GROUP OF INSTITUTIONS

Meerut Institute of Engineering & Technology

Department of Computer Science & Engineering

Dr. A.P.J. Abdul Kalam Technical University, Lucknow

LAB MANUAL

Subject Code: KCS-451

Subject Name: Operating System Lab

INDEX

Vision of the Institute

To be an outstanding institution in the country imparting technical education, providing need based, value based and career based programmes and producing self-reliant, self-sufficient technocrats, capable of meeting new challenges.

Mission of the Institute

To educate young aspirants in various technical fields to fulfill global requirement of human resources by providing sustainable quality education, training and invigorating environment, also molding them into skilled competent and socially responsible citizens who will lead the building of a powerful nation.

Vision of Department

To be an excellent department that imparts value based quality education and uplifts innovative research in the ever-changing field of technology.

Mission of Department

a) To fulfill the requirement of skilled human resources with focus on quality education.
b) To create globally competent and socially responsible technocrats by providing value and need based training.
c) To improve Industry-Institution Interaction and encourage the innovative research activities.

Program Educational Objectives

1. Students will have the successful careers in the field of computer science and allied sectors as an innovative engineer.
2. Students will continue to learn and advance their careers through participation in professional activities, attainment of professional certification and seeking advance studies.
3. Students will be able to demonstrate a commitment to life-long learning.
4. Students will be ready to serve society in any manner and become a responsible and aware citizen.
5. Establishing students in a leadership role in any field.

Program Outcomes

1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations
4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes

1. Ability to apply and analyze computational concepts in the areas related to algorithms, machine learning, cloud computing, web designing and web services.
2. Ability to apply standard practices and methodologies in software development and project management.
3. Ability to employ fundamental concepts and emerging technologies for innovative research activities, carrier opportunities & zeal for higher studies.

Course Outcomes

CO-1	To apply the basic LINUX commands, process concepts and system calls.
CO-2	To implement various CPU scheduling algorithm for a given problem
CO-3	To implement the concepts of deadlock and multiprogramming system
CO-4	To implement various page replacement algorithms.

List of Experiments

Exp. No.	Experiment Name	Course Outcome
1	a) Write Basic Commands for VI-Editor used in Linux. b) Write a basic program on VI Editor in Linux.	CO1
2	Write a program to show the process ID and parent process ID of a process.	CO1
3	d) Write a program to create a child process using fork () system call. e) Write a program to create child process and verify the child's parent ID and parent process ID.	CO1
4	Write a program to show the orphan process concept.	CO2
5	Write a program to show the status of processes [Zombie(Z), sleeping(S), running(R)].	CO2
6	Write a program to show the concurrent execution of child and parent process using fork() system call.	CO2
7	a) Write a program to simulate FCFS scheduling algorithm without arrival time. b) Write a program to simulate FCFS scheduling algorithm with arrival time. c) Write a program to simulate Shortest Job First (SJF) scheduling algorithm without arrival time. d) Write a Program to simulate Non- Preemptive Priority Scheduling Algorithm	CO2
8	Write a Program to simulate LEAST RECENTLY USED page replacement algorithm and OPTIMAL page replacement algorithm	
9	Write a Program to simulate Round Robin Scheduling Algorithm	CO2
Value added Programs		
10	Write a program to simulate Banker's Algorithm.	CO3

LAB-1.1

Objective: Write Basic Commands for VI-Editor used in Linux.

Commands	Description
i	Instruction mode
a	Append to right mode
/word	Move to the occurrence of "word"
n	Locate next occurrence
w	Advance to next mode
e	Advance to the next end of the word
b	Move to the previous word
3b	Move backwards 3 words
YY	Delete line
3dd	Delete 3 lines
D	Deletes remainder of a line
dw	Deletes a word
X	Delete character
o (small)	Open space for new line below the cursor line
O (big)	Opens a line above the cursor
Ctrl-w	Move back a word in append mode
u (small)	Undo last
U	Undo all changes to current line
Esc	Command mode
:w new file name	Save the file to the new file name from the command mode
:wq	Save and quit
q!	Quit without saving
	r Replaces then type a character to be replaced with r then return to break up a line
J	Joins 2 lines
	S Substitute (sentence) type text over a character, Esc when done
	cw Change word
	c Change part of line from the cursor to the end of line
	cc Substitute new text for a line, Esc when done
	h Move the cursor back one space
	H Move the cursor to the highest line on the space
	L Move the cursor to the lowest line on the screen
	M Position the cursor at the midpoint of the screen
	G Last line in the file
	0 (ZERO) Move the cursor to the beginning of the line it is on
	View filename Open a file for view only
	set number Turn on line number
	set no number Turn on line number
	:n Access the next file for editing
	:wq filename.c save command
	gcc space –o space filename space filename.c compilation command
	./filename run command
	Vi return terminal(new)
	Vi space filename.c (return program)

LAB -1.2

Objective: Write a basic program on VI Editor in Linux.

```
#include <stdio.h>
#include <sys/types.h>

main ()
{
  int n;

  printf("Enter an integer\n");

 scanf("%d",&n);

  if ( n%2 == 0 )
    printf("Even\n");
  else
    printf("Odd\n");

  return 0;
}
```

LAB -2

Objective: Write a program to show the process ID and parent process ID of a process.

```
#include <stdio.h>
#include <sys/types.h>

main ()
{
  int pid;
  pid=getpid();
  ppid=getppid();
  printf("id of the process is=%d\n", getpid());

  printf("id of the process is=%d\n", getppid());
 }
```

LAB -3.1

Write a program to create a child process using fork () system call.

```
#include <stdio.h>
#include <sys/types.h>

main ()
{
  int pid;
  pid=fork();
  printf("id of the process is=%d\n", getpid());
  printf("hello");


 }
```

LAB -3.2

Objective: Write a program to create child process and verify the child's parent ID and parent process ID.

```
#include <stdio.h>
#include <sys/types.h>
main ()
{
  int pid;
  pid=fork();
  if(pid==0)
  {
  printf("id of the child  process is=%d\n", getpid());

  printf("id of the parent  process is=%d\n", getppid());
  }
 else

 {
  printf("id of the parent  process is=%d\n", getpid());

  printf("id of the parent of parent  process is=%d\n", getppid());
 }
}
```

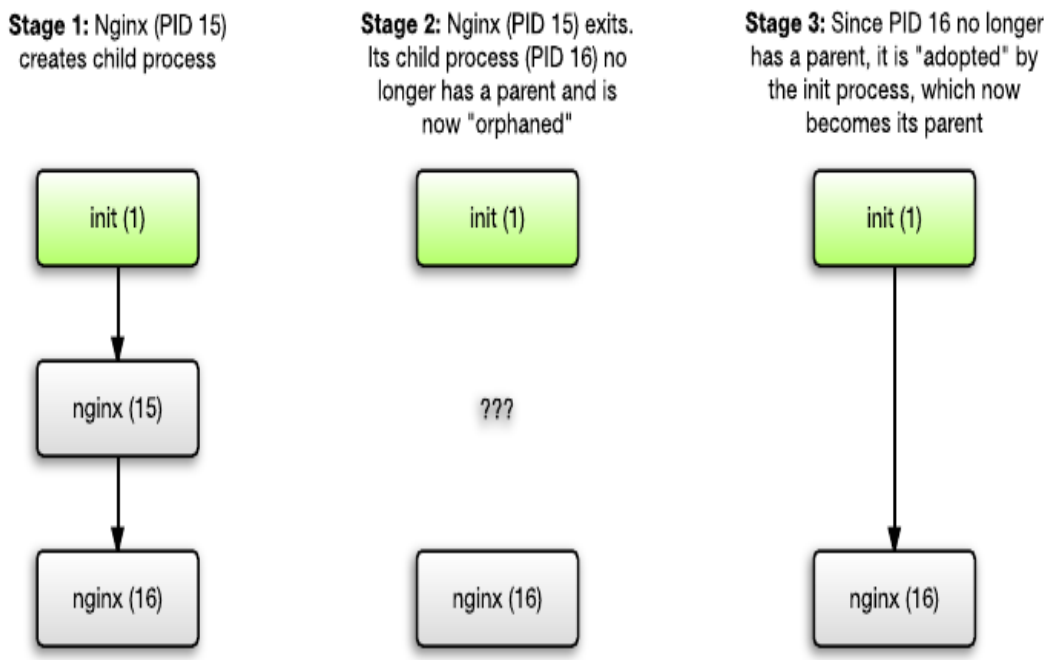
LAB-4

Objective: Write a program to show the orphan process concept.

Algorithm/FlowChart:

A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process.

a child process whose parent process terminates before it does becomes an *orphan process*. Such situations are typically handled with a special "root" (or "init") process, which is assigned as the new parent of a process when its parent process exits. This special process detects when an orphan process terminates and then retrieves its exit status, allowing the system to deallocate the terminated child process.



Parent finishes execution and exits while the child process is still executing and is called an orphan process now.

However, the orphan process is soon adopted by init process, once its parent process dies.

LAB-5

Objective: Write a program to show the status of processes [Zombie(Z), sleeping(S), running(R)].

Algorithm:

A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process. A child process always first becomes a zombie before being removed from the process table. The parent process reads the exit status of the child process which reaps off the child process entry from the process table.

1. Create new Child process using fork system call.
2. Print ID of the process.
3. If(pid !=0)
   Print "parent process"
   Else
   print "Child process".
4. Call Sleep method.

Some of the salient points related to zombie processes are as follows:

- All the memory and resources allocated to a process are deallocated when the process terminates using the exit() system call. But the process's entry in the process table is still available. This process is now a zombie process.
- The exit status of the zombie process zombie process can be read by the parent process using the wait() system call. After that, the zombie process is removed from the system. Then the process ID and the process table entry of the zombie process can be reused.
- If the parent process does not use the wait() system call, the zombie process is left in the process table. This creates a resource leak.
- If the parent process is not running anymore, then the presence of a zombie process indicates an operating system bug. This may not be a serious problem if there are a few zombie processes but under heavier loads, this can create issues for the system such as running out of process table entries.

LAB-6

Objective: Write a program to show the concurrent execution of child and parent process using fork() system call.

Algorithm:

Fork system call use for creates a new process, which is called ***child process***, which runs concurrently with process (which process called system call fork) and this process is called ***parent process***. After a new child process created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.
It takes no parameters and returns an integer value. Below are different values returned by fork().

Negative Value: creation of a child process was unsuccessful.
Zero: Returned to the newly created child process.
Positive value: Returned to parent or caller. The value contains process ID of newly created child process.

Concurrent execution of child and parent process can be done as follows:

1. Create new Child process using fork() system call .
2. Print id of the process using getpid()//if id=0 child process will execute, parent process otherwise.

LAB -7.1

Objective: Write a program to simulate FCFS scheduling algorithm without arrival time

Algorithm/Pseudocode:

Given n processes with their burst times, the task is to find average waiting time and average turnaround time using FCFS scheduling algorithm.
First in, first out (FIFO), also known as first come, first served (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes in the order that they arrive in the ready queue.
In this, the process that comes first will be executed first and next process starts only after the previous gets fully executed.
Below we have a few shortcomings or problems with the FCFS scheduling algorithm:

1. It is **Non Pre-emptive** algorithm, which means the **process priority** doesn't matter. If a process with very least priority is being executed, more like **daily routine backup** process, which takes more time, and all of a sudden some other high priority process arrives, like **interrupt to avoid system crash**, the high priority process will have to wait, and hence in this case, the system will crash, just because of improper process scheduling.
2. Not optimal Average Waiting Time.
3. Resources utilization in parallel is not possible, which leads to **Convoy Effect**, and hence poor resource(CPU, I/O etc.) utilization.

Here we are considering that arrival time for all processes is 0.

1- Input the processes along with their burst time (bt).

2- Find waiting time (wt) for all processes.

3- As first process that comes need not to wait so

waiting time for process 1 will be 0 i.e. wt[0] = 0.

4- Find **waiting time** for all other processes i.e. for process i ->
wt[i] = bt[i-1] + wt[i-1] .

5- Find **turnaround time** = waiting_time + burst_time for all processes.

6- Find **average waiting time** =
total_waiting_time / no_of_processes.

7- Similarly, find **average turnaround time** =
total_turn_around_time / no_of_processes.

LAB -7.2

Objective: Write a program to simulate FCFS scheduling algorithm with arrival time

Algorithm/Pseudocode:

We have already discussed FCFS Scheduling of processes with same arrival time. In this post, scenario when processes have different arrival times are discussed. Given n processes with their burst times and arrival times, the task is to find average waiting time and average turnaround time using FCFS scheduling algorithm.
FIFO simply queues processes in the order they arrive in the ready queue. Here, the process that comes first will be executed first and next process will start only after the previous gets fully executed.

1. Completion Time: Time at which process completes its execution.
2. Turn Around Time: Time Difference between completion time and arrival time. Turn Around Time = Completion Time – Arrival Time
3. Waiting Time(W.T): Time Difference between turn around time and burst time. Waiting Time = Turn Around Time – Burst Time

Changes in code as compare to code of FCFS with same arrival time:
To find waiting time: Time taken by all processes before the current process to be started (i.e. burst time of all previous processes) – arrival time of current process
wait_time[i] = (bt[0] + bt[1] +…… bt[i-1]) – arrival_time[i]

Ago:

1- Input the processes along with their burst time(bt)

and arrival time(at)

2- Find waiting time for all other processes i.e. for

a given process i:

wt[i] = (bt[0] + bt[1] +..... bt[i-1]) - at[i]

3- Now find **turn around time**
= waiting_time + burst_time for all processes
4- **Average waiting time** =
total_waiting_time / no_of_processes
5- **Average turnaround time** =
total_turn_around_time / no_of_processes

LAB-7.3

Objective: Write a program to simulate Shortest Job First (SJF) scheduling algorithm.

Shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN is a non-preemptive algorithm.

Shortest Job first has the advantage of having minimum average waiting time among all scheduling algorithms.
It is a Greedy Algorithm.
It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of aging.
It is practically infeasible as Operating System may not know burst time and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times. SJF can be used in specialized environments where accurate estimates of running time are available.

Algorithm:

1- Sort all the processes in increasing order
according to burst time.
2- Then simply, apply FCFS.

How to compute below times in SJF using a program?

1.Completion Time: Time at which process completes its execution.
2.Turn Around Time: Time Difference between completion time and arrival time. Turn Around Time = Completion Time – Arrival Time
3.Waiting Time(W.T): Time Difference between turn around time and burst time. Waiting Time = Turn Around Time – Burst Time

LAB-7.4

Objective: Write a program to simulate Priority Scheduling Algorithm without arrival time.

Algorithm:

Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems. Each process is assigned a priority. Process with the highest priority is to be executed first and so on.

Processes with the same priority are executed on first come first served basis. Priority can be decided based on memory requirements, time requirements or any other resource requirement.

First input the processes with their burst time
and priority.

2- Sort the processes, burst time and priority according to the priority.

3- Now simply apply FCFS algorithm.

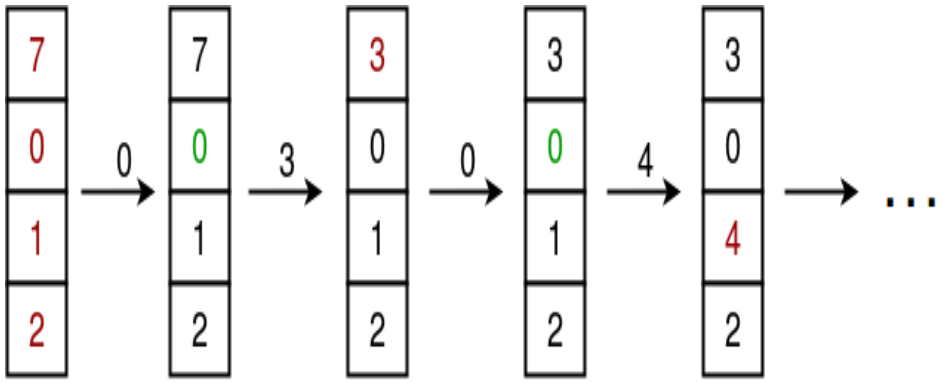
LAB -8

Objective: Write a Program to simulate FCFS,LRU and OPTIMAL Page Replacement Algorithm

Given n processes with their burst times, the task is to find average waiting time and average turn around time using FCFS scheduling algorithm.
First in, first out (FIFO), also known as first come, first served (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes in the order that they arrive in the ready queue.
In this, the process that comes first will be executed first and next process starts only after the previous gets fully executed.
Here we are considering that arrival time for all processes is 0.

How to compute below times in Round Robin using a program?

1. Completion Time: Time at which process completes its execution.
2. Turn Around Time: Time Difference between completion time and arrival time. Turn Around Time = Completion Time – Arrival Time
3. Waiting Time(W.T): Time Difference between turn around time and burst time. Waiting Time = Turn Around Time – Burst Time
4. In operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.
5. In **L**east **R**ecently **U**sed (LRU) algorithm is a Greedy algorithm where the page to be replaced is least recently used. The idea is based on locality of reference, the least recently used page is not likely
6. Let say the page reference string 7 0 1 2 0 3 0 4 2 3 0 3 2 . Initially we have 4 page slots empty.
   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.



Total Page faults = 6

In operating systems, whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.
In this algorithm, OS replaces the page that will not be used for the longest period of time in future.

Examples :

Input : Number of frames, fn = 3

Reference String, pg[] = {7, 0, 1, 2,

0, 3, 0, 4, 2, 3, 0, 3, 2, 1,

2, 0, 1, 7, 0, 1};

Output : No. of hits = 11

No. of misses = 9

Input : Number of frames, fn = 4

Reference String, pg[] = {7, 0, 1, 2,

0, 3, 0, 4, 2, 3, 0, 3, 2};

Output : No. of hits = 7

No. of misses = 6

LAB -9

Objective: Write a Program to simulate Round Robin Scheduling Algorithm.

Algorithm:

Round Robin is a CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way.

- It is simple, easy to implement, and starvation-free as all processes get fair share of CPU.
- One of the most commonly used technique in CPU scheduling as a core.
- It is preemptive as processes are assigned CPU only for a fixed slice of time at most.
- The disadvantage of it is more overhead of context switching.

Completion Time: Time at which process completes its execution.

1. Turn Around Time: Time Difference between completion time and arrival time. Turn Around Input the processes along with their burst time(bt)

and arrival time(at)

2- Find waiting time for all other processes i.e. for
a given process i:
wt[i] = (bt[0] + bt[1] +..... bt[i-1]) - at[i]

3- Now find **turn around time**
= waiting_time + burst_time for all processes

4- **Average waiting time** =
total_waiting_time / no_of_processes

5- **Average turn around time** =
total_turn_around_time / no_of_processes

424226392. Time = Completion Time – Arrival Time

424227064. Waiting Time(W.T): Time Difference between turn around time and burst time. Waiting Time = Turn Around Time – Burst Time

In this post, we have assumed arrival times as 0, so turn around and completion times are same.

The tricky part is to compute waiting times. Once waiting times are computed, turn around times can be quickly computed.

1- Create an array **rem_bt[]** to keep track of remaining
burst time of processes. This array is initially a
copy of bt[] (burst times array)
2- Create another array **wt[]** to store waiting times
of processes. Initialize this array as 0.
3- Initialize time : t = 0
4- Keep traversing the all processes while all processes
are not done. Do following for i'th process if it is
not done yet.
a- If rem_bt[i] > quantum
(i) t = t + quantum
(ii) bt_rem[i] -= quantum;
c- Else // Last cycle for this process
(i) t = t + bt_rem[i];
(ii) wt[i] = t - bt[i]
(ii) bt_rem[i] = 0; // This process is over

LAB -10

Objective: Write a program to simulate Banker's Algorithm.

Algorithm: The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Following **Data structures** are used to implement the Banker's Algorithm:

Let **'n'** be the number of processes in the system and **'m'** be the number of resources types.

Available :

- It is a 1-d array of size **'m'** indicating the number of available resources of each type.
- Available[j] = k means there are **'k'** instances of resource type $\mathbf{R_j}$

Max :

- It is a 2-d array of size '**n*m'** that defines the maximum demand of each process in a system.
- Max[i, j] = k means process $\mathbf{P_i}$ may request at most **'k'** instances of resource type $\mathbf{R_j}$.

Allocation :

- It is a 2-d array of size **'n*m'** that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process $\mathbf{P_i}$ is currently allocated **'k'** instances of resource type $\mathbf{R_j}$

Need :

- It is a 2-d array of size **'n*m'** that indicates the remaining resource need of each process.
- Need [i, j] = k means process $\mathbf{P_i}$ currently need **'k'** instances of resource type $\mathbf{R_j}$

  for its execution.

- Need [i, j] = Max [i, j] – Allocation [i, j]

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1) Let Work and Finish be vectors of length ‘m’ and ‘n’ respectively.
Initialize: Work = Available
Finish[i] = false; for i=1, 2, 3, 4….n

2) Find an i such that both
a) Finish[i] = false
b) $Need_i$ <= Work
if no such i exists goto step (4)

3) Work = Work + Allocation[i]
Finish[i] = true
goto step (2)

4) if Finish [i] = true for all i
then the system is in a safe state

Resource-Request Algorithm

Let $Request_i$ be the request array for process P_i. $Request_i$ [j] = k means process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

1) If $Request_i$ <= $Need_i$
Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If $Request_i$ <= Available
Goto step (3); otherwise, P_i must wait, since the resources are not available.

3) Have the system pretend to have allocated the requested resources to process Pi by modifying the state as
follows:
Available = Available – Requesti
$Allocation_i$ = $Allocation_i$ + $Request_i$
$Need_i$ = $Need_i$– $Request_i$