CPU Deadlock Performance Report

# By: Dylan Silva-Rivas

# Deadlock Overview

I will explain all of the necessary deadlock conditions and explain their relevance.

1. Mutual exclusion: A single process can use a resource at a time.
2. Hold and wait: Process is holding one or more resources while waiting for resources.
3. No Preemption: Resource can’t be allocated unless another process releases the resource.
4. Circular wait: A set of processes waiting for resources while taking resources in a circular fashion.

While looking into CPU scheduling algorithms, none of them inherently cause deadlock. The way a CPU scheduling algorithm is implemented can cause deadlock. Which is another reason why the deadlock conditions are necessary for this simulation.

# Deadlock Detection

The system that was created has given resource requirements, which made Banker’s algorithm a good choice for the deadlock detection algorithm. The algorithm was implemented by creating a struct data type called Process\_With\_R, which stands for process with resources. The struct is made up of seven different attributes: Process ID (String), resource allocation (Integer array), maximum resource request (Integer array), resource availablity (Integer array), resource need (Integer array), process ran (Boolean), and run time (integer). To calculate the safe sequence, the main function first calls the bankers\_Algo\_Avoidance function which takes in a reference to the process array, a reference to the safe sequence array, the size of process array, and the resource count as parameters. The function first calculates the total for all process allocations, second, calculates the need matrix, then finds the safe sequence relative to the need matrix. The safe sequence calculation has a time complexity of O(n^2\*m). n = process count, and m = resource count. The safe sequence array will contain the index locations of the processes, which will be used later to print out the processes.

# Simulation of Deadlock Scenarios

The program will first display a menu with three options: Low processes, medium processes, and high processes. Low processes will simulate five processes, medium processes will simulate ten processes, and large processes will simulate twenty processes. Each of the options will access the threading function if the safe sequence is detected, if not, it won’t run, which communicates that deadlock was detected. Let’s say the safe sequence is generated, this will mean the process array will start creating threads for each process, which will run in the safe sequence order. The threads will access a function that’s managed by mutual exclusion, limiting one thread to access the critical section at a time. During the running process, the user will be notified which process is currently running. When all processes are done running and are all synchronized, the runtime of Banker’s algorithm will be calculated and showed to the user.

# Deadlock Handling

Banker’s algorithm can be used to handle and detect deadlocks. The reasoning behind this answer is because the safe sequence that Banker’s algorithm creates determines if a system runs all the way through or not, which technically prevents the deadlock before it even happens. The way this was implemented is an extension of bankers\_Algo\_Avoidance function, the extension being a return value of 1 or 0. If the safe sequence is calculated, the function returns a 1, if not it returns a 0. This determines if processes will runs as threads.

# Performance Impact

The deadlock detection and handling techniques are the same, but there is some performance issues relative to Banker’s algorithm. The main issue with the algorithm is its time complexity, which is O(n^2\*m). This performance measurement causes huge amount of overhead, relative to the amount of processes and resources there are in a system. This is heavy on computational resources to calculate but the wait time for each process isn’t affected. The reason behind this is because processes don’t run until Banker’s algorithm is done finding the safe sequence. And to add on to performance, when a user chooses all three options, the time it takes for each algorithm to execute increases by a multiple of half or two when the amount of processes double. Here is some data that I received when I simulated the system:

|  |  |  |  |
| --- | --- | --- | --- |
| Program Executions | Low Processes (5) | Medium Processes (10) | High Processes (20) |
| 1 | 2084 | 2224 | 3096 |
| 2 | 651 | 1573 | 2845 |
| 3 | 1203 | 1302 | 2305 |
| 4 | 822 | 1293 | 2144 |
| 5 | 832 | 1824 | 2265 |
| 6 | 802 | 1313 | 2274 |
| 7 | 811 | 1403 | 2354 |
| 8 | 792 | 1342 | 2365 |
| 9 | 1052 | 1232 | 1824 |
| 10 | 791 | 1583 | 2355 |
| Total: | 9840 | 15089 | 23827 |
| Average: | 984 | 1508.9 | 2382.7 |

# Comparison of Detection and Handling

Since the detection and handling techniques are the same there was no difference in performance between the two identification schemes. So, I will describe the pros and cons of Banker’s algorithm.

Pros:

* Can used as a deadlock detection and handling scheme.
* Avoid deadlocks reliably.
* Easy implementation.

Cons:

* A lot of overhead.
* Doesn’t scale well because of its time complexity.
* Can’t be applied to a preemptive system.
* Need to know resources beforehand.

# Conclusion

Observations of this algorithms performance has lead to a mixed review on it. Yes, the algorithm gets its job done reliably, it’s easy to implement; but, it has a lot of overhead and isn’t scalable. Through analysis, the Banker’s algorithm is completely efficient in a small system, that has known resources. For bigger systems, other algorithms will have to be applied such as wait-for graph algorithm for detection and rollback and abort algorithm for deadlock handling.