Project 2: Multi-Process Web Browser: IPC and File I/O

CSCi 4061: Introduction to Operating Systems - Weissman – Fall 2022

*To be completed by your group of **size 3,** see Piazza for details*

Posted	Oct. 12, 05:00 PM
Intermediate Submission Due	Oct. 21, 11:59 PM
Final Submission Due	Oct. 28, 11:59 PM

1 Introduction

The browser of project #1 was limited in several important ways. First, the URL rendered in a tab window could not be changed. Two, self-killed tabs (via X) would become zombies until the controller exited. Third, tab indices were not re-used. Fourth, the browser lacked any modern useful features such as favorites/bookmarks and browsing history. To solve these problems you will improve the browser. You will utilize inter-process communication in the form of pipes, communication protocols, and file I/O. We have provided a template that we recommend you use.

2 Description

The URL tab process was unable to respond to any new commands from the controller since it was blocked. We have modified `render` in three ways: (1) it is not blocked and can receive communications (commands) from the ***controller***, (2) it now has its own callback to handle a URL selected to be a favorite, and its API has changed – it no longer receives a URL when it is launched, as tab creation and URL assignment are now decoupled. Instead, it receives two pipes (4 descriptors) that enable it to communicate in two directions with the controller. It is not possible to run `render` now from the shell, it must be `execl`'ed with two `pipe`s that the controller will create. As before, you will focus on the controller

```
render (<tab #: 1, ... >, <inbound pipe><outbound pipe>)
```

only.

The pipe inbound and outbound are named from the perspective of the tab process. So the `inbound` pipe is the pipe that the ***controller*** will use to send commands to the tab, and the `outbound` pipe is the pipe that the controller will use to read commands from the ***controller***. There is a pair of such pipes between the controller and each tab, held in an array in the controller. **Tabs now start at 1, as the controller will use tab 0 for its private pipe. The read-end of all pipes must be set to non-blocking.** We have also changed the ***browser*** interface such that it takes no command-line arguments. It obtains its blacklist from a file called *.blacklist* and its favorites from a file called *.favorites* in the local directory. The controller has also fundamentally changed: instead of blocking in gtk, it now continually checks if any commands have come in from each tab via that tab specific pipe. Callbacks still work as before. The type of all commands flowing in either direction is `req_t`. **Note that `req_t` has several fields that may be required for you to set for the command to be properly executed later.**

You will use a new stream-lined wrapper **BUT you will not modify anything in *wrapper.c* (thus only wrapper.o given to you).** In *wrapper.h*, we have marked the types and functions that you will use. We have also provided a utility (util.o) for useful functions (see *util.h* for details) that you will use. We have provided the **render** executable, and **browser_sol**, solution executable as before.

3 Tabs and URLs

Tabs are now created prior to specifying a URL by clicking the + button. This will invoke a callback `new_tab_created_cb` that you must write. This function will determine an available tab #, create an `inbound` and `outbound` pipe for the tab process, and then create the tab process via **render**. As before you will want to bookkeep the tab information, a data-structure called `tab_list` is provided for this purpose. Since you will properly be removing tabs, you should re-use deleted tab #'s (e.g. if tab 1 is removed, the index 1 can be given to a new tab). This callback must still check that you have not exceeded the tab limit. All bookkeeping state should be kept in the controller process (as before).

Once a tab is created, you can provide it a URL in a separate action on the ***controller***. To do this, you identify the tab by inputting a tab # in the tab window, then enter a URL in the URL window and hit return. At this point, the callback `uri_entered_cb` will be called by *gtk*. This callback must send the command NEW_URI_ENTERED to the specific tab process using the tab's `inbound` pipe. If the controller is X'ed, the wrapper sends a PLEASE_DIE command to the controller (how this happens isn't your concern).

4 Termination

Termination is now **much** cleaner although it creates an interesting flow of commands. When the controller receives a PLEASE_DIE command on its private pipe, it 1) sends a PLEASE_DIE command to each of its tabs using their `inbound` pipe (which will cause them to die), 2) waits for all tab children to die, and 3) exits. The controller is also immediately notified if a tab is self-killed (via

X on the tab window) by receiving a TAB_IS_DEAD command from the tab using the `outbound` pipe.

5 Favorites

The browser will now maintain a persistent list of favorites that can be displayed via a menu. A favorite is created by clicking + fav button on the tab which will send an IS_FAV command to the controller. You must handle this command. We have provided code that updates the favorite's menu, but it is your job to update the *.favorites* file and the variables for maintaining favorite URLs. There is **no way to delete** a favorite **nor do you have to check for bad URLs** (i.e. URLs that are in a proper format but do not render, e.g. https://fdjsdxxfdsdjfsdjf.com). You select a favorite by simply picking a favorite from the menu and a tab to render it in. **There is also a placeholder for browsing history, but this is not required.**

6 Controller main loop

The largest conceptual change in project #2 is that the controller is now in a loop so that it can both respond to commands and handle callbacks simultaneously. In this loop, it should continually check for commands from itself (e.g. PLEASE_DIE) via `comm[0]` and commands from each valid tab (e.g. TAB_IS_DEAD) via **each** tab's `outbound` pipe. The controller main is like before, it should do initialization of all data-structures, mark all tabs as free, and create its own pipe, before calling `run_control`.

7 Intermediate Submission

You get you started, you must submit an intermediate submission due Oct. 21, 11:59 PM. You will enable a new tab process to be created and a URL to be subsequently provided to that tab in the URL window. No error checking, favorites, termination, or constraints need to be handled at this step. I would implement `main`, basic `handle_uri`, `new_tab_created`, and `uri_entered_cb`. They all need to be implemented to get the preliminary to work. This is more work that the preliminary for project #1.

8 Possible split

This is a hard one to break down at least for the preliminary where everything needs to be working together. Work together closely for the preliminary. After that, functionality could be decomposed based on helper functions, favorite handling, termination.

9 Grading Criteria

10% - Intermediate submission

20% - Documentation of code, coding style. (Indentations, readability of code, use of defined constants, good names, modularity, etc.), following submit instructions (see below)

70% - Functionality. Constraints, and Error Handling.

1. New tab window (10%)
2. URL rendered in new tab window (10%)
3. Self-termination of tab (X) (10%)
4. Self-termination of controller (X) (10%)
5. Favorites including max favorites check (10%)
6. Tab reuse and proper max tab check (10%)
7. Error checking for all system calls (10%)

10 Getting Started

1. Read this description very carefully.
2. Then read it again with the template code alongside, paying attention to the places that are marked for you.
3. Confer with the group to see that everyone understands what is needed in the preliminary.
4. Confer with the group to see that everyone understands what is needed for the full project.
5. Start coding.

11 Documentation

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.

12 Deliverables:

A **tar file** containing your code and a **Makefile** that can be used to build and execute your code, and a **README** file. The README indicates how the work was partitioned within your group. There could be overlap. We will check this to see that everyone participated. You submit ONLY ONE tar file per group. You will be docked points if you do not follow these instructions.

```
/* CSCI-4061 Fall 2022 – Project #2
 * Group Member #1: <name> <x500>
 * Group Member #2: <name> <x500>
 * Group Member #3: <name> <x500> */
```