Lab 3 — Queue and Process Management

November 19, 2018

This and the remaining labs should be done in pairs – only **one** lab should be submitted.

1 Objectives

Following completion of this lab you should be able to:

- Create and use a Makefile to compile code.
- Use the GNU Debugger, gdb, to debug your code.
- Implement a queue data structure in C.
- Use the `fork` system call to spawn child processes.
- Use the `exec` family of system calls to overlay the process image of a child process with another executable program.
- Use the `gettimeofday` system call to generate timing statistics for the creation and execution of processes.
- Explain process creation, process termination and process management.

2 Tasks

2.1 Makefile

Checkout the `maketutorial` project in the Examples directory from your repository onto your local machine. Follow the tutorial. A solutions is posted

for your convenience.

2.2 GNU Debugger

Follow the instructions in the gdb tutorial to familiarize yourself with using gdb to debug your code. gdb is a convenient utility that you should use to diagnose and correct bugs in your code.

2.3 CharsStringsFiles (15 points)

Checkout the `CharsStringsFiles` project from your repository repository onto your local machine. Type your answers to the questions below in the file `ANSWERS.txt`.

The `CharsStringsFiles` project is inspired by many high-level string processing methods provided by Python's String class. These methods include `capitalize`, `lower`, `upper`, `swapcase`, `capwords`, `strip`, and `center`, to name a few. In order to provide similar functionality in C, we designed and implemented this project.

We have provided all of the test code in the file `csf-main.c`. We have also provided support for reading data from files and displaying data to the console.

The following questions all refer to the code in the `CharsStringsFiles` project. The code consists of three C source files and two header files as follows.

`file-functions.c` `file-functions.h`	Implements file reading functions. Three different versions of `printFileToConsole` are provided. The first reads a file and prints it to the console one character at a time. The second reads and prints a line at a time. The final version reads the entire file into an array, then prints the array.
`string-functions.c` `string-functions.h`	Implements a number of string functions such as `capitalize` and `center`.
`csf-main.c`	Implements the sample application of chars, strings and files. Sample input is provided in the **Data** folder and expected output is captured in `EXPECTED_OUTPUT.txt`.

1. Your instructor will complete with you a makefile to compile your code for this assignment.

2. (5 points) `csf-main` does not work as intended. In fact, in it's current state, `csf-main` generates an error when it is run. Use gdb to identify which error is received and where.

   (a) What error is received?

   (b) Where (what line number and which instruction) is the error received?

3. (5 points) What is the root cause of the error in Problem **??**?

   Hint: Use `gdb`.

4. (5 points) Make the necessary modifications to correct the root cause of the error you identified in Problems **??** and **??**. Be sure to commit your changes to your repo.

   Hint: You shouldn't need to understand the whole codebase to make the change - the solution only requires adding one line. But make sure you at least understand the function you're editing.

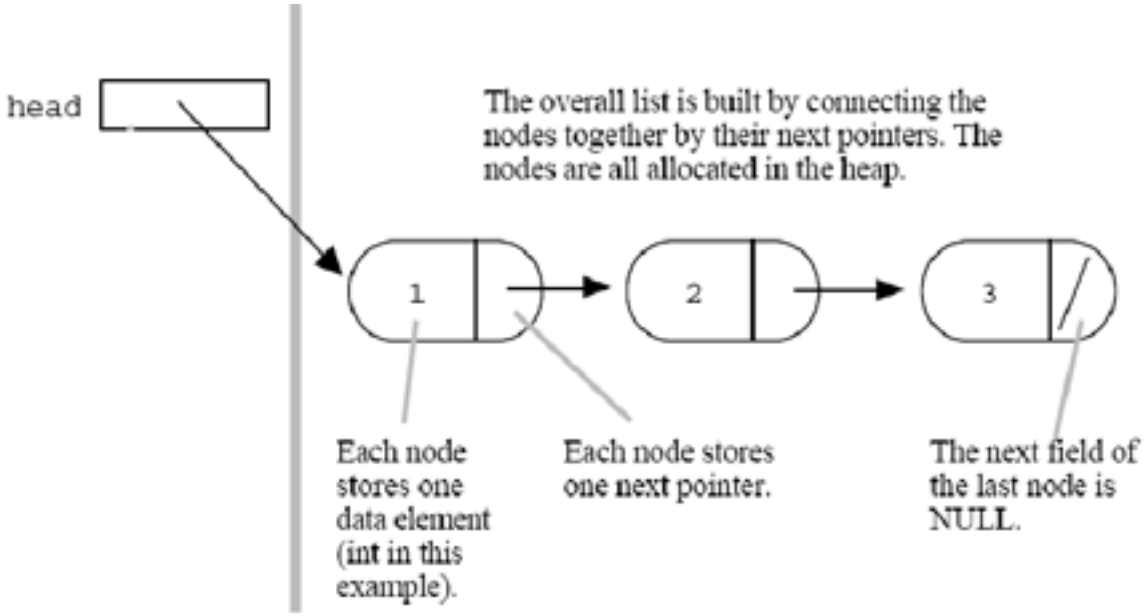
2.4 Queue (100 points)

In the exercise, you will use the notion of a linked-list to implement a queue data structure. Queues are heavily used in operating systems for various tasks. You will use pointers to create and maintain the queue. The functions to be implemented are described in the header file included in the project that you will checkout from your repository.

Linked-list: A linked list is a data structure that contains a list of nodes. A node has two parts:

1. Data
2. A pointer to next node

Typically, the last node in the list will not point to any other node; hence, its next node pointer will point to `NULL`. Using the next node pointer, it is possible to keep track of the entire linked-list and access any node in the list. A linked-list typically has a head pointer, as shown in the figure below. The head pointer points to the first node in the list. The figure below shows a singly-linked-list.



Using a singly-linked-list to implement a queue: A queue is a data structure in which a node is always added to the **tail** of the list and removed from the **head** of the list. It is a data structure with a *First-In First-Out (FIFO)* discipline. To model this behavior, implement a singly-linked-list with a **head** and a **tail** pointer. The **tail** pointer points to the last node in the list or `NULL` if the list is empty. The **head** pointer points to the first node in the list or `NULL` if the list is empty. Each node in the list will have a pointer to the next node in the list or `NULL` if there is no next node.

The structure for a queue instance, the structure for a node, function prototypes and their descriptions are available in `queue.h`.

1. Checkout the **Queue** project in the **lab3** directory from your repository onto your local machine.

2. In `queue.c`, implement the functions described in `queue.h`. You must use pointers as described in the header file and you must use a singly-linked-list with head and tail pointers. You are provided with `driver.c` to test your implementation. You may **not** modify `queue.h`.

3. Create a **Makefile** for the Queue project that will build your executable.

4. Commit your work, including the Makefile (but excluding your executables, `*.o` files, and other vestigial files), to your repository with an appropriate commit message.

Sample output for the printQueue function is provided below:

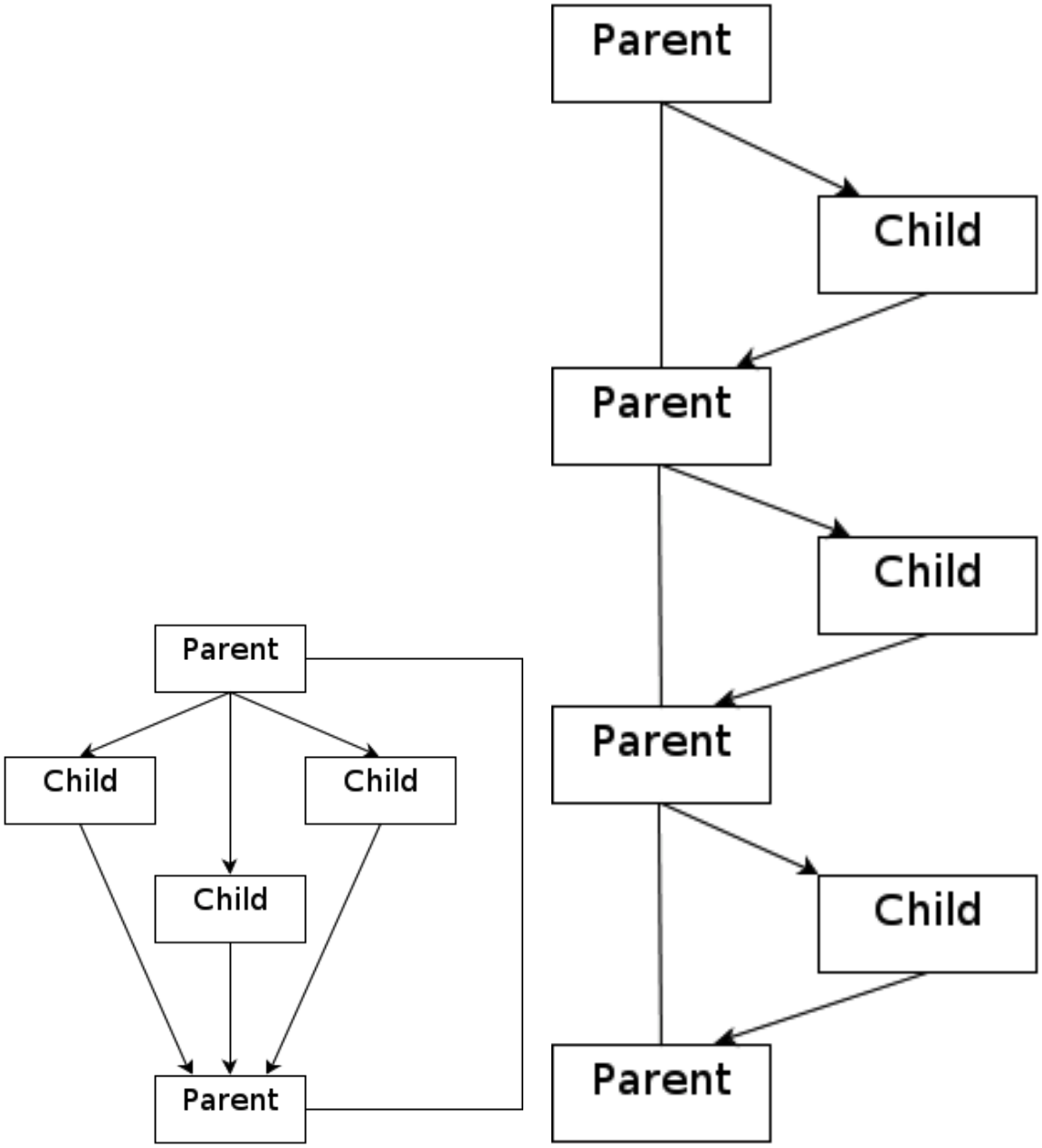
Process id	Arrival Time	Service Time	Remaining Time
1	0	10	10
2	1	11	11
3	2	12	12
4	3	13	13
5	4	14	14

2.5 createProcesses

Checkout the **createProcesses** project in the **lab3** directory from your repository onto your local machine. The files in this project should be used as sample files for this lab. The lecture slides on Process operations may also be helpful.

2.6 Processes (100 points)

There are two different ways to organize a parent-children process hierarchy. The first is a process chain, in which the parent waits for each child process to complete before creating a new child. The second is a process fan, which we will be using for in lab. In a process fan the parent creates all of the child processes at once allowing them to execute in parallel. The figures below show a process fan, on the left, and a process chain, on the right.



a) Process fan b) Process chain

Figure 1: Spawning child processes a): Process fan, b) Process chain

Checkout the **Processes** project in the **lab3** directory from your reposi-

tory onto your local machine.

1. Write a C program in the files `processes.c` and `processes.h` that creates child processes in a fan:

   (a) Accept three command line arguments: 1) n the number of child processes to create; 2) x a delay factor in microseconds for each child process; and 3) the name of a source file containing individual file names, one per line, to pass to the child processes.

   `Files.txt` included in the project is an example.

   (b) Use a loop to create a process fan of n child processes that will run in parallel.

   Recall: In a process fan, the children processes are all created by the parent process at once. You need to enforce this fact by making sure that no child process forks.

   (c) Each child process should be assigned a local id named **index**, which corresponds to the order in which it was created. The first child process will have an **index** of 1, second will have an **index** of 2, and so on.

   (d) Each process, including the parent process, will also have a start time and an end time.

   (e) The process **pid** (the id returned to the parent from **fork()**), process start time and process end time for each child process should be stored in an appropriate structure.

   (f) An array of such structures should be created to store process info for the n child processes. The entry for each child process will be indexed by the process's **index**.

   (g) Immediately before each child process is created, the parent should capture the current time and store it in the process info structure as the start time for that child process. The process **pid** should also be stored in the structure. The process start time should be captured using the **gettimeofday(2)** system call ("man -S 2 gettimeofday" describes its usage).

   (h) Each child process should start by sleeping ((n-**index**)*x) microseconds. Use a value of 100,000 microseconds for x initially. This will be varied later.

(i) In each child process, read the corresponding line in the file passed into the program (e.g. `Files.txt`). For example, the first process created should read the first line in the input file. If n is larger than the number of lines present in the file, the program should wrap back around to the first line in the input file, using the same file more than once.

(j) During execution and before terminating each child process should do the following:

   i. Sleep ((n-**index**)*x) microseconds.

   ii. Print its **index** to the console.

   iii. Overlay its process image with the "`myCopy`" program using one of the exec family of system calls. Remember that "`myCopy`" takes two command line arguments: 1) the name (or path) of the input file (read from the source file by the child process); and 2) the name of the corresponding output file. If the input file name is `filename.txt`, then the output file name should be `filename_index_out.txt`. For example, `FileOne.txt` will correspond with `FileOne_1_out.txt` if the child process with index 1 produces the output file.

(k) After a child terminates, the parent should print the child's **index** and its total time from creation to end of execution to the console in the following form:

```
Time for process <index> = <calculated time> microseconds.
```

Note that the parent must get the termination time of the terminated process and the appropriate entry from the array of process info structures to compute the total time for that process. The end time for the terminated process should also be stored in its process info structure.

(l) The parent process should measure its total execution time using **gettimeofday(2)**. After all of the children have finished execution, the parent should print to the console that the children have finished, that the current process is the parent process, and the measured total execution time before it exits. Ultimately, a parent's total execution time should be less-than or equal to the sum of the execution times for all its child processes. Can you think of why that might be?

2. Once your program works with the fan, experiment with different x values for the sleep times of each process. By looking at the program output, you should see when processes begin to call **myCopy** simultaneously and when they begin to reorder themselves again.

3. When you turn in your assignment it should include the following:

   (a) Makefile and source code both `processes.c` and `myCopy.c`. Make sure that your Makefile creates **both** executables.

   (b) A README file containing usage instructions.

   (c) Console printout results for at least three test cases with different values of n and x.

   (d) The input files you used for the **processes** and **myCopy** executables.

3 Linting your code before you make your final submission

We will use the Google C++ style. Use **cpplint.py** (this version has a couple of small local modifications to make it work better for C code) to make sure your code conforms to a reasonable style. You should download `cpplint.py` to your home directory (accesible via `cd ~`). You might need to make `cpplint.py` executable (e.g. by running "`chmod a+x cpplint.py`"). You can check `foo.c` by running "`~/cpplint.py hw01.c`" assuming `cppylint.py` is in your home directory. If `cpplint.py` is in a different directory (which is not recommended), you'll need to include the path to it. If it's in the current directory the path to it is "`./cppylint.py`." Note: you should place CPPLINT.cfg in your home directory (preferred option) OR the root of your CSSE332 directory tree to automatically load local configuration options. This config file will eliminate the copyright warning.

A simple way to download these files to your home directory is by entering the commands below at the terminal:

```
> cd ~
> wget http://classes.csse.rose-hulman.edu/csse332/resources/cpplint.py
> wget http://classes.csse.rose-hulman.edu/csse332/resources/CPPLINT.cfg
> chmod a+x cpplint.py
```

4 Turning It In

This lab is due no later than 11:59 PM on the due date.

You will receive credit for each project **only** if it compiles and runs on the course linux distribution.

1. Only **one** lab should be submitted by each **pair**. You **must** include both partner's names in a comment at the top of all files you submit. Commit a "`README`" file in the repository of the partner not submitting the code which lists the names of both partners.

2. Add any new files/directories you created to your repository.

3. Commit and push your solutions to each project to your repository with appropriate commit messages.

4. Use the git status command and/or point your web browser to your repository to verify that you submitted your solutions.

Note: It is good practice to commit often.

Once the labs are graded, you can do a *git pull* to pull comments and grader feedback.