CSCI 315 Lab 7 Exercise

March 24, 2010

Objectives: In this lab you will work with a memory management simulation. The goal is to investigate the behavior and performance of various contiguous allocation algorithms and their effects on external fragmentation.

References:

- **Standard Deviation:** Bring the formula for standard deviation to lab with you (you should have seen it in your earlier course work, particularly the statistics course). You will have to write a method to compute the standard deviation of the values in an array.
- **Text:** Bring your textbook to lab. You may have to reference descriptions of various contiguous memory allocation schemes.

Prelab: There is no prelab this week.

Problems:

- **A Simple Memory Manager**

  Copy the file ~`cs315/Labs10/Lab07/Driver.java` and the directory ~`cs315/Labs10/Lab07/MemManage` and its contents to your workspace. **NOTE:** If you use Eclipse you may need to work out a way to import the Java code into Eclipse. The `Driver.java` file imports the contents of the `MemManage` directory.

  Look over the file `Driver.java` and the files `Memory.java` and `SimpleMM.java` for the memory manager. Notice that:

  - The program `Driver.java` contains code for processing command line arguments.
  - The simulation is driven by the random numbers generated in various parts of the `Driver.java` code. You can change the character of the simulation by changing the values to which the random numbers are compared .

    Note also that the random number generator `randGen` is constructed using an argument that fixes the seed. This insures that each run will use the same random values. This is useful in development and debugging, and also in grading. You can make the program behave differently on different runs

by deleting the argument. **However, complete all your test runs using the seed that is given.**

- Note that there are two different ways in which memory request amounts are generated. One, labeled "Random requests", is used in the code as provided to you. The other, "Bimodal requests", is commented out. You will use both in the course of the lab.

Do the following: Modify `Driver.java` so that when it completes it will print **the average size of the memory allocation requests** and **the number of allocation requests (note that this value is computed for you)**. Also, **comment out any other print statements** so that only the average size and number of allocation requests is printed. (You might actually want to wait to comment out the other print statements until you get your simulations working.)

- **Collecting Statistics**

  Examine the file `MemManage/SimpleMM.java`. This memory manager works by scanning through the blocks in memory (both allocated and unallocated) whenever it tries to allocate memory for a request. This is not the most efficient or sophisticated way to do things, but it works for our purposes, and our examples are small enough that inefficiency is not a problem.

  In the following activities you will be testing the memory manager. As indicated earlier, `Driver` has been set up to take optional parameters off the command line. The first parameter specifies a memory size and the second specifies the number of operations (allocation or deallocation) to attempt. You can play around with these parameters, but **you must use the following values for your hand-in runs:**

  ```
  java Driver 10000 800
  ```

  Try running the program both with and without command line arguments. You don't have to hand in anything from these runs.

  Now carry out the following tasks:

  - **Modify the implementation:** Make the following changes to `Driver.java` and `SimpleMM.java`:
    - Examine the method `allocate()` in `SimpleMM.java`. Note that allocation requests can fail for two reasons: First, the total amount of memory available (`AmtAvail`) may be insufficient. Second, there may be enough memory available, but it may be fragmented into blocks that are each too small to handle the request. Add three new, appropriately-named data members to accumulate the following data:
      - the number of requests that fail because total memory available is insufficient;
      - the number of requests that fail because of external fragmentation, even though sufficient memory is available; and

· the sum of the sizes of the requests that fail for the second reason.

Provide accessor methods For these values. Add code to **allocate()** so that these measures are accumulated correctly. Then add code to `Driver.java` to print the following information derived from those data members at the end of a run:

```
XXX allocation requests failed due to insufficient memory.

YYY allocation requests failed due to external fragmentation;
    the average size of these requests was ZZZ bytes.
```

After you get this scheme working, run the simulation with the specified command line parameters and have the instructor verify that you have the correct output values.

* Add a new public method `measureWaste()` to `SimpleMM.java`. This method should scans the memory and determine (and **print**) the average size of the **available** blocks and the standard deviation of these block sizes. This provides a measure of the memory fragmentation at the time this method is called. [**Hint:** Introduce a new data member in `SimpleMM` that keeps track of the number of available blocks. You will need to adjust this count in allocate, where available blocks are allocated and sometimes split, and in deallocate, where allocated blocks become available again. This will provide the information you need to create a local array of the correct size in `measureWaste()`. As you scan the memory, store the size of each block in a position of this array. When the scan is complete you can calculate the mean and standard deviation using the values in the array.]
* Modify `Driver.java` so that after every *twentieth allocation attempt* it calls `measureWaste()`.
* Modify the `allocate()` method in the `SimpleMM` class to support *best fit* and *worst fit* allocation. The current version of this method implements *first fit* allocation. You should be able to comment out two of the allocation methods and run the third. You can make the changes either by including the appropriate code within the current `allocate()` method, or by making three different copies of `allocate()`. **Be sure to include comments labeling each of your allocation schemes.**

– **Run the simulations:** Having made (and tested) these additions, run the memory manager simulation (`Driver.java`) using the **random requests** scheme for generating requests. Run the simulation once for each of the three allocation algorithms in turn. **Keep the seed value for the random number generator unchanged. Use the command line arguments 10000 and 800 as discussed above.** Make sure you are printing only the reports at the end of the run (information about successful and unsuccessful allocations), and the outputs of the calls to `measureWaste()`. **Be sure to cut and paste the results from your runs in your handin file, and be**

sure to label the results of each run with the allocation method (FF, BF, WF) and the request generation method (random requests).

Now comment out the code for random requests and uncomment the **bimodal requests** code. **Repeat the procedure above, labeling your results appropriately.**

– **Analyze your results:** When you have completed your experimental runs, analyze your results and write them up for your handin. There are lots of interesting issues to consider, and we strongly encourage you to consider your results and comment on what you see. However, do consider the following issues:

  * Which algorithm, according to your simulations, seems to do the best job of minimizing fragmentation (i.e., being able to allocate the most requests)? Is the difference large enough to claim that one or two of the algorithms do substantially better than the other(s)? Remember that you've run one trial with a fixed seed. Do you have enough information to draw a good conclusion?
  * What do your results suggest about worst fit?
  * How do the fragmentation statistics change over the course of your runs? Does this imply anything interesting?
  * Comment on the failures due to insufficient total memory. When and why do they occur? Do they indicate a problem with the allocation method? Explain why or why not.

Combine the following output and code into a single file, print it using a2ps, and hand it in:

- the final version of `SimpleMM.java`, including the three different allocation schemes and the `measureWaste()` method;
- the final version of `Driver.java`;
- the outputs from your runs (include only outputs from calls to `measureWaste()` and the final reports printed by the driver) and your analysis;