Deadlock Avoidance and Detection in Real-Time Operating Systems

Sidharth Nair
University of Texas at Austin

1 Introduction

Real-time operating systems are critical in embedded systems, ensuring timely execution of tasks and efficient resource management [Li et al. (1997)]. However, the occurrence of deadlocks can severely impact system performance and reliability. Deadlocks arise when multiple processes compete for resources, leading to a situation where each process is waiting for a resource held by another process, resulting in a cyclic waiting dependency. Without mechanisms to handle this, a safety-critical system could be stuck until reset by an administrator.

This project aims to enhance the functionality of a real-time operating system (RTOS) designed for the TM4C123G micro-controller [*EK-TM4C123GXL Evaluation board — TI.com* (n.d.)], by implementing deadlock avoidance and detection mechanisms. For avoidance, we explore Banker's algorithm to ensure safe resource allocation in a system with a pre-defined amount of threads and knowledge on the maximum request limits of each thread. However, this knowledge is often not known ahead of time, so we also explore periodically detecting (and breaking) potential deadlocks by constructing a wait-for graph. This report presents the design, implementation, and evaluation of these techniques within the RTOS environment. The code for this project can be found at https://github.com/sidharthNair/deadlock-detection.

2 Background

2.1 Deadlocks

A deadlock occurs when a set of processes are blocked, waiting on resources that each other hold such that none of the processes can execute. There are four necessary conditions for deadlock to occur [Shub (2003)]:

Mutual exclusion: no two processes can exist in a critical section at any given point of time.

Hold and wait: a process is holding one or more resources while also waiting for another resource.

No preemption: a resource cannot be taken away from a process unless it voluntarily releases it.

Circular wait: there are a set of of processes waiting on each other in a circular fashion.

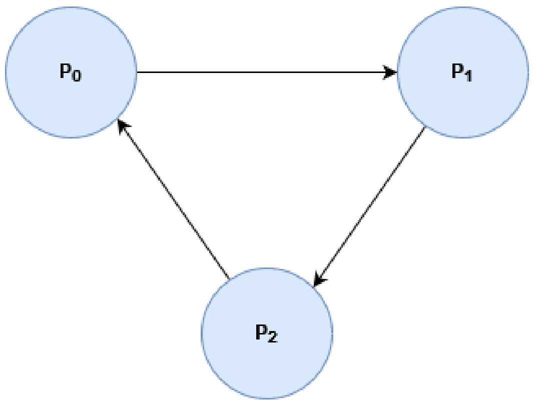

Figure 1: Simple deadlock example. An arrow from process i to process j indicates that process i is waiting for a resource held by process j

To give a simple example, say we have a set of processes $\{P_0, P_1, P_2\}$ and a set of resources $\{R_0, R_1, R_2\}$. If P_0 holds R_0 and is waiting for R_1, P_1 holds R_1 and is waiting for R_2, and P_2 holds R_2 and is waiting for R_0, then we can construct a dependency graph ("wait-for" graph) as shown in figure 1. Here it is clear to see the cyclic dependencies, and that it is impossible for this system to make progress.

2.2 Real-Time Operating Systems

Operating systems are an important layer of software that manages system resources (CPU, memory, etc.) and facilitates communication between software applications and hardware. They provide features like task scheduling, interrupt handling, and resource management, simplifying the development process and ensuring consistent behavior across different projects. Real-time operating systems (RTOS) are specialized software systems designed to meet strict timing requirements for tasks and processes in embedded systems. Unlike general-purpose operating systems, RTOS prioritize deterministic behavior and timely response to events. As a result, they are commonly used for industrial automation, automotive systems, medical devices, aerospace, and telecommunications.

2.3 Deadlock Prevention

A number of techniques for deadlock prevention have been developed in the past decades, and there has not been many changes algorithmically-wise in these approaches. In RTOS environments, where hardware resources are often limited and timing constraints are strict, the overhead introduced by deadlock detection may outweigh its benefits. However, when the system is not hardware-bound, deadlock detection can be valuable for ensuring system reliability and availability. Additionally, deadlock detection mechanisms can serve as valuable debugging tools, helping developers identify and resolve potential issues in system design and resource allocation. In the following sections we cover two approaches for handling deadlocks: avoidance and detection.

2.3.1 Deadlock Avoidance

With deadlock avoidance we want to proactively ensure that deadlock conditions cannot arise in the system. This is achieved by carefully managing resource allocation and scheduling to avoid the circular waiting dependency among processes that leads to deadlock. Here we focus on the Banker's algorithm to ensure safe resource allocation and prevent deadlock situations from occurring.

The Banker's Algorithm Dijkstra's Banker's algorithm ensures deadlock avoidance by rejecting resource requests that could lead to unsafe system states [Dijkstra (1968)]. The algorithm assumes that there are a set of n processes and r types of resources in the system. Additionally, each process must specify its maximum potential resource requirement in advance. Every process commits to completing its tasks and returning resources to the system if it receives its maximum requested amount. To detect whether a request may cause a deadlock, we simulate what would happen *if the request was granted* and check whether there exists a safe sequence of process completions (a process requests its remaining resources and then releases them) such that every process can get access to their requested resources. If so, the request is granted; otherwise, it is rejected and the process must wait until more resources become available.

Algorithm 1: Banker's Algorithm; ran before granting any request

```
Input: Available: array[r] of int, number of available instances of each resource type.
       Max: matrix[n][r] of int, maximum number of resources each process may request.
       Allocation: matrix[n][r] of int, number of resources currently allocated to each process.
       Need: matrix[n][r] of int, remaining resources needed by each process for completion.
Output: Safe or unsafe state
1  while all $P_i$ have not been marked as completed do
2      forall $P_i$ do
3          if $P_i$ is not completed and Need[i] $\leq$ Available then
4              Available += Allocation[i]
5              Mark $P_i$ as completed
6          end
7      end
8      if no processes were marked as completed in this iteration then
9          return System is in an unsafe state (deadlock may occur)
10     end
11 end
12 return System is in a safe state
```

The key idea of the Banker's algorithm is that knowing the maximum resource requirements of each process upfront allows the system to proactively determine if deadlock is possible. A RTOS is well suited for this since often we know exactly what code each thread will be running (and how they will be scheduled) on our system. However, in applications where we don't have this information upfront, the Banker's algorithm is not very useful. Additionally, the safety check (which needs to be ran on every resource request) has a worst-case time complexity of O(n^2r), where n is the number of threads in the system and r is the number of resource types, which can make this an expensive operation as the number of threads grows. The space complexity of the algorithm is O(nr).

2.3.2 Deadlock Detection

Deadlock detection involves periodically examining the system to determine whether a deadlock has occurred. Unlike deadlock avoidance, where the system actively avoids deadlock by careful resource allocation, deadlock detection focuses on identifying deadlock situations after they have occurred. Once a deadlock is detected, appropriate actions can be taken to recover from it, such as terminating one or more processes or rolling back their actions.

Wait-for Graph The key idea in most deadlock detection algorithms is finding cycles in a ”wait-for” graph that represents dependencies between processes [Ni et al. (2009)], similar to what is shown in figure 1. If we can construct this graph, then cycle detection can be performed using a graph traversal algorithm like BFS or DFS traversal. For this project, we simplify this problem by only performing deadlock detection with locks, which can only be held by one thread at a time and a thread can only be waiting on one lock at a time. This means that each process can only have at most one outgoing edge, and the graph can be traversed in the same manner as a linked list.

Upon detecting a deadlock, the system needs to recover to a consistent state. Recovery mechanisms may include:

- Terminating one or more processes involved in the deadlock to free resources and break the deadlock.
- Temporarily preempting resources from one or more processes to allocate them to others.
- Rolling back the actions of one or more processes to a previous consistent state.

In this project, we use the first approach and kill *all* threads that are involved in a deadlock. We make sure to free all resources that the thread is holding on to in its blocked state, so that other threads can continue.

Deadlock detection does have a non-negligible overhead due to the periodic checks required to examine the system state for deadlock. The complexity of this check is is linear with the number of nodes and edges in the wait-for graph (in our case, linear with the number of threads in the system). While this cost may grow, the system reliability it provides often justifies these expenses. Additionally, we can increase the detection period to avoid running the algorithm too often.

3 Implementation

This section covers the implementation details of the described deadlock prevention mechanisms in a RTOS built for the TM4C123G micro-controller. The base RTOS supports periodic interrupt routines, priority scheduling, blocking semaphores, and dynamic memory allocation to name a few of the key features that were extended and/or used for this project.

3.1 Banker's Algorithm

Banker's algorithm was implemented as a sub-module in our RTOS, similar to the heap manager and the file system. The API and implementation can be found in `bankers.h` and `bankers.c` respectively. The implementation mainly follows the pseudo-code provided in section 2.3.1, and all data structures (vectors and matrices) were allocated on the heap. Additionally, all accesses to shared data structures are protected by a semaphore to ensure no data races.

We provide blocking and non-blocking versions of the resource request functions to allow flexibility in how these are called. Additionally, a debug flag can be enabled to print out (via UART) the state of each data structure and whether or not a safety sequence was found for each resource request/release operation. One important extension to our operating system that was needed to implement the above was adding signal all functionality to the semaphore library; specifically, we implemented `OS_SignalAll()` which will release all threads waiting on a semaphore. This is used whenever a release request is made, since there may be multiple threads with rejected requests who are waiting to try again.

3.2 Cycle Detection

For cycle detection we first implemented a Lock structure on top of our semaphore library to be in a setting where only one thread can hold a resource. Additionally, no thread other than the holder is allowed to release the resource. The API for this structure can be found in `OS.h`. Each Lock structure internally contains a semaphore, thread control block (TCB) pointer, and a next Lock pointer. The TCB pointer is updated to the current holder (which also contains a pointer to the current Lock it is blocked on) whenever the lock is acquired and is used to maintain the wait-for graph that will be used for deadlock detection. The next Lock pointer is used to maintain a linked list of all Locks that a thread holds, allowing us to efficiently release all resources acquired by a thread in the event that it is killed early.

We set up a periodic timer interrupt to run every 3 seconds (configurable) to run our cycle detection algorithm. The interrupt iterates over all threads in the system and checks how long each thread has been blocked for (blocked starting time is stored in the thread's TCB when a lock is acquired). If the thread has been blocked longer than the period of the ISR, then we run our cycle detection algorithm starting from that thread. This heuristic ensures we only run our algorithm when it might actually be needed; if a thread has been blocked for over 3 seconds, then it is very likely it is involved in a deadlock.

With the data structure changes mentioned above, the cycle detection algorithm is quite straight forward. From the starting thread's TCB, we simply keep traversing to the TCB of the holder of the

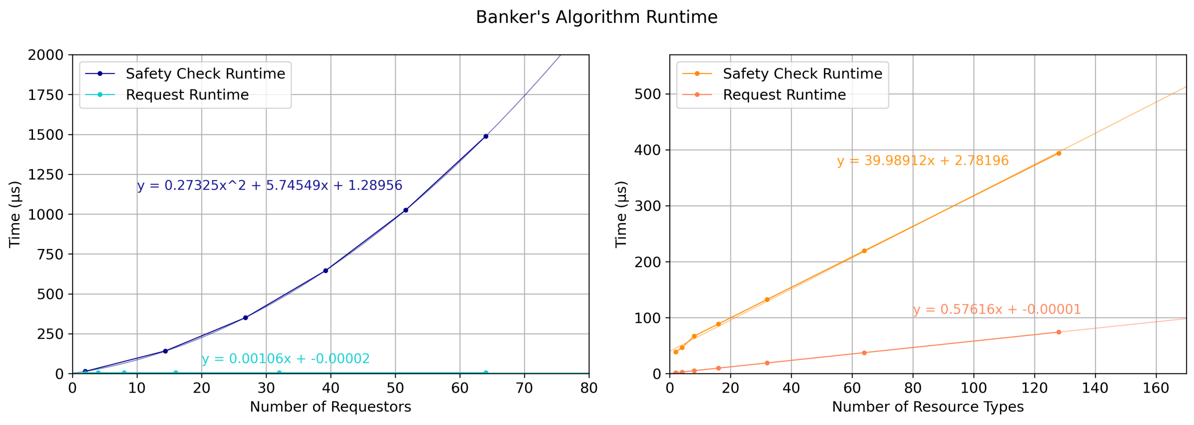

Figure 2: Runtime of Banker's algorithm vs. number of requestors and number of resource types. Request Runtime is the time to update the allocation, need, and available vectors, and Safety Check Runtime is the time required to see if the update will put the system into an unsafe state.

lock that the current thread's TCB is waiting on. This is equivalent to traversing the wait-for graph, and we continue until we reach `NULL` or a TCB that we have seen before. In the latter case we know we have found a cycle and can perform the recovery logic. For this we simply kill all threads in the cycle, releasing all their held resources while doing so. A debug flag can be set to print information about cycle detection and killed threads.

4 Performance Results

In this section we go over the benchmarking results for each of the algorithms implemented to prevent deadlocks. Code was profiled using pin toggling and a logic analyzer to minimally impact execution. The data for the graphs shown in this section can be found in appendices A and B.

4.1 Banker's Algorithm

For Banker's algorithm, in figure 2, and tables 1 and 2, we show performance scaling against both number of requestors and number of resource types in the system. Number of resource types and requestors was held constant (at a value of 8) in the respective benchmarks. For each measurement, we constructed a test case that would result in the worst-case time for Banker's algorithm. This corresponds to the case where the safe sequence is $n-1, n-2, ..., 2, 1$. Additionally, we show the time made for the request, before the safety check is executed.

We see that request time scales linearly with number of resource types, and the safety check time scales quadratically with number of requestors and linearly with number of resource types. This is expected from our complexity analysis in section 2.3.1. Banker's algorithm clearly gets quite expensive as the number of threads grow. Additionally, this cost is incurred on *every* request, which should be considered if used in an application where resources are frequently requested and released. However, given that Banker's

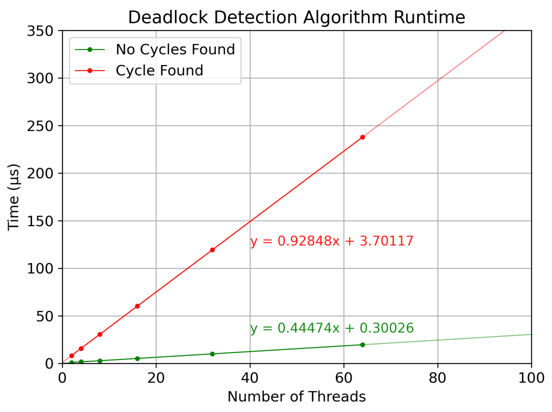

Figure 3: Runtime of the deadlock (cycle) detection algorithm vs. number of threads in the system. We report the time where a cycle is not found and the time where one is found (includes the time to kill the threads in the cycle).

algorithm was originally intended to be used to manage access to hardware resources, it is unlikely that we would see such a large amount of requestors and resource types, which is why this algorithm is suitable for a RTOS.

4.2 Cycle Detection

For the cycle detection algorithm, we used the classical Dining Philosophers [Dijkstra (1971)] problem to benchmark the system, with number of threads equalling the number of philosophers. We report the runtime of the interrupt service routine when deadlock is and is not detected in figure 3 and table 3. Both are relevant since they form an upper and lower bound on the time we should expect to see from the algorithm with the corresponding number of threads and resources. Additionally, deadlocks should generally not be very frequent so we should expect to see the runtime with no cycles more often than with a cycle.

We see that the runtime scales linearly with number of threads, which is expected since we need to traverse a linked list of all threads during the check. This cost is incurred every 3 seconds (deadlock detection period), but it is generally low since we almost never detect a cycle.

4.3 Analysis

Given that the cycle detection algorithm scales better with number of threads and has a lower space complexity, both of which are quite important in a RTOS setting, it seems beneficial to use it over Banker's algorithm. However, some critical applications may need guarantees that the system can never get into such a state, since it does take time for the detection algorithm to run. In such cases, Banker's algorithm may be the better choice (if we have knowledge on the number of threads and resource requests in advance) to enforce that resources are requested safely. As a result, the choice of one of these over the other should be decided on a case-by-case basis, depending on how the hardware constraints of the system and what level of strictness we need from our deadlock prevention mechanism.

5 Conclusion

The implemented deadlock prevention mechanisms, Banker's algorithm for avoidance and cycle detection algorithm, were evaluated for their performance in a real-time operating system environment.

Banker's algorithm showed significant scalability challenges, particularly with an increase in the number of requestors and resource types. As the number of threads grew, the runtime of safety checks increased quadratically, making it less suitable for systems where frequent resource requests occur. On the other hand, the cycle detection algorithm exhibited linear scalability with the number of threads. Although it incurs periodic overhead due to the detection check, this overhead remains relatively low and acceptable, especially considering the critical importance of avoiding deadlocks in real-time systems.

In conclusion, while Banker's algorithm may provide stronger guarantees against deadlocks, its scalability limitations and performance overhead make it less preferable for many real-time operating system applications. Cycle detection, offers a practical and efficient solution for deadlock prevention in such environments. Ultimately, the choice between these approaches should be guided by requirements of the system and application being developed.

References

Dijkstra, E. W. (1968). Co-operating sequential processes. In F. Genuys (Ed.), *Programming languages: Nato advanced study institute: lectures given at a three weeks summer school held in villard-le-lans, 1966* (pp. 43–112). Academic Press Inc.

Dijkstra, E. W. (1971). Hierarchical ordering of sequential processes. *Acta Informatica*, *1*, 115–138. DOI: 10.1007/BF00289519

Ek-tm4c123gxl evaluation board — ti.com. (n.d.). Retrieved 2024-04-30, from https://www.ti.com/tool/EK-TM4C123GXL (Accessed Apr. 30, 2024)

Li, Y., Potkonjak, M., & Wolf, W. (1997). Real-time operating systems for embedded computing. In *Proceedings international conference on computer design vlsi in computers and processors* (p. 388-392). DOI: 10.1109/ICCD.1997.628899

Ni, Q., Sun, W., & Ma, S. (2009). Deadlock detection based on resource allocation graph. In *2009 fifth international conference on information assurance and security* (Vol. 2, p. 135-138). DOI: 10.1109/IAS.2009.64

Shub, C. (2003, 10). A unified treatment of deadlock. *Journal of Computing Sciences in Colleges*, *19*, 194-204.

A Performance Results for Banker's Algorithm

Requestors	**Request Execution (μs)**	**Safety Check Execution (μs)**
2	5.167	13.958
4	5.167	26.583
8	5.167	66.750
16	5.208	163.958
32	5.167	463.958
64	5.167	1488.417

Table 1: Runtime of Banker's algorithm vs. number of requestors. Number of resource types = 8.

Resources	**Request Execution (μs)**	**Safety Check Execution (μs)**
2	1.750	38.583
4	2.875	46.458
8	5.167	66.917
16	9.792	88.750
32	19.000	132.375
64	37.417	219.542
128	74.167	393.917

Table 2: Runtime of Banker's algorithm vs. number of resource types. Number of requestors = 8.

B Performance Results for Cycle Detection Algorithm

Threads	**No Cycle (μs)**	**Found Cycle (μs)**
2	1.042	8.250
4	1.667	15.792
8	2.833	30.542
16	5.250	60.167
32	10.042	119.375
64	19.667	237.792

Table 3: Runtime of the deadlock detection algorithm vs. number of threads