Bilkent University

Department of Computer Engineering

CS342 Operating Systems

Project 3 Report

Authors:	*ID-s:*
Naisila Puka	21600336
Kunduz Efronova	21600469

April 21, 2019

Experiments

Set up

We have written a program called `experiment.c`, in which we initiate the resource allocation library with M resources. Then, the program creates N threads, and in each thread function, resources are frequently requested and released. The parent program waits for all threads to finish, and the time elapsed for all threads to finish is recorded and printed out.

In order to determine the overhead of deadlock avoidance to a certain extent, the experiment program initiates the library with `DEADLOCK_AVOIDANCE` in order to evaluate the overhead of deadlock avoidance compared to not using deadlock avoidance, in which case the library is initiated with `DEADLOCK_NOTHING`.

At the same time, the number of processes (N) and the number of resources (M) can be changed inside the program to better see the results using various parameters. Below you may find how to change the parameters:

–In lines **16** and **17**, change the number of resources as desired:

```
#define M 10 // number of resource types
int exist[10] = 12, 8, 10, 8, 10, 12, 8, 12, 8, 10; // resources
```

–In line **19**, change the number of threads as desired:

```
#define N 12 // number of processes - threads
```

–In line **96**, change the deadlock handling method as you desire:

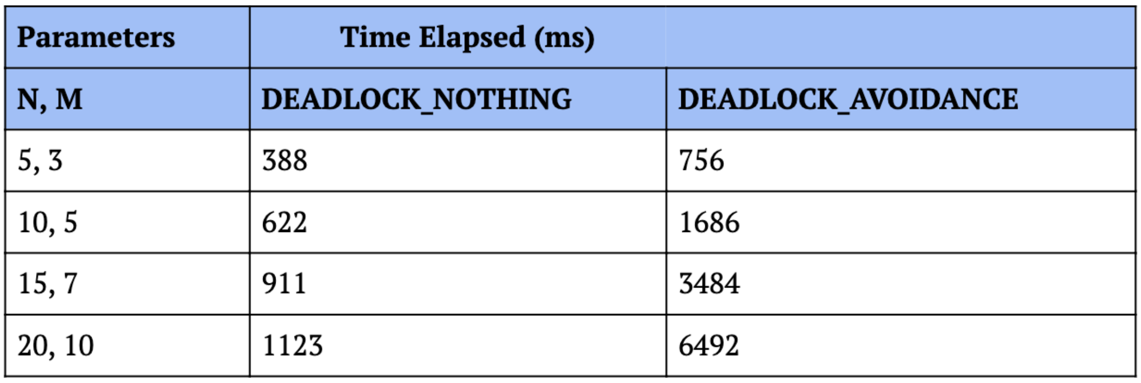
```
handling_method = DEADLOCK_AVOIDANCE;
```

By the way in which the program allocates resources, there is no possibility of deadlock, so the threads will ultimately finish. However, if we use `DEADLOCK_AVOIDANCE`, there is obvious overhead to using `DEADLOCK_NOTHING`, as will be explained in the following section.

DEADLOCK_AVOIDANCE vs DEADLOCK_NOTHING

In this subsection we will show results for the effect of frequently calling the deadlock avoidance algorithm which checks whether the system is left at a safe state after allocating requested resources to a specific program.

NOTE: *ALL TIMES ARE IN MICROSECONDS*



Parameters	**Time Elapsed (ms)**	
N, M	**DEADLOCK_NOTHING**	**DEADLOCK_AVOIDANCE**
5, 3	388	756
10, 5	622	1686
15, 7	911	3484
20, 10	1123	6492

Figure 1: Table of comparing AVOIDANCE with NOTHING

It is obvious that usage of `DEADLOCK_AVOIDANCE` has overhead compared to usage of `DEADLOCK_NOTHING`. We have also increased the number of processes and the number of resources in order to see how the overhead increases in different methods. We observe in the table in Figure 1 that overhead in usage of deadlock avoidance algorithm increases more than linearly, whereas usage of no algorithm at all increases by a constant factor of almost 300 ms.

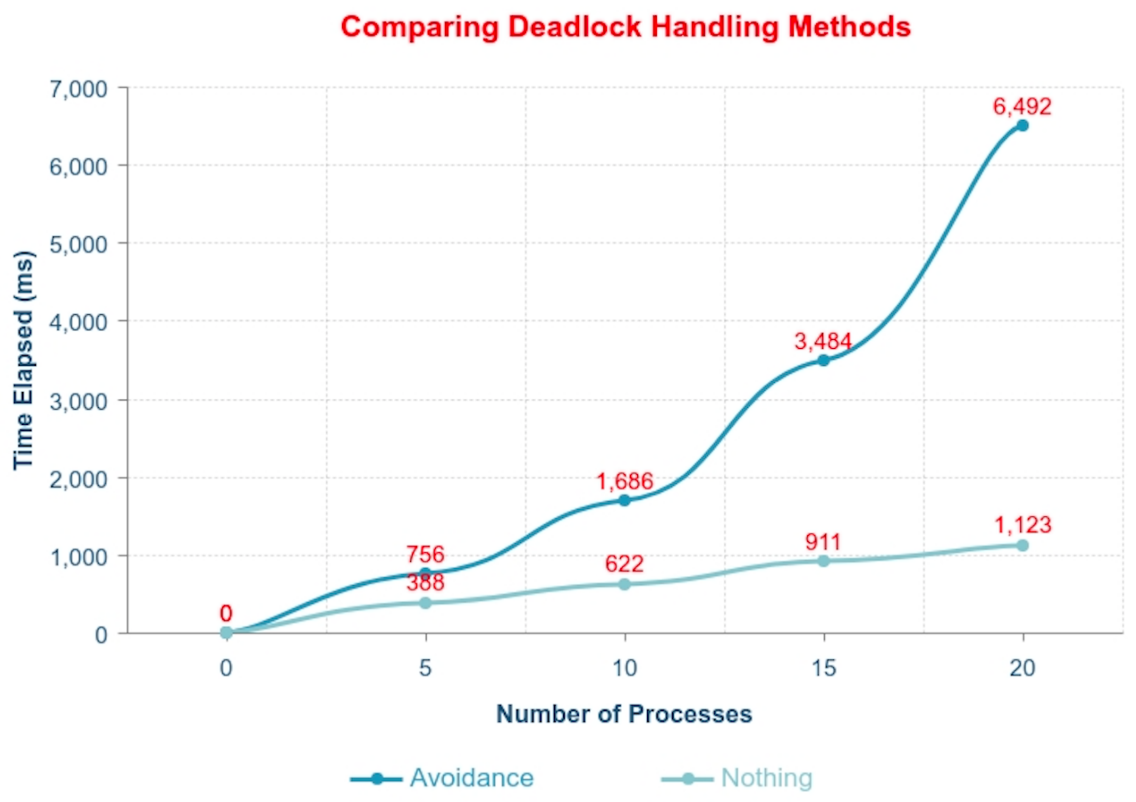
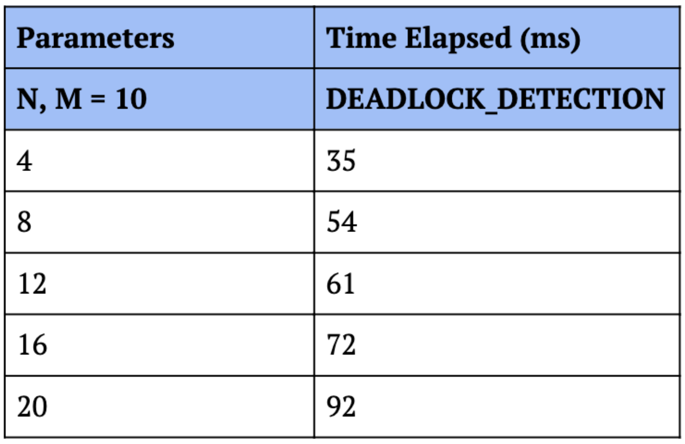


Figure 2: Graph of comparing AVOIDANCE with NOTHING

Using the values in Figure 1, where N stands for the number of processes and M stands for the number of resources, we can clearly see how time elapsed increases in each case in the graph in Figure 2. We observe that by increasing the number of processes (and resources simultaneously), the overhead of using deadlock avoidance algorithm becomes quite significant. Although the algorithm assures that there will be no deadlock occurrence, it consumes a significant amount of time in the overall procedure, even in the cases where there will be no deadlock (like in the program in `experiment.c`).

DEADLOCK_DETECTION

In our experiments, we measured the time it takes to perform a single `ralloc_detection`. Results are shown in Figure 3. Values are calculated using a constant amount of resources ($M = 10$).



Parameters	Time Elapsed (ms)
N, M = 10	DEADLOCK_DETECTION
4	35
8	54
12	61
16	72
20	92

Figure 3: Time Elapsed for a single `ralloc_detection`

Usage of this function depends on the programmer. It is expected that detection for deadlocks will be performed periodically, based on program preferences, so the programmer should consider the overhead and choose accordingly. We can see that for a single call to the function, `ralloc_detection` takes between 30 and 100 ms to complete, which is not significant in case the function will no be called significantly. However, if we call `ralloc_detection` at least 100 times, the overhead of this check may reach up to 10000 ms (based on the number of threads and resources), which becomes significant.