CSCI 4061: Intro to Operating Systems
Project 2: Multi-Process Web Browser

Posted Mar 1 – Due Mar 22, 11:55 pm Groups of 3

1 Introduction

Unlike traditional web browsers which run as a multi-threaded process, modern web-browsers such as Internet Explorer and Google Chrome adopt the more robust and parallel multi-process architecture. In this project, we explore this new architecture using *pipes* as IPC mechanism. **Note:** We will provide all browser graphics, event handling (e.g., clicks and input retrieval), and new tab creation code.

2 Description

You will design a multi-process web-browser in this assignment given some initial code. The multi-process web-browser uses a separate process for each of its tabs to guard against errors in the web-page rendering tab. Your web-browser will consist of *one* main parent process called the ROUTER process, *one* CONTROLLER child process, and *zero or more* URL-RENDERING child process(es) corresponding to each opened tab. We briefly introduce the roles of these processes in this section, and will provide detailed explanation of their implementations in Section 4.

2.1 ROUTER process

The ROUTER process is the main parent process which runs when the web-browser program is started. The ROUTER process is responsible for *forking* the CONTROLLER process. Once done, the ROUTER keeps *polling* (i.e. continuously checking for) requests from its child processes (CONTROLLER and URL-RENDERING processes) through non-blocking read of pipes in a loop. **Note:** the ROUTER process is not involved in any of the browser rendering graphics and is only intended to function as a *router* between the CONTROLLER process and URL-RENDERING processes. The ROUTER is not displayed in a tab.

2.2 CONTROLLER process

The CONTROLLER process is a child of the ROUTER process. It renders the Controller Tab window (shown in Fig 1) to the user. The window has following regions:

1. *URL-region*: A text field where the destination URL (e.g., `http://www.google.com`) is entered.
2. *Tab-selector* region: The tab number in which the URL is to be rendered. It accepts an integer value less than the number of opened tabs.
3. *Create-new-tab* button: The control for creating a new URL-RENDERING tab.
4. *Close-tab* button: The control for closing the tab.

When a user wants to open a URL in a new tab, she does the following in order:

1. Clicks the *Create-new-tab* button of the Controller Tab. This opens a new URL-RENDERING window (shown in Fig 2).
2. Enters the tab number where the requested URL is to be rendered.
3. Enters the URL (e.g., http://www.google.com) in the *URL-region* of the Controller Tab and then hits the keyboard Enter key in the *URL-region*. **Note:** "http://" must be included in the URL.

The webpage will be loaded in the specified tab. **Note:** The user must hit the enter key in the *URL-region* for the webpage to be rendered in the specified tab. Hitting the enter key anywhere else will not work.

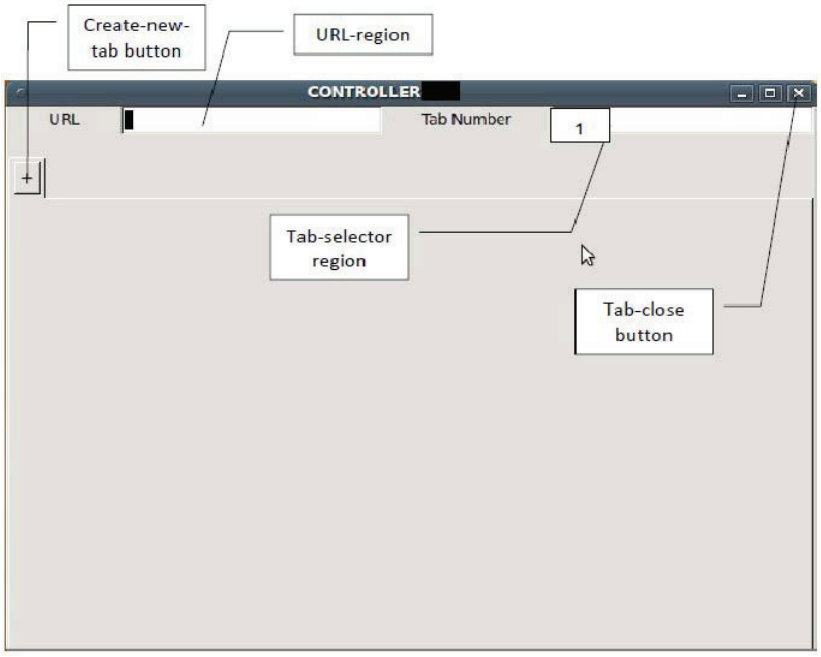

Figure 1: Controller Tab

Figure 2: URL-RENDERING Tab

2.3 URL-RENDERING process:

The URL-RENDERING tab process(es) are also children of the ROUTER process. They render requested URLs specified in the CONTROLLER windows. Fig 2 shows the layout of a URL-RENDERING tab. **Note:** The URL-RENDERING tab does not have any control where the URL can be entered. All URLs rendered in the URL-RENDERING tab are specified in the CONTROLLER with tab index. A URL-RENDERING process is created when the user clicks the *Create-new-tab* button. When you click the *Create-new-tab*, a new window will be created rather a tab in the same window.

3 Forms of IPC

The messaging between the CONTROLLER, ROUTER and URL-RENDERING processes should be implemented using pipes. A pipe is a channel of communication between parent and child process. In fact, you will use a set of pipes as in the knock-knock example in class to enable communication flow in both directions.

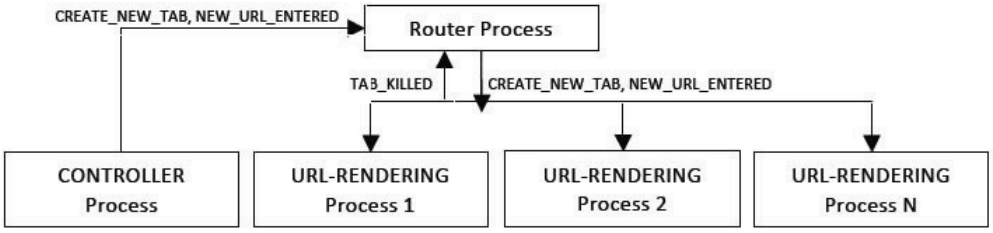

Figure 3: IPC among various browser process

Fig. 3 demonstrates the IPC between these processes. There are three kinds of messages:

1. CREATE_TAB: This request is sent by the CONTROLLER process to the ROUTER process when the user clicks the *Create-new-tab* button (see Fig. 1).
2. NEW_URI_ENTERED: The CONTROLLER process sends this message to the ROUTER process when the user hits the enter button after entering the URL in the *URL-region* (see Fig. 1). Then, the ROUTER process sends this message to the URL-RENDERING process with the specified tab number to render the URL.
3. TAB_KILLED: URL-RENDERING/CONTROLLER processes send this message to the ROUTER process when the user closes the tab by clicking the 'close' button in the tab-window.

See Appendix A for the request format and Section 4 on how each of the involved processes handles this message.

4 Program Flow:

4.1 ROUTER Process:

When you open your browser, the process that runs the `main` function is the ROUTER process. In the `main` function, the ROUTER process performs the following steps:

1. Create the CONTROLLER process:
   (a) First, the ROUTER process creates two *pipes* for bi-directional communication with the CONTROLLER process.
   (b) Second, the ROUTER *forks* a new child process, the CONTROLLER process.

2. Polling for requests from child processes:

   (a) The ROUTER process then *polls* (via non-blocking *read*) on the set of open *pipe* file descriptors created before forking the child process, in a loop. At this stage, there is at least one child process (the CONTROLLER process), which the ROUTER polls for messages. The termination condition for the loop is when there exists no child process for the ROUTER process i.e., all children processes have finished their execution.

   (b) The non-blocking read will read data from the pipe. If no data is available, the *read* system call will return immediately with return value of -1 and *errno* set to EAGAIN. In that case, simply continue with the *polling*. To reduce CPU consumption in the loop, you will use `usleep` between polls.

   (c) If *read* returns with some data, read the data into a buffer of type `child_req_to_parent` (see Appendix A for its definition). There are three types of messages that the ROUTER process may *read* from the pipe.

      i. CREATE_TAB: Upon receiving this message, the ROUTER process *forks* a URL-RENDERING process. Again, the ROUTER process first creates 2 *pipes* for bidirectional communication with the child process and then *forks* the URL-RENDERING child process.

      ii. NEW_URI_ENTERED: This message specifies the URL to be rendered along with tab index on which the URL should be rendered. Upon receiving this message, the ROUTER process simply writes this message on the *pipe* connecting the ROUTER process with the corresponding URL-RENDERING process whose index is specified in the message.

      iii. TAB_KILLED: This message contains the index of the tab that has been closed. Upon receiving this message, the ROUTER process checks: if the tab-index belongs to the CONTROLLER process, this indicates that the entire web browser should be closed. The ROUTER will send TAB_KILLED messages to every alive URL-RENDERING processes and then call `waitpid` with each URL-RENDERING process' pid. This prevents the child URL-RENDERING process from becoming a Zombie process. Then the ROUTER calls `exit` to terminate the entire program.
      If the tab-index belongs to a URL-RENDERING process, the ROUTER should send a TAB_KILLED message to the specific URL-RENDERING process and call `waitpid` with that process' pid. Then the ROUTER closes the file descriptors of the corresponding *pipes* which were created for communication with the closed tab. **Note:** This change should also be reflected in the list of *pipe* descriptors which the ROUTER process *polls* for messages.

4.2 CONTROLLER Process:

1. Upon its creation, the process invokes `controller_process` which calls `create_browser` to create the CONTROLLER window and invokes the `show_browser` function of the wrapper-library that we provide to you (see Appendix B). `show_browser` is a blocking call which returns when the user closes the CONTROLLER window. The CONTROLLER process exits with the return status of 0 (for success) when the user closes the CONTROLLER window. How does the CONTROLLER respond to requests if it is blocked? It uses callbacks described below.

2. Two callback functions are registered when creating the CONTROLLER window: `create_new_tab_cb` and `uri_entered_cb`. These get executed in the context of CONTROLLER process:

   (a) `create_new_tab_cb`: The basic template for this function is available in Appendix C. You should append your code in the template for this callback. The CONTROLLER process simply sends a CREATE_TAB message to the ROUTER process. You should supply the code for creating this message and writing it to the pipe descriptor connecting the CONTROLLER to the ROUTER. Code for extracting the pipe descriptor is provided in the template.

   The `child_to_parent_fd` field of `browser_window.channel` structure stores the pipe descriptor through which CONTROLLER communicates with the ROUTER - this descriptor gets set when you invoke the `create_browser` function of the wrapper library. See Appendix B for details. Note that the template code will only work if you invoked the `create_browser` wrapper-library function.

(b) `uri_entered_cb`: The basic template for this function is also available in Appendix C. You should append your code in the provided template for this callback. This callback function receives two input parameters: (1) the URL that the user has entered in url-region (see Fig. 1) and (2) auxiliary data (a `void*` which should be type-casted to a pointer of type `browser_window*)`. The template code extracts the pipe descriptor from the `browser_window` data-structure. Your code should prepare and send a NEW_URI_ENTERED request message to the ROUTER process. You can use `strcpy` to copy the URL to the `new_uri_req.uri` string buffer.

4.3 URL-RENDERING Process:

1. URL-RENDERING process is the child of the ROUTER process and is created when the user clicks the *create-new-tab* button in the CONTROLLER window. This causes the `create_new_tab_cb` callback function for the CONTROLLER process (which you register when you call `create_browser` function) to be invoked. The `create_new_tab_cb` callback function of the CONTROLLER process is expected to then send a 'CREATE_TAB' message to the ROUTER process. On receiving this message, the ROUTER process *forks* a new child process which calls `url_rendering_process` to create the URL-RENDERING window (Refer to Appendix B for details). Note that the `show_browser` function is never called by the URL-RENDERING process. Lastly, `render_web_page_in_tab` has only two parameters. It doesn't need `tab_index`.

2. Upon its creation, the URL-RENDERING process waits for messages from the ROUTER by reading (non-blocking) the pipe descriptor that the ROUTER process created for communication with the child processes as explained in bullet 2(a) of Section 4.1. It performs the following two tasks in a loop within `url_rendering_process`. The termination condition for the loop is the TAB_KILLED message from the ROUTER process.

   Task 1: Process GTK events: Invoke `process_single_gtk_event` wrapper-library function, if no data was received from the ROUTER process.

   Task 2: Handle two kinds of messages that it might receive:

   (a) NEW_URI_ENTERED: Upon receiving this message, the URL-RENDERING process renders the requested URL by invoking `render_web_page_in_tab` of the wrapper-library (see Appendix B).

   (b) TAB_KILLED: Invoke `process_all_gtk_events` function of the wrapper library (see Appendix B) and exit the process with the return status of 0 (for success).

   (c) If it reads a message of any other type, interprets this as a bogus message and you should do some error handling (for example, you may ignore the read message, or you may print an error message on the screen etc.). You have the choice of deciding upon the recovery strategy.

5 Error Handling:

You are expected to check the return value of all system calls that you use in your program to check for error conditions. In addition, if your program encounters an error (for example, an invalid tab number supplied in the tab selector region), a useful error message should be printed to the screen. Your program should be robust; it should try to recover from errors if possible (such as when any URL-RENDERING processes would crash potentially). If the error prevents your program from functioning normally, then it should exit after printing the error message. (The use of the `perror` function for printing error messages is encouraged.) You should also take care of zombie or/and orphaned processes (e.g., if controller exits, make sure all other child processes are removed). Upon closing all the tabs (of the web-browser), the main process must exit properly, cleaning-up any used resources. Use of the *wait* system call will be useful. **Note:** To test crash, check for running processes (ps -u $< uid >$) and kill them at the shell with kill -9 $< pid >$.

6 Grading Criteria (Approximately)

5% README file. See below.

20% Documentation with code, Coding and Style. (Indentations, readability of code, use of defined constants rather than numbers, modularity, non-usage of global variables etc.)

75% Test cases

1. Correctness (40%): Your submitted program does the following tasks correctly:
   (a) Creates new tabs when instructed. (5%)
   (b) Renders specified URL in the correct tab when instructed. Credits will only be granted if you implement the IPC correctly for this functionality.(20%)
   (c) Closing the tab terminates the associated process. Credits will only be granted for this aspect of correctness if after closing the tab, no zombie OR orphaned processes remain in the system. (15%)
2. Error handling(25%):
   (a) Handling invalid tab index specification in CONTROLLER tab.(5%)
   (b) Closing the CONTROLLER tab should close all the other tabs. (12%)
   (c) Error code returned by various system/wrapper-library calls. (3%)
   (d) There should be no "Broken-Pipe" error when your program executes. Also, appropriate cleanup must be done whenever any of the child-processes (CONTROLLER/URL-RENDERING process) terminates. For eg., closing the pipe ends.(5%)
3. Robustness (5%) Abrupt crashing of one URL-RENDERING process should not stall the browser. Main browser should be able to render web-pages correctly for current and/or future tabs.
4. Miscellaneous (5%): For any non-listed aspect of correctness of your submitted program.

7 Documentation

You must include a README file which describes your program. It needs to contain the following:

1. Who did what
2. How to compile the program
3. How to use the program from the shell (syntax)
4. Any explicit assumptions you have made

Within your code you should use one or two sentences to describe each function that you write. You do not need to comment every line of your code. However, you might want to comment portions of your code to increase readability. At the top of your README file and main C source file please include the following comment:

```
/* CSci4061 S2017 Assignment 2
* date:  mm/dd/yy
* name:  full_name1, full_name2, full_name3 (for partner)
* id:  id_for_first_name, id_for_second_name, id_for_third_name */
```

8 Deliverables:

1. Files containing your code
2. A README file (readme and c code should indicate this is assignment 2).

All files should be submitted using the SUBMIT utility. You can find a link to it on the class website. This is your official submission that we will grade. We will only grade the most recent and on-time submission.

9 Tips

All the code changes should be made in the "browser.c" file. You do not need to change any code of "wrapper.c", "wrapper.h" and "Makefile".

9.1 Compilation

We rely on a third party library GTK+ whose compilation and linking can be tricky (so follow our instructions). The base code we have provided can compile on the system specification required for this course's projects on the cselab machines: Ubuntu 16.04 + gcc 5.4.0. You are on your own for other configurations. You can safely ignore all the warnings you get after compiling the base code as they refer to variables we have declared for you but that are not (yet) used. When you use them, the errors will go away. Also, make sure you use the provided Makefile to compile your project code. It correctly specifies the compilation flags and linked library for you. If you want to manually compile the code using the gtk+ library, this web page may be useful: http://stackoverflow.com/questions/11546877/compiling-and-linking-gtk-3-with-c-project-on-ubuntu.

9.2 X11 forwarding

This Project involves a Graphical User Interface. If you are using ssh to connect to machines in CSELab, make sure to enable X11 forwarding. This can be done using -X flag in ssh command, and opening a X server on your computer. For more details see http://cseit.umn.edu/how-to/remote-linux-applications-over-ssh.

9.3 Partitioning the work

One way to split the tasks is to assign the implementation of each process to a team member. Before you implement though, you will need to identify the interactions between processes. It is difficult to test each process code separately due to frequent interactions between processes. So, after each process is implemented, you will need to exchange your code and stitch them together, which is the integration phase.

In the integration phase, tasks can also be partitioned among group members based on scenarios. There are 3 basic scenarios: 1) launch the browser and then open a new tab, 2) input URL and tab number then open a web page, and 3) close a URL-RENDERING tab. Every scenario requires all three processes to be work properly. Make sure scenario 1 passes testing first, since scenario 2 and 3 are dependent on scenario 1.

9.4 Working steps

At a first glance, this project may be daunting given the long description! However, the description is used to explain how the browser works and the process model. We advise you to approach the project following these steps to get started:

1. Run the provided solution executable file, understand what the end product should look like;
2. Read the appendix;
3. Look at the provided code and map the writeup to it.