Assignment 3: Resource Allocation Denial Simulation (Bankers Algorithm)

CSci 430: Introduction to Operating Systems

Fall 2020

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 determins 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.

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*.

Unit Test 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.

1. You should start by implementing the `needsAreMet()` member function. This member function takes a process id/index as its first parameter, and an array of integers represent the current number of resources available of each resource type. This function returns a boolean result of true if the process needs are met by the `currentAvailable` resources, and false if they are not. 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.

2. The next test case tests the `findCandidateProcess()` member function. This function should 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. 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 should 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.

3. The next test case tests the `releaseAllocatedResources()` member function. This funciton will be called after a process is selected to run to completion, to return its currently allocated resources back to the `currentAvailable` resources. This funciton takes a process id/index as its first parameter, and the same `currentAvailable` array as its second parameter. The `allocation` array of the `State` class contains the current allocations of resources for each process. In this funciton 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.

4. 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

Once all of the unit tests are passing, you can begin working on the system tests.

As with the previous assignment, the `assg03-sim.cpp` creates program that expected command line arguments, and it uses

```
$ ./sim
Usage: sim state.sim
Run Resource Allocation Denial (Banker's Algorithm) on simulation
state file.  Return safe if the state is safe, or unsafe if not.


state.sim    Filename describing system state to load and
             test if it is safe or unsafe.
```

For this assignment you should not have to perform any additional work for the system tests. The system tests simply load a `State` from a file, and call the `isSafe()` method to determine if the state in the file is safe or not. If your `isSafe()` method is working correctly then the system tests should be passing.

Assignment Submission

In order to document your work and have a definitive version you would like to grade, a MyLeoOnline submission folder has been created named Assignment-03 for this assignment. There is a target in your `Makefile` for these assignments named `submit`. When your code is at a point that you think it is ready to submit, run the submit target:

```
$ make submit
tar cvfz assg03.tar.gz State.hpp State.cpp
State.hpp
State.cpp
```

The result of this target is a tared and gziped (compressed) archive, named `assg03.tar.gz` for this assignment. You should upload this file archive to the submission folder to complete this assignment. I will probably be also directly logging into your development server, to check out your work. But the submission of the files serves as documentation of your work, and as a checkpoint in case you keep making changes that might break something from when you had it working initially.

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. 20 pts each (80 pts) for completing each of the 4 listed steps in this assignment to write the functions needed to implement the Banker's Algorithm.

3. 10 pts if all given unit tests are passed by your code.
4. 10 pts if all system tests pass and your hypothetical machine produces correct output for the given system tests.

Program Style and Documentation

This section is supplemental for the first assignment. If you use the VS Code editor as described for this class, part of the configuration is to automatically run the `uncrustify` code beautifier on your code files every time you save the file. You can run this tool manually from the command line as follows:

```
$ make beautify
uncrustify -c ../../config/.uncrustify.cfg --replace --no-backup *.hpp *.cpp
Parsing: HypotheticalMachineSimulator.hpp as language CPP
Parsing: HypotheticalMachineSimulator.cpp as language CPP
Parsing: assg01-sim.cpp as language CPP
Parsing: assg01-tests.cpp as language 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 larget 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 docs
doxygen ../../config/Doxyfile 2>&1
  | grep warning
  | grep -v "\file statement"
  | grep -v "\pagebreak"
```

```
  | sort -t: -k2 -n
  | sed -e "s|/home/dash/repos/csci430-os-sims/assg/assg01/||g"
```

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 docs` 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 docs
doxygen ../../config/Doxyfile 2>&1
  | grep warning
  | grep -v "\file statement"
  | grep -v "\pagebreak"
  | sort -t: -k2 -n
  | sed -e "s|/home/dash/repos/csci430-os-sims/assg/assg01/||g"

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.