Design

Deadlock Detection Simulator

Sagalpreet Singh

13th October, 2021

Introduction

1. What’s it?
2. Logic
3. Heuristics Used for selecting victim thread
4. Results

What’s it?

The program simulates the deadlock scenario in cases where resources (multiple instances) are acquired by processes (threads in the case of program) in such a way that it becomes impossible for all the processes to make progress. The program handles deadlocks caused in such a manner by:

a) Detecting that deadlock has occurred

b) Identifying the processes(simulated as threads) involved in the deadlock

c) Terminating some of these processes to release resources bound to them so that other processes can progress and the deadlock is broken

d) For each of the deadlock, we need to keep a track of the duration between consecutive deadlocks while using different heuristics in selecting the process to be terminated first

e) The number of resources of different types that are available have to be taken as command line arguments

f) A thread is used to simulate a process. A thread continuously puts forward a set of resources (generated randomly) required by it. It tries to get those resources (with some random delay in between acquiring different resources) and the moment it has all the required resources, it waits for some time to simulate the process execution. After the wait is over, the thread releases all the resources acquired by it and starts off with a new set.

g) Whenever a process is terminated, essentially a thread is killed in simulation and a new thread is spawned in its place i.e the number of threads remains constant throughout the execution of the program.

h) Checking and handling of deadlocks is done on a dedicated separate thread which calls the function to detect deadlocks every "d" microseconds, where d is taken as a command line argument.

i) The program runs indefinitely until terminated forcefully by an interrupt

Logic

a) Simulating Processes: Using Threads. After taking the number of threads as command line arguments, a corresponding number of threads are spawned, each associated with a thread function.

This thread function randomly generates a set of required resources (with corresponding quantity). It tries to acquire these resources as and when available by constantly looking out for them. Each thread constantly keeps a check on if it is instructed by the deadlock handler to kill itself. The communication happens through shared memory.

After getting all the resources, it sleeps for some random time between 0.7d to 1.5d microseconds where d is the delay argument taken as a command. This simulates the process execution.

After this, the thread releases all the acquired resources. (it increments the number of freely available resources accordingly to simulate this)

b) Simulating resources: Resources are simulated using structures that contain the name of the resource, its total quantity available and the number of instances that are freely available to be acquired at present

c) Simulation of deadlock detection: The routine runs after every d microseconds where d is the delay argument taken as a command. To simulate this, it sleeps for d microseconds in every iteration of the while loop. To detect deadlock, the following algorithm is used:

- Try to go through all the available processes and check if any of these can complete successfully with the resources it has already acquired and the resources available freely to it.

- If such a resource is found, add the resources it has already acquired to the pool of free resources and mark it to be not involved in a deadlock. (Note that this step is performed on the copy of the available resources and the copy of resources acquired and not the original data structures as the process is still under progress while deadlock is being detected)

- Repeat the above two steps until no such process is found which can proceed further with given availability.

- At this point, all the processes which are not marked as deadlock free are involved in deadlock and can be reported.

- Now through some heuristic approach (which is discussed in detail in the Design document), it is decided as to which process to kill (i.e which thread to terminate), and that thread is asked to terminate itself. The communication happens through shared memory. (Communication via signals is not a good idea for threads)

- Now all the above steps are repeated again until the deadlock is resolved.

It is worth noting here that deadlock may resolve in a lot of different manners, depending upon how randomness proceeds. Say some thread if terminated resolves the deadlock completely and by heuristics, that thread is selected, still it may so happen that before the next check for deadlock existence, that thread has not terminated, so another thread will be instructed to terminate itself, although even if it wouldn't have done so, the deadlock would have been resolved in some time.

d) Spawning new threads when older ones die: Threads are spawned only by the main thread in the implementation for obvious reasons of avoiding complicacies. Whenever a thread terminates itself, it communicates to the main thread (parent thread essentially) to spawn a new thread in its place. The communication again takes place through shared memory space. The main thread continuously looks for such messages.

e) Inter thread communication: The communication between threads happens through shared memory via status bits (technically bytes, since char is used to store them). For example: To indicate to the main thread that the worker thread is exiting, it will unset the 'active' bit i.e active[i] = 0 where i is the mapping to id corresponding to that thread. Main thread observes this and spawns a new thread, storing the new thread id in place of the previous thread.

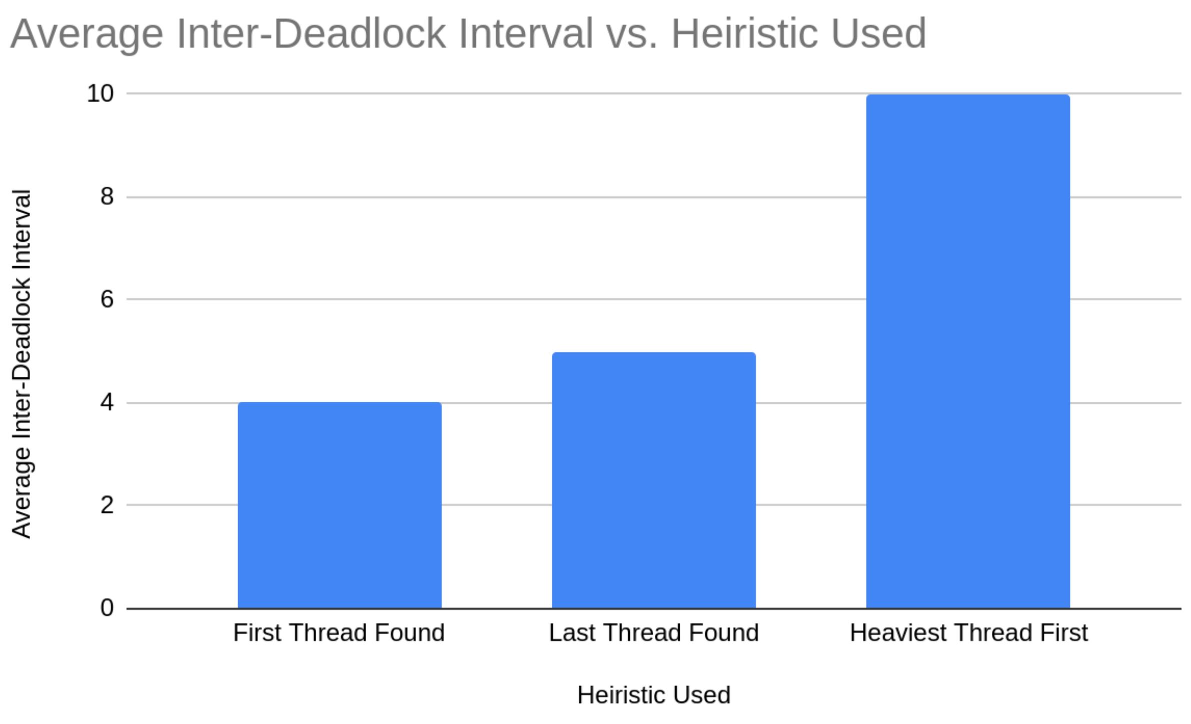
f) Tracking time between deadlocks: To track the time between deadlocks, all details related to the deadlocks are logged into a file with timestamp of when and because of which threads, the deadlock occured. Which threads had to be killed to resolve the deadlock etc.

NOTE: In the implementation, proper care is taken to keep the program thread safe by using mutex locks as and when required.

Heuristics Used for selecting victim thread

1) **First found Thread:** The first thread which was found to be involved in deadlock is selected as victim. This approach is a natural choice as it is simple and spends minimal amount of time in figuring out the best possible victim thread.
2) **Last found Thread:** This one is just a simple variation in the implementation of method 1 with the only difference being the order in which the victim thread is hunted for.
3) **Heaviest Thread First:** Here, the thread which currently acquires the maximum number of resources in total is chosen as victim with the reasoning that maybe freeing up such a heavy thread might facilitate progress of several smaller threads.

Results



(time in seconds)