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

Posted Oct 15 – Due Oct 29, 11:59 pm Groups of 2

1 Purpose

The purpose of this assignment is to get you, the intrepid programmer, familiar with interprocess communication (IPC) – namely, pipes. One very interesting application of IPC is used every time you open a modern browser like Chrome, Internet Explorer, or Firefox. The browser has several different tabs running independently of each other, each of which must communicate with the master browser process. This has two huge benefits, namely robustness and parallelism. When web pages are isolated, one can crash (due to a bad plug-in or looping JavaScript scripts) without affecting others. We are also able to keep one web page doing intensive computations or other machinations while we work with a different process on a different page. This is very attractive because our browser now allows us to take serious advantage of our multi-core systems.

This project will take IPC concepts and apply them to the web browser architecture and make both an interesting and useful project. We will provide all browser graphics and all code related to displayed page contents; you will need to implement the user-to-browser and process-to-process interaction.

You will also see some comments regarding system calls we will provide, especially concerning the browser functions that render web pages. This uses the open-source GTK library (see www.gtk.org) and you can look through it if you want; however, understanding the GTK sources will not be required for this project.

This is a complex lab; read the program description repeatedly and very carefully. We will discuss this in class, so pay attention and take notes.

2 Description

The multi-process web browser has one process per page, with a parent process overseeing user-to-tab requests. Your browser will take advantage of some initial code (e.g., code for displaying the browser and catching user clicks) to create three basic process types:

- Router (listens for process-to-process communication)
- Controller (shows the parent web page that allows user to open new tabs)
- URL Rendering (one per open tab, shows a web page after receiving a URL)

2.1 The ROUTER process

The ROUTER process is the main parent process; it runs when we launch the browser program. Its purpose is to:

- *Fork* the CONTROLLER process
- *Fork* any requested URL-RENDERING processes
- Wait for requests from those child processes (either CONTROLLER or URL-RENDERING processes)
- Create a set of pipes for processes to communicate with
- Keep checking (a.k.a. polling) pipes for new messages

Note that the ROUTER does not display anything and has **no** associated window; all it does is serve as a mail carrier between the CONTROLLER process, which users will interact with to open new tabs, and those URL-PROCESSING tabs.

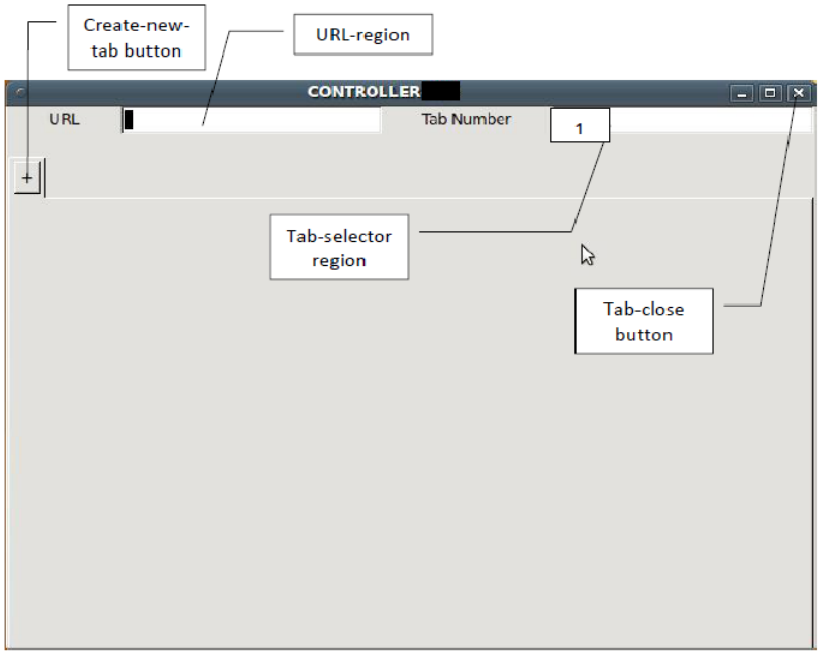

Figure 1: Controller Tab

2.2 The CONTROLLER process

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

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

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

- Clicks the 'create-new-tab' button (this opens a new URL-RENDERING window (shown in Fig. 2))
- Enters the URL in the 'URL' region of the CONTROLLER followed by the tab-number where the requested URL should go (e.g., http://www.google.com, 1)
- Hits the keyboard Enter key in the 'URL region.' (*If you hit Enter anywhere else it will not work!!*)

2.3 The URL-RENDERING process

The ROUTER spawns a URL-RENDERING tab process every time the user clicks the 'create-new-tab' button. Any URL-RENDERING tab can show a new URL; to do this, the user will write the URL and index (of the tab to use) in the CONTROLLER window and click the create-new-tab button. The ROUTER will read the URL and the tab index from the CONTROLLER window and write it as a new message to the URL-RENDERING process. See the URL-RENDERING tab in Fig. 2. The process is:

- The CONTROLLER process receives the message that the user requested a new browser (via the create-new-tab button)

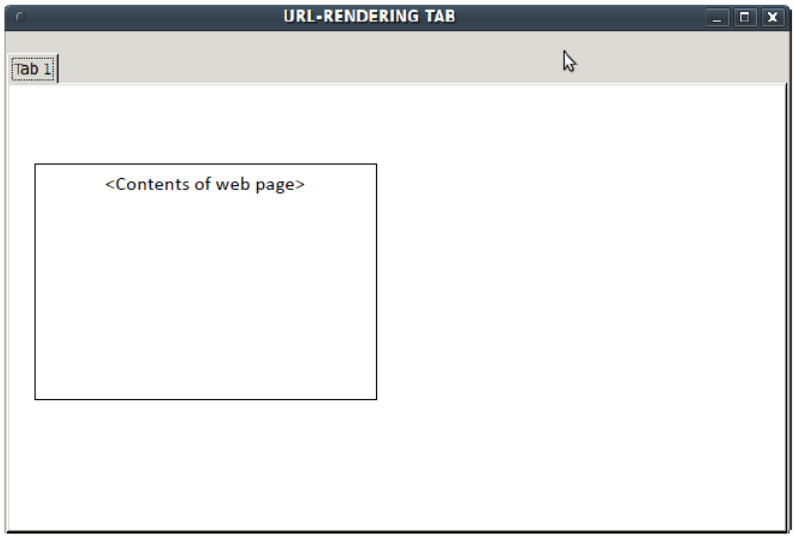

Figure 2: URL-RENDERING Tab

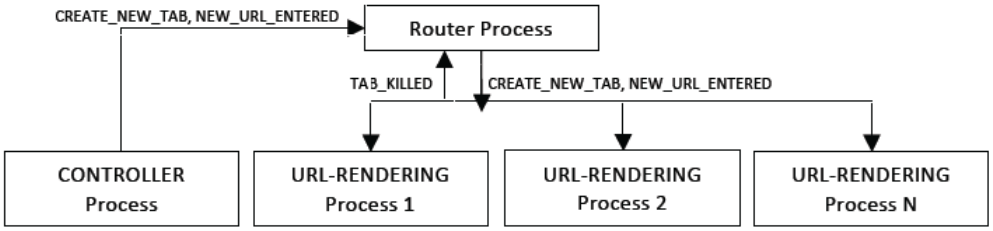

Figure 3: IPC among various browser process

- The CONTROLLER sends the new-tab request to the ROUTER process
- The ROUTER process forks a new process for URL rendering
- The new URL-RENDERING process launches a new browser window
- The URL-RENDERING process must show the URL given by the ROUTER (a more thorough description is given in Section 4.3)

3 Forms of IPC

Your multi-process browser must implement pipes. A pipe is a channel of communication between parent and child process; either process may write into one end of the pipe, and the other process will read from the other end of the pipe. All messaging between the CONTROLLER, ROUTER and URL-RENDERING processes will be implemented with pipes (see ”man 2 pipe” for the signature description of `pipe`).

Fig. 3 demonstrates the IPC among the browser processes. There are three kinds of messages exchanged among the various browser processes. The request format for each is described in Appendix A, and Section 4 shows how each of involved processes handles the message.

1. CREATE_NEW_TAB: The CONTROLLER process sends this to the ROUTER process when the user clicks the create-new-tab button (see Fig. 1).

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).

3. TAB_KILLED: Individual URL-RENDERING/CONTROLLER processes pass this message to the ROUTER process when the user closes the tab by clicking the 'X' at the top-right corner of the tab-window (see Fig. 1).

4 Program Flow

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

4.1 ROUTER Process:

When you invoke your browser, the `main()` function of your program starts executing. This will serve as your ROUTER process, and it must perform the following tasks:

1. Create pipes and `fork` the CONTROLLER process:
   (a) Create two *pipes* for bi-directional communication with the CONTROLLER process.
   (b) `fork` the first new child process, the CONTROLLER process.

2. Wait for requests from child processes:
   (a) Poll all of the open pipe file descriptors for new messages. Because not all pipes will contain new information, we must make sure the `read` call is non-blocking; that is, it returns so that we can go check other pipes. Initially, only the CONTROLLER exists, so ROUTER only polls that pipe. As more child processes are created, the ROUTER must keep track of all of them and poll each for changes in a loop. The loop ends when there no more child processes; e.g, when all the CONTROLLER and URL-RENDERING processes have finished execution.
   (b) The non-blocking read will read data from the pipe. If no data is available, `read` will return immediately with return value of -1 and set `errno` to EAGAIN. If this happens, ROUTER simply continues to poll the pipes, but should reduce CPU consumption by using the function `usleep` between reads.
   (c) If *read* returns 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: This means the ROUTER process must *fork* a URL-RENDERING process. Again, the ROUTER process must first create 2 *pipes* for bidirectional communication with the child process and then *forks* the URL-RENDERING child process.
      ii. NEW_URI_ENTERED: This message specifies a URL to load and the tab index of the window to use. After receiving this message, the ROUTER process simply writes this message on the proper *pipe* leading to the URL-RENDERING process specified by the tab index.
      iii. TAB_KILLED: This message contains the tab index of the recently closed tab. After receiving this message, the ROUTER process simply closes the file descriptors of the corresponding tab's communication *pipe*. **Note:** This change should be now be reflected in the list of *pipe* descriptors which the ROUTER process polls for messages. The actual end of the process is not handled by the ROUTER, but ROUTER needs to `wait` for the end of the process.

3. When all browsing windows (including the CONTROLLER window) are closed (that is, ROUTER has no more child processes), the ROUTER process exits with a return status of 0 for total success.

4.2 CONTROLLER Process

Before explaining the CONTROLLER process, you must understand a particular function type called a *callback function*. A callback function is a piece of code passed as an argument to other code (much like you have used other data types, like `int`, `char*`, or `struct node`, as function arguments). Callback functions allow the calling program to give up control to the callback function (you can also think of this as a subroutine within the main program). One benefit of using callback functions is that we can change the callback function without changing the calling code - an example of good code modularization! You will use several callback functions in this project.

1. The CONTROLLER process is created by the ROUTER as described in Section 4.1.
2. It launches the initial browser window via a blocking call that does not return until the user closes the CONTROLLER window. CONTROLLER is completely event-driven; it only responds to the user's interaction with the given fields and buttons and keyboard commands. The CONTROLLER process exits with the return status of 0 (for success) when the user closes the CONTROLLER window. The CONTROLLER is able to respond to requests (even though it is blocked) with 2 callback functions more fully described in Appendix B and the browser.c code provided. Basically, the functions will either:
   (a) Handle a new URL by helping the CONTROLLER process send a CREATE_TAB message to the ROUTER process (the `new_tab_created_cb` function).
   (b) Handle a new tab request by writing a NEW_URI_ENTERED request message to the ROUTER. The request will contain the URL from the url-region (see Fig. 1) and a tab index (the `uri_entered_cb` function).

4.3 URL-RENDERING Process

1. URL-RENDERING process is the child of the ROUTER process and is created every time the user clicks the 'create-new-tab' button in the CONTROLLER window.
2. After creation, it performs two tasks in a loop. It terminates when it receives a 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 (created by the ROUTER for this communication – see bullet 2(a) of Section 4.1) for messages from the ROUTER process. There are three kinds of messages that it might receive:
   (a) CREATE_TAB: URL-RENDERING process ignores this message, since only the CONTROLLER needs to deal with it.
   (b) NEW_URI_ENTERED: This means the URL-RENDERING process should bring up a new webpage (by invoking `render_web_page_in_tab()` explained in Appendix B).
   (c) TAB_KILLED: Finish processing any pending events and exit the process with the return status of 0 (for success). We provide the system call you'll use, and it's explained more in Appendix B.
   (d) If it reads a message of any other type, interprets this as a bogus message – this is where you need to do some error handling (for example, ignore the message, print an error message on the screen and continue, etc.). You may decide your own recovery strategy.

   Task 2: Process web browser events, via a system call we will provide.
3. As mentioned earlier, the URL-RENDERING process exits on receiving the TAB_KILLED message from the ROUTER process.

5 Error Handling

You must check the return value of all system calls that you use in your program for error conditions. If your program encounters an error (for example, invalid tab indexes or bogus message types), a useful error message should be printed to the screen. Your program should be robust; it should try to recover from errors if possible (we mentioned one example late in Section 4.3). If the error prevents your program from functioning normally, then it should print a useful error message and then exit. (We recommend using the perror() function for printing error messages.) You should also take care of zombie and/or orphaned processes (e.g. if controller exits, all other tab / child processes are properly removed). After closing all browser tabs, 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 F2012 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 */
```

For example, Marie's project code would include:

```
   /* CSci4061 F2012 Assignment 2
* section:  2
* date:  10/20/12
* name:  Marie Manner, no partner
* id:  123456 */
```

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 Moodle's submit utility. You can find a link to it on Moodle; we will grade whatever you have posted by the deadline.