

CS-303 Assignment 2

=====

Submitter name: Preetesh Verma

Roll No.:2018eeb1171

Course:Operating System

=====

Problem Statement

As part of this assignment we were asked to implement a program that mimics how deadlocks occur and are detected in an OS. The program simply creates situations where it is very likely for the system to get into a deadlock and then the deadlock needs to be detected and some action takes place to remove it. The program has to be developed in C programming language on a linux OS.

The problem statement had the following main components:

1. Thread Process Simulator
2. Request Generator
3. Allocator
4. Deadlock Detection and Termination of Thread if necessary

Thus, the problem statement asked us to create a thread pool where each of these threads would randomly be asking for random resources with a certain amount of pauses between the requests. This would lead to deadlocks being occurring and then on another thread an infinite running algorithm to detect the Deadlock is functioning which after every 'd' seconds checks for potential deadlocks in the system.

If the deadlock is not found then the threads work normally till they have acquired all the resources that they need and after this they randomly generate a new request after releasing all their resources. If a deadlock is found then the algorithm needs to terminate a few threads based on different kind of metrics and release their resources and spawn new threads in their place and the thread start functioning again. It's an infinitely running program.

Approach to Solution

Overview

As part of the solution I have used the `<pthread.h>` library for multithreading.

First I created all the necessary vectors and matrices necessary to keep a track of the allocated resources and the required resources for each of the threads. Then after implementing the thread pool each of these threads would run parallelly and randomly request for locks (mutual exclusion) on these above created data structures and if acquired then would generate random requests of resources with a certain amount of pause between successive requests.

If the asked amount of resources are currently available then the system will allocate them to the threads or else simply ask the thread to ask for another request. If a thread is allocated all that it wanted then it simply releases its resources and then makes a new request for resources.

In parallel to these threads, there is another thread which is also running infinitely and simply looks for potential deadlocks in these threads after every d seconds. If deadlock is found then the threads are arranged according to current heuristics and threads are being terminated and the resources which they held are released which would remove the potential deadlock. I have implemented three type of heuristics:

[1]-thread with total most resources is released first [2]-thread with total least resources is released first [3]-thread with a most amount of resource is released first

From the results observed the first heuristic performs the best as the time between the deadlocks is most and the number of threads terminated is least.

The third one does the second best and it's quite logical as well since in a majority of cases the threads which hold the single largest value of a resource also generally contains the largest sum.

The second one is not very good performing as the deadlocks occur quite frequently.

Directory Structure

```
|-- README.md
|-- main.c
|-- test.c
|-- Readme.pdf
|-- images
|-- |-- figure.png..
```

Detailed Explanation of the solution and contents of each file

main.c

This is the main file which starts the program. The file takes the following arguments upon execution and if not provided the server would not start.

1. The number of threads in the thread pool.
2. Types of Resources
3. Maximum number of instances of each resource
4. Time Delay
5. Heuristic Approach

Upon receiving the above mentioned arguments the program starts by creating the resource allocation matrix and the requirement vectors for each of the threads. Each of threads created in the thread pool then parallelly start their execution trying to acquire locks on these vectors. If acquired the locks the threads generate resource requests by passing a random value less than the need for the thread as a request for each of the resource. Implementation of Allocation of resources to the thread which will only happen if there are enough resources available to be allocated to that process else no resources are allocated for that resource to that thread. It is checked if the asked amount is

available or not and is needed by that resource or not or if it has already acquired that much amount of resource through prior request.

The deadlock checker function is responsible for checking for any potential deadlock among the threads by seeing the matrix and the requirements of the threads. Then based on heuristics threads are terminated. Remove the resources held by these threads in `threads_in_deadlock` and then check if deadlock is removed

1. if removed then resume the normal processing of the threads.
2. continue releasing resources from the threads.

Implementing the heuristics as requested by the user The purpose of this node is to help in applying different types of heuristics to the threads in deciding which thread to terminate.

```
struct Node {  
    int thread; // Name of a dynamically loaded library  
    int resources; // Name of a function to call from the DLL  
};
```

Sort the structure in decreasing order of resources each thread in the deadlock holds the resources.
Sort the structure in increasing order of resources each thread in the deadlock holds the resources.
Sort the structure in decreasing order of the single largest amount of resource each thread in the deadlock holds. Results from the heuristics are:

1. From the results observed the first heuristic performs the best as the time between the deadlocks is most and the number of threads terminated is least.
2. The third one does the second best and its quite logical as well since in a majority of cases the threads which hold the single largest value of a resource also generally contain the largest sum.
3. The second one is not very good performing as the deadlocks occur quite frequently.

Note -- In order to close the execution you need to press Ctrl+C.

test.c

This is the file containing the unit tests written by me to test the functions. The tests check the working of the heuristic, multithreading functions. The test cases basically run tests on heuristic operations such as sorting. The random generation of requests for each type of resource. Proper allocation of items to a request if available. The use of pthread library to create threads.

Procedure to run the files

To run the solution only one file needs to run. `main.c` is the file containing the entire codebase.

Commands to compile and run the `main.c` and a standard argument list which could be changed by the user are provided below.

```
gcc main.c -pthread -o main  
./main
```

Then provide the arguments as needed by the function.

Enter the number of threads:

2

Enter resource count:

3

Enter total quantity of each resource:

5 5 5

Enter the time delay

3

Enter the heuristic approach you want to see There are three approaches as implemented by me

[1]-thread with total most resources is released first

[2]-thread with total least resources is released first

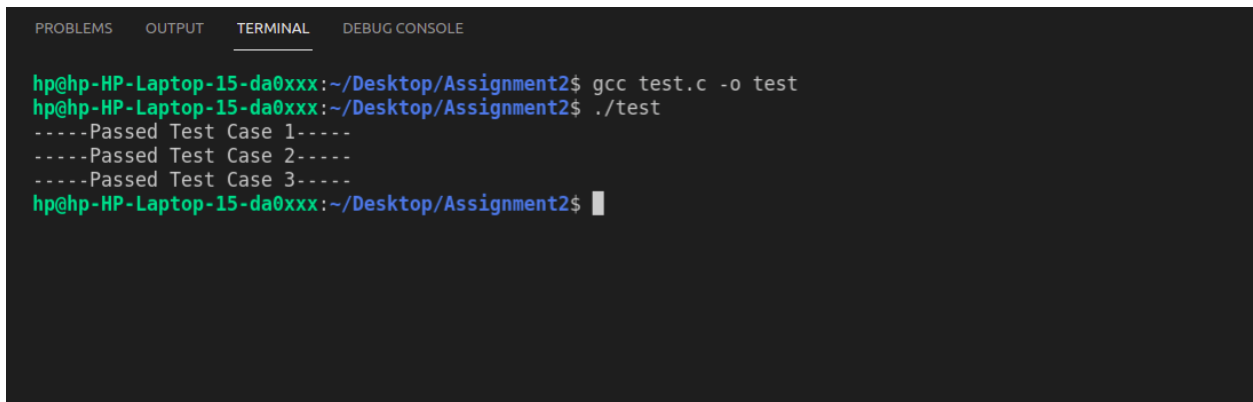
[3]-thread with a most amount of resource is released first

1

Commands to run the unit test file code

```
gcc test.c -o test
```

```
./test
```



```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

hp@hp-HP-Laptop-15-da0xxx:~/Desktop/Assignment2$ gcc test.c -o test
hp@hp-HP-Laptop-15-da0xxx:~/Desktop/Assignment2$ ./test
-----Passed Test Case 1-----
-----Passed Test Case 2-----
-----Passed Test Case 3-----
hp@hp-HP-Laptop-15-da0xxx:~/Desktop/Assignment2$
```

Snapshots of the results

Main program asking for arguments

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
hp@hp-HP-Laptop-15-da0xxx:~/Desktop/Assignment2$ gcc main.c -pthread -o main
hp@hp-HP-Laptop-15-da0xxx:~/Desktop/Assignment2$ ./main
Enter the number of threads:
2
Enter resource count:
3
Enter total quantity of each resource:
5 5 5
Enter the time delay
3
Enter the heuristic approach you want to seeThere are three approaches as implemented by me
[1]-thread with total most resources is released first
[2]-thread with total least resources is released first
[3]-thread with a most amount of resource is released first
1
```

Main program running

```
Detecting potential deadlock
Requesting resources for process 1
The ask array is: 0 4 0
Allocating resources if possible
The allocation vector is
2 4 0
Done Allocation
The required vector is
0 0 5
Requesting resources for process 0
The ask array is: 5 0 0
Allocating resources if possible
The allocation vector is
0 0 0
Done Allocation
The required vector is
5 0 0
Requesting resources for process 0
The ask array is: 0 0 0
Allocating resources if possible
The allocation vector is
0 0 0
Done Allocation
The required vector is
5 0 0
Requesting resources for process 1
The ask array is: 0 0 5
Allocating resources if possible
The allocation vector is
2 4 5
Done Allocation
The required vector is
0 0 0
Requesting resources for process 0
The ask array is: 0 0 0
Allocating resources if possible
```

Deadlock detected and thread terminated

```
Detecting potential deadlock
#####
Deadlock Found here
#####
Deadlock occurred at 0.037941 seconds
Threads in Deadlock are:2
Thread number 0
Thread number 1
The details of structure after sorting
1 4
0 1
The thread terminated is 1
Deadlock has been removed for now and normal processing can proceed
```

Threads received all the resources and now released their resources.

```
~~~~~
Thread 1 has finished. Releasing resources of the same
~~~~~
Detecting potential deadlock
The required vector is
5 3 4
The hold vector is
0 0 0
Requesting resources for process 1
The ask array is: 5 0 0
Allocating resources if possible
The allocation vector is
0 0 0
Done Allocation
The required vector is
5 3 4
~~~~~
Thread 0 has finished. Releasing resources of the same
~~~~~
The required vector is
1 5 5
The hold vector is
0 0 0
Requesting resources for process 0
The ask array is: 1 0 0
Allocating resources if possible
The allocation vector is
1 0 0
Done Allocation
The required vector is
0 5 5
```

A small demo video can be found in the images folder.

References

<https://www.geeksforgeeks.org/multithreading-c-2/>