COMPSCI 230 S1 C - Assignment 3

Due Date: Friday 3rd June, 2011 (9:00pm)

Assessment

- Due: Friday 3rd June, 2011 (9:00 pm)
  - 15% penalty if up to 24 hours late
  - 25% penalty if between 24 to 48 hours late
  - Assignment drop box closes automatically 9pm Sunday, no more submissions allowed.
- Worth: 6% of your final mark

Files

- A copy of this document, as well as relevant files for this assignment can be obtained from the CompSci 230 assignments page on the web:
  http://www.cs.auckland.ac.nz/courses/compsci230s1c/assignments/
- Submit your files using the web drop box (https://adb.ec.auckland.ac.nz/adb/).
- See at the end of this document for what files to submit.

Aims of the assignment

The purpose of this assignment is to give you some practice building responsive GUI applications. This will involve getting familiar with multi-threading. You will also extend this multi-threading to develop a parallel application to take advantage of multi-core processors.

Please note

- The work done on this assignment must be your own work. Think carefully about any problems you come across, and try to solve them yourself before you ask anyone for help. Under no circumstances should you take or pay for an electronic copy of someone else's work.
- Penalties for copying will be severe – to avoid being caught copying, don't do it.
- To ensure you are not identified as cheating you should follow these points:
  - Always do individual assignments by yourself.
  - Never give another person your code.
  - Never put your code in a public place (e.g., forum, your web site).
  - Never leave your computer unattended. You are responsible for the security of your account.
  - Ensure you always remove your USB flash drive from the computer before you log off.

Your name, UPI and other notes

- In this assignment your UPI must appear in the title of the frame (i.e. replace "upi123" with your actual UPI).
- All your files must include your name, UPI and a comment at the beginning of each file (except for those in the com.jhlabs package).
- All your program files must compile without requiring any editing.
- All your program files must include good layout structure.

1 Apply filters on images

The application given to you allows you to apply filters on images. Unfortunately, the application given has not been multi-threaded (click the red button). This means that the GUI thread (also known as Event Dispatch Thread, or EDT) will perform the filters on the images. During this time, the application is not responsive to other user actions. For example, try to click the coloured "Responsive?" button. Does this increment while a job is in progress? If not, it means your application is not responsive! The application has been given to you with the respective filters. It is rather primitive and untidy in many aspects.

Your job is to make it look prettier and more interactive. Since you are now multi-threading, you should think about what thread-safety mechanisms need to be implemented as discussed in lectures. Be sure not to remove any buttons, you should keep all the buttons (i.e. don't get rid of the red, orange or "Responsive?" buttons, etc). The reason being we want to demonstrate the difference between a sequential application, a concurrent application and a parallel application.

Remember that all GUI computations (e.g. JPanels, JProgressBars, etc) must be performed by the EDT. Remember that the actionPerformed() method is an event handler. As such, it is called by the EDT automatically. However, you want this method to be as short as possible so that the EDT can respond to other events. This is achieved by off-loading the computation to other thread(s). However, when you need to update the GUI again, this needs to be performed by the EDT. Therefore, you should always ensure this is the case (SwingUtilities.isEventDispatchThread()). Code can be scheduled to execute on the EDT by using SwingUtilities.invokeLater(), or SwingUtilities.invokeAndWait(), placing in the done() method of a SwingWorker instance, or using the notify (or notifyGUI) clause from ParaTask. We will discuss all these in lectures.

2 Tasks

2.1 *[Task 1]* Improve the GUI (5 marks)

Make the GUI of the top panel prettier. Maybe use a different layout, organise the buttons differently, increase the width of the input/output text field. Make the filter combo box span longer. It should be tidy, make use of the space. Make the counting button spread the whole row. You will get full marks for this if it is a tidy and better improvement of the current layout. Feel free to get as creative as you like. But don't waste too much time! Make sure you keep the "Responsive?" button, because the markers will use this to test that your application doesn't hang and remains responsive! Again, don't get rid of any buttons, just make things tidier.

2.2 *[Task 2]* Add a multi-image filter (10 marks)

Create a new filter. This filter will take a list of images and tiles them together to create a large image. Have a look at TileImageFilter.java. This almost achieves what you want, except it only tiles copies of the same image together. Note that this filter has inherit parallelism potential. For example, given a directory of large images, we can create thumbnails of them in parallel and then combine the smaller images into a single image. Your final image will be made up of 10x10 images (i.e. 100 thumbnail images). If too few images are provided, then use a round-robin schedule to fill the large image's boundary (so, if 5 images were provided as input, then the final image will be composed of 100 small thumbnails, 20 of each image). For efficiency, you shouldn't be saving the intermediate thumbnails as Files, since this will introduce unnecessary overhead and slow down your application.

So, given a directory of input images, you first create thumbnails of those images (i.e. 64x64 pixels) – but make sure you don't save those thumbnails (in the form of BufferedImages) to Files since this will create overhead due to disk access. You then combine the BufferedImages into a tile. You should be able to copy-paste a lot from TileImageFilter.java, but you won't be able to use the filter(BufferedImage src, BufferedImage dst) method since you want this filter to provide a list of BufferedImages as source.

Hint! Think of what you have learnt in OOP! Have a look at the method applyFilter(BufferedImageOp, ArrayList<File>, File, int). If you declare another method called applyFilter(TileMultiImageFilter, ArrayList<File>, File, int), then it gets called whenever you use a TileMultiImageFilter object (i.e. method overloading!). So, inside this method you would implement what this TileMultiImageFilter will do. The skeleton of the methods and calling it has been provided, you just need to create the respective class and complete this method.

2.3 *[Task 3]* Job progress panel (15 marks)

A "job" is when the user presses the "sequential", "concurrent" or "parallel" buttons. The job might be for a single image, or for a directory full of images. Regardless, the job is considered complete when all the images have had the respective filter applied. We wish to improve the application by displaying progress bars for each respective job. At the moment, we can only run one job at a time since the application is sequential (i.e. it becomes unresponsive while a job is running). Add a panel to the bottom of the frame, so that every time a new job is started, the job gets added. You are free to implement this however you like, so long as there is an entry for each job showing at least the following information:

- the job's ID number (it increments when a new job is started),
- a progress bar showing the status of each respective job. Calculate this as a percentage of the number of images that have had the respective filter applied already, and
- when the job completes, you should then add the total time it took for that job. This allows us to see all the jobs and all the times it took for them. The time should start from the moment the button was clicked, and ends as soon as the last respective filter was applied.

Word of warning! Even though you implement your progress bar, it actually won't work in the sequential version! This is because the EDT (the one that updates the progress bar) is actually too busy applying the filters on the images! Therefore, you won't start seeing your progress bar work until you implement the orange and green buttons! Also, validate() might be useful when you add new panels to the GUI and you want it to update (again, this is a GUI computation and must be performed by the EDT).

2.4 *[Task 4]* Implement the concurrent button (25 marks)

You will notice that so far only the red sequential button is working. Your job now is to implement the orange button so that the computations are off-loaded to a SINGLE thread. This keeps the GUI thread responsive to respond to other events. You should not parallelise the application just yet (i.e. make sure each job is on one thread only). You are allowed to create a new thread for each new job (i.e. you don't need to recycle threads), but don't make more than 1 thread for each job (this will be the purpose of the green button! ;-)

Remember that all GUI computations must be performed by the EDT. In particular, this means the progress bar and panel added earlier. Therefore, the progress bar must be updated by the EDT and not by any other thread. Your progress bar should be getting updated each time another image in the job is completed. You can off-load the task to a new Thread, or you can use a SwingWorker instance. The whole job is one task. Either way, your SwingWorker or Thread needs a reference to the panel so that it can update the progress bar (but this update must be done by the EDT).

2.5 *[Task 5]* Parallelise the filters (25 marks)

You have multi-threaded the application to improve interactivity and make it responsive (i.e. the orange button). However, you should now multi-thread the application to take advantage of multi-core processors (you will now implement the green button). This means even launching a single job should cause multiple threads to co-operate on the same job (assuming the job is for mroe than one image). Eg. resizing a directory of images will result in a team of threads sharing the images. If you "hack" this part by creating a thread for each image, you will gain marks for parallelism, but lose marks for efficiency. You should instead be making use of a threadpool with tasks (e.g. SwingWorker, Executor, ParaTask, etc).

This involves dividing the job into smaller parts so that multiple images may be worked on simultaneously. You should first parallelise the simple filters. You shouldn't have to do anything to the individual filters, just create multiple tasks that get executed by a team of threads. This step is rather easy to parallelise, because there are no dependencies between the tasks. Each image has the filter applied to it independently of other images. For now, ignore the parallelisation of the multi-image filter – you'll do this in the next step. Again, think OOP when parallelising (i.e. your parallelisation should be re-usable for all filters).

But be careful. This part includes making sure that the progress bars of each respective job is updated correctly too. It also requires your code is thread-safe whenever something is shared. In particular, this includes the list of images – you cannot share the ArrayList between different threads. You should also always ensure that the GUI components are only being accessed by the EDT. This is all part of multi-threading! And remember, jobs should get queued if the worker threads are already busy applying other filters in other jobs. How will you distribute the images amongst the threads? Will you use a static or dynamic distribution (dynamic distribution is probably best for this application)? You can't share a List of files amongst multiple threads, so you now must place the files in a thread-safe concurrent collection (look at java.util.concurrent), or use the Parallel Iterator.

This task (and the next) is worth a lot of marks because you must ensure your code is truly thread-safe, and not just hope that it works "99%" of the time. This means data accessed by multiple threads is thread-safe, and GUI components are only accessed by the EDT. But at the same time, you want to maximise the parallelism.

2.6 *[Task 6]* Parallelise the multi-image filter (15 marks)

You should now parallelise the multi-image filter. This filter is slightly different when it comes to parallelisation, because we have an extra task at the end of the smaller tasks that depends on them. In other words, you must wait for all the thumbnails to be ready (each an independent and parallelisable task) before you can combine the results into a tile. For example, if you have 3 images, then you end up with 4 tasks (3 tasks to generate the thumbnails, and 1 task to combine the thumbnails into a tile). Your thumbnails should be executed in parallel on a thread pool team (e.g. SwingWorker, Executor, ParaTask, or your own threadpool implementation). Note the dependencies – you cannot create the final image until the thumbnails of all the images are completed. As in the sequential version, make sure that you don't store the intermediate thumbnails as Files to disk, otherwise this will introduce unnecessary overhead.

Files to submit

- A3.txt
- all your *.java files (even if some files you didn't need to modify)
- all other files necessary to run your application. Assume the markers have nothing.

Please double check that you have included all the files required to run your program and the A3.txt in the zip file before you submit it. No marks will be awarded if your program does not compile and run.

What to include inside the A3.txt file

You must include a text file named A3.txt in your submission. There will be a 5 mark penalty for not doing so. This text file must contain the following information:

- Your name
- Your login name and ID number
- How much time did the assignment take overall?
- What areas of the assignment did you find easy?
- What areas of the assignment did you find difficult?
- Which topics of the course did the assignment most help you understand?
- Any other comments you would like to make.

Marking

The marks are outlined above for the respective tasks (total of 95 marks). In addition to this, there are 5 marks given for good programming style (i.e. indentation and meaningful variable names). There will be a 5 mark penalty for not including a completed A3.txt file.

The marking schedule given to the markers will be fairly simple. Primarily, you must ensure that the correct threading approach is used for each respective button, i.e:

- Red button: the application is unresponsive while a job executes,
- Orange button: the application is responsive while other (orange or green) jobs are executing, but only the job is assigned as a single task,
- Green button: the application is responsive while other (orange or green) jobs are executing, and even executing a single job (with multiple images) will result in all multi-cores being busy.

You should also ensure that the appropriate thread-safety is adhered to. This means access to all shared data by multiple threads must be thread-safe, and all GUI computations must remain on the EDT. In the case of the orange and green buttons, you should minimise the amount of work performed by the EDT (i.e. it should never be part of the team of threads working on a job).