Assignment 3: Resource Allocation Denial Simulation (Bankers Algorithm)

CSci 430: Introduction to Operating Systems

Overview

In this assignment we will be implementing the Resource Allocation Denial algorithm used as a deadlock avoidance scheme. The RAD is also more commonly known as the Banker's algorithm. The Banker's algorithm is described in chapter 6 of our textbook, and in particular we are implementing the figure 6.9c `safe()` pseudo-code that takes a described state and determines if the state is *safe* or *unsafe*.

Questions

- How do deadlock avoidance schemes work in practice to keep deadlocks from occurring in a computing system?
- How can we programatically implement an algorithm to test whether a resource request would be safe to grant or not?
- What information is needed about processes needs and requests in order to define their state and determine if new resource requests are safe to grant or not?

Objectives

- Explore deadlock avoidance mechanisms and in particular the Banker's algorithm for deadlock avoidance.
- Practice using C arrays and multi-dimensional arrays to represent the vectors and matrices used to describe and analyze system state for this algorithm.

Introduction

In this assignment we will be implementing the Resource Allocation Denial algorithm, also known as the Banker's algorithm. This is an example of a deadlock avoidance mechanism. The Banker's algorithm is described in chapter 6 of our textbook, and in particular we are implementing the pseudo-code shown in figure 6.9. In this assignment we break down the 6.9c `safe()` function into several smaller subfunctions. You will implement the small subfunctions, and then put them together to implement the full `isSafe()` method which can determine if a given system `State` is currently *safe* or *unsafe*.

Most of the work to define, describe and load a system state has been done for you. In this assignment there is a single class named `State`, described and implemented in the standard `State.hpp` header file and `State.cpp` source code implementation file. The `State` object mainly consists of 3 matrices (implemented as regular 2-dimensional arrays of integers), that hold what our textbook calls the Claim (C), Allocation (A) and the need (C-A) matrices. There are also 2 vectors, which are just 1-dimensional arrays of integers, describing the total Resource vector (R) and the Available resource vector (V). A function named `loadState()` has been implemented that reads in a system state from a given file, and fills in all of these matrices and vectors with the state information to use to implement the Banker's Algorithm. Additional functions have also been implemented that allow you to display the `State` as a string (useful to send to an output stream), and some helper functions for copying vectors and other tasks. For example, a the `state-01.sim` state file, which describes the same state as our textbook discussion of the Banker's Algorithm (Figures 6.7 and 6.8), we have the following state:

```
# This is state from figure 6.7a.  This is a safe state
# number of processes / number of resources
4 3

# total Resources vector R
```

```
9 3 6

# Claim matrix C
3 2 2
6 1 3
3 1 4
4 2 2

# Allocation matrix A
1 0 0
6 1 2
2 1 1
0 0 2
```

This state consists of 4 processes in the system, and 3 resources for the system. In our simulation, we start indexing processes and indexes by 0, so we have processes `P0, P1, P2, P3` in this system and resources `R0, R1, R2`. Notices that we only provide the total resources (R) and the claim matrix (C) and allocation matrix (A) in the input files. The need matrix is inferred from the claim and allocation matrices, e.g. need = C - A. Likewise we can infer the current number of available resources, because we can subtract the allocations for each resource from the total for that resource to determine the number of that resource that is still available.

Your task in this assignment will be to implement the `isSafe()` member function, which is the function that analyzes the current system state and make a determination of whether the state is currently *safe* or *unsafe*.

Overview and Setup

For this assignment you will be implementing missing member methods of the `State.[hpp|cpp]` class. As usual before starting the assignment tasks proper, you should make sure that you have completed the following setup steps:

1. Accept the assignment and copy the assignment repository on GitHub using the provided assignment invitation link for 'Assignment 03 Resource Allocation Denial Simulation' for our current class semester and section.
2. Clone the repository using the SSH URL to your host file system in VSCode. Open up this folder in a Development Container to access and use the build system and development tools.
3. Confirm that the project builds and runs, though no tests will be defined or run initially. If the project does not build on the first checkout, please inform the instructor. Confirm that you C++ Intellisense extension is working, and that your code is being formatted according to class style standards when files are saved.
4. You should create the issue for Task 1 and/or for all tasks for the assignment now before beginning the first task. On your GitHub account, go to issues, and create it/them from the issue templates for the assignment. Also make sure you are linking each issue you create with the `Feedback` pull request for the assignment.

Assignment Tasks

There is only the single `State` class given in this assignment. You will be simply implementing 4 functions that have not yet been implemented in the `State` class. In the given unit tests, the first test case simply tests the `loadState()` function to make sure the class is still correctly able to load a state from a file and represent it as a string.

So for this assignment, as usual, you should start by getting all of the unit tests to pass, and I strongly suggest you work on implementing the functions and passing the tests in the given order, starting with the second test case that test the implementation of the `needsAreMet()` member function. You will need to perform the following tasks.

Task 1: Implement `needsAreMet()` member method

You should start by defining the tests for task1 and declaring and implementing the `needsAreMet()` member function. This member function takes a process id/index as its first parameter, and an array of integers that represent the current number of resources available of each resource type. The total number of resources in the system can be found in the `numResources` member variable, so it is not passed in as a parameter to this function. You should look at the unit tests for this function to see how the array representing the currently available resources is declared and passed into this member function. This function returns a boolean result of true if the process needs are met by

the `currentAvailable` resources, and false if they are not. Make sure you put the function signature in the header file, and the implementation in the .cpp implementation file. Also, as a reminder, you are required to have function documentation for all functions, so when you add this function you need to create the function documentation for it (it has not been given to you this time).

Basically, in the `State` object there is a matrix called `need`, which holds the (C - A) information. Each row of this matrix is the information for a particular process. For the `needsAreMet()` function, you need to check each of the resource needs for the indicated process, and see if the are all <= to the indicated `currentAvailable` resource. If all of the needs are less than or equal to the currently available, then the process can have its needs met, and the function should return true. But if any need is greater than a current available resource, then the answer should be false from this function.

Once you are satisfied with your implementation, make sure you commit your changes for task 1 and push them to your Feedback pull request in your GitHub repository.

Task 2: Implement `findCandidateProcess()` member method

The next test case tests the `findCandidateProcess()` member function. You should define the task 2 tests and work on this after completing task 1.

This function is required to use the previous `needsAreMet()` function to perform its work. This function takes an array of boolean flags as its first parameter, and an array of `currentAvailable` resources, the same as used by `needsAreMet()`. The boolean array are a set of flags that indicate whether a particular process has completed its work yet or not. This function returns a process id / index as its result, e.g. the process id of the first candidate whose needs can be met by the currently available resources. This function needs to return the defined constant `NO_CANDIDATE` if no process can currently meet its needs with the available resources.

When determining if a state is safe, we start by assuming all processes are still running, so all process start off with completed as false. When a process can run, we mark its completed as true. For the `findCandidateProcess()` function, you need to search through all of the processes in the system. The first process you find that is not yet completed, but whose needs can be met by the `currentAvailable` resources should be returned as the candidate process. As mentioned, you need to use the `needsAreMet()` function to determine whether or not a particular process can have its needs met and is thus a candidate to be run to completion.

The pseudo-code for the `findCandidateProcess()` function thus would look something like this:

```
for each process:
   if process not yet completed and processes needs are met
      return the process id

otherwise if no process was found, return NO_CANDIDATE
```

Once you are satisfied with your implementation and your tests are passing, push your commit to the Feedback pull request and continue on to the next task.

Task 3: Implement `releaseAllocatedResources()` member method

The next test case tests the `releaseAllocatedResources()` member function. Define the task3 test cases and declare and implement the function for this task.

This function will be called after a process is selected to run to completion, to return its currently allocated resources back to the `currentAvailable` resources. This function takes a process id/index as its first parameter, and the same `currentAvailable` array as its second parameter that we have been using int the previous function.

The `allocation` array of the `State` class contains the current allocations of resources for each process. In this function you basically need to add the allocations of the indicated process to the `currentAvailable` vector of resources, which simulates the resources being released and returned back to the system to be used by other processes. This function is a void function, so it doesn't return anything explicitly, but of course it does change the `currentAvailable` vector to update it with the released resources.

The pseudo-code to implement the `releaseAllocatedResources()` member functions would look something like the following:

```
for each resource in the system
   add the number of allocated resources for the indicated process to the current available resources
```

Once you are satisfied with your implementation, push your commit to the Feedback pull request in your GitHub repository and then continue on to task 4.

Task 4: Implement `isSafe()` member method

The final test case is for the main `isSafe()` member method. This method will implement the actual Banker's algorithm to determine if the `State` is a *safe* or *unsafe*. This function doesn't take any input, it uses the current state defined by the matrices and vectors of the `State` object to perform its task. The function does return a boolean value of `true` if the state is safe, or `false` if the state is unsafe.

This member function will use the previous 3 functions you wrote, and maybe others from the `State` class to perform its tasks. The pseudo-code of the steps you need to perform are as follows:

```
1. Make a copy of the State resourceAvailable vector using the copyVector() function.
   This will be the currentAvailable vector, which is initially equal to the
   available resources, but if/when processes complete we release resources back to the
   system and keep track of the currently available resources in this vector.

2. Create a list of all processes called completed. Represent the list as an array of bool flags.
   The completed array keeps track of which processes have been selected and run to completion.
   Thus initially all processes should be marked as false, as all processes start out as
   not completed initially.

3. Search for a candidate process from the uncompleted processes.  This will be a loop
   that should keep being performed until no candidate process is found that can be selected
   to run to completion.  You should use the findCandidateProcess() function to search for
   the next potential candidate process to run.

    3.1 If we found a candidate process, release its allocated resources back to the available
        resources using releaseAllocatedResources().  Also mark the candidate process as
        completed.

    3.2 If no candidate was found, terminate the search loop

4. As a final test, after the search completes, if all processes were marked as
   complete, then the state is safe, so return true.  Otherwise if 1 or more
   processes did not complete, then the state is unsafe and you return false.
```

System Tests: Putting it all Together

As with the previous assignment, the `assg03-sim.cpp` creates a program that expects command line arguments, and it uses the `State` class you created to load and run a simulation from a simulation file. The command line resource allocation denial safe state program expects 1 argument, The name of the input simulation file that will be used to specify the state that is checked to see if it is safe or not.

```
$ ./sim
Usage: sim xxx
TBD
```

So for example, you can run the simulation from the command line for the first simulation state like this

```
TBD
```

If you get your output correct, you can see if your system tests pass correctly. The system tests work simply by doing a `diff` of the simulation output with the correct expected output for a simulation. You can run all of the system tests like this.

```
$ make system-tests
./run-system-tests
TBD
```

Assignment Submission

For this class, the submission process is to correctly create a pull request with changes committed and pushed to your assignment repository for grading and evaluation. For the assignments, you may not be able to complete all tasks and have all of the tests successfully finishing. This is ok. However, you should endeavor to have as many of the tasks completed before the deadline for the assignment as possible. Also, try and make sure that you only push commits that are building and able to run the tests. You may loose points for pushing a broken build, especially if the last build you submit is not properly compiling and running the tests.

In this problem, up to 50 points will be given for having at least 1 commit that compiles and runs the tests (and at least some attempt was made to work on the first task). Thereafter 10 to 15 points are awarded for completing each of the 4 tasks. However you should note that the autograder awards either all point for passing all tests, or no points if any test is failing for one of the tasks. Also note that even if you pass all tests, when the instructor evaluates your assignment, they may remove points if you don't follow the requirements for implementing the code (e.g. must reuse functions here as described, need to correctly declare parameters or member functions as `const` where needed, must have function documentation correct). You may also loose points for style issues. The instructor may give back comments in pull requests and/or create new issues for you if you have issues such as these, so it is good to have work committed early before the due date, so that the instructor may give feedback requesting you to fix issues with your current submission.

Requirements and Grading Rubrics

Program Execution, Output and Functional Requirements

1. Your program must compile, run and produce some sort of output to be graded. 0 if not satisfied.
2. 40 points for keeping code that compiles and runs. A minimum of 50 points will be given if at least the first task is completed and passing tests.
3. 15 points are awarded for completing each subsequent tasks 1-4.

Program Style and Documentation

This section is supplemental for the first assignment. If you uses the VS Code editor as described for this class, part of the configuration is to automatically run the `clang-format` code style checker/formatter on your code files every time you save the file. You can run this tool manually from the command line as follows:

```
$ make format
clang-format -i include/*.hpp src/*.cpp
```

Class style guidelines have been defined for this class. The `uncrustify.cfg` file defines a particular code style, like indentation, where to place opening and closing braces, whitespace around operators, etc. By running the beautifier on your files it reformats your code to conform to the defined class style guidelines. The beautifier may not be able to fix all style issues, so I might give comments to you about style issues to fix after looking at your code. But you should pay attention to the formatting of the code style defined by this configuration file.

Another required element for class style is that code must be properly documented. Most importantly, all functions and class member functions must have function documentation proceeding the function. These have been given to you for the first assignment, but you may need to provide these for future assignment. For example, the code documentation block for the first function you write for this assignment looks like this:

```
/**
 * @brief initialize memory
 *
 * Initialize the contents of memory.  Allocate array large enough to
 * hold memory contents for the program.  Record base and bounds
 * address for memory address translation.  This memory function
```

```
 * dynamically allocates enough memory to hold the addresses for the
 * indicated begin and end memory ranges.
 *
 * @param memoryBaseAddress The int value for the base or beginning
 *   address of the simulated memory address space for this
 *   simulation.
 * @param memoryBoundsAddress The int value for the bounding address,
 *   e.g. the maximum or upper valid address of the simulated memory
 *   address space for this simulation.
 *
 * @exception Throws SimulatorException if
 *   address space is invalid.  Currently we support only 4 digit
 *   opcodes XYYY, where the 3 digit YYY specifies a reference
 *   address.  Thus we can only address memory from 000 - 999
 *   given the limits of the expected opcode format.
 */
```

This is an example of a `doxygen` formatted code documentation comment. The two `**` starting the block comment are required for `doxygen` to recognize this as a documentation comment. The `@brief`, `@param`, `@exception` etc. tags are used by `doxygen` to build reference documentation from your code. You can build the documentation using the `make docs` build target, though it does require you to have `doxygen` tools installed on your system to work.

```
$ make refdocs
Generating doxygen documentation...
doxygen config/Doxyfile 2>&1 | grep -A 1 warning | egrep -v "assg.*\.md" | grep -v "Found unknown command"
Doxygen version used: 1.9.1
```

The result of this is two new subdirectories in your current directory named `html` and `latex`. You can use a regular browser to browse the html based documentation in the `html` directory. You will need `latex` tools installed to build the `pdf` reference manual in the `latex` directory.

You can use the `make refdocs` to see if you are missing any required function documentation or tags in your documentation. For example, if you remove one of the `@param` tags from the above function documentation, and run the docs, you would see

```
$ make refdocs
doxygen config/Doxyfile 2>&1 | grep -A 1 warning | egrep -v "assg.*\.md" | grep -v "Found unknown command"

HypotheticalMachineSimulator.hpp:88: warning: The following parameter of
HypotheticalMachineSimulator::initializeMemory(int memoryBaseAddress,
    int memoryBoundsAddress) is not documented:
  parameter 'memoryBoundsAddress'
```

The documentation generator expects that there is a description, and that all input parameters and return values are documented for all functions, among other things. You can run the documentation generation to see if you are missing any required documentation in you project files.