**SVKM’s NMIMS University**

**Mukesh Patel School of Technology Management & Engineering**

OPERATING SYSTEM, MBATech IT Sem IV

# **Practical 6-Bankers Algorithm [** Deadlock Avoidance]

|  |  |  |
| --- | --- | --- |
| Name | Roll\_No | SAP-ID |
| Jal Bafana | K005 | 70102300054 |

**AIM:**Deadlock Avoidance.

**STUDY ON:**

Implementing and analysing any one Algorithm for Deadlock Avoidance.

**PROGRAM:**

6.1 Implement a C program to simulate an algorithm for deadlock avoidance.

**JOURNAL:**

**This Experiment would consist of:**

1. Briefly explain Deadlock Detection, Prevention, Avoidance and the four conditions stated for deadlock.

ANS:

**Deadlock:**

* The computer system uses may types of resource which are then used by various processes to carry out their individual functions.
* But problem is that the number of resources available is limited and many processes needs to use it.
* A set of process is said to be in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The event can be resource acquisition, resource release etc. The resource can be physical (printers, memory space) or logical (semaphores, files)

The necessary and sufficient conditions for deadlock to occur are:

* **Mutual Exclusion**
  + A resource at a time can only be used by one process.
  + If another process is requesting for the same resource, then it must be delayed until that resource is released.
* **Hold and Wait**
  + A process is holding a resource and waiting to acquire additional resources that are currently being held by other processes.
* **No Pre-emption:**
  + Resources cannot be pre-empted
  + Resource can be released only by the process currently holding it based on its voluntary decision after completing the task
* Circular wait
  + A set of processes { P0,P1,….,Pn-1,Pn } such that the process P0 is waiting for resource held by P1,P1 is waiting for P2, and Pn is waiting for P0 to release its resources.
  + Every process holds a resource needed by the next process.

All the four above mentioned conditions should occur for a deadlock to occur.

The three major approaches for handling deadlocks are as follows:

a) **It ensures Deadlock Prevention:**

* That the system never enters a deadlock state.
* It provides a set of methods to make sure that at least one of the four necessary conditions for a deadlock is never satisfied.
* Mutual Exclusion - this condition is needed to be checked for non-sharable resources (e.g., Printer)
* Hold and Wait - It requires a process to request a resource and get allocated before execution or allow process to request resources when the process has none.
* No pre-emption - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
* Circular Wait-we can impose a total ordering of all resource types, and ask that each process requests resources in an increasing order of enumeration.
* Disadvantage is that it can lead to low device utilization and reduced system throughput.

b) **Deadlock Avoidance:**

* It is the simplest and most friendly method.
* It requires that each process declare the maximum number of resources of each type that it will need.
* The deadlock-avoidance algorithm dynamically checks the resource-allocation state to ensure the system can never be in a circular-wait condition.
* When a process requests a resource, the system must make sure that the allocation would leave the system in a safe state.
* It the system is in a safe state, then there would be no deadlock. However, if it is in an unsafe state, that there is a possibility (not certainty) of a deadlock.
* The avoidance approach requires that knowledge of all processes, all the resources available, the resources allocated presently and the future requests by the processes.
* For a single instance of a resource type, we use the resource allocation graph.
* For multiple instances of a resource type, we use the banker’s algorithm.
* A major drawback of this method is that it is difficult to know at the beginning itself of the maximum resource required.

c) **Deadlock detection and recovery:**

* Here we allow the system to enter into a deadlock state and then try to recover the system back from it
* This method has two parts:

i. Detection: an algorithm to check the system state whether in deadlock

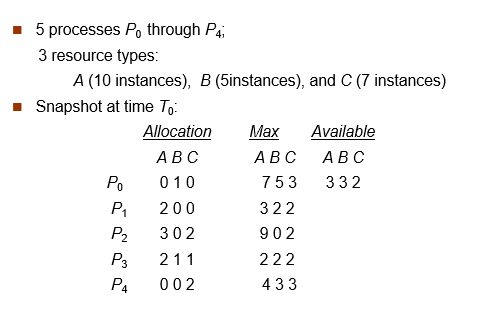
ii. Recovery: initiated when deadlock detected, to recover from that deadlock.

* This method doesn’t require the maximum need MAX or the Need matrix. It works using the Available, Allocation and Request data structures
* For single instance of a resource type, we use the wait-for graphs and search for cycles in the graph.
* For multiple instances of each resource type, we use the Deadlock-detection algorithm.
* To recover we have two options

i. Abort one or more processes

ii. Pre-empt one or more resources from one or more deadlocked process

2. Attach the program with output.



P = 5 # Number of processes

R = 3 # Number of resources

allocation = [[0] \* R for \_ in range(P)]

max\_resources = [[0] \* R for \_ in range(P)]

available = [0] \* R

need = [[0] \* R for \_ in range(P)]

# Function to calculate the need matrix

def calculate\_need():

for i in range(P):

for j in range(R):

need[i][j] = max\_resources[i][j] - allocation[i][j]

# Function to check if the system is in a safe state and also return the safe sequence

def is\_safe():

work = available[:]

finish = [False] \* P

safe\_sequence = []

count = 0

while count < P:

found = False

for p in range(P):

if not finish[p]:

can\_finish = True

for i in range(R):

if need[p][i] > work[i]:

can\_finish = False

break

if can\_finish:

for i in range(R):

work[i] += allocation[p][i]

finish[p] = True

safe\_sequence.append(p)

count += 1

found = True

break

if not found:

return False, [] # Not in a safe state

return True, safe\_sequence # In a safe state

# Main function to input matrices and check for safety

def main():

print("Enter Allocation Matrix:")

for i in range(P):

allocation[i] = list(map(int, input().split()))

print("Enter Maximum Matrix:")

for i in range(P):

max\_resources[i] = list(map(int, input().split()))

print("Enter Available Resources:")

available[:] = list(map(int, input().split()))

# Calculate Need Matrix

calculate\_need()

# Check if the system is in a safe state and get the safe sequence

safe\_state, safe\_sequence = is\_safe()

if safe\_state:

print("System is in a Safe State.")

print("Safe Sequence:", safe\_sequence)

else:

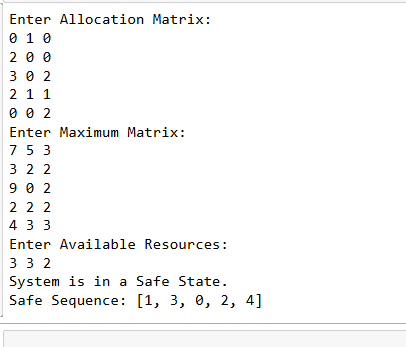
print("System is in an Unsafe State.")

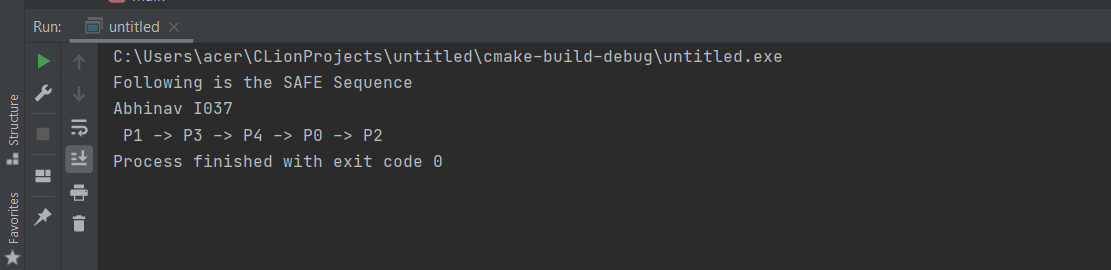
if \_\_name\_\_ == "\_\_main\_\_":

main()

**OUTPUT:**

**Sample below**

****

****

import threading

import time

read\_mutex = threading.Semaphore(1)

write\_mutex = threading.Semaphore(1)

shared\_data = 0

read\_count = 0

def reader(reader\_id, num\_reads, read\_time):

global read\_count

reads\_done = 0

while reads\_done < num\_reads:

read\_mutex.acquire()

read\_count += 1

if read\_count == 1: # First reader acquires write\_mutex

write\_mutex.acquire()

read\_mutex.release()

print(f"Reader {reader\_id} reads shared data: {shared\_data}")

time.sleep(read\_time)

read\_mutex.acquire()

read\_count -= 1

if read\_count == 0: # Last reader releases write\_mutex

write\_mutex.release()

read\_mutex.release()

time.sleep(1)

reads\_done += 1

def writer(writer\_id, num\_writes, write\_time):

global shared\_data

writes\_done = 0

while writes\_done < num\_writes:

write\_mutex.acquire()

shared\_data += 1

print(f"Writer {writer\_id} writes shared data: {shared\_data}")

write\_mutex.release()

time.sleep(write\_time)

writes\_done += 1

def main():

print("Performed By: Jal Bafana - K005")

num\_readers = int(input("Enter the number of reader threads: "))

num\_writers = int(input("Enter the number of writer threads: "))

read\_time = float(input("Enter the time (in seconds) for a reader to read: "))

write\_time = float(input("Enter the time (in seconds) for a writer to write: "))

num\_reads = int(input("Enter the number of times each reader will read: "))

num\_writes = int(input("Enter the number of times each writer will write: "))

reader\_threads = [threading.Thread(target=reader, args=(i, num\_reads, read\_time)) for i in range(num\_readers)]

writer\_threads = [threading.Thread(target=writer, args=(i, num\_writes, write\_time)) for i in range(num\_writers)]

for t in reader\_threads + writer\_threads:

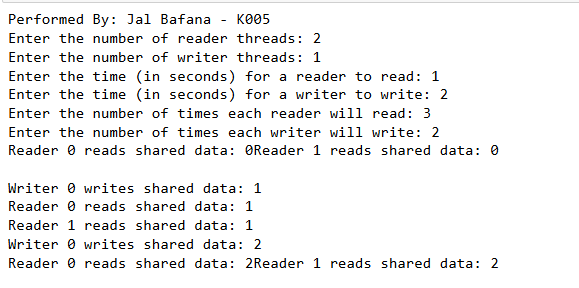
t.start()

for t in reader\_threads + writer\_threads:

t.join()

if \_\_name\_\_ == "\_\_main\_\_":

main()



## **Conclusion: -**

**Banker’s Algorithm** is a method used to avoid deadlock by ensuring the system remains in a **safe state**. It requires knowledge of each process’s maximum resource demand and its current allocation. The system checks whether a resource request can be granted without entering an unsafe state.

**Implementation:**

When working with a banker's algorithm, it requests to know about three things:

How much each process can request for each resource in the system. It is denoted by the [MAX] request.

How much each process is currently holding each resource in a system. It is denoted by the [ALLOCATED] resource.

It represents the number of each resource currently available in the system. It is denoted by the [AVAILABLE] resource.

**Conclusion:**

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. It helps you to identify whether a loan will be given or not. This **algorithm** is used to test for safely simulating the allocation for determining the maximum amount available for all resources.