CS 367 Fall 2021
Project #4: Text Processing System
Due: Wednesday, December 1, 2021, 11:59pm

This is to be an individual effort. No partners.
No late work allowed after 48 hours; each day late automatically uses up one of your tokens.

1. Introduction
For this assignment, you are going to use C to implement a text processing system called the **AAH** Text Processing System. Once running, AAH maintains several text buffers (you can think of these like open files), and would be able to accept/execute commands from the user to modify the contents of these buffers. This assignment will help you to get familiar with the principles of process management in a Unix-like operating system. Our lectures on processes, signals, and Unix-IO as well as Textbook Ch.8 (in particular 8.4 and 8.5) and 10.3 will provide good references to this project.

2. Project Overview
A typical command shell receives line-by-line instructions from the user in a terminal. In this project, the shell is the interface to our text processing system. The shell would support a set of built-in instructions, which will then be interpreted by the shell, and acted on accordingly. In some cases, the instructions would be requests for the system to execute other programs. In that case, the shell would fork a new child process and execute the program in the context of the child.

The text processing system also has the responsibility to maintain a number of open buffers, and to keep them organized. The user is able to open, save, display, and close buffers, as well as to select which buffer is currently active. The user may run programs to fill a text buffer or to process the contents of an existing buffer. If the user runs commands on an active buffer, the user interface waits until the command is complete before doing anything else; however, if the user runs commands on a different buffer, the command takes place in the "background," and the user may do other things in the meantime. The shell provides some instructions to help manage and list open buffers, as well as some instructions to execute and control running commands.

For this assignment, your implementation should be able to perform the following:

- Accept a single line of instruction from the user and perform the instruction.
    - The instruction may involve creating or closing a text buffer.
    - The instruction may involve reading from or writing to a file.
    - The instruction may involve loading and running a user-specified program.
- The system must support any arbitrary number of simultaneous running processes.
    - AAH is able to both wait for a processes to finish, or let them run in the background.
- Perform basic management of open buffers and active buffers;
- Use pipes to send text to a running child process and collect the child's output.
- Use signals to suspend/resume or terminate running processes, and track child activity.

We will describe each aspect of the system in more details with some examples below.

Specifications Document: **(Chapter 3 at the end has some guidance on starting your design)**
This document has of a breakdown of each of the features, looking at specific details, required logging, and sample outputs. This is an **open-ended** project that will require you to make your own design choices on how to approach a solution. **Read the whole document before starting.**

2.0 Implementation Responsibility

Your project handout consists of several files:

- The starting template code in `textproc.c`.
- Headers, helper functions and logging functions in `logging.c`, `logging.h`, `parse.c`, `parse.h`, `util.c`, `util.h`, and `textproc.h`.
- A `Makefile` to build all components of the project.
- Several utility programs which you can use to help you test your text processing system.

You may take the existing starting template code in `textproc.c` and modify it. **This is the only file which you should modify**, and is the only file which you will be turning in. You should not need to include any additional header files, and you will not be allowed to use headers which change the project's linking requirements.

All of your code will be tested on the Zeus system, so you are expected to do your development on Zeus. Even if you have a Linux or Mac system at home, there are subtle differences in the implementation of signal handling and process reaping from system to system. You are responsible for making sure that your code functions correctly on Zeus.

When testing your code, there are some cases where you will be running external commands from within your system shell. You are allowed to use any normal commands on the system (e.g. `ls` or `grep`), any of the provided utility programs, or any programs you have written yourself. It is **not** recommended that you try to execute any commands which have an interactive interface (e.g. `vim`) from within your shell.

2.1 Use of the Logging Functions

In order to keep the output format consistent, all of your output will be generated by calling the provided logging functions at the appropriate times to generate the right output from your program. Do not use your own print statements, unless it is for your own debugging purposes. All logging output is encoded with a unique header which enables us to keep track of the activities of our shell. **The generated output from log calls will also be used for grading.**

The files `logging.c` and `logging.h` provide the functions for you to call from your code. Most of the log functions require you to provide additional information such as the Buffer ID (`bufid`), or possibly other info (e.g. process ID, file name) to make the call. We will explain more details how and when each log function is used in the specifications below.

2.2 Prompt, Accepting, and Parsing User Instructions

Once started, the shell prints a welcome message and a prompt, and waits for the user to input an instruction. Each line from the user is considered as one instruction.

Logging Requirements:

- The user prompt must be printed by calling `log_prompt()`.
    - The call to `log_prompt()` is already present in the starting template code, so will not need to take any additional action to display the prompt correctly.

If a command line input is empty, it will be ignored by the shell. Otherwise, you will need to parse the user input into useful pieces. We provide a `parse()` function in `parse.h`. (the implementation is in `parse.c`). The provided template `textproc.c` has already included the code which calls `parse()`. Feel free to use the provided `parse()` as is, or to implement your own parsing facility. **Check Appendix A for a detailed description of the input, output, and examples of the provided** `parse()`.

In testing, make sure you use user commands following these rules:

- Every item in the line must be separated by one or more spaces;
- Every line must start with the name of a shell instruction;
- Most instructions allow an optional buffer ID argument.
    - When a buffer ID is not supplied, the default buffer is the current active buffer.
- Some instruction (`open` and `write`) include a file name argument.
- Some instructions (`new` and `exec`) allow the user to include a program to execute.
    - The program can be any real program, e.g. `ls`.
    - The program name is optionally followed by its command line arguments.

For this assignment, you can make the following **assumptions**:

- All user inputs are valid command lines (no need for format checking in your program).
- You may assume a bounded input line size and number of command arguments.
    - The maximum number of characters per input line is `100`.
    - The maximum number of arguments per program is `25`.
    - Check `shell.h` for relevant constants defined for you.
- A command will not specify the path.
    - For example, you may see "`ls`" but not "`/usr/bin/ls`" in the input.

After calling `parse()`, the provided `textproc.c` leaves the design and implementation up to you as an **open-ended** project for you to solve. You are encouraged to write many helper functions as well and you may add additional code in to `main()` as needed.

2.3 Basic Shell Instructions

A typical shell program supports a set of **built-in commands** (internal functions that are built-in to the shell itself). If a built-in command is received, the shell process must execute that directly **without** forking any additional process. The most basic commands supported by our AHA system can be executed directly without the need to interact with any other parts of the system:

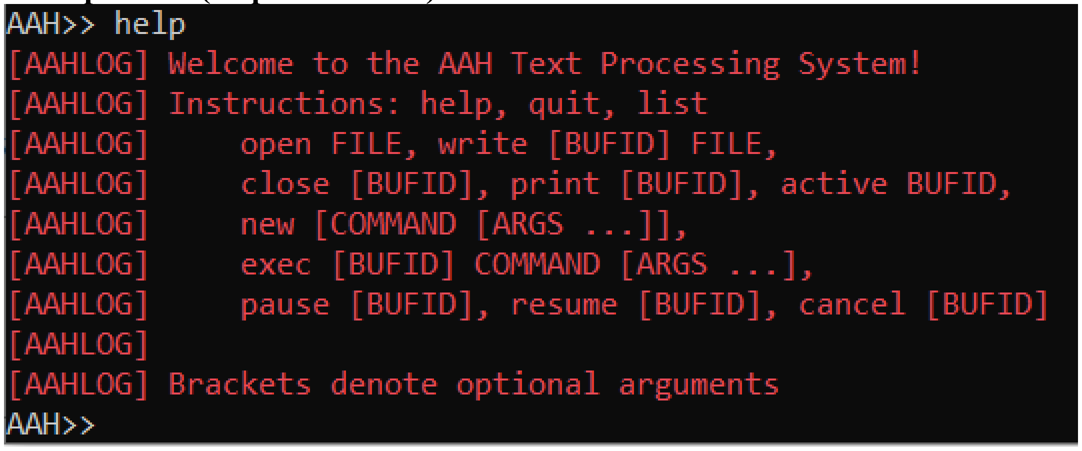
2.3.1 The "help" Built-In Instruction

`help`: when called, your shell should print on terminal a short description of the system, including a list of built-in instructions and their usage.

Logging Requirements:

- You must call `log_help()` to print out the predefined information.

Example Run (help instruction):



```
AAH>> help
[AAHLOG] Welcome to the AAH Text Processing System!
[AAHLOG] Instructions: help, quit, list
[AAHLOG]     open FILE, write [BUFID] FILE,
[AAHLOG]     close [BUFID], print [BUFID], active BUFID,
[AAHLOG]     new [COMMAND [ARGS ...]],
[AAHLOG]     exec [BUFID] COMMAND [ARGS ...],
[AAHLOG]     pause [BUFID], resume [BUFID], cancel [BUFID]
[AAHLOG]
[AAHLOG] Brackets denote optional arguments
AAH>>
```

2.3.2 The "quit" Built-In Instruction

`quit`: when called, your shell should terminate.

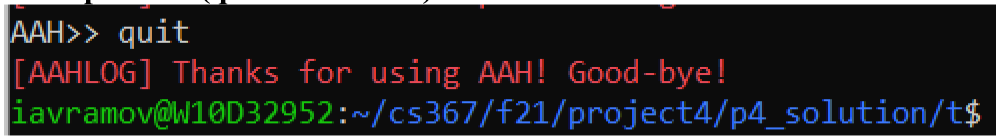
Logging Requirements:

- You must call `log_quit()` to print out the predefined information.
- You will then need to exit your shell program, using exit code 0.

Assumptions:

- You can assume there are no non-terminated background processes when calling `quit`.
    - In other words: you may quit immediately; you are not responsible for clean-up.

Example Run (quit instruction):



```
AAH>> quit
[AAHLOG] Thanks for using AAH! Good-bye!
iavramov@W10D32952:~/cs367/f21/project4/p4_solution/t$
```

2.4 Buffer Management Instructions

This program is a text processing system, so it has the ability to maintain several open text buffers at any time. Each open buffer is assigned a buffer ID, which is a positive integer value. In addition, every open buffer has current text contents, which can be represented as an ordinary zero-terminated C string. Finally, unless there are no buffers currently open on the system, one of the buffers is always designated the **active** buffer. Whenever a buffer-related instruction does not specify a buffer ID, the command will default to using the current active buffer.

Any open buffer is in one of three states: **Ready**, **Working**, or **Paused**. A buffer will typically be in the Ready state. However, if it is currently using an external command to process its text, then it will be in the Working or Paused state.

2.4.1 Buffer ID and Active Buffer Assignment

Whenever a new buffer is created or closed, we must follow certain rules about buffer IDs:

- Any new buffer is assigned a buffer ID which is one greater than the largest open buffer ID.
    - For example, if current open buffers are 1, 3, and 5, the next buffer will have ID 6.
- If there are no open buffers, then the next new buffer will be assigned buffer ID 1.
- If a buffer is closed, the remaining buffer IDs are not renumbered.
- If an active buffer is closed, the new active buffer will be the largest open buffer ID.
- If no buffers remain open, then there is no active buffer.

2.4.2 The "new" and "list" Built-In Instructions

`new`: when called, creates a new open buffer.

Logging Requirements:

- You must call **`log_open(buf_id)`** to indicate which buffer ID was created.
- You must call **`log_activate(buf_id)`** to indicate that the buffer was set active.
- In a later section, we will describe additional logging requirements for executing commands.

Assumptions:

- The buffer ID of the new buffer should be assigned consistent with the rules from 2.4.1.
- The buffer's initial string contents should be an empty (zero-length) string.
- The newly created buffer should be set as the current active buffer.
- The newly created buffer will be in the Ready state.
- We will revisit the **`new`** instruction when we discuss external commands – see 2.5.3.

`list`: lists all of the currently opened buffers.

- Includes the total number of open buffers and the current active buffer.

Logging Requirements:

- First call **`log_buf_count(num)`** to indicate the number of open buffers.
- Then, for every open buffer, call **`log_buf_details(buf_id, state, pid, cmd)`**.
- Finally, call **`log_show_active(buf_id)`** to show which buffer is currently active.

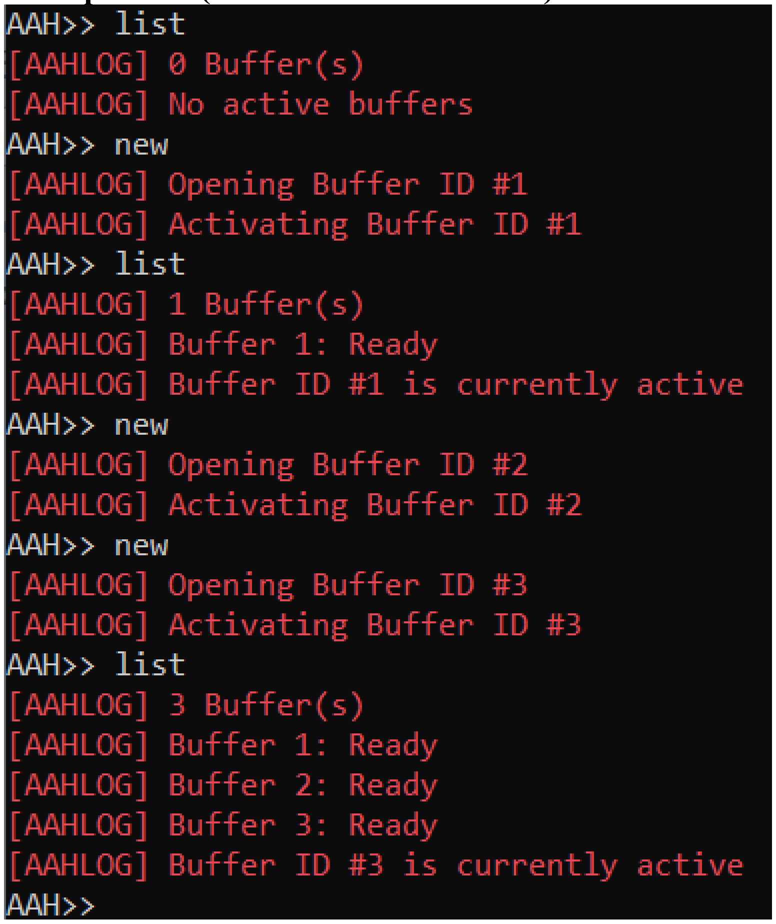
Assumptions:

- List the buffers in order of increasing buffer ID.
- Most buffers will begin in the state **`LOG_STATE_READY`** – but see 2.5.3.
- The **`pid`** and **`cmd`** can be 0 and **`NULL`**, respectively, for now – but see 2.5.2.
- The number of open buffers begins as zero, but increases if new buffers are opened.
- If there are any open buffers, then there will always be an active buffer.
- If there are no open buffers, then 0 should be passed into **`log_show_active()`**.

Implementation Hints:

- You will eventually need to be able to add or remove open buffers. Consider using a data structure which lets you add or remove arbitrary elements easily.

Example Run (new and list instructions):



```
AAH>> list
[AAHLOG] 0 Buffer(s)
[AAHLOG] No active buffers
AAH>> new
[AAHLOG] Opening Buffer ID #1
[AAHLOG] Activating Buffer ID #1
AAH>> list
[AAHLOG] 1 Buffer(s)
[AAHLOG] Buffer 1: Ready
[AAHLOG] Buffer ID #1 is currently active
AAH>> new
[AAHLOG] Opening Buffer ID #2
[AAHLOG] Activating Buffer ID #2
AAH>> new
[AAHLOG] Opening Buffer ID #3
[AAHLOG] Activating Buffer ID #3
AAH>> list
[AAHLOG] 3 Buffer(s)
[AAHLOG] Buffer 1: Ready
[AAHLOG] Buffer 2: Ready
[AAHLOG] Buffer 3: Ready
[AAHLOG] Buffer ID #3 is currently active
AAH>>
```

2.4.3 The "active" Built-In Instruction

`active` ***`BUFID`***: sets the new active buffer to ***`BUFID`***.

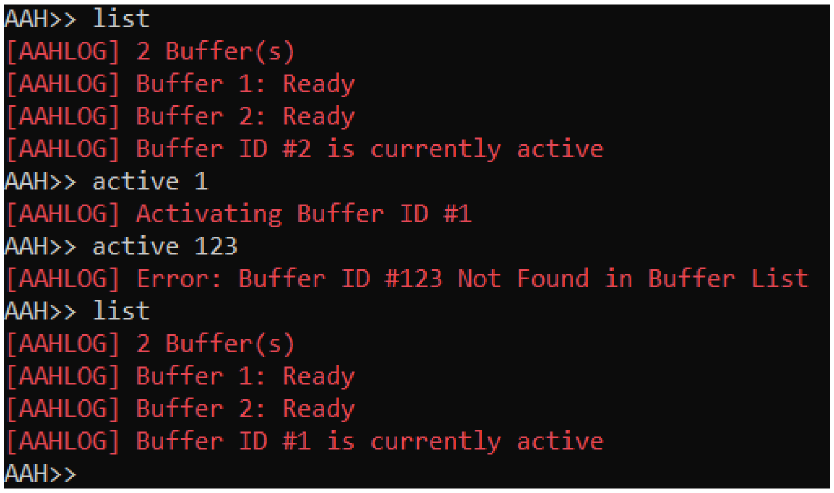
Logging Requirements:

- You must call `log_activate(buf_id)` to indicate that a buffer has been set as active.
- If the new active buffer ID is invalid, `log_buf_id_error(buf_id)` call instead.

Assumptions:

- If the buffer ID of the input is invalid, do nothing except print the log message.
- If there are no open buffers, do nothing except print the log message.
- If the argument is zero, treat is as if it was the buffer ID of the active buffer.
- The new active buffer may have a running process – see 2.5.2.

Example Run (active instruction):



```
AAH>> list
[AAHLOG] 2 Buffer(s)
[AAHLOG] Buffer 1: Ready
[AAHLOG] Buffer 2: Ready
[AAHLOG] Buffer ID #2 is currently active
AAH>> active 1
[AAHLOG] Activating Buffer ID #1
AAH>> active 123
[AAHLOG] Error: Buffer ID #123 Not Found in Buffer List
AAH>> list
[AAHLOG] 2 Buffer(s)
[AAHLOG] Buffer 1: Ready
[AAHLOG] Buffer 2: Ready
[AAHLOG] Buffer ID #1 is currently active
AAH>>
```

2.4.4 The "close" Built-In Instruction

`close` ***`BUFID`***: closes buffer ***`BUFID`***.

`close`: closes the active buffer.

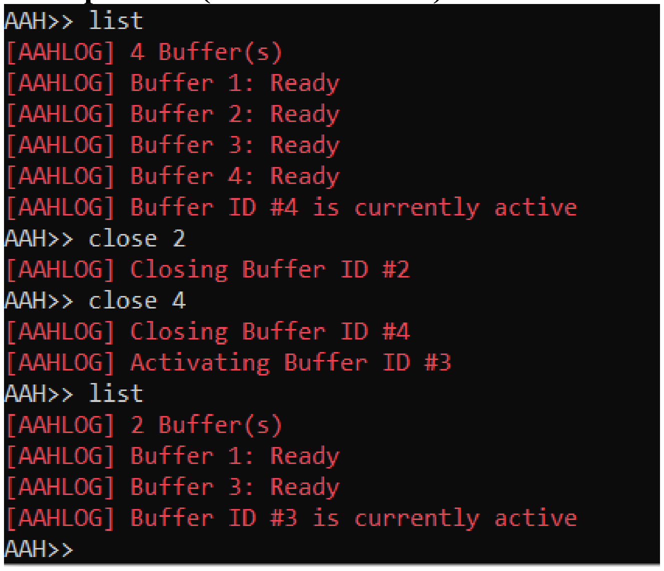
Logging Requirements:

- On a successful close, call **`log_close(buf_id)`**.
- If the selected buffer is invalid, call **`log_buf_id_error(buf_id)`** instead.
- If the buffer is not in Ready state and cannot be closed, call **`log_close_error(buf_id)`**.
- If the active buffer changes, see the logging requirements for **`active`** in 2.4.3.

Assumptions:

- Closing a buffer removes it from the list of buffers.
- Only a buffer which is in Ready state can be closed; otherwise make an error log call.
- If the active buffer is closed, make the current largest buffer ID the new active buffer.

Example Run (close instruction):



```
AAH>> list
[AAHLOG] 4 Buffer(s)
[AAHLOG] Buffer 1: Ready
[AAHLOG] Buffer 2: Ready
[AAHLOG] Buffer 3: Ready
[AAHLOG] Buffer 4: Ready
[AAHLOG] Buffer ID #4 is currently active
AAH>> close 2
[AAHLOG] Closing Buffer ID #2
AAH>> close 4
[AAHLOG] Closing Buffer ID #4
[AAHLOG] Activating Buffer ID #3
AAH>> list
[AAHLOG] 2 Buffer(s)
[AAHLOG] Buffer 1: Ready
[AAHLOG] Buffer 3: Ready
[AAHLOG] Buffer ID #3 is currently active
AAH>>
```

2.4.5 File I/O and Buffer Text: "open," "write," and "print" Built-In Instructions

`open` ***`FILE`***: creates a new buffer and fills it with text from the file ***`FILE`***.

Logging Requirements:

- Creating a new buffer has the same logging requirements as `new` in 2.4.2.
- If the file is successfully read, call `log_read(buf_id, file)`.
- On an error opening the file, call `log_file_error(LOG_FILE_OPEN_READ, file)` instead.
- The buffer is set active, with the same logging requirements as `active` in 2.4.3.

Assumptions:

- The only I/O error we are concerned about are errors opening the file.
- Opening a buffer from a file will set it active.
- Aside from reading from a file, the assumptions are the same as `new` in 2.4.2.

Implementation Hints:

- You may use the `fd_to_text(fd)` function to create a C string from an open file descriptor.
    - The function is provided for you in `util.h`, which is part of the P4 handout package.
    - You will need to open and close the file descriptor yourself with `open()`, `close()`.
    - You will need to free the returned string yourself.

`write` ***`BUFID FILE`***: writes the text contents of buffer ***`BUFID`*** into file ***`FILE`***.
`write` ***`FILE`***: writes the text contents of the active buffer into file ***`FILE`***.

Logging Requirements:

- If the file is successfully written, call `log_write(buf_id, file)`.
- On an error opening the file, call `log_file_error(LOG_FILE_OPEN_WRITE, file)` instead.
- If the selected buffer is invalid, call `log_buf_id_error(buf_id)` instead.

Assumptions:

- The only I/O error we are concerned about are errors opening the file.
- The file name will not be a number (to avoid confusion with the buffer ID number).
- If the file does not exist, create the file (use `O_CREAT`).
- If the file exists, overwrite the file (use `O_TRUNC`).
- Even if the buffer state is not Ready, we still want to do the `write` operation.
- Writing to a file does **not** close/remove the buffer.

Implementation Hints:

- You may use the `text_to_fd(text, fd)` function to write a C string to an open file.
    - The function is provided for you in `util.h`, which is part of the P4 handout package.
    - You will need to open and close the file descriptor yourself with `open()`, `close()`.
- Make sure to use `0600` as the third argument of `open()` as needed.
    - With `open()`, the third argument sets the file's reading/writing/executing permission.
    - It will set the file to be readable and writable by the owner of the file only.

`print BUFID`: writes the text contents of buffer `BUFID` to the terminal.
`print`: writes the text contents of the active buffer to the terminal.

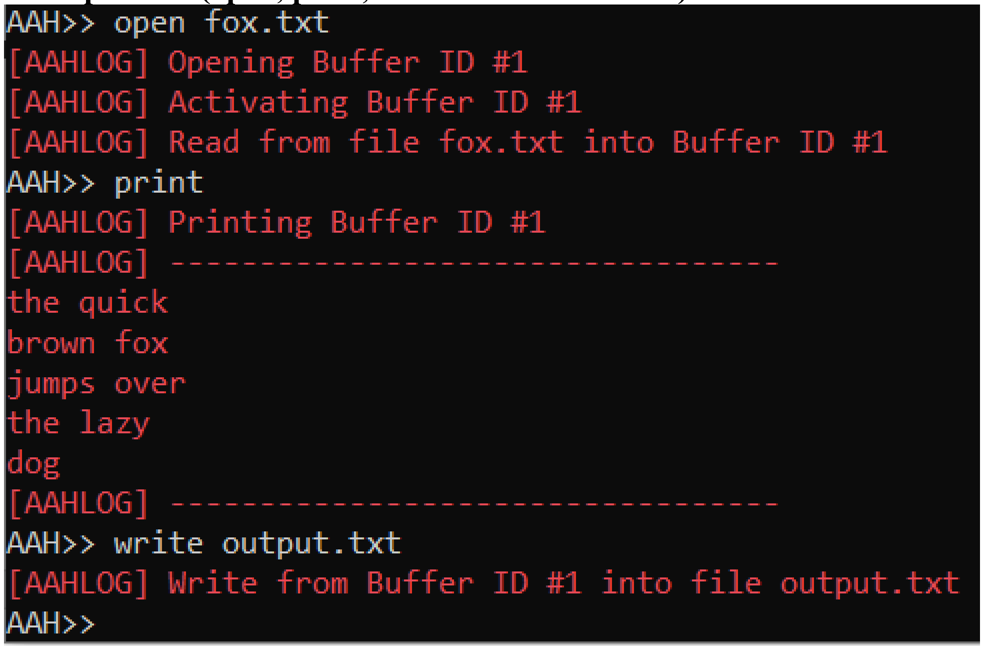
Logging Requirements:

- Write the text contents of the buffer using `log_print(buf_id, text)`.
- If the selected buffer is invalid, call `log_buf_id_error(buf_id)` instead.

Assumptions:

- Every open buffer ID has text contents, even if they are empty.

Example Run (open, print, and write instructions):



```
AAH>> open fox.txt
[AAHLOG] Opening Buffer ID #1
[AAHLOG] Activating Buffer ID #1
[AAHLOG] Read from file fox.txt into Buffer ID #1
AAH>> print
[AAHLOG] Printing Buffer ID #1
[AAHLOG] ---------------------------------
the quick
brown fox
jumps over
the lazy
dog
[AAHLOG] ---------------------------------
AAH>> write output.txt
[AAHLOG] Write from Buffer ID #1 into file output.txt
AAH>>
```

2.5 Process Execution Instructions

Since this is a text *processing* system, we need to include capabilities for processing text. In order to enable this capability, we will give our shell the ability to run external commands as separate processes. We can run an external command by first forking a child process, and then – within the context of the child – using one of the `exec()` variants to actually run the program.

Our system will allow us to run multiple concurrent processes. In fact, every open buffer is allowed to have its own running process to filter its text. If a buffer is currently working on text processing, it will be in the Working (or Paused) state, otherwise it will be in the Ready state. Any buffer cannot have more than one process at work filtering its text.

2.5.0 Execution Paths

When we execute external commands using `execv` or `execl`, we will also need to know the full path of the command to satisfy its first argument. To generate this, we will need to check two different paths for each command. These are: "`./`" and "`/usr/bin/`". Both of these paths must be checked, in this order, for an entered command. A command as entered on the command line will not have any path to begin with.

For example, if the user enters the command "`ls -al`", we will try both "`./`" and "`/usr/bin/`" as the path argument to `execv` or `execl`, **in that order**. We would first try to execute "`./ls`", and failing in that, we then execute the correct path in "`/usr/bin/ls`". Check the error code on `execv` or `execl` to see if the path was not found before checking the next one. If neither path leads to a valid program, then we would handle it as a path error and issue the appropriate log function.

Since the path argument of `execv` or `execl` needs to be modified from the original command by concatentating in "`./`" or "`/usr/bin/`", we will simply keep the original command name as **argv[0]**. So, if the user inputs "`ls -al`", then the path may be either "`./ls`" or "`/usr/bin/ls`" depending on which one works, but our `argv[0]` will still need to be "`ls`", which is what the user typed in.

2.5.1 Pipes

When a process is used to filter buffer contents, it first uses the current buffer contents as input. When it is done processing, it takes the output of the command – if it exited successfully – and uses that to replace the buffer contents. The key question is how we pass data to and from the child process. The answer is that we can use pipes.

A pipe is like a double-ended file: we can write to one end of the pipe and read from the other end of the pipe. If we create a pipe before forking a child, then the child will inherit the pipe, and the parent and child can use the ends of the pipe to send messages to one another. **We would need two different pipes** (four different file descriptors), one to send the initial text from the parent to the child's input, and a second pipe to send the child's output back to the parent.

When using a pipe, we should close the end of the pipe which we are not using (typically, the parent would close one end of the `pipe` after forking, while the child would close the other end). Also, as we would with any file, we should close the end of the pipe which we are using after we are done using the pipe.

Other than that, a pipe is used like any other file. We can make use of the utility functions `text_to_fd()` and `fd_to_text()` from the provided file `util.h`.

2.5.2 The "exec" Built-In Instruction

`exec` ***`BUFID CMD [ARGS …]`***: executes the external command ***`CMD`*** on the text contents of buffer ***`BUFID`***, storing the output back into the buffer.
`exec` ***`CMD [ARGS …]`***: executes the external command ***`CMD`*** on the text contents of the active buffer, storing the output back into the buffer.

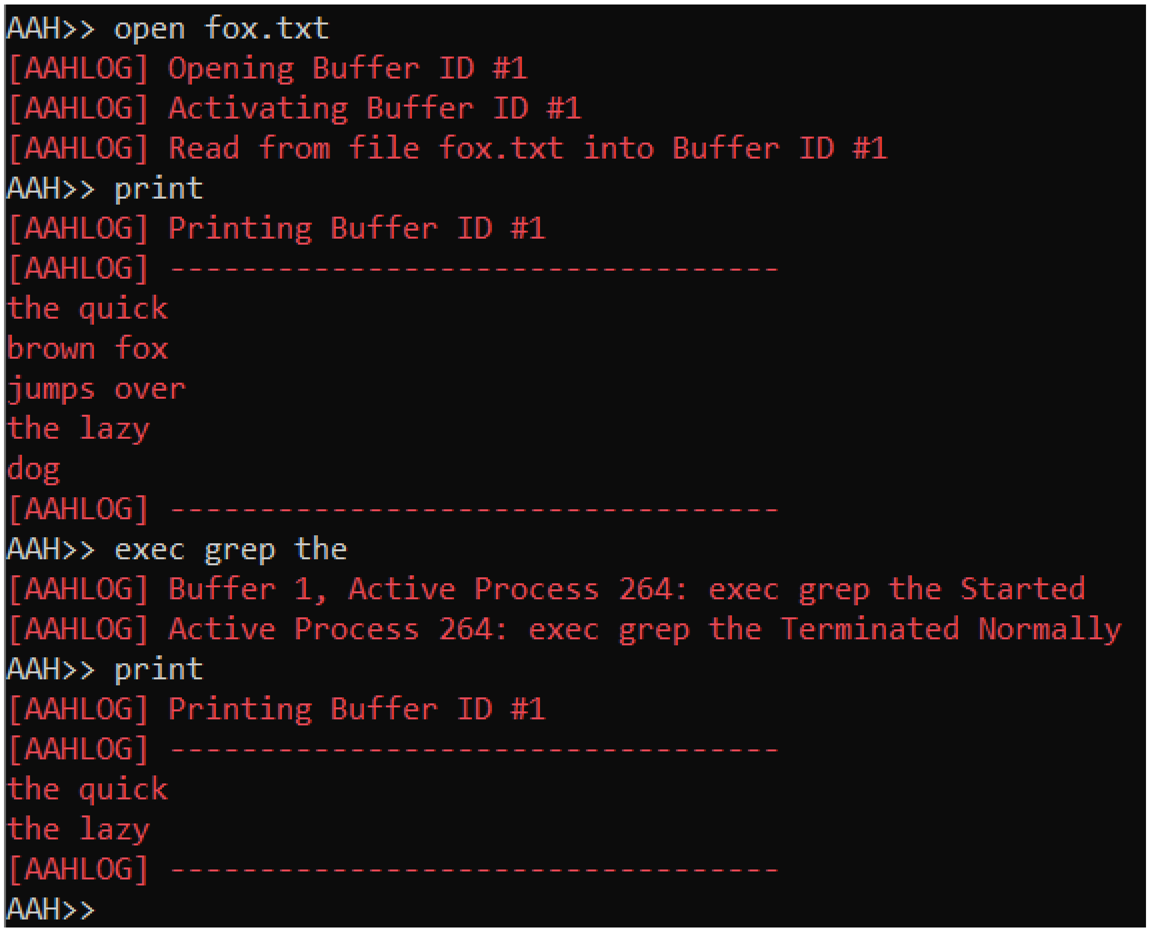
Logging Requirements:

- If the command is successfully started, call `log_start(buf_id, pid, type, cmd)`.
    - The `type` is `LOG_ACTIVE` or `LOG_BACKGROUND`.
    - Use the full input line as `cmd`.
- If the command cannot be executed, call `log_command_error(cmd)` instead.

  - Use the full input line as `cmd`.
- If the buffer is not in Ready state, call `log_cmd_state_conflict(buf_id, state)` instead.
  - State is the integer state value (Ready, Working, or Paused, see `logging.h`)
- If the selected buffer is invalid, call `log_buf_id_error(buf_id)` instead.
- When the process terminates, there should be a log message; see 2.6.3.

Assumptions:
- The buffer contents should only be replaced if the command terminates successfully.
  - It must terminate normally, **and** its exit code must be zero.
- The `exec` instruction can only be successfully run if the buffer is originally in Ready state.
- The exec will **not** be used on interactive programs (e.g. `vim`).
- All commands will be entered without a path (the path is entered by the shell).
- If the active buffer is selected, we must wait for the process to finish before moving on.

Implementation Hints:
- We can use `fork()` to create a new child process.
  - See **Appendix C** for information about something you should do after forking.
- We can use either `execl()` or `execv()` to load a program and execute it in a process.
- Though `execl` or `execv` do not normally return, they **will** return with a `-1` value on error.
  - Example: if the path or command cannot be found.
  - Use the man pages for the command you wish to use to see the details.
  - Check both valid paths (`./` and `/usr/bin`) with a command before calling it an error.
- When first trying to implement the command, run it without reassigning input or output.
- Once that works, try to reassign input using a pipe.
  - The `grep` command (keyword search) is good for testing if it works.
- Once input works, reassign output as well, using a pipe.
- When replacing the buffer text with the command output, do not forget to free the original.
  - The function `fd_to_text()` will copy text from an FD into a C string for you.
    - The function is provided for you in `util.h`, which is part of the P4 handout.
  - You will need to open and close the file descriptor yourself with `open()`, `close()`.
  - Once you have done this step, you will not be able to see the output directly.
    - If it has executed correctly, you will be able to see the output using `print`.
- If an active buffer is used for the instruction, use `waitpid()` to wait for it to finish.
  - Either way, signals are relevant to process completion; see 2.6.3 and 2.6.6.
- Use the `pipe()` command to create the two ends of a single pipe.
  - We will need to use two pipes, one for input from the child, and one for output.
  - Use `dup2()` to change the standard input/output of the child **after** forking.



```
AAH>> open fox.txt
[AAHLOG] Opening Buffer ID #1
[AAHLOG] Activating Buffer ID #1
[AAHLOG] Read from file fox.txt into Buffer ID #1
AAH>> print
[AAHLOG] Printing Buffer ID #1
[AAHLOG] ---------------------------------
the quick
brown fox
jumps over
the lazy
dog
[AAHLOG] ---------------------------------
AAH>> exec grep the
[AAHLOG] Buffer 1, Active Process 264: exec grep the Started
[AAHLOG] Active Process 264: exec grep the Terminated Normally
AAH>> print
[AAHLOG] Printing Buffer ID #1
[AAHLOG] ---------------------------------
the quick
the lazy
[AAHLOG] ---------------------------------
AAH>>
```

2.5.3 The "new" Built-In Instruction, Revisited

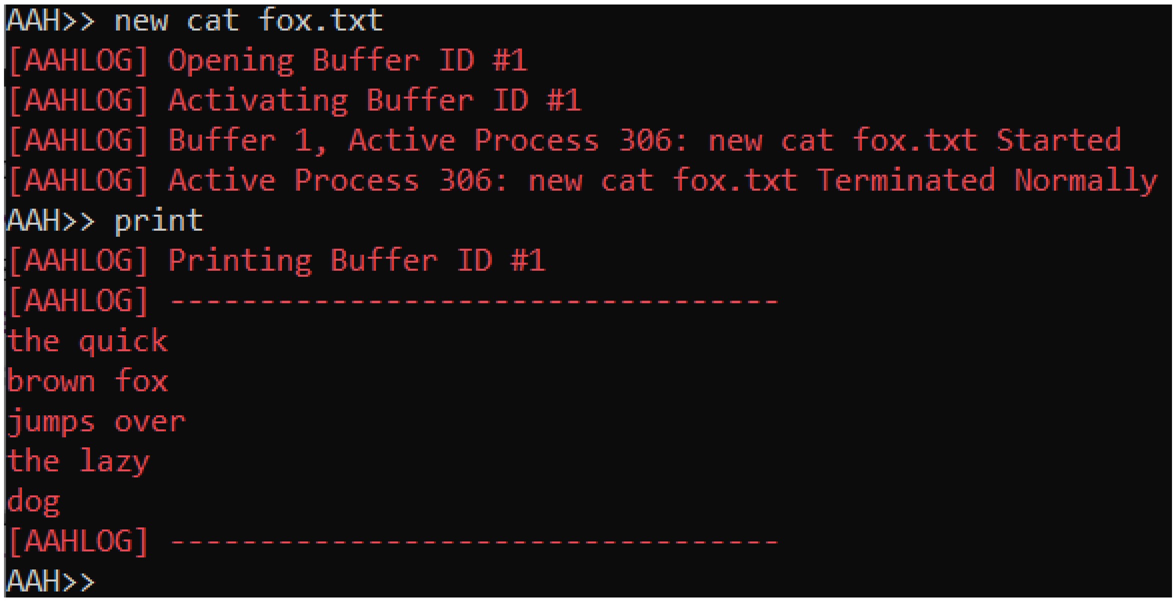
`new` ***`CMD`*** `[`***`ARGS …`***`]`: creates a new buffer and populates it with the text output of the external command ***`CMD`***.

Logging Requirements:

- This command is identical to if we had first called **`new`** followed by **`exec`**.
    - The logging requirements are the same.

Implementation Hints:

- This "upgraded" version of **`new`** is literally like first calling **`new`** followed by **`exec`**.
- When the buffer is first created, its text contents are a zero-length empty string.
    - This "empty string" is still the input to the **`exec`** call.



```
AAH>> new cat fox.txt
[AAHLOG] Opening Buffer ID #1
[AAHLOG] Activating Buffer ID #1
[AAHLOG] Buffer 1, Active Process 306: new cat fox.txt Started
[AAHLOG] Active Process 306: new cat fox.txt Terminated Normally
AAH>> print
[AAHLOG] Printing Buffer ID #1
[AAHLOG] ----------------------------------
the quick
brown fox
jumps over
the lazy
dog
[AAHLOG] ----------------------------------
AAH>>
```

2.6 Process Control and Signaling

In addition to simply being able to execute external processes, we are also interested in being able to control running processes. We may want to cancel a process before it is complete. Or we may want to pause (stop) a running process and resume (continue) it later. This especially applies if we attempt using `^C` or `^Z` on the active buffer's process, from the terminal.

Finally, if we switch our active job to a different job which may already have a running process, then we will need to wait for the new active process to finish.

2.6.1 The "cancel", "pause", and "resume" Built-In Instructions

`cancel` ***`BUFID`***: terminates the process running in buffer ***`BUFID`*** using **`SIGINT`**.
`cancel`: terminates the process running in the active buffer using **`SIGINT`**.

Logging Requirements:

- When signaling the cancel, call **`log_cmd_signal(LOG_CMD_CANCEL, buf_id)`**.
- If the selected buffer is invalid, call **`log_buf_id_error(buf_id)`** instead.
- If no process is running, call **`log_cmd_state_conflict(state, buf_id)`** instead.
- As a side effect, the child may exit, which leads to additional logs; see 2.6.3.

Assumptions:

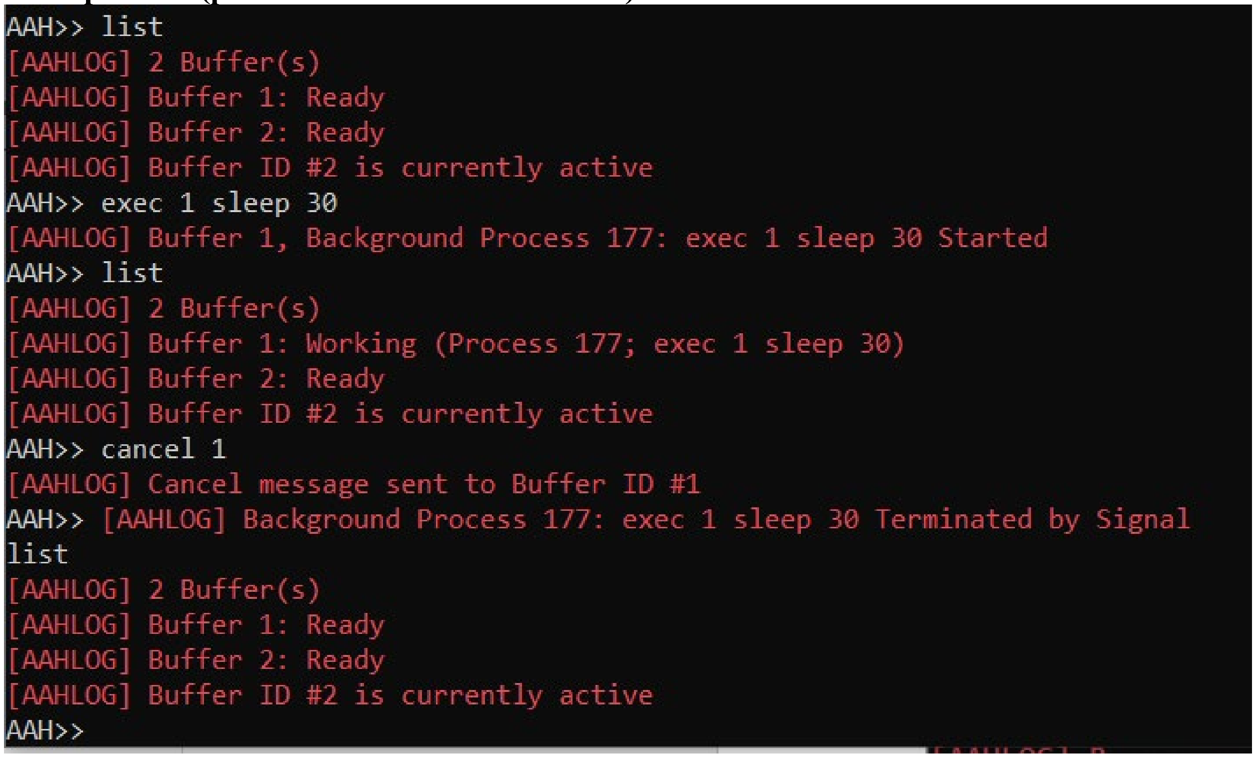
- Working or Paused processes can be canceled; a Ready process will produce the error log.

Implementation Hints:

- A signal can be sent to a child process by using the **`kill()`** function.
- The signal which we want to send is **`SIGINT`**.
- This instruction only needs to signal the need to cancel to the process.

- Termination and clean-up would be handled separately; see 2.6.3.

Example Run (pause and resume instructions):



```
AAH>> list
[AAHLOG] 2 Buffer(s)
[AAHLOG] Buffer 1: Ready
[AAHLOG] Buffer 2: Ready
[AAHLOG] Buffer ID #2 is currently active
AAH>> exec 1 sleep 30
[AAHLOG] Buffer 1, Background Process 177: exec 1 sleep 30 Started
AAH>> list
[AAHLOG] 2 Buffer(s)
[AAHLOG] Buffer 1: Working (Process 177; exec 1 sleep 30)
[AAHLOG] Buffer 2: Ready
[AAHLOG] Buffer ID #2 is currently active
AAH>> cancel 1
[AAHLOG] Cancel message sent to Buffer ID #1
AAH>> [AAHLOG] Background Process 177: exec 1 sleep 30 Terminated by Signal
list
[AAHLOG] 2 Buffer(s)
[AAHLOG] Buffer 1: Ready
[AAHLOG] Buffer 2: Ready
[AAHLOG] Buffer ID #2 is currently active
AAH>>
```

`pause` ***`BUFID`***: suspends the process running in buffer ***`BUFID`*** using `SIGTSTP`.
`pause`: suspends the process running in the active buffer using `SIGTSTP`.

Logging Requirements:
- When signaling the pause, call `log_cmd_signal(LOG_CMD_PAUSE, buf_id)`.
- If the selected buffer is invalid, call `log_buf_id_error(buf_id)` instead.
- If no process is running, call `log_cmd_state_conflict(state, buf_id)` instead.
- As a side effect, the child may suspend, which leads to additional logs; see 2.6.3.

Assumptions:
- Working or Paused processes can be paused; a Ready process will produce the error log.

Implementation Hints:
- A signal can be sent to a child process by using the `kill()` function.
  - Yes, it is still called "kill" if we are doing something benign like pausing the process.
- The signal which we want to send is `SIGTSTP`.
- This instruction only needs to signal the need to pause to the process.
  - The suspension event would be handled separately; see 2.6.3.

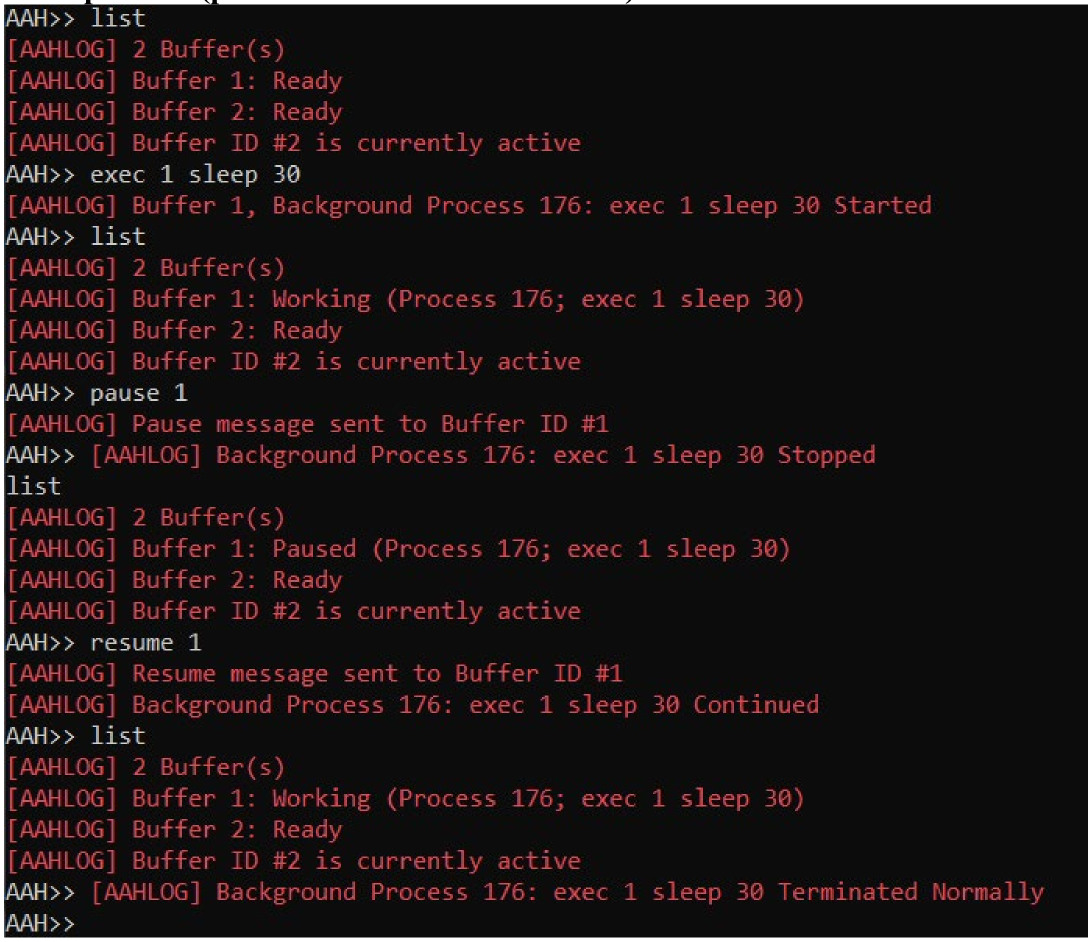
`resume` ***`BUFID`***: resumes the suspended process running in buffer ***`BUFID`*** using `SIGCONT`.
`resume`: resumes the suspended process running in the active buffer using `SIGCONT`.

Logging Requirements:
- When signaling the resume, call `log_cmd_signal(LOG_CMD_RESUME, buf_id)`.
- If the selected buffer is invalid, call `log_buf_id_error(buf_id)` instead.
- If no process is running, call `log_cmd_state_conflict(state, buf_id)` instead.
- As a side effect, the child may resume, which leads to additional logs; see 2.6.3.

Assumptions:
- Working or Paused processes can be resumed; a Ready process will produce the error log.
- It is ok to resume a process which is not paused – it is unlikely to have an effect, though.

Implementation Hints:
- A signal can be sent to a child process by using the `kill()` function.
  - Yes, it is still called "kill" if we are using it to resume a process.
- The signal which we want to send is `SIGCONT`.
- This instruction only needs to signal the need to resume to the process.
  - The resume event itself would be handled separately; see 2.6.3.
- A process in the active buffer may need to be resumed, which requires a wait; see 2.6.2.

Example Run (pause and resume instructions):



```
AAH>> list
[AAHLOG] 2 Buffer(s)
[AAHLOG] Buffer 1: Ready
[AAHLOG] Buffer 2: Ready
[AAHLOG] Buffer ID #2 is currently active
AAH>> exec 1 sleep 30
[AAHLOG] Buffer 1, Background Process 176: exec 1 sleep 30 Started
AAH>> list
[AAHLOG] 2 Buffer(s)
[AAHLOG] Buffer 1: Working (Process 176; exec 1 sleep 30)
[AAHLOG] Buffer 2: Ready
[AAHLOG] Buffer ID #2 is currently active
AAH>> pause 1
[AAHLOG] Pause message sent to Buffer ID #1
AAH>> [AAHLOG] Background Process 176: exec 1 sleep 30 Stopped
list
[AAHLOG] 2 Buffer(s)
[AAHLOG] Buffer 1: Paused (Process 176; exec 1 sleep 30)
[AAHLOG] Buffer 2: Ready
[AAHLOG] Buffer ID #2 is currently active
AAH>> resume 1
[AAHLOG] Resume message sent to Buffer ID #1
[AAHLOG] Background Process 176: exec 1 sleep 30 Continued
AAH>> list
[AAHLOG] 2 Buffer(s)
[AAHLOG] Buffer 1: Working (Process 176; exec 1 sleep 30)
[AAHLOG] Buffer 2: Ready
[AAHLOG] Buffer ID #2 is currently active
AAH>> [AAHLOG] Background Process 176: exec 1 sleep 30 Terminated Normally
AAH>>
```

2.6.2 Activating Processes With the "resume" and "active" Built-In Instructions

When we use the `resume` or `active` instructions, it may result in a running process being brought into the foreground. If we `resume` a process which is associated with the active buffer, the process is brought to the foreground. Likewise, if we use the `active` instruction to activate a buffer in the Working state, then it will result in that buffer's process being brought to the foreground.

A process running in the foreground simply means that we will wait for it to finish, instead of letting the child run while we do other things. We can wait using the `waitpid()` call. A process running in the foreground due to a `resume` or `active` call is no different than a process running in the foreground due to a `new` or `exec` call.

Processes which do not run in the foreground will eventually signal that they are done and will be processed via the signal processing mechanism (see 2.6.3 and 2.6.6).

Implementation Hints:

- A `waitpid()` may be interrupted by a signal; if so, we will need to call wait again.

2.6.3 Reaping Child Processes

When we exit a child process or change the state of a process, we will need to reap the process to determine its status. When a process ends, our main shell process will need to reap it at some point - this generally involves a call to `waitpid()`. We still have decisions to make regarding when and where the waiting takes place. We may perform the waiting after making a process active (e.g. if we are executing a process in the active buffer; see 2.6.2). We may also perform waiting whenever we receive a signal from the child (see 2.6.6).

No matter which way we do the reaping, there are several things we know about the process. One, we can use the same call to detect for normal process termination as we do for signaled termination (e.g. `^C`), suspended processes, and resumed process. More importantly, when we control process state (e.g. by suspending and pausing it; section 2.6.1), we only need to send the signal to the process; later, when we have a chance to use `waitpid()`, we can do the real work of updating the buffer/process state in our program. This means that our `waitpid()` calls come with additional logging requirements:

Logging Requirements:

- When the process state is affected, call `log_cmd_state(pid, type, cmd, transition)`.
    - `pid` is the Process ID of the process (not the buffer ID).
    - `type` is `LOG_ACTIVE` for process in the active buffer, or `LOG_BACKGROUND` otherwise.
    - `cmd` is the original user input which ran the process, exactly as entered.
    - `transition` indicates how the process status was affected.
        - Can be: exited; terminated by signal; stopped; continued.
        - Uses `LOG_CANCEL`, `LOG_CANCEL_SIG`, `LOG_PAUSE`, or `LOG_RESUME`.

Assumptions:

- Process state does not need to be updated when first signaled; it can defer until `waitpid()`.

- A `waitpid()` can be used to detect all process state changes of interest, not just termination.

Implementation Hints:

- By default, `waitpid()` blocks until a process finishes; we can make it poll for results instead.
  - If we use the `NOHANG` option, then it exits immediately if no process has finished yet.
  - Additional options can (and should) use to check for stopped/continued processes.
  - Existing macros will help up read the status; see the `man` page for details.
    - `WIFEXITED`, `WIFSTOPPED`, `WIFSIGNALED`, `WIFCONTINUED`, etc.
- A call to `waitpid()` can be interrupted if a signal arrives.
  - If this happens, it may be necessary to restart the wait; be sure to check error codes.
- Multiple processes could end at roughly the same time; if so, wait multiple times in a row.
  - It may make sense to use a loop; keep waiting until no more processes are returned.

2.6.4 Keyboard Interaction

A number of keyboard combinations can trigger signals to be sent to the group of active processes. We will use these keyboard signals to help us control our current active process – otherwise we would have no way to enter commands to be able to stop or pause our active process. Our command shell will use signal handling to detect keyboard inputs, and to forward those signals to the appropriate child process.

For this assignment, we need to support two keyboard combinations:

- `ctrl-C`: A `SIGINT` (value 2) is sent to the active buffer's process to terminate its execution.
- `ctrl-Z`: A `SIGTSTP` (value 20) is sent to the active buffer's process to pause its execution.
- If there is no active buffer process when these combinations are input, they should be ignored (i.e. they should not affect the execution of the command shell or any of the buffers).

Logging Requirements:

- Call `log_ctrl_c()` to report the arrival of `SIGINT` triggered by `^C`.
- Call `log_ctrl_z()` to report the arrival of `SIGTSTP` triggered by `^Z`.
- If there is no active process, the log calls should still be made.

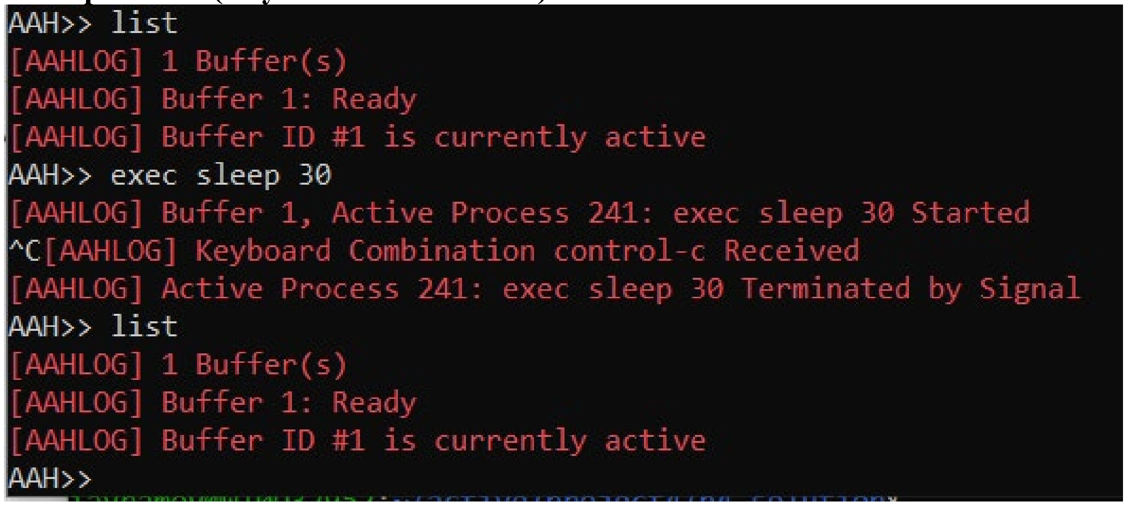
Assumptions:

- Assume that `SIGINT` signals received by the shell process have only been triggered by `^C`.
- Assume that `SIGTSTP` signals received by the shell process have only been triggered by `^Z`.
- If the buffer's process state is not Working, the signal should be ignored.

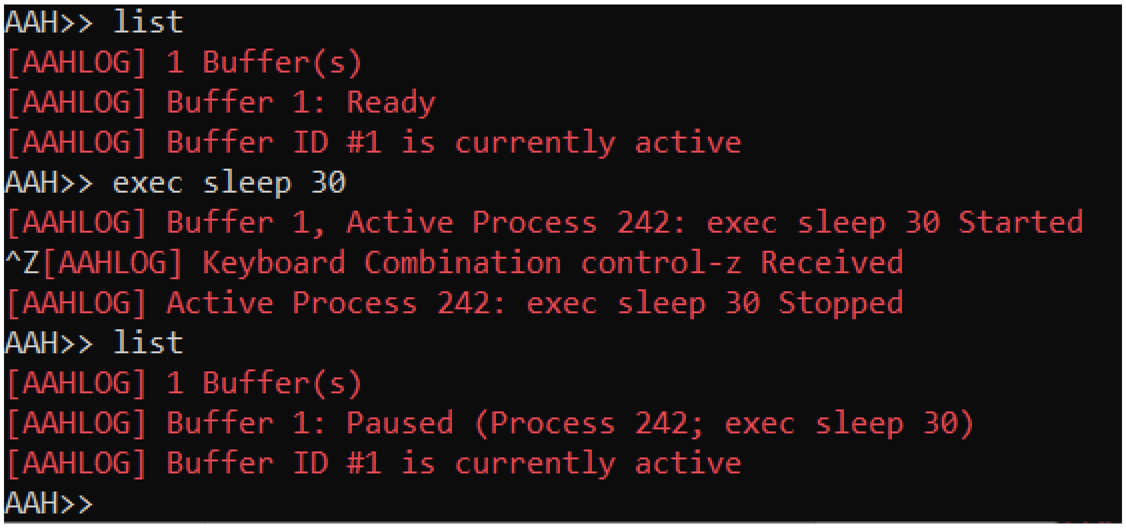
Implementation Hints:

- By default, a keyboard-triggered signal gets sent to *all* processes, including children.
  - We can avoid this by putting different processes in different groups using `setpgid()`.
  - See **Appendix C** for specific instructions.
- Use `sigaction()` to change the default response to a particular signal.
- Use `kill()` to send or forward a signal we have received to another process.

Example Runs (Keyboard ctrl-c / ctrl-z):



```
AAH>> list
[AAHLOG] 1 Buffer(s)
[AAHLOG] Buffer 1: Ready
[AAHLOG] Buffer ID #1 is currently active
AAH>> exec sleep 30
[AAHLOG] Buffer 1, Active Process 241: exec sleep 30 Started
^C[AAHLOG] Keyboard Combination control-c Received
[AAHLOG] Active Process 241: exec sleep 30 Terminated by Signal
AAH>> list
[AAHLOG] 1 Buffer(s)
[AAHLOG] Buffer 1: Ready
[AAHLOG] Buffer ID #1 is currently active
AAH>>
```



```
AAH>> list
[AAHLOG] 1 Buffer(s)
[AAHLOG] Buffer 1: Ready
[AAHLOG] Buffer ID #1 is currently active
AAH>> exec sleep 30
[AAHLOG] Buffer 1, Active Process 242: exec sleep 30 Started
^Z[AAHLOG] Keyboard Combination control-z Received
[AAHLOG] Active Process 242: exec sleep 30 Stopped
AAH>> list
[AAHLOG] 1 Buffer(s)
[AAHLOG] Buffer 1: Paused (Process 242; exec sleep 30)
[AAHLOG] Buffer ID #1 is currently active
AAH>>
```

2.6.5 Signal Concurrency Considerations

We will start to experience the fun and challenge of concurrent programming in this assignment. In particular, if your design includes a global buffer list, be alert that **race conditions** might occur. A race condition is we call it if one process performs an action too soon, or another one takes too long, and the two interfere in unexpected ways. A typical race in this project might happen if our main shell process is in the middle of updating the buffer list when the signal handler is triggered due to a child process completing. Due to the list being in an unstable state, the signal handler fails while trying search the same list. For example, the following sequence is possible if no synchronization is provided:

1. The shell (parent) requests to close buffer 2, just before the running process in buffer 3 (child process) closes;
2. The parent unlinks buffer 2 from the list, and is just about to relink the remaining list.
3. `SIGCHLD` handler is executed, and attempts to find buffer 3 in the list;
4. The handler fails because the list is cut off, and might even get caught seeking through deallocated memory.

We recommend an approach where we protect ourselves against concurrency issues by **blocking** the `SIGCHLD` signal (and other signals that might trigger the updates of the global list) whenever we are about to perform a sensitive operation, and unblocking it when we are done. Any signals which were sent while the signal was blocked are delivered right after the signal is unblocked.

It is a good idea to block signals before the call to `fork()` and unblock them only after the processes information has been updated in the buffer list. In fact, it is a good idea to block signals right before any update to any global data (such as the buffer list), and unblock afterwards. Both blocking and unblocking of signals can be implemented with `sigprocmask()`. If we create functions for blocking and unblocking, it is easy to trigger them when needed.

Implementation Hints:

- Make functions to block and unblock signals on request.
    - Blocking can be implemented using the `sigprocmask()` function.
- Block signals right before any update to a global data structure (e.g. a buffer list).
    - Unblock when done with the update.
    - Includes right before forking.
- Children inherit the blocked set of their parents.
    - They are responsible for unblocking any signals blocked by their parents.
    - In particular, unblock all signals before calling `execl()` or `execv()`.

2.6.6 Signal Handling with SIGCHLD

If you have already implemented the keyboard interrupts described in section 2.6.4, then you already have signal handling built into your code. If you have taken the signal blocking precautions described in 2.6.5, then your code will be reasonably well-prepared to handle concurrent processing scenarios which may arise due to signaling and forked children. Let us talk about one more important signal which you will be responsible for handling.

Whenever a child process changes its state, it automatically (without any direct action on our part) sends out a `SIGCHLD` signal to the parent. This includes when the process terminates, or is killed, or suspends or resumes. In order to find out when our processes end, or change state in some other way, we should set up a signal handler to catch and handle these child events. It is true that we can wait on a process to find out when it ends, but our program will not spend all of its time waiting; it is more reliable to use signals to determine when it ends.

Assumptions:

- The only signals we are responsible for handling are `SIGINT`, `SIGTSTP`, and `SIGCHLD`.

Implementation Hints:

- A child inherits its parent's handlers, so it should first reset the handlers to the default.
    - See **Appendix D** for more details.
- Use blocking (see 2.6.4) outside the handler to protect your sensitive logic from signals.
- Signals which were raised multiple times while blocking are delivered once after unblocking.
    - If `SIGCHLD` is triggered, reap in a loop until you are sure there are no more processes.
    - A `waitpid()` with correct arguments exit immediately, so we can use it to poll.

- Commands such as `cancel` or `pause` do not need to directly quit or suspend the process.
    - If they merely signal the process, the action can take place in the signal handler.
    - See section 2.6.3.
- The textbook uses `signal()`, **which has been deprecated** and replaced by `sigaction()`.
    - Do not use `signal()`. Use `sigaction()` instead.
- Avoid using `stdio.h` functions inside a signal handler.
    - The handler's call may interfere with an interrupted call from the main code.
    - Especially applies to the `errno` of a call; two different calls use the same `errno`.
    - If you need to print debug statements, use `write()` with `STDOUT_FILENO` directly.

3. Getting Started

First, get the starting code (`project4_handout.tar`) from the same place you got this document. Once you un-tar the handout on Zeus (using `tar xvf project4_handout.tar`), you will have the following files in the `p4_handout` directory:

- `textproc.c` – **This is the only file you will be modifying (and submitting).** There are stubs in this file for the functions that will be called by the rest of the framework. Feel free to define more functions if you like but put all of your code in this file!
- `textproc.h` – This has some basic definitions and includes necessary header files.

- `logging.c` – This has a list of provided logging functions that you need to call at the appropriate times.
- `logging.h` – This has the prototypes of logging functions implemented in `logging.c`.

- `parse.c` – This has a list of provided parsing functions that you could use to divide the user command line into useful pieces.
- `parse.h` – This has the prototypes of parsing functions implemented in `parse.c`.

- `util.c` – This includes utility functions which you may call to automate the process of sending text to or from a file descriptor (including text files and pipes)
- `util.h` – This has the prototypes of parsing functions implemented in `util.c`.

- `Makefile` – to build the assignment (and clean up).

- `fox.txt` – a simple text file for convenient testing (of file loading).

- `my_pause.c` – a C program that can be used as local program to load/execute in shell. It will not terminate normally until SIGINT has been received for three times. Feel free to edit the C source code to change its behavior.

- `slow_cooker.c` – a C program that can be used as local program to load/execute in shell. It will slowly count down from 10 to 0 then terminate normally. You can specify a different starting value and/or edit the C source code to change its behavior.

- `my_echo.c` – a C program that can be used as local program to load/execute in shell. It takes an integer argument and use it as the return value / exit status. The default return value / exit status is 0. You can change the value to test exit status with shell.

To get started on this project, read through the provided code in `textproc.c` and the constants and definitions in `textproc.h`, `parse.h`, `util.h`,and `logging.h`. Make sure you understand the input/output of the provided parsing facility, in particular the structure of `argv[]` and `Instruction`.

It is a very good idea to **start early**, and not try to do the whole project in one swoop. Implement feature by feature and test the features as you implement them. The more complex features build off of the simpler features, so it is a good idea to make sure that the earlier parts of the project work reliably before attempting something more involved.

If you are not sure where to begin, the order of topics in this design document serves as a reasonable suggestion for a potential order of implementation: begin with instructions which have no relation to the rest of the project; add buffer management capability; add process execution; add signal handling and refined process control. Spend some time designing the overall layout on paper before starting to code. Once you have this in your design, add the various features in an order which lets you ensure that your code works.

For testing, you may use any of the programs which are included in your handout; any programs you write yourself; or any utilities which are already available on the system. Some programs which may be helpful for debugging include, but are not limited to: `ls`, `grep`, `tr`, `cat`, or `sleep`.

After this, make sure all of the details in each section of this document are met, such as all of the required logging is present (**this is critically important – all grading is done from these log calls)**, that you have all the cases handled, and that all features are incorporated. The more modular you make your design, the easier it will be to debug, test, and extend your code.

4. Submitting & Grading

Submit this assignment electronically on Blackboard. Note that the only file that gets submitted is `textproc.c`**. Make sure to put your G# and name as a commented line in the beginning of your program.**

You can make multiple submissions; but we will test and grade ONLY the latest version that you submit (with the corresponding late penalty, if applicable).

Important: Make sure to submit the correct version of your file on Blackboard! Submitting the correct version late will incur a late penalty; and submitting the correct version 48 hours after the due date will not bring any credit.

Questions about the specification should be directed to the CS 367 Piazza forum.

Your grade will be determined as follows:

- **20 points** - code & comments. Be sure to document your design clearly in your code comments. This score will be based on reading your source code.
- **80 points** – correctness. We will be building your code using the `textproc.c` code you submit along with our code.
  - If you program does not compile, **we cannot grade it**.
  - If your program compiles but does not run, **we cannot grade it.**
  - We will give partial credit for incomplete programs that compile and run.
  - We will use a series of programs to test your solution.

Appendix A: User Input Line

The provided `parse.c` includes a `parse()` function that divides the user command into useful pieces. To use it, provide the full command line as input in `cmd_line`, and previously allocated data structures `inst` and `argv`. The `parse()` function will then populate `inst` and `argv` based on the contents of `cmd_line`.

```
void parse(const char *cmd_line, Instruction *inst, char *argv[]);
```

- `cmd_line`: the line typed in by user **WITHOUT** the ending newline (`\n`).
- `inst`: the pointer to an `Instruction` record which is used to record the additional information extracted from `cmd_line`. The detailed definition of the struct is as below.

```
typedef struct instruction_struct{
        char *instruct;   // the instruction we're running
        int id;           // the buffer ID associated with the command, or 0 if default
        char *file;       // the filename associated with the command
} Instruction;
```

- `argv`: an array of NULL terminated char pointers with the similar format requirement as the one used in `execv()`.
  - `argv[0]` should be the name of the program to be loaded and executed;
  - the remainder of `argv[]` should be the list of arguments used to run the program
  - `argv[]` must have NULL following its last argument member.

Assumptions: You can assume that all user inputs are valid command lines (no need for format checking in your program). You can also assume that the maximum number of characters per command line is `100` and the maximum number of arguments per executable is `25`. Check `textproc.h` for relevant constants defined for you.

Notes of Usage:

- The provided `parse.c` also has supporting functions related to command line parsing, including the initialization/free of `argv` and `inst`. Check `parse.h` for details.
- The provided `textproc.c` already includes necessary steps to make the call to `parse()`.
- The provided `textproc.c` also has a `debug_print_parse()` function you could use to check the return of `parse()`. It's for debugging only.

Appendix B: Useful System Calls

Here we include a list of system calls that perform process control and signal handling. You might find them helpful for this assignment. The system calls are listed in alphabetic order with a short description for each. Make sure you check our textbook, lecture slides, and Linux manual pages on `zeus` to get more details if needed.

- `int dup2(int oldfd, int newfd);`
  - It makes `newfd` to be the copy of `oldfd`; useful for file redirection.
  - Textbook Section **10.9**
  - Manual entry: ***man dup2***

- `int execv(const char *path, char *const argv[]);`
- `int execl(const char *path, const char *arg, ...);`
  - Both are members of exec() family. They load in a new program specified by `path` and replace the current process.
  - Textbook Section **8.4**
  - Manual entry: ***man 3 exec***

- `void exit(int status);`
  - It causes normal process termination.
  - Textbook Section **8.4**
  - Manual entry: ***man 3 exit***

- `pid_t fork(void);`
  - It creates a new process by duplicating the calling process.
  - Textbook Section **8.4**
  - Manual entry: ***man fork***

- `int kill(pid_t pid, int sig);`
  - Used to send signal `sig` to a process or process group with the matching `pid`.
  - Textbook Section **8.5.2**
  - Manual entry: ***man 2 kill***

- `int open(const char *pathname, int flags);`
- `int open(const char *pathname, int flags, mode_t mode);`
  - Opens a file and returns the corresponding file descriptor.
  - Textbook Section **10.3**
  - Manual entry: ***man 2 open***

- `int pipe(int pipefd[2]);`
  - It creates a new pipe, initializing file descriptor at either end of the pipe.
  - Manual entry: ***man pipe***

- `int setpgid(pid_t pid, pid_t pgid);`
  - It sets the group id for the running process.
  - Textbook Section **8.5.2**
  - Manual entry: ***man setpgid***

- `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`
  - Used to change the action taken by a process on receipt of a specific signal.
  - Textbook Section **8.5.5 (pp.775)**
  - Manual entry: ***man sigaction***

- `int sigaddset(sigset_t *set, int signum);`
- `int sigemptyset(sigset_t *set);`
- `int sigfillset(sigset_t *set);`
  - The group of system calls that help to set the mask used in `sigprocmask`.
  - `sigemptyset()` initializes the signal set given by `set` to empty, with all signals excluded from the `set`.
  - `sigfillset()` initializes `set` to full, including all signals.
  - `sigaddset()` adds signal `signum` into `set`.
  - Textbook Section **8.5.4**
  - Manual entry: ***man sigsetops***

- `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);`
  - Used to fetch and/or change the signal mask; useful to block/unblock signals.
  - Textbook Section **8.5.4, 8.5.6**
  - Manual entry: ***man sigprocmask***

- `unsigned int sleep(unsigned int seconds);`
  - It makes the calling process sleep until `seconds` seconds have elapsed or a signal arrives which is not ignored.
  - Note: `sleep` measures the elapsed time by absolute difference between the start time and the current clock time, regardless whether the process has been stopped or running. This means if you suspend and resume it, it will check to see if x seconds have passed since starting.
    - So, if you use sleep 5, then ctrl-Z 1 second into the run, wait 30 seconds, and then resume it, it will see at least 5 seconds have elapsed since it started and will immediately quit, even though it only 'ran' for 1 second.
  - Textbook Section **8.4.4**
  - Manual entry: ***man 3 sleep***

- `pid_t waitpid(pid_t pid, int *status, int options);`
  - Used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed.
  - Textbook Section **8.4.3**
  - Manual entry: ***man waitpid***

- `ssize_t write(int fd, const void *buf, size_t count);`
  - It writes up to `count` bytes from the buffer pointed `buf` to the file referred to by the file descriptor `fd`. The standard output can be referred to as `STDOUT_FILENO`.
  - Textbook Section **10.4**
  - Manual entry: ***man 2 write***

Appendix C: Process Groups

Every process belongs to exactly one process group.

```
#include <unistd.h>

pid_t getpgrp(void);
```

The **`getpgrp()`** function shall return the process group ID of the calling process.

When a parent process creates a child process, the child process inherits the same process group from the parent.

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

- The **`setpgid()`** function shall set process group ID of the calling process.
  - In particular, **`setpgid(0,0)`** will create a new process group with just itself.

In practical terms, the one place that your code must use either of these calls will be a call to **`setpgid(0,0)`**, from the child process, immediately after forking.

References:

- http://man7.org/linux/man-pages/man3/getpgrp.3p.html
- http://man7.org/linux/man-pages/man3/setpgid.3p.html
- Textbook Section 8.5.2 (pp.759)

Appendix D: System call sigaction()

The **sigaction()** system call is used to change the action taken by a process on receipt of a specific signal. The **sigaction()** function has the same basic effect as **signal()** but provides more powerful control. It also has a more reliable behavior across UNIX versions and is recommended to be used to replace **signal()**.

```
#include <signal.h>
int sigaction (int signum, const struct sigaction *action,
          struct sigaction *old_action);
```

- **signum** specifies the signal.
  - It can be any valid signal except **SIGKILL** and **SIGSTOP**.
- For non-**NULL** **action**, a new action for signal **signum** is installed from **action**.
  - It could be the name of the signal handler.
- If **old_action** is non-**NULL**, the previous action is saved in **old_action**.

Example program:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void sigint_handler(int sig) {   /* signal handler for SIGINT */
  write(STDOUT_FILENO, "SIGINT\n", 7);
  exit(0);
}

int main(){
  struct sigaction new;   /* sigaction structures */
  struct sigaction old;

  memset(&new, 0, sizeof(new));
  new.sa_handler = sigint_handler;     /* set the handler */
  sigaction(SIGINT, &new, &old);       /* register the handler for SIGINT */

  int i=0;
  while(i<100000){                     /* this will loop for a while */
        fprintf(stderr, "%d\n", i);    /* break loop by Ctrl-c to trigger SIGINT */
        sleep(1);   i++;
  }
  return 0;
}
```

References:

- https://www.gnu.org/software/libc/manual/html_node/Sigaction-Function-Example.html
- http://man7.org/linux/man-pages/man2/sigaction.2.html