Project 3

CS342 - Operating Systems

Efe Acer - 21602217
Yusuf Dalva - 21602867

Bilkent University, CS

Contents

Introduction

A *deadlock* is a state of a computer in which the processes running are stuck waiting for resources held by each other. If a deadlock occurs, there is no way of recovering to a safe (deadlock free) state without killing the waiting processes or deallocating resources from them.

Computer scientists developed ways to and avoid deadlocks. A common algorithm for deadlock avoidance and deadlock detection is the *Banker's Algorithm*. The algorithm basically tries to find an order of execution that allows each process to use and release the resources it has requested. If the algorithm finds such an order of execution, then it is said that the system is safe, in other words the system is free of deadlocks. Note that the algorithm cannot say how long it will take for all processes to terminate but it says that the processes will eventually terminate assuming that each process uses its resources for a finite amount of time. The Banker's Algorithm can also be used for deadlock detection with minor modifications in implementation.

Real Operating Systems usually do not implement deadlock avoidance or detection but they rather give the responsibility of building deadlock free programs to the developers.

In this project we simulate the behavior of an Operating System Kernel in terms of allocating resources. The Kernel we simulate can do deadlock avoidance or detection depending on its mode of operation. One can also choose for the Kernel to do nothing in case of managing deadlocks.

Implementation

As explained in the previous section, our Kernel can perform deadlock avoidance and deadlock detection. It performs both by the help of a resource allocation library, `ralloc`.

The `ralloc` library works as a *monitor*. A monitor is an abstract data type that can be thought as a class whose methods are guaranteed to be executed by a single caller at any instance. This requirement of a monitor is implemented using `mutex locks` from the `pthread` library. Each function in the `ralloc` library that can be potentially called by multiple processes (threads) has an `acquire lock` statement in its entry section and a `release lock` statement in its exit section; every statement in between is considered as a critical section that must be free of race conditions.

A monitor has queues containing the callers that are waiting to execute a function. These queues are implemented using a `condition variable`, again from the `pthread` library. A process is enqueued to a waiting queue if it calls wait on the condition variable, the process located in the head of the waiting queue is dequeued when a signal call is made on the condition variable. Also a broadcast operation dequeues all processes waiting in the queue.

The `ralloc` library implements safety checks using Banker's algorithm. In case of a resource request, the library checks the current system state's safety; if the state is safe the resources are given to the requesting process, otherwise the process is waited on the condition variable. Of course, the safety checks are different for the mode of operation that is set during library initialization.

When a process releases the resources it owns, the library updates the system state by deallocating the resources from the releasing process. Then the waiting processes are broadcasted on the condition variable, meaning all of them are dequeued. The dequeued processes are then continued from the point that they have started waiting. All checks are done inside while loops to enforce a continued process to pass the safety checks to further execute.

Experiment

We conducted a simple experiment to see the effect of the mode of operation of the `ralloc` library on its performance. Formally, the runtime of a safe resource allocation scheme is the experiment's dependent variable and the mode of operation (deadlock avoidance, deadlock detection or no operation (deadlock nothing)) is the independent variable. Hence, we set up a safe resource allocation scheme with 3 resource types and 5 processes. We set up timers in our experiment to measure the time elapsed between system initialization and the final state in which all processes finish their execution. We performed 10 runs for each mode of operation, the results of the experiment is given below:

Table 1. Execution time of sample case with different deadlock handling mechanisms (in miliseconds)

Trials	DEADLOCK_NOTHING	DEADLOCK_DETECTION	DEADLOCK_AVOIDANCE
1	250	281	427
2	305	333	423
3	308	344	356
4	249	274	357
5	243	285	355
6	241	341	404
7	252	341	457
8	254	278	474
9	255	337	511
10	307	279	354

Looking at the table above we see that the different modes of operations correspond the following range of runtime values (in milliseconds):

- Deadlock Nothing: 241-308
- Deadlock Detection: 274-344
- Deadlock Avoidance: 354-511

It is obvious that doing nothing in case of a deadlock provides the best performance. This is a strong expectation since no checks are made in this case. The checks in `ralloc` library are usually time consuming, since we need to simulate multiple allocation iterations to resolve whether a system is safe or not.

Deadlock Detection provides a better performance then Deadlock Avoidance, the reason for

this is again the number of checks made in the library. Deadlock Avoidance runs the safety check in each request, while the Deadlock Detection only runs the safety check after a process (thread) performs all its requests.

We conclude that the performance is negatively affected by the overhead caused by the safety checks.

Conclusion

As a result of our experiment, we conclude that an increase in the number of safety checks reduce the performance of a program using the `ralloc` library. Deadlock Avoidance performs the most number of checks, then comes Deadlock Detection and the least number of checks is made when there is no mechanism checking for deadlocks. The overhead is significant in the sense that it nearly doubles the runtime of a program. This is one of the reasons why real Operating Systems usually do not implement deadlock avoidance/detection algorithms.