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

Posted Oct 12 – Due Oct 31, 2:00 pm Groups of 2-3

1 Introduction

Most traditional web browsers run as a multi-threaded process, rendering web pages on user request. However, a multi-process architecture for web-browsers has emerged, e.g. Internet Explorer 8.0 and Google Chrome. Each browser process manages one or more tabs. This design architecture provides two primary benefits:

1. Robustness: Web-page isolation via separate processes makes the web-browser immune to crashes by providing address-space isolation where each web-page is rendered by a separate process. A plug-in crash (one of the main reasons for browser crashes) in one web-page will have then have no effect on other web-pages being managed by other processes.

2. Parallelism: Web-pages are becoming ever more complex with contents ranging from JavaScript code to full-blown web-applications like Google docs. With the functionality of a web-browser transforming into a platform where web-applications can run independently and in parallel, a multi-process architecture for the browser is attractive on multi-node/multi-core systems. In this project, we will explore this new architecture for a web-browser using our new mechanisms for IPC using `pipes`. Note: we will provide all browser graphics and event handling code (e.g. clicks and input retrieval).

2 Description

The multi-process web-browser uses a separate process for each of its tabs to guard against error in the web-page rendering engine. **You will design a multi-process web-browser** in this assignment given some initial code. 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 open tab.

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, it waits for requests from its child processes (CONTROLLER/URL-RENDERING processes) in a loop (*polling* the *pipe* descriptors via non-blocking read). Note that 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.

2.2 CONTROLLER process

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

1. *URL* region: A text field where the destination URL (eg., `http://www.google.com`) is entered.
2. *Tab-selector* region: The tab where the URL is to be rendered. It accepts an integer value >0 and less than the number of opened tabs.
3. *Create-new-tab* button: The control for creating a new tab.
4. *Close-tab* button: The control for closing the tab.

When a user wants to open a URL in a new tab, she:

1. Clicks the 'create-new-tab' button (this opens a new URL-RENDERING window (shown in Fig 2))

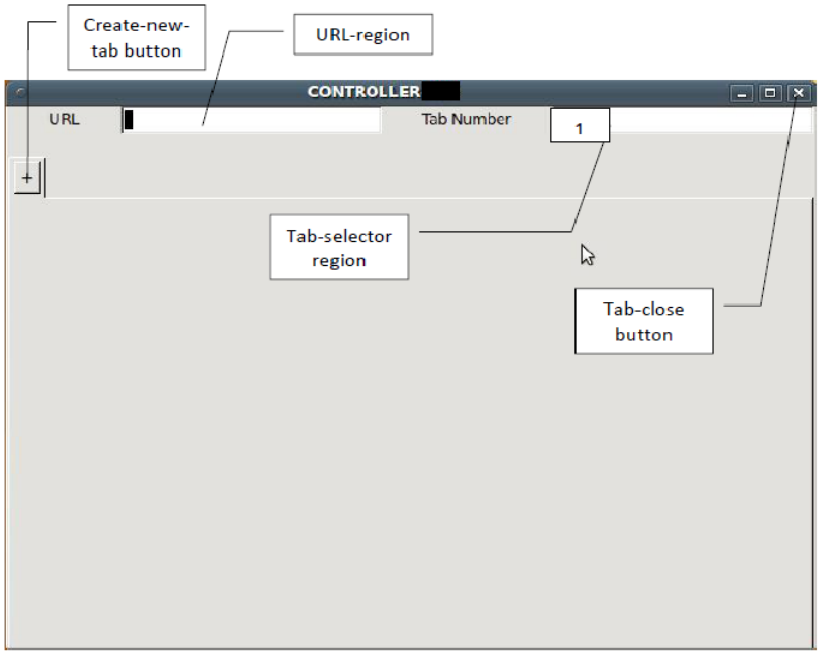

Figure 1: Controller Tab

2. Enters the URL in the 'URL' region of the CONTROLLER followed by the tab-number where the requested URL is to be rendered (e.g., http://www.google.com, 1)

3. Hits the keyboard Enter key in the 'URL region.'

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

2.3 URL-RENDERING process:

The URL-RENDERING tab process(es) are also children of the ROUTER process. They render the requested URL (specified in the CONTROLLER windows). Fig. 2 shows the layout of the URL-RENDERING tab. Note that 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 appropriate tab index.

The URL-RENDERING process is created when the user clicks the 'create-new-tab' button. This causes the `new_tab_created_cb` callback function for the CONTROLLER process (which you register when you create the CONTROLLER window via create_browser()) to be invoked. The `new_tab_created_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 invokes `create_browser()` to create URL-RENDERING window (Refer Appendix B for the parameters you need to pass to create a URL-RENDERING window). Once created, the functionality of URL-RENDERING process is described in Section 4.3. 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.

3 Forms of IPC

You will implement the following forms of IPC while implementing this multi-process browser.

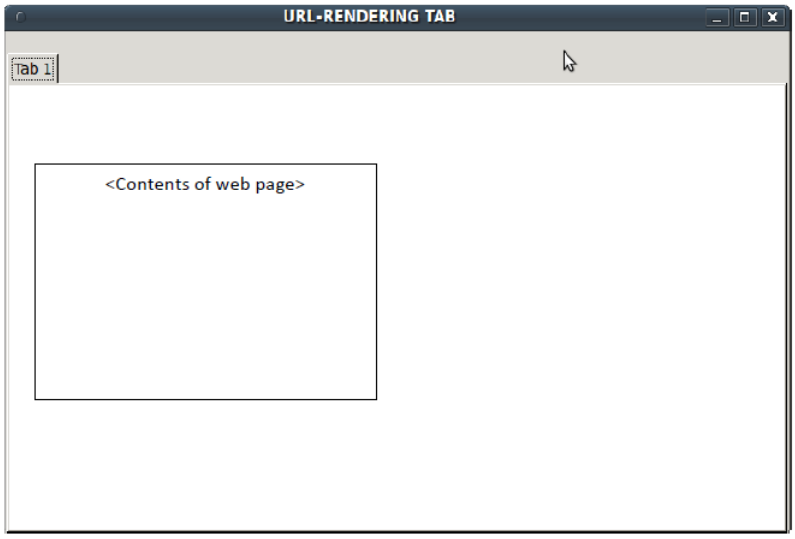

Figure 2: URL-RENDERING Tab

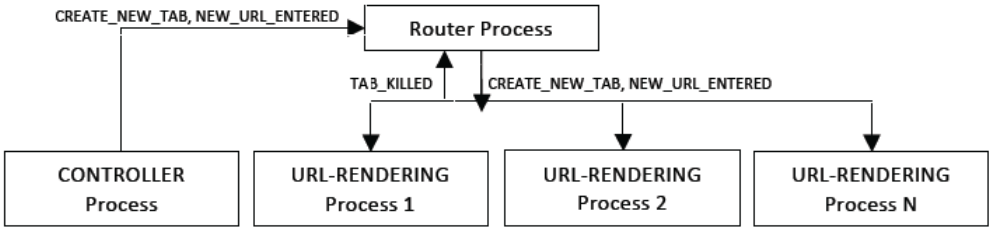

Figure 3: IPC among various browser process

1. Pipes: Pipe is a channel of communication between parent and child process. The messaging between the CONTROLLER, ROUTER and URL-RENDERING processes will be implemented using pipes. The IPC in Section 3.1 is to be implemented using 'pipes.'

3.1 Inter-Process Communication Infrastructure:

Fig. 3 demonstrates the IPC among the browser processes. There are three kinds of messages exchanged among the various browser processes:

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

2. NEW_URL_ENTERED: The CONTROLLER process passes this message to the ROUTER process when the user hits the enter button after entering the URL in the URL region (see Fig. 1). See Appendix A for the request format and Section 4 on how each of the involved processes handles this message.

3. TAB_KILLED: Individual URL-RENDERING/CONTROLLER processes pass this message to the ROUTER process when the user closes the tab by clicking the 'cross' at the top-right corner of the tab-window (see Fig. 1). See Appendix A for the request format and Section 4 on how each of the involved processes handles this message.

4 Program Flow:

This section describes the flow of each of the involved processes.

4.1 ROUTER Process:

When you invoke your browser, the `main()` function of your program starts executing. Upon its invocation, the ROUTER process performs the following tasks:

1. *fork* the CONTROLLER process:
   (a) Before this, the ROUTER process creates two *pipes* for bi-directional communication with the CONTROLLER process.
   (b) Second, it *forks* the new child process, the CONTROLLER process. See Section 4.2 for a description of the CONTROLLER.
2. Waits 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 (see step b above), 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 child 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* (see step 1(b)). To reduce CPU consumption in the loop, you will use `usleep` between reads.
   (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. See Section 4.3 for a description of the URL-RENDERING process's functionality.
      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 tab-index of the recently closed tab. Upon receiving this message, the ROUTER process simply closes the file descriptors of the corresponding *pipe* which was created for communication with the closed tab. **Note:** This change should be now be reflected in the list of *pipe* descriptors which the ROUTER process *polls* for messages.
3. When all browsing windows (including the CONTROLLER window) are closed (which implies there are no child processes for the ROUTER process), the ROUTER process exits with a return status of 0 (for success), marking the successful completion of the program.

4.2 CONTROLLER Process:

1. The CONTROLLER process is created by the ROUTER as described in Section 4.1.
2. Upon its creation, it invokes the show_browser() function of the wrapper-library that we provide to you (see Appendix B). This 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 now.

3. Two callback functions are registered when creating the browser]window (CONTROLLER window) e.g., `new_tab_created_cb` and `uri_entered_cb`. These get executed in the context of CONTROLLER process:

   (a) `uri_entered_cb`: The basic template for this function is available in Appendix C. You should append your code in the end of the 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 supplied code should prepare and send a NEW_URI_ENTERED request message to the ROUTER process). The `child_to_parent_fd` field of browser_window 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) `new_tab_created_cb`: The basic template for this function is also available in Appendix C. You should append your code in the end of provided template for this callback. The template code extracts the URL along with the tab-index where user wishes to render the URL. 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.

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.

2. Upon its creation, it performs the following two tasks in a loop. The termination condition for the loop is the TAB_KILLED message from the ROUTER process.

   Task 1: Wait for messages from the ROUTER: the URL-RENDERING process reads (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) for incoming messages from the ROUTER process. There are three kinds of messages that it might receive:

   (a) CREATE_TAB: URL-RENDERING process ignores this message.

   (b) 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).

   (c) 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).

   (d) 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.

   Task 2: Process GTK events: Invoke `process_single_gtk_event()` wrapper-library function.

3. As mentioned earlier, the URL-RENDERING process exits on receiving the TAB_KILLED message from the ROUTER process.

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. If your program encounters an error (for example, invalid tab-index 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

(one such instance is given in Point 2 (Task 1, sub-point iv) of Section 4.3). 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.

6 Grading Criteria

5% README file. Be explicit about your assumptions for the implementation.

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. The purpose of your program
2. How to compile the program
3. How to use the program from the shell (syntax)
4. What exactly your program does
5. Any explicit assumptions you have made
6. Your strategies for error handling

The README file does not have to be very long, as long as it properly describes the above points. Proper in this case means that a first-time user will be able to answer the above questions without any confusion.

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 F2011 Assignment 2
* section:  one_digit_number
* section:  one_digit_number
* date:  mm/dd/yy
* name:  full_name1, full_name2 (for partner)
* id:  d_for_first_name, id_for_second_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.