[ ]	5: Well Done [ ]

Signature of the Instructor **Date of Completion : ___/____ /____**

Assignment 4
Page Replacement Algorithms

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when new page comes in. When page fault occurs, system try to load the corresponding page into one of the available free frames. If no free frame is available them system, try to replace one of the existing pages in frame with new page this is called as Page Replacement. Frame selected for page replacement is as victim frame.

Page replacement algorithm of demand paging memory management scheme will decide which frame will be selected as victim frame. There are various page replacement algorithms as

There are various page replacement algorithms as:

1. First In First Out Page Replacement (FIFO)
2. Optimal Page Replacement (OPT)
3. Least Recently Used Page Replacement (LRU)
4. Most Recently Used Page Replacement (MRU)
5. Most Frequently Used Page Replacement (MFU)
6. Least Frequently Used Page Replacement (LFU)
7. Second Change Page Replacement's

Some of the Important Terminology related to Page Replacement Algorithm are as follows.

1. **Page Fault**
   When process requests some page during execution and that page in not available in physical memory then it is called as Page Fault.Whenever page fault occurs Operating system check really it is a page fault or it is a invalid memory reference. If page fault has occurred, then system try to load the corresponding page in available free frame.

2. **Page Replacement**
   When page fault occurs, system try to load the corresponding page into one of the available free frames. If no free frame is available them system, try to replace one of the existing page in frame with new page. This is called as Page Replacement. Frame selected for page replacement is as victim frame. Page replacement algorithm of demand paging memory management scheme will decide which frame will be selected as victim frame.

3. **Input to Page Replacement Algorithm**
   a. **Reference String:** It is the list of numbers which represent the various page numbers demanded by the process.
   b. **Number of Frames:** It represents the number of frames in physical memory. Ex. Reference String: 12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15, 19, 8It means first process request the page number 12 then page number 15 then 12 and so on.

4. **Data Structure**
   a. Array of memory frames which is used to maintain which page is loaded in which frame. With each frame we may associate the counter or a time value whenever page is loaded in that frame or replaced.
   b. Array of reference String.
   c. Page Fault Count.

5. **Output**

   The output for each page replacement algorithm should display how each next page is loaded in which frame. Finally, it should display the total number of page faults

Page Replacement Algorithms

1. **First In First Out Page Replacement (FIFO)**
   - In this page replacement algorithm, the order in which pages are loaded in memory, in same order they are replaced.
   - We can associate a simple counter value with each frame when a page is loaded in memory.
   - Whenever page is loaded in free frame a next counter value is also set to it.
   - When page is to be replaced, select that page which has least counter value.
   - It is most simple page replacement algorithm.

2. **Optimal Page Replacement (OPT)**
   - This algorithm looks into the future page demand of a process.
   - Whenever page is to be replaced it will replace that page from the physical which will not be required longer time.
   - To implement this algorithm whenever page is to be replaced, we compare the pages in physical memory with their future occurrence in reference string. The page in physical memory which will not be required for longest period of time will be selected as victim.

3. **Least Recently Used Page Replacement (LRU)**
   - This algorithm replaces that page from physical memory which is used least recently.
   - To implement this algorithm, we associate a next counter value or timer value with each frame/page in physical memory wherever it is loaded or referenced from physical memory. When page replacement is to be performed, it will replace that frame in physical memory which has smallest counter value.

4. **Most Recently Used Page Replacement (MRU)**
   - This algorithm replaces that page from physical memory which is used most recently.
   - To implement this algorithm, we associate a next counter value or timer value with each frame/page in physical memory wherever it is loaded or referenced from physical memory. When page replacement is to be performed, it will replace that frame in physical memory which has greatest counter value.

Set A

1. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n as the number of memory frames.

   Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

   i. Implement FIFO

2. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n as the number of memory frames.

   Reference String :3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6

   i. Implement FIFO

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of the Instructor **Date of Completion ____/____/_____**

Set B

1. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n as the number of memory frames.

   Reference String : 12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8

   i. Implement OPT

2. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n as the number of memory frames.

   Reference String :7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2

   i. Implement LRU

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of the Instructor **Date of Completion ____/____/_____**

Set C

1. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n as the number of memory frames.

   Reference String :12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8

   i. Implement LRU

2. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n as the number of memory frames.

Reference String : 2,5,2,8,5,4,1,2,3,2,6,1,2,5,9,8

ii. Implement MRU

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of the Instructor **Date of Completion ____/____/_____**

Assignment 5
A* and AO* Algorithm

A* Algorithm-

A * algorithm is a searching algorithm that searches for the shortest path between the *initial and the final state.* It is used in various applications, such as *maps*.

In *maps* the A* algorithm is used to calculate the shortest distance between the source (initial state) and the destination (final state).

A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster.

- A* Algorithm is one of the best and popular techniques used for path finding and graph traversals.
- A lot of games and web-based maps use this algorithm for finding the shortest path efficiently.
- It is essentially a best first search algorithm.

Working-

A* Algorithm works as-

- It maintains a tree of paths originating at the start node.
- It extends those paths one edge at a time.
- It continues until its termination criterion is satisfied.

A* Algorithm extends the path that minimizes the following function-

$$f(n) = g(n) + h(n)$$

Heuristics function: Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal. The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time.

Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

The form of the heuristic function for A* is

f(n) = g(n) + h(n)

where,

g(n)= the cost

Algorithm of A* search:

Step1: Place the starting node in the OPEN list.

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise

Step 4: Expand node n and generate all of its successors and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.

Step 6: Return to **Step 2**.

AO* Search: (And-Or) Graph

The Depth first search and Breadth first search given earlier for OR trees or graphs can be easily adopted by AND-OR graph. The main difference lies in the way termination conditions are determined, since all goals following an AND nodes must be realized; where as a single goal node following an OR node will do. So for this purpose we are using AO* algorithm.

Like A* algorithm here we will use two arrays and one heuristic function.

OPEN:

It contains the nodes that has been traversed but yet not been marked solvable or unsolvable.

CLOSED:

It contains the nodes that have already been processed.

Algorithm:

1. Let G consists only to the node representing the initial state call this node INIT. Compute h' (INIT).

2. Until INIT is labeled SOLVED or hi (INIT) becomes greater than FUTILITY, repeat the following procedure.

(I) Trace the marked arcs from INIT and select an unbounded node NODE.

(II) Generate the successors of NODE. if there are no successors then assign FUTILITY as h' (NODE). This means that NODE is not solvable. If there are successors then for each one called SUCCESSOR, that is not also an ancester of NODE do the following

(a) add SUCCESSOR to graph G
(b) if successor is not a terminal node, mark it solved and assign zero to its h ' value.
(c) If successor is not a terminal node, compute it h' value.

(III) propagate the newly discovered information up the graph by doing the following. letS be a set of nodes that have been marked SOLVED. Initialize S to NODE. Until S is empty repeat the following procedure.

(a) select a node from S call if CURRENT and remove it from S.

(b) compute h' of each of the arcs emerging from CURRENT, Assign minimum h' to CURRENT.

(c) Mark the minimum cost path a s the best out of CURRENT.

(d) Mark CURRENT SOLVED if all of the nodes connected to it through the new marked have been labeled SOLVED.

(e) If CURRENT has been marked SOLVED or its h ' has just changed, its new status must be propagate backwards up the graph. hence all the ancestors of CURRENT are added to S.

Set A :

1) Given an initial state of a 8-puzzle problem and final state to be reached-

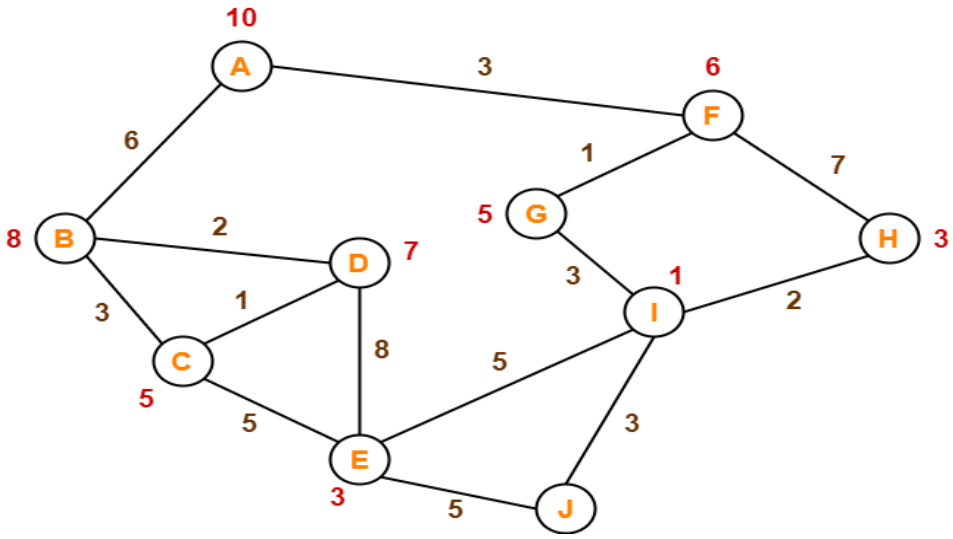
Initial State

Final State

Find the most cost-effective path to reach the final state from initial state using A* Algorithm in C/Python.

Consider g(n) = Depth of node and h(n) = Number of misplaced tiles.

2) Consider the following graph-



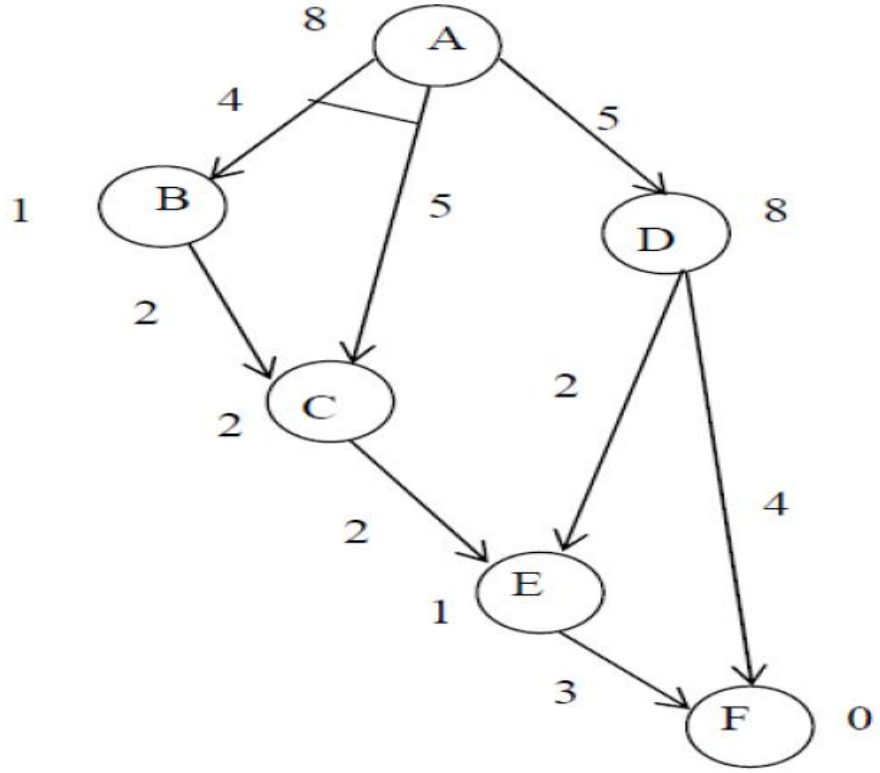
The numbers written on edges represent the distance between the nodes.

The numbers written on nodes represent the heuristic value.

Implement A* algorithm in C/Python for above graph and find out most cost-effective path from A to J.

Set B:

Implement AO* algorithm in C/ python for following graph and find out minimum cost solution.



Assignment Evaluation

0: Not Done [] 1: Incomplete [ ] 2: Late Complete [ ]

3: Needs Improvement [ ] 4: Complete [ ] 5: Well Done [ ]

Signature of the Instructor **Date of Completion ____/____/_____**