21.04.2019

CS342 OPERATING SYSTEMS
PROJECT 3

Elif Gülşah Kaşdoğan – 21601183

Sıla İnci – 21602415

SECTION 2

Our implementation of deadlock avoidance method uses the safety algorithm. We pass the current system to avoid_helper function and it checks whether the system will stay safe if we grant the current request. The helper function loops through the *finish[i]* array which indicates each process' completion. If all can be successfully completed, the function returns 1 (safe) else 0 (unsafe). In the request function, if "DEADLOCK_AVOIDANCE" is selected as the handling method, we call this helper function and if the demand is safe to be granted, we update the *available* and the process' *allocation*. If the state is not safe we call the *wait* operation on that demand. When another process makes a release action, to give a chance to all the waiting processes by calling *broadcast.*

If the "DEADLOCK_DETECTION" method is selected, we do not check whether upon granting the request we end up in a deadlock or not. We grant the request if there are available resources and report the number of deadlocks in that state if the ralloc_detection function is called by the user.

In our app.c, in order to test and evaluate the cost of each method (meaning time duration) we have given a system that does not create any deadlock. Thus we are able to measure the time without the program stopping because it has endured some deadlocks.

We tested our library with different numbers of processes. Each process makes multiple request and release calls. Experiment results are recorded to Table1.

In order to measure the cost of detection operation, we called the detection function after each release and measured the time. Experiment results are recorded to Table2.

Table1

N	Avoidance time(ms)	Detection time(ms)	Nothing time(ms)
20	24009.080000	18007.815000	18005.699000
15	20008.496000	15006.680000	15009.929000
10	15007.469000	15005.985000	15005.738000
5	15005.652000	15005.563000	15006.450000

Table2

N	Without detection	With detection
20	16008.137000	20006.977000
18	16007.779000	18009.812000
15	15005.909000	15007.672000
10	15006.724000	15006.596000

When we examine the Table1 we can see that avoidance is more costly than the others. Detection and Do nothing works in the same way for request and release. Since we execute safety algorithm each time a process makes a request, avoidance costs more than Detection and Do nothing.

Also, the number of processes (number of threads in our case) affect the runtime of each algorithm. As process count decreases all three methods give similar elapsed time values. We can say that for a small set of processes, performance will not be affected by the method we use but a number of processes increase avoidance may become a overhead.

In order to understand the cost of deadlock detection, we performed two runs. We performed *request* and *release* calls consecutively in one of them and called detection after request in the other one. Again we can see that the number of processes is an important factor for distinguishing the difference between two runs and see the cost of detection. We can see that for a large number of processes, detection results in a significant cost.

As a conclusion, deadlock avoidance can be useful if we have a set of processes which create deadlocks frequently. Deadlock detection can be used if we have a system which creates deadlocks and if we want to know which processes are causing the deadlock. But if we have a system which deadlocks are occurring rarely, avoidance may become an overhead. Furthermore, the detection algorithm is costly if we have a large set of processes.