CSCI 315 Lab 8 Exercise

April 1, 2010

Objectives: In this lab you will do further work with memory management simulations to explore issues related to *internal fragmentation*. Recall that internal fragmentation is the amount of space within an allocated block of memory that is not actually in use. The goal for this week's lab is to implement measurement and reporting of the amount of internal fragmentation for both a contiguous allocation scheme and for a paged memory allocation scheme. A second goal is to gain practice with instrumentation of code.

Note that you will be using simulations based on data that probably isn't a good approximation of any system's behavior. The point of this exercise is not to imitate a specific system, but to look at the tradeoffs involved in dealing with internal fragmentation by changing system parameters.

Problems:

- **Internal Fragmentation for Contiguous Allocation:**

  Copy the file ~`cs315/Labs10/Lab08/Contig/ContigDriver.java` and the directory
  ~`cs315/Labs10/Lab08/Contig/MemManage` and its contents to your workspace. The warning to Eclipse users from last week applies again to these files.

  Look over `SimpleMM.java`. The file has been modified to use only a first-fit allocation scheme, and to enforce a minimum block size for allocations:

  - If an allocation request is smaller than the minimum block size, the request is handled as if it was for the minimum block size. The original request is retained so that you will be able to compute the resulting amount of internal fragmentation.
  - When an allocation is made, a block is not split if that would result in a block smaller than the minimum block size.

  A new constructor has been provided that allows the minimum block size to be passed in as a parameter.

  Also take a look at `ContigDriver.java`. Note that it now can take a third command line argument that specifies a minimum block size for the simulation. This will allow you to run experiments without having to edit and recompile between runs.

  You have two coding tasks to complete before you run some simulations:

  - Implement the measurement of the total amount of internal fragmentation within allocated blocks.
  - Implement a method that prints out both the amount of internal fragmentation and the fraction of memory taken up by internal fragmentation **expressed as a percentage**. Once you have it completed, moddify the driver program to call it every 40th iteration.

Measuring the amount of internal fragmentation is not difficult, but there are a number of issues to consider:

- You can compute the total internal fragmentation either by keeping a running count in a new data member, or by scanning through the array and reading internal fragmentation values out of allocated blocks. Either way, you will need to store a block's amount of internal fragmentation (or 0 if there is no internal fragmentation) within that block. This is obvious in the scanning scheme; in the running total scheme you will need this value to be able to update your total when you deallocate a block.

  Note that the second word in each block is not currently used (if the block starts at position `pos`, the second word is at `mem[pos+1]`). This word can be used to store the block's internal fragmentation.

- There are two reasons why internal fragmentation occurs: because a requested amount of memory is smaller than the minimum block size, or because a block is not split to avoid leaving a block smaller than the minimum. You will need to make sure you include both types in your measurement. To help you with the first type, `allocate()` has both the value of the original request (the parameter `reqAmt`) and the amount being allocated (`amt`). If a block is not split, `amt` is adjusted further.

Implement and test you internal fragmentation measurement code. Once you've completed that, run a sequence of experiments as follows:

- Run your experiments with a memory size of 10,000 and 1,000 repetitions.
- Do a run for each of the following minimum block sizes: 5, 10, 20, 25, 40, 50, 80, and 100. For a minimum block size of 40, for example, you would enter

  ```
  java ContigDriver 10000 1000 40
  ```

  on the command line.

Analyze your results, and write up your analysis. What do you observe happening as the minimum block size increases? What happens with failures due to no large-enough block? With failures due to not enough total memory? Why do these things happen? Is there a minimum block size you think works better than others? What is it and why?

- **Internal Fragmentation for Paged Allocation:**

  Copy the files ∼`cs315/Labs10/Lab08/Paged/PagedDriver.java` and the directory
  ∼`cs315/Labs10/Lab08/Paged/PagedMM.java` and its contents to your workspace. Note that there are no package and directory issues for these files.

  You again need to write a method to compute the total amount of internal fragmentation in allocated frames, and modify `PagedDriver.java` to print it every 40 iterations. The coding you need to do here is simpler because `PagedMM.java` already includes the code to store the amount of internal fragmentation of each frame in the second word of the frame. This is zero for every frame in an allocation except possibly the last one.

Your method only needs to scan through the frames, accumulating the internal fragmentation values for the **allocated** frames. Note that frames with no internal fragmentation have their second word set to zero by default, so it is safe to do the sum over every allocated frame.

Once you have these modifications implemented and tested, again run a series of experiments. Note that `PagedDriver`'s `main()` method takes a third command line argument that specifies the frame size. Follow the following guidelines:

- Run your experiments with a memory size of 10,000 and 1,000 repetitions.
- Do a run for each of the following frame sizes: 5, 10, 20, 25, 40, 50, 80, and 100. For a frame size of 40, for example, you would enter

  ```
  java PagedDriver 10000 1000 40
  ```

  on the command line.

Analyze your results, and write up your analysis. What do you observe happening as the page size increases? Which frame size worked best? Do you think choosing this relative frame size would work well on a real system? Think about how the choice of frame size can affect efficiency (e.g., time to allocate or deallocate) and complexity (what are the effects on the sizes of page tables, e.g.).

Hand in the output from your experimental runs and your discussions of these results by next Monday. Also hand in a copy of your modified `SimpleMM.java` file from the first problem and `PagedMM.java` file from the second problem.