BURAK KORKMAZ
21601296

CS 342 OPERATING SYSTEMS PROJECT 3

At the beginning of this report I want to mention that, I did this project alone. For the first part of the project I implemented a resource allocation library (libralloc.a) which simulates like a kernel in terms of resource allocation and deadlock handling. It has all the functions that mentioned in the project description document. I use a mutex and a condition to not to face up with a race condition, so functions can be called by multiple threads.

Later on I implemented a Multi-threaded application which is called app.c and it creates 5 threads simulating concurrently running processes which are requested from the library (libralloc.c). In my implementation each 5 threads has 10.000, 100.000, and 1.000.000 requests respectively. The reasons behind these request numbers is to show the time difference among deadlock handling types (nothing, avoidance, detection). For the table below Handling Type 1 is Deadlock Nothing, Handling Type 2 is Deadlock Detection, and Handling Type 3 is Deadlock Avoidance.

threads created = 5 with Handling Type 1 in time = 0.075500
threads created = 5 with Handling Type 2 in time = 0.042508
threads created = 5 with Handling Type 3 in time = 0.370409

threads created = 5 with Handling Type 1 in time = 0.784041
threads created = 5 with Handling Type 2 in time = 0.414232
threads created = 5 with Handling Type 3 in time = 3.417382

threads created = 5 with Handling Type 1 in time = 6.549100
threads created = 5 with Handling Type 2 in time = 3.584708
threads created = 5 with Handling Type 3 in time = 33.865135

In my implementation, I thought the Deadlock Detection handling type as an option for the user to check if there is a deadlock or not. It has a time complexity O(N^2*M^2) where N is number of processes and M is number of resource type. However it do not perform any context switch which is highly costly. That is the reason why its total time is the lowest for all three experiments.

Time Complexity

Deadlock Nothing (Type 1) = O (M) + context switch

Deadlock Detection (Type 2) = O (N^2 * M) (no context switch)

Deadlock Avoidance (Type 3) = O (N^2 * M) + context switch

As the similarity between the time complexity table that I constructed and experiments. Deadlock Nothing handling type is the smallest run time in a non deadlock case. However in a deadlock situation Deadlock Avoidance handling type is far more better than Deadlock Nothing. Because it wont wait for resolving deadlock.

In my opinion an optimal solution would be using Deadlock Detection than Implementing Resources. Because it wont perform a context switch then it performs a request.

On the following pages I put the examples from the book to test all three handling types.

Example that I use for the correctness of Deadlock Avoidance on page

7.5.3.3 An Illustrative Example

To illustrate the use of the banker's algorithm, consider a system with five processes P_0 through P_4 and three resource types A, B, and C. Resource type A has ten instances, resource type B has five instances, and resource type C has seven instances. Suppose that, at time T_0, the following snapshot of the system has been taken:

	Allocation	***Max***	***Available***
	$A\ B\ C$	$A\ B\ C$	$A\ B\ C$
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

The content of the matrix ***Need*** is defined to be ***Max*** − ***Allocation*** and is as follows:

	Need
	$A\ B\ C$
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

We claim that the system is currently in a safe state. Indeed, the sequence $<P_1, P_3, P_4, P_2, P_0>$ satisfies the safety criteria. Suppose now that process P_1 requests one additional instance of resource type A and two instances of resource type C, so $Request_1$ = (1,0,2). To decide whether this request can be immediately granted, we first check that $Request_1 \leq$ ***Available***—that is, that (1,0,2) $\leq$ (3,3,2), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	Allocation	***Need***	***Available***
	$A\ B\ C$	$A\ B\ C$	$A\ B\ C$
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence $<P_1, P_3, P_4, P_0, P_2>$ satisfies the safety requirement. Hence, we can immediately grant the request of process P_1.

You should be able to see, however, that when the system is in this state, a request for (3,3,0) by P_4 cannot be granted, since the resources are not available. Furthermore, a request for (0,2,0) by P_0 cannot be granted, even though the resources are available, since the resulting state is unsafe.

We leave it as a programming exercise for students to implement the banker's algorithm.

Example that I used for the correctness of Deadlock Nothing and Dead Detection on page

instances. Suppose that, at time T_0, we have the following resource-allocation state:

	Allocation	***Request***	***Available***
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence $< P_0, P_2, P_3, P_1, P_4 >$ results in ***Finish***[i] == *true* for all i.

Suppose now that process P_2 makes one additional request for an instance of type *C*. The ***Request*** matrix is modified as follows:

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

We claim that the system is now deadlocked. Although we can reclaim the resources held by process P_0, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.