CSCI4061 Spring 2017 | Assignment 2: Unix processes management

Due:

Due on Sunday, 02/26/2017 by 11:59 PM (Online, no hard copy)

Purpose:

The purpose of this assignment is to write a C program that creates UNIX processes and runs programs.

Scenario:

The scenario is similar to Assignment 1. You will be provided with a readable input directory which has a collection of image files in various formats (png, gif, bmp, etc.). You must make the images accessible through HTML pages.

There are subtle changes when compared to assignment 1. Please read the description below carefully.

In this assignment, you must write a program to create thumbnails for these images and copy all the thumbnails from the input directory provided to a single new directory. You must store the thumbnails in the new directory in a standard format (JPG). HTML pages must be generated for each thumbnail in the output directory. Additionally, in this assignment, each of these HTML pages must switch at a fixed time interval (time = 1 or 2 seconds).

Task 1: Programming

In this task you MUST write a C program that automates the image conversion process. Submissions with shell scripts will not be accepted for Task 1.

- Your program must convert all the images to the standard format by executing multiple processes in parallel or in some combination of parallel and serial operations. This will be determined by the user input variable *convert_count* indicating the number of processes to be created in parallel at a given time.
- The input directory will contain images with extensions png, gif and bmp. These are considered valid image types and must be converted to thumbnails of type jpg.
- The conversion must be done by executing the ***convert*** command. (*Hint*: Refer ImageMagick manual docs)
- The input directory will also contain some junk files which do not have any of the 3 image type extensions (gif, png, bmp). For these files create a single text file in the output directory called 'nonImage.txt'. List the name of each of these junk files in this text document. Then delete the junk file from the input directory.
- To perform the above tasks, your main process will spawn some child processes. You must retrieve the process id of all new processes created and if their process id
    - is divisible by 2 then these processes should convert png files only
    - is divisible by 3 then these processes should convert bmp files only
    - is divisible by both 2 and 3 then these processes should convert gif files only
    - The processes which are not divisible by 2 or 3 will handle the junk files
- At all times, as long as there are remaining input files to process, you must make an attempt to keep *convert_count* copies of *convert* running in parallel.

- Your program must not exit until all child processes have finished.
- In addition, you must run a simple experiment to determine the number of parallel operations that optimize the performance of the convert operation (see more details in Task 2: Experiment section).

Program Usage: Make sure your final executable is of the following format including the names for our testing and grading purposes

```
usage: parallel_convert convert_count output_dir input_dir
```

For example,

```
$ ./parallel_convert  2  ./myweb  ./input_dir
```

where the user inputs the following information as command line parameters

parallel_convert: must be the name of your final executable or your main program

convert_count: is the maximum number of child processes that can exist simultaneously. It can take values between 1 - 10.

e.g. if *convert_count* is 5, you should create 5 child processes to convert the input images in parallel. If the number of images to be converted is more than five, you need to create a new child processes after one of five running child processes ends and has be waited for. If the number of files are lesser than 5 then you create only the required number of processes.

output_dir: Each image is to be converted to JPG format and stored in this directory. Thumbnail images must also be generated from each image.

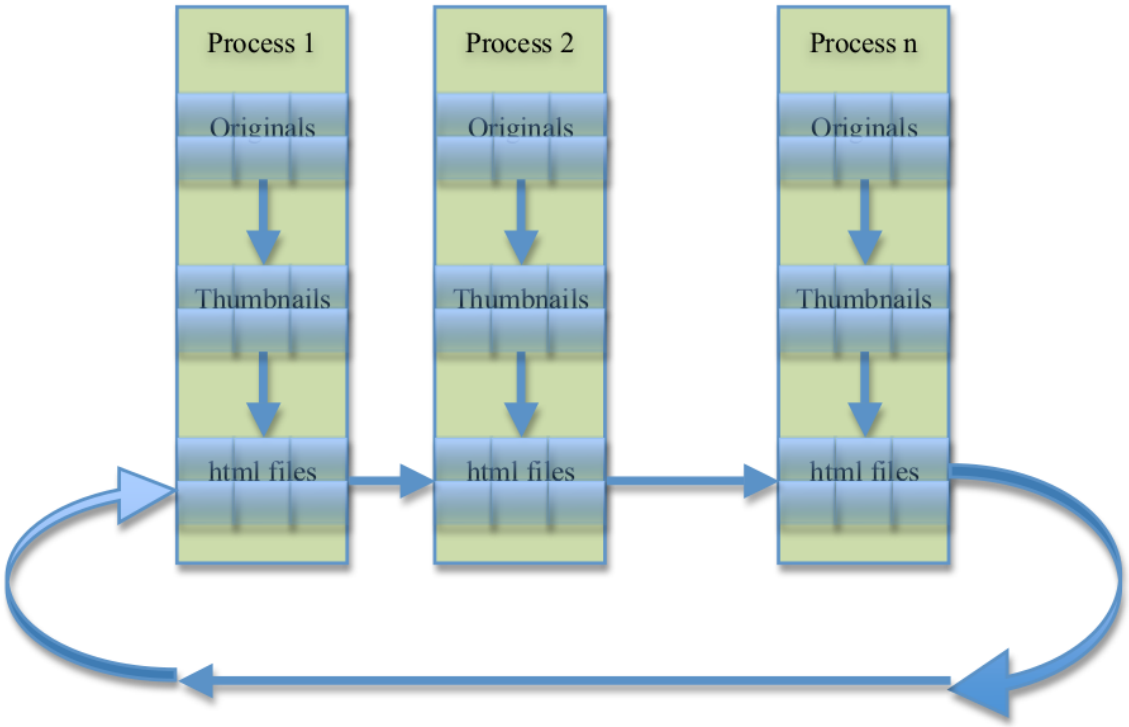
input_dir: list of files to convert is present in this directory. Your program must iterate through the input files in parallel. For each valid image, it must *execute* a copy of *convert* command. (*Hint*: Refer fork and exec APIS manual docs)

The command you execute will in parts look like this

```
usage:  convert   input_file    output_dir/input_file_<processId>.<extn>
```

So if you have an input file called millenium_falcon.png, the result would be an output file in the specified output directory, called millenium_falcon_123456.jpg with dimensions of 200*200 pixels. (See ImageMagick manual docs for information on how to set the dimensions)

```
e.g. convert   millenium_falcon.png   output_dir/millenium_falcon_123456.jpg
```



Generic flow of processes generating thumbnails and html files

HTML Generation

- From the diagram above, observe that the HTML files are created by the same process, which generates the corresponding thumbnail.
- The HTML file for the final thumbnail, should redirect to the first file.
- Hint: Use the meta redirect tag in HTML in order to link the different HTML files with each other. A refresh tag in HTML would be something similar the following

```
<meta http-equiv='refresh' content="2;URL=file:///$absolute_filepath_output_folder/$next_page.html">
```

Task 2: Experiment

Since the value of *convert_count* controls the number of parallel processes, a question naturally arises: **How many should you run to get the job done in the minimum amount of time?**

Your job is to answer that question and report the results. You will need to perform at least 10 runs, converting at least 40 images each time, with values of convert_count from 1 through 10. You can measure the total amount of time with the shell command *time* (Refer time manual docs).

Your report will be graded on what is written in the first TWO pages only so remember to limit your answers to the following

- Description of your experimental setup. Please include details of how you ran the tests, measured time, number of images used etc
- A graph of convert_count (x axis) versus time(y axis).

- A table of the timing data you obtained
- Your interpretation of the result. The description of the setup should include any factors you think might influence the results.

To prevent performance from being dominated by sending output to the screen (or to your remote login), you should redirect standard output to a file when you run your experiment. Ideally, you will write a shell script/C program that runs your experiment and reports the results for you. If you do write a shell script/C program, you are not required to turn it in, but you could briefly describe it in your report.

Test Data:

You may use images from assignment #1 for test data, or you may use your own images. A new set of images is also provided so as to maintain a set larger than 200 pixels in each dimension. You may use multiple copies of the same images (Rename the files to img1, img2, img3 i.e., different names), because you need at least 40 images for your experiment. Make sure to use the same set of images for each run of the experiment.

Logging

Each time a conversion completes, you must write to standard output and log file the name of the input file that was converted, and the process ID of the process that did the conversion. You may provide additional logging for usability. For e.g., `> Writing into log.file`

```
> image1.png converted to jpg of size 200x200 by process with id : 123456
```

```
>All files were successfully completed
```

Error Handling:

- Errors must be handled gracefully, with informative error messages on standard error.
- You cannot count on your user always giving you the right number of arguments. If the wrong number of arguments is given, then your program should print a *usage* message and exit with a status of 1. e.g. Every command-line argument in the usage statement should be a single word, with no spaces. If you want to write an argument as multiple words, join the words together using underscores. For e.g., a general usage message unrelated to this assignment is shown below. (Reference on last page)

```
e.g. usage:  prog_name  input_file,  output_file
```

- You must also check for errors on the system calls you use, and use the perror() function to report them. e.g. perror messages will print the word error followed by the error message generated by the system

```
Error: : No such file or directory
```

- You may add your own error reporting if you feel it was necessary. For example: You must have an error message printed if the value of *convert_count* lesser than 1 and greater than 10.

Valid Assumptions:

- You may assume the input folder will not have any other folders within. Only valid image and junk files.
- There will be different types of files in the input folder. There are three fixed file formats, namely png, bmp and gif that you have to consider for this assignment. They may be present in uppercase or lowercase. You must handle both cases.
- The converted files must be of type JPG. You may name it in uppercase or lowercase.
- You will be provided the set of images in fixed file size. You need not have to test it against other file sizes.
- Only files with extensions – gif, bmp or png are to be converted. Any other file in the input_dir, even jpg and jpeg are to be considered junk files.
- You can assume that input files exist and are readable, that the output directory is usable (if it exists), and you may ignore filename conflicts on the output side.
- You may assume none of the input files would have any spaces in the name and each file will have a unique filename
- You may assume length of file names will not exceed 64 characters
- You may assume total files won't exceed 100 in number
- You may assume a fixed file size of 200*200 for all thumbnails
- It is understood that a bmp extension in the filename means that the file is a valid image file. You do not need to check whether a file with extension bmp is indeed a bmp file.
- There are no soft / hard links in the directory tree. There are only files.
- The number of files for each type will be equal.
- It is possible to have ZERO files of one or more type. Your program does not need to treat this case any differently.
- You must create the output directory if it doesn't already exist with read and write permissions.
- You do NOT need to deal with subdirectories in this assignment. No need to create new directories.
- All input files would be in one input_dir and you put all files in output_dir.
- The output_dir must have the following contents within it
    - Thumbnails files.
    - HTML files of all generated thumbnails
    - Log File
    - The 'nonImage.txt' file

Platform:

You may work initially on the platform of your choice; Linux, Solaris, Irix, and Mac OSX should all work. Windows might work under the cygwin package, but only if you have the Posix compatibility package installed. However, you must test under CSELabs UNIX Machines Linux box (machine list is here), and we will test all submissions using Linux on one of the boxes when we grade your code.

Teamwork:

You may do this assignment individually or with one partner.

Deliverables:

See the course syllabus for general requirements and assignment page for submission guideline. The specific requirements of this assignment are as following:

You need to include the following files for this assignment:

1. README.txt should include
   a. Personal information for the group
   b. CSELab machine you tested your code on.
   c. Syntax and usage pointers for your program (details on how to run/use your program)
   d. Any special test cases or anomalies handled / not handled by your program.
2. Source code files. All *.h, *.c files required to execute your program. Your main program must be in parallel_convert.c and this is mandatory. You may choose to split your program to smaller modules/files
3. MakeFile for compilation
4. The experiment report. All reports must be in pdf form.
5. Your commentary as described in the syllabus. This should explain, briefly, what your code does, how it works, how you use it, and how to interpret its output. It should be no longer than TWO pages in length. This is a separate pdf file.
6. You do not need to submit any test files, images or generated html files. These should be created when the program is run.
7. Your code will be tested by another program. To make sure that your information is read correctly, the beginning of each parallel_convert.c file must adhere to the format that is given below. If you are not working in a group you should specify only one name and id.

```
/* Information
CSci 4061 Spring 2017 Assignment 2
Name1=FullName
Name2=FullName
StudentID1=ID for First name
StudentID2=ID for Second name
Commentary=description of the program, problems encountered while coding, etc.
*/
```

Program Usage requirements:

```
username@Directory/path:$ make
username@Directory/path:$ parallel_convert [convert_count] [output_dir] [input_dir]
```

Grading:

In this assignment, 70% is allocated to the code (of which, 20% on style and 50% on correctness), and 30% is allocated to the experiment report.

The general grading criteria are given in the course syllabus. For this exercise the "style" points will be based on readability. Use meaningful names if you create variables, and use consistent indentation to display any control structure in your script. Unless mentioned, each subsection involves creation of the HTML page of all expected thumbnails. Following are details of the grading criterion:

Command line argument processing

- Wrong # of arguments (5pts)

Converting files

- convert_count is 1 (5pts)
- convert_count is 0 (5pts)
- convert_count equals to the number of files (5pts)
- convert_count is smaller than the number of files (5pts)
- convert_count is bigger than the number of files (5pts)

Process information

- Designated process does the conversion prints the correct correspondence between process pid and file name. Parent process works (5pts).
- Prints child pid and filename AFTER child finishes (5pts)
- Waits for all children before exiting and prints wait message appropriately (5pts)

Usability

- Provide informative error messages on standard output. (5 pts)

Reporting

- A description of your experimental setup (5pts)
- The timing data graph you obtained (15pts)
- Your interpretation of the result (10pts)

Coding Style

- Informative description at beginning of the program (5pts)
- Informative comments within the program (5pts)
- Use meaningful names if you create variables (5pts)
- Use consistent indentation to display any control structure in your script. (5pts)

Note: Test data used while grading will not be provided.

Resources and Hints

The resource page contains several pointers related to C programming and programming style.

C library functions that could be useful for this project, other than what we've discussed in class, might include perror(), itoa(), exec() family of functions, fork(), vfork(), fprintf(), basename(), strlen(), strcmp() and strncat(). You can read more about these functions here http://pubs.opengroup.org/onlinepubs/9699919799/functions/contents.html

Usage Message: http://en.wikipedia.org/wiki/Usage_message