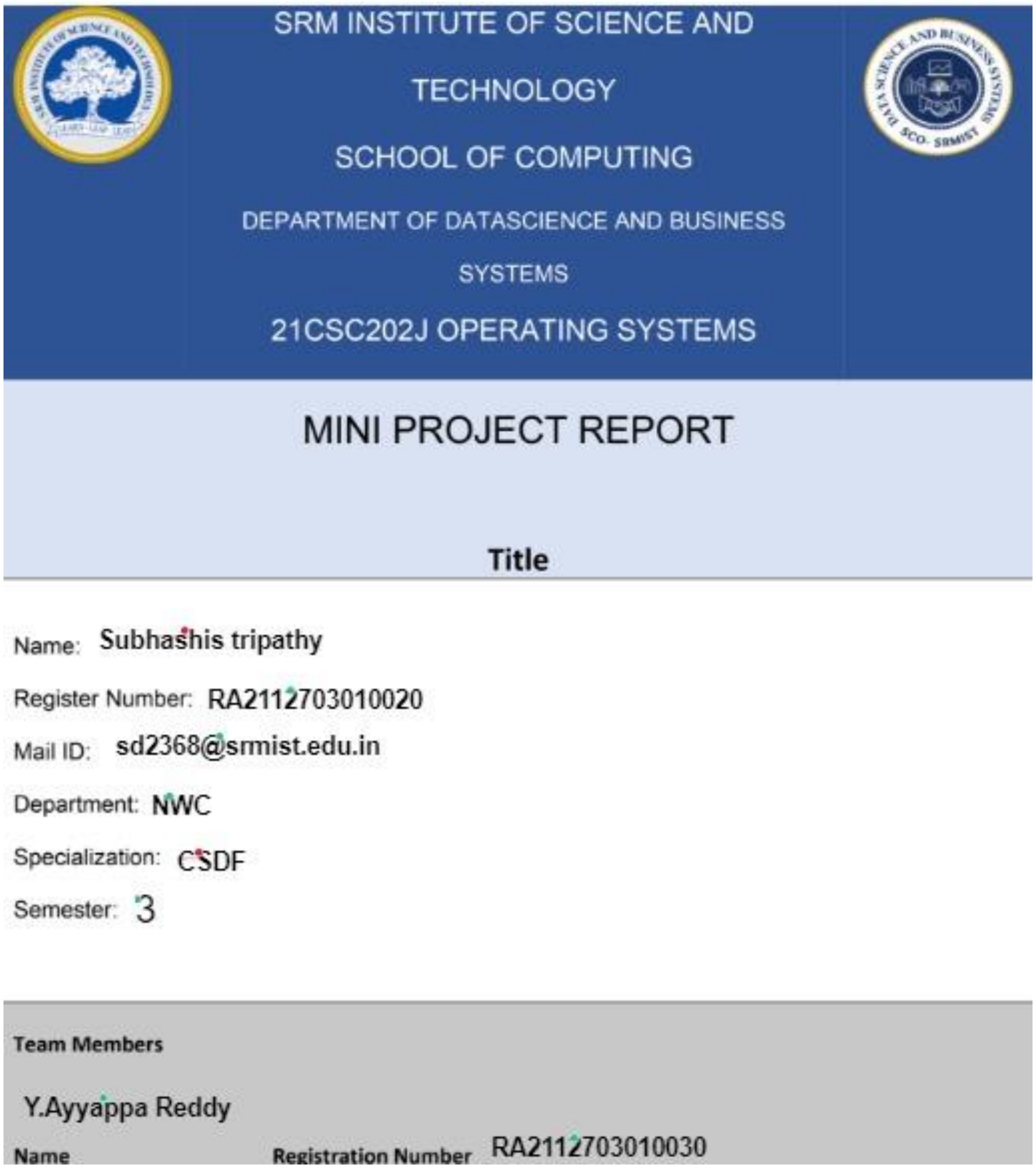


SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

SCHOOL OF COMPUTING

DEPARTMENT OF DATASCIENCE AND BUSINESS SYSTEMS

21CSC202J OPERATING SYSTEMS

MINI PROJECT REPORT

Title

Name: Subhashis tripathy

Register Number: RA2112703010020

Mail ID: sd2368@srmist.edu.in

Department: NWC

Specialization: CSDF

Semester: 3

Team Members

Y.Ayyappa Reddy

Name **Registration Number** RA2112703010030

ABSTRACT

Dekker's algorithm guarantees mutual exclusion, freedom from deadlock, and freedom from starvation. Let us see why the last property holds. Suppose p0 is stuck inside the while wants_to_enter loop forever. There is freedom from deadlock, so eventually p1 will proceed to its critical section and set turn = 0 (and the value of turn will remain unchanged as long as p0 doesn't progress). Eventually p0 will break out of the inner while turn ≠ 0 loop (if it was ever stuck on it). After that it will set wants_to_enter to true and settle down to waiting for wants_to_enter to become false (since turn = 0, it will never do the actions in the while loop). The next time p1 tries to enter its critical section, it will be forced to execute the actions in its while wants_to_enter loop. In particular, it will eventually set wants_to_enter to false and get stuck in the while turn ≠ 1 loop (since turn remains 0). The next time control passes to p0, it will exit the while wants_to_enter loop and enter its critical section.

If the algorithm were modified by performing the actions in the while wants_to_enter[1] loop without checking if turn = 0, then there is a possibility of starvation. Thus all the steps in the algorithm are necessary.

PROBLEM STATEMENT :

The problem of this first version of Dekker’s algorithm is implementation of lockstep synchronization. It means each thread depends on other to complete its execution. If one of the two processes completes its execution, then the second process runs. Then it gives access to the completed one and waits for its run. But the completed one would never run and so it would never return access back to the second process. Thus the second process waits for infinite time

INTRODUCTION

DEKKER'S ALGORITHM:

If two processes attempt to enter a critical section at the same time, the algorithm will allow only one process in, based on whose turn it is. If one process is already in the critical section, the other process will busy wait for the first process to exit. This is done by the use of two flags, wants_to_enter[0] and wants_to_enter[1], which indicate an intention to enter the critical section on the part of processes 0 and 1, respectively, and a variable turn that indicates who has priority between the two processes.

Processes indicate an intention to enter the critical section which is tested by the outer while loop. If the other process has not flagged intent, the critical section can be entered safely irrespective of the current turn. Mutual exclusion will still be guaranteed as neither process can become critical before setting their flag (implying at least one process will enter the while loop). This also guarantees progress as waiting will not occur on a process which has withdrawn intent to become critical. Alternatively, if the other process's variable was set the while loop is entered and the turn variable will establish who is permitted to become critical. Processes without priority will withdraw their intention to enter the critical section until they are given priority again (the inner while loop). Processes with priority will break from the while loop and enter their critical section.

Dekker's algorithm guarantees mutual exclusion, freedom from deadlock, and freedom from starvation. Let us see why the last property holds. Suppose p0 is stuck inside the while wants_to_enter[1] loop forever. There is freedom from deadlock, so eventually p1 will proceed to its critical section and set turn = 0 (and the value of turn will remain unchanged as long as p0 doesn't progress). Eventually p0 will break out of the inner while turn ≠ 0 loop (if it was ever stuck on it). After that it will set wants_to_enter[0] to true and settle down to waiting for wants_to_enter[1] to become false (since turn = 0, it will never do the actions in the while loop). The next time p1 tries to enter its critical section, it will be forced to execute the actions in its while wants_to_enter[0] loop. In particular, it will eventually set wants_to_enter[1] to false and get stuck in the while turn ≠ 1 loop (since turn remains 0). The next time control passes to p0, it will exit the while wants_to_enter[1] loop and enter its critical section.

If the algorithm were modified by performing the actions in the while wants_to_enter[1] loop without checking if turn = 0, then there is a possibility of starvation. Thus all the steps in the algorithm are necessary.

The solution to critical section problem must ensure the following three conditions:

- Mutual Exclusion
- Progress
- Bounded Waiting

First version

- Dekker’s algorithm succeeds to achieve mutual exclusion.
- It uses variables to control thread execution.
- It constantly checks whether critical section available.

Second Version

In Second version of Dekker’s algorithm, lockstep synchronization is removed. It is done by using two flags to indicate its current status and updates them accordingly at the entry and exit section.

Third Version

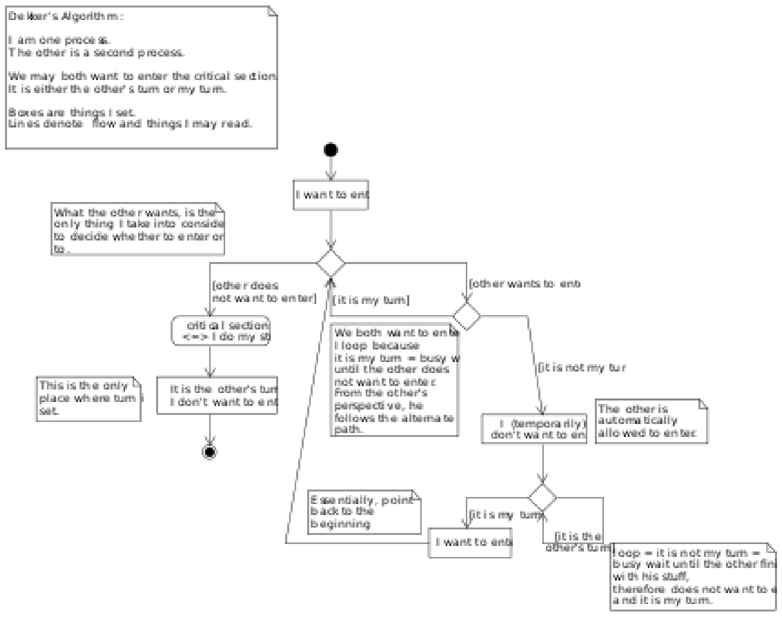
In this version, critical section flag is set before entering critical section test to ensure mutual exclusion.

Fourth Version

In this version of Dekker’s algorithm, it sets flag to false for small period of time to provide control and solves the problem of mutual exclusion and deadlock.

Fifth Version (Final Solution)

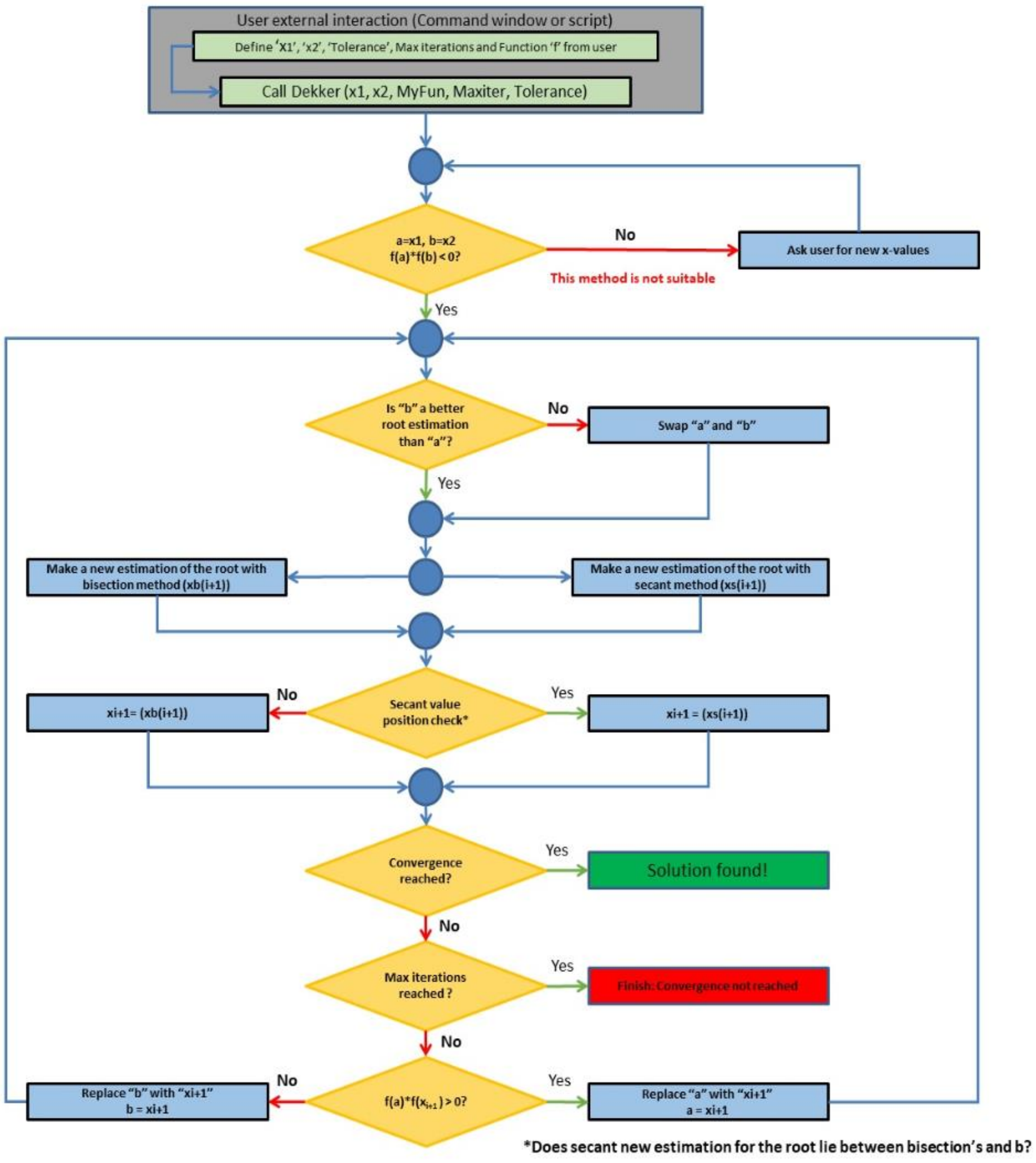
In this version, flavored thread motion is used to determine entry to critical section. It provides mutual exclusion and avoiding deadlock, indefinite postponement or lockstep synchronization by resolving the conflict that which thread should execute first. This version of Dekker’s algorithm provides the complete solution of critical section problems.


Dekker's Algorithm:
I am one process.
The other is a second process.
We may both want to enter the critical section.
It is either the other's turn or my turn.
Boxes are things I set.
Lines denote flow and things I may read.
I want to ent
What the other wants, is the only thing I take into conside to decide whether to enter or to.
[other does not want to enter]
[it is my turn]
[other wants to ent
critical section <=> I do my st
We both want to ente I loop because it is my turn = busy w until the other does not want to enter. From the other's perspective, he follows the alternate path.
[it is not my tur
This is the only place where turn set.
It is the other's tur I don't want to ent
I (temporarily) don't want to en
The other is automatically allowed to enter.
Essentially, point back to the beginning
[it is my turn]
I want to ent
[it is the other's turn]
loop = it is not my turn = busy wait until the other fin with his stuff, therefore does not want to e and it is my turn.

LIMITATIONS OF DEKKER ALGORITHM

One disadvantage is that it is limited to two processes and makes use of busy waiting instead of process suspension. (The use of busy waiting suggests that processes should spend a minimum amount of time inside the critical section.)

FLOW CHART


User external interaction (Command window or script)
Define 'X1', 'x2', 'Tolerance', Max iterations and Function 'f' from user
Call Dekker (x1, x2, MyFun, Maxiter, Tolerance)
a=x1, b=x2
f(a)*f(b) < 0?
No
This method is not suitable
Ask user for new x-values
Yes
Is "b" a better root estimation than "a"?
No
Swap "a" and "b"
Yes
Make a new estimation of the root with bisection method (xb(i+1))
Make a new estimation of the root with secant method (xs(i+1))
Secant value position check*
No
xi+1= (xb(i+1))
Yes
xi+1 = (xs(i+1))
Convergence reached?
Yes
Solution found!
No
Max iterations reached ?
Yes
Finish: Convergence not reached
No
f(a)*f(x_{i+1}) > 0?
No
Replace "b" with "xi+1"
b = xi+1
Yes
Replace "a" with "xi+1"
a = xi+1
*Does secant new estimation for the root lie between bisection's and b?

WORKING

Dekker’s algorithm will allow only a single process to use a resource if two processes are trying to use it at the same time. The highlight of the algorithm is how it solves this problem. It succeeds in preventing the conflict by enforcing mutual exclusion, meaning that only one process may use the resource at a time and will wait if another process is using it. This is achieved with the use of two "flags" and a "token". The flags indicate whether a process wants to enter the critical section (CS) or not; a value of 1 means TRUE that the process wants to enter the CS, while 0, or FALSE, means the opposite. The token, which can also have a value of 1 or 0, indicates priority when both processes have their flags set to TRUE.

This algorithm can successfully enforce mutual exclusion but will constantly test whether the critical section is available and therefore wastes significant processor time. It creates the problem known as lockstep synchronization, in which each thread may only execute in strict synchronization. It is also non-expandable as it only supports a maximum of two processes for mutual exclusion.

Implementation requirements

1.Importing libraries in the system

2.Uploading / Selecting the Image in the System

3.Running the Triple DES (3DES) Algorithm

4.Running the Encryption Process

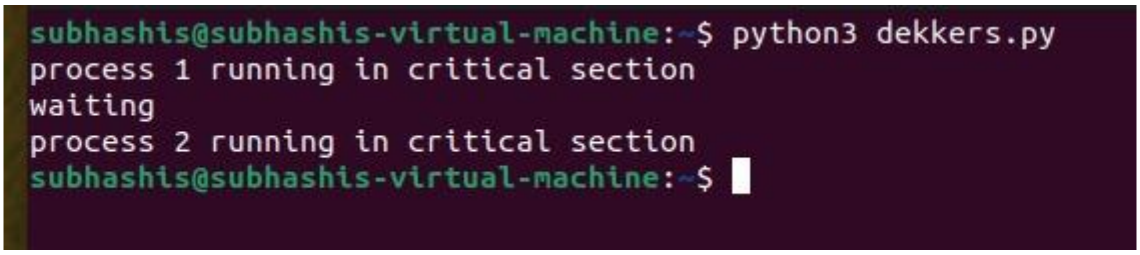
5.Running the decryption process

SOURCE CODE :

```
import threading

import random

import time


class DekkersAlgorithm():

    #  flags to indicate if each prcoess is in queue to enter its critical section

    flag = [False, False]

    # to denote which prcoess will enter next

    turn = 0
```

```
def process0(self):
    #process 0 wants to enter
    self.flag[0] = True
    while self.flag[1] == True:
        #if process1 is already running..
        if self.turn == 1:
            # gives access to other prcoess  wait for random amount of time
            self.flag[0] = False
            while self.turn == 1:
                #process 0 is waiting for process 1 to complete
                print("waiting")
            self.flag[0] = True
    #critical section
    print("process 1 running in critical section")

    # remainder section
    self.turn = 1
    self.flag[0] = False



def process1(self):
    #process 1 wants to enter
    self.flag[1] = True
    while self.flag[0] == True:
        #if process0 is already running..
        if self.turn == 0:
            # gives access to other prcoess  wait for random amount of time
            self.flag[1] = False
            while self.turn == 0:
                #process 1 is waiting for process 0 to complete
                print("waiting")
            self.flag[1] = True
        #critical section
        print("process 2 running in critical section")

        # remainder section
        self.turn = 0
        self.flag[1] = False

    def main(self):
        while True:
            t1 = threading.Thread(target = self.process0)
            t1.start()
            t2 = threading.Thread(target = self.process1)
            t2.start()
            exit(5);
if __name__ == "__main__":
    d = DekkersAlgorithm()
    d.main()
exit(5);
```

OUTPUT



```
subhashis@subhashis-virtual-machine:~$ python3 dekkers.py
process 1 running in critical section
waiting
process 2 running in critical section
subhashis@subhashis-virtual-machine:~$ 
```

CONCLUSION:

We saw that Dekker's algorithm while quick to state, actually resulted in a complex system with a very large number of states compared to its program graph. Starting with 10 program graph 'states' for a single thread it generated over 150 transition system states for two threads.