CS 367 Fall 2022
Project #3: Task Controller
Due: Friday, November 11, 2022, 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 task controller called the **KI** Task Controller. Once running, KI maintains a list of several tasks, which can be executed, inspected, or otherwise managed (e.g. by killing or suspending a running task). 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 task controller. 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 task controller also has the responsibility to maintain a list of tasks of interest, and to keep them organized. The user is able to enter programs, then execute them in the *foreground* (wait until the task completes) or *background* (allow the process to run while moving on to other things). The user can also control existing tasks by temporarily suspending them, killing them, or deleting them from the list altogether. Finally, the user will have some additional capabilities, like redirecting the input or output of a process to or from a file, or piping the output of one task to another. The task controller will allow the user to check the exit code of completed tasks. The shell provides some built-in instructions to help manage and list tasks, 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, deleting or listing tasks.
    - 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.
    - KI is able to both wait for a processes to finish, or let them run in the background.
- Perform basic management of tasks, whether they are in ready mode, running, or complete;
- Use file redirects and pipes to read input from or send output to a file or another process.
- 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 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 `taskctl.c`.
- Headers, helper functions and logging functions in `logging.c`, `logging.h`, `parse.c`, `parse.h`, `util.c`, `util.h`, and `taskctl.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 `taskctl.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 Zeus is your expected development platform. 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 Task Number (`task_num`), 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_kitc_prompt()`.
  - The call to `log_kitc_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 `taskctl.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;
- Many built-in instructions expect an additional Task Number argument.
- The `pipe` built-in expects two Task Number arguments.
- Some built-in instruction (`exec` and `bg`) allow file redirection.
- Any instruction which is not a built-in represents a command to be executed.
  - The command 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 `taskctl.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 `taskctl.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 Responsible Coding

Our processes management will require us to fork processes and receive signals. Every process we create uses system resources – sometimes we do not realize how many processes we have left lying around. To compound the situation, whenever we redefine our signal handlers, it may sometimes impact our process's natural ability to shut down cleanly. Hopefully we won't write any unintended fork bombs, but even then, we want to do our best to keep from creating too many unwanted processes. Here are some suggestions which will help.

2.3.1 Killing the Currently Running Process

Often, we may be in the middle of running our program, and either got stuck in a loop or simply want to exit out quickly. If this is the case, **do not use** `ctrl-z`. If you do this, it will not kill the process, it will simply put it to sleep, while still retaining all of its resources in memory. Instead, use `ctrl-c`. If the does `ctrl-c` not work (often because we have redefined the signal handler), try `ctrl-\` instead. The former uses `SIGINT` whereas the latter uses `SIGTERM`. Either will, by default, terminate the program.

2.3.2 Checking for Running Process

When we are back at the `bash` prompt, we may be interested in knowing which processes we are currently running. That way, we can know if we have left behind any residual processes. The easiest way to do that is with the `ps ux` command. This will list out all processes under our name (including `bash` itself).

2.3.3 Killing Residual Processes

If we discover that we have left behind processes, we can use the `pkill` command at the `bash` prompt to kill of processes by name. For example, we would use `pkill taskctl` in order to kill of all of our `taskctl` processes. If `pkill` fails to kill one of our processes (often due to a corrupted signal handler), we can instead use `pkill -KILL taskctl` in order to send a firmer and impossible to override kill message.

2.4 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 KI system can be executed directly without the need to interact with any other parts of the system:

2.4.1 The "help" Built-In Instruction

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

Logging Requirements:

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

Example Run (help instruction):

```
kitc$ help
[KITC-LOG] Instructions:
[KITC-LOG]     COMMAND [ARGS...],
[KITC-LOG]     help, quit, list, purge TASK,
[KITC-LOG]     exec TASK [<INFILE] [>OUTFILE],
[KITC-LOG]     bg TASK [<INFILE] [>OUTFILE],
[KITC-LOG]     pipe TASK1 TASK2,
[KITC-LOG]     kill TASK, suspend TASK, resume TASK
[KITC-LOG]
[KITC-LOG] Brackets denote optional arguments
kitc$
```

2.4.2 The "quit" Built-In Instruction

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

Logging Requirements:

- You must call `log_kitc_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):

```
kitc$ quit
[KITC-LOG] Thanks for using the KI Task Controller! Good-bye!
[cs367@zeus-2 p3_solution]$
```

2.5 Basic Task Management Instructions

This program is a task management system, so it has the ability to maintain several tasks in various states at any time. Each task is assigned a Task Number, which is a positive integer value. In addition, every task has some associated metadata, such as its current state and Process ID, and exit code.

Any existing task is in one of five states: **Ready**, **Running**, **Suspended**, **Finished**, or **Killed**. Before it is run, a task will begin in the **Ready** state. However, running the program will cause it to enter one of the other states.

A task has both a *Task Number* and a *Process ID*, not to be confused. The *Task Number* is a number we have assigned our task in order to make it easier to keep track of. The *Process ID*, on the other hand, is a number assigned by the operating system to uniquely define the child process.

2.5.1 Task Number Assignment

Whenever a new task is created or deleted, we must follow certain rules about Task Numbers:

- Any new task is assigned the first available currently unused Task Number.
    - For example, if current tasks are 1, 3, and 5, the next task will have Task ID 2.
- If there are no tasks, then the next new task will be assigned Task ID 1.
- If a task is purged, the remaining Task Numbers are **not** renumbered.

2.5.2 User Commands and the "list" Built-In Instruction

Any command which is not a built-in instruction should be interpreted as a user command. When a user command is entered into KI, this will create and initialize a new task entry which describes the command. Entering the command will not immediately result in executing the command, but merely creating a task entry.

Logging Requirements:

- Call `log_kitc_task_init(task_num, cmd)` to indicate which Task Number was assigned.
  - The `cmd` is for the **complete** command line string which was provided as input.

Assumptions:

- The Task Number of the new task should be assigned consistent with the rules from 2.5.1.
- When a new command is entered, it is **not** immediately forked or executed.
  - Entering the command will only create a new task entry.
  - We would later use the `exec`, `bg`, or `pipe` built-ins to run the task.
- The newly created tasks will be in the **Ready** state.

Implementation Hints:

- You are required to have the ability to maintain an arbitrary number of tasks. As we have seen in an earlier example, a new task may be numbered in between two existing task, and arbitrary tasks may be deleted. Consider using a data structure which lets you add an unlimited number of elements, and to add or remove arbitrary elements easily.

`list`: lists all of the currently existing tasks.

- Includes the total number of tasks.

Logging Requirements:

- First call `log_kitc_num_tasks(num)` to indicate the current number of tasks.
- Call `log_kitc_task_info(task_num, status, exit_code, pid, cmd)` once per task.
  - Tasks should be listed in order of increasing Task Number.
  - The `status` should be one of the `LOG_STATE_*` constants found in `logging.h`.
  - The `exit_code` should be 0 unless the process has already completed execution.
  - The Process ID (`pid`) should be 0 while in the task is in the **Ready** state.
  - The `cmd` is the **complete** command line of the task.

Assumptions:

- Tasks will begin in the state `LOG_STATE_READY` – but see 2.6.
- The `pid` and `exit_code` can be 0 for now – but see 2.7.2
- The number of tasks begins as 0, but increases if new user commands are entered.

Example Run (new command and list instructions):

```
kitc$ ls -al
[KITC-LOG] Adding Task #1: ls -al (Ready)
kitc$ cal -3
[KITC-LOG] Adding Task #2: cal -3 (Ready)
kitc$ slow_cooker
[KITC-LOG] Adding Task #3: slow_cooker (Ready)
kitc$ list
[KITC-LOG] 3 Task(s)
[KITC-LOG] Task #1: ls -al (Ready)
[KITC-LOG] Task #2: cal -3 (Ready)
[KITC-LOG] Task #3: slow_cooker (Ready)
kitc$
```

2.5.3 The "purge" Built-In Instruction

`purge` ***`TASKNUM`***: removes ***`TASKNUM`*** from the list of tasks.

Logging Requirements:

- On a successful delete, call `log_kitc_purge(task_num)`.
- If the selected task does not exist, call `log_kitc_task_num_error(task_num)` instead.
- If the task is currently busy, call `log_kitc_status_error(task_num, status)` instead.
  - A busy task is a **Running** or **Suspended** task; do not delete the task in this case.
  - The status would be `LOG_STATE_RUNNING` or `LOG_STATE_SUSPENDED`.

Assumptions:

- Only a buffer which is idle (**Ready**, **Finished**, or **Killed**) can be deleted.
- Once purged, the task no longer exists and information about the task need not be retained.

Example Run (purge instruction):

```
kitc$ list
[KITC-LOG] 4 Task(s)
[KITC-LOG] Task #1: ls -al (Ready)
[KITC-LOG] Task #2: cat taskctl.c (Ready)
[KITC-LOG] Task #3: cal -3 (Ready)
[KITC-LOG] Task #4: slow_cooker (Ready)
kitc$ purge 2
[KITC-LOG] Purging Task #2
kitc$ purge 3
[KITC-LOG] Purging Task #3
kitc$ purge 5
[KITC-LOG] Error: Task #5 Not Found in Task List
kitc$ my_echo 10
[KITC-LOG] Adding Task #2: my_echo 10 (Ready)
kitc$ list
[KITC-LOG] 3 Task(s)
[KITC-LOG] Task #1: ls -al (Ready)
[KITC-LOG] Task #2: my_echo 10 (Ready)
[KITC-LOG] Task #4: slow_cooker (Ready)
kitc$ _
```

2.6 Process Execution Instructions

Our system would not be a task controller without the capability to execute tasks. 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 existing task in the list is allowed to have its own corresponding process running as a child. If a task is currently running, it will be in the **Running** (or **Suspended**) state. Before running, it starts in the **Ready** state. After the process completes, it will be in the **Finished** or **Killed** state, depending on how it terminated.

Assumptions:

- Any task will not have more than one associated child process at a time.

2.6.1 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.

2.6.2 The "exec" Built-In Instruction

`exec TASKNUM [< INFILE] [> OUTFILE]`: executes an external command.

- The command is the external command associated with task ***TASKNUM.***
- Runs as a foreground process, meaning that the shell waits for the process to finish.
- If ***INFILE*** is specified, then the child process's input should be redirected from the file.
- If ***OUTFILE*** is specified, then the child process's output should be redirected to the file.
- The process status changes as it runs and eventually completes; see 2.5.
- When the process completes, the task should record the process's exit code; see 2.7.2

Logging Requirements:

- If the selected task does not exist, call `log_kitc_task_num_error(task_num)`.
- If the task is currently busy, call `log_kitc_status_error(task_num, status)` instead.
  - A busy task is a **Running** or **Suspended** task; do not execute the task in this case.
  - The status would be `LOG_STATE_RUNNING` or `LOG_STATE_SUSPENDED`.
- Call `log_kitc_status_change(task_num, pid, LOG_FG, cmd, LOG_START)`.
  - This should be called by the parent after forking a new process.
  - Use the task's full command line as `cmd`.
- If the command cannot be executed (`exec` failed), call `log_kitc_exec_error(cmd)`.
  - Use the task's full command line as `cmd`.
  - Terminate the child with exit code 1.
- If a redirection is performed, call `log_kitc_redir(task_num, redir_type, filename)`.
  - `redir_type` is `LOG_REDIR_IN` or `LOG_REDIR_OUT` depending on which type.
- If a redirect file cannot be opened, call `log_kitc_file_error(task_num, filename)`.
  - Terminate the child with exit code 1.
- When the process terminates, there should be a log message; see 2.7.2.

Assumptions:

- The `exec` instruction can only be successfully run if the task is not currently busy.
- 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).

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.
- If a file is specified, open the file and redirect the child's standard in or out to the file.
  - Use `dup2()` to change the standard input or output the child **after** forking.
  - It is legal to request a redirect of both input **and** output at the same time.
  - Implement `exec` built-in without redirection before adding the redirect feature.
- Use `wait()` or `waitpid()` to wait for the child process to finish.
  - Either way, signals are relevant to process completion; see 2.7.5
- Yes, running a completed task would result in re-running the command.

Example Run (exec instruction):

```
kitc$ ls -a
[KITC-LOG] Adding Task #1: ls -a (Ready)
kitc$ daffodil soup
[KITC-LOG] Adding Task #2: daffodil soup (Ready)
kitc$ grep the
[KITC-LOG] Adding Task #3: grep the (Ready)
kitc$ exec 1
[KITC-LOG] Foreground Process 1805600 (Task 1): ls -a (Started)
.         fox.txt    logging.o  my_pause    parse.h      slow_cooker.c  taskctl.h  util.h
..        logging.c  my_echo    my_pause.c  parse.o      taskctl        taskctl.o  util.o
Makefile  logging.h  my_echo.c  parse.c     slow_cooker  taskctl.c      util.c
[KITC-LOG] Foreground Process 1805600 (Task 1): ls -a (Terminated Normally)
kitc$ exec 2
[KITC-LOG] Foreground Process 1805613 (Task 2): daffodil soup (Started)
[KITC-LOG] Error: daffodil soup: Command Cannot Load
[KITC-LOG] Foreground Process 1805613 (Task 2): daffodil soup (Terminated Normally)
kitc$ exec 3 < fox.txt
[KITC-LOG] Redirecting input from fox.txt for Task #3
[KITC-LOG] Foreground Process 1805617 (Task 3): grep the (Started)
the quick brown
the friendly dog
[KITC-LOG] Foreground Process 1805617 (Task 3): grep the (Terminated Normally)
kitc$ list
[KITC-LOG] 3 Task(s)
[KITC-LOG] Task #1: ls -a (PID 1805600; Finished; exit code 0)
[KITC-LOG] Task #2: daffodil soup (PID 1805613; Finished; exit code 1)
[KITC-LOG] Task #3: grep the (PID 1805617; Finished; exit code 0)
kitc$
```

2.6.3 The "bg" Built-In Instruction

`bg TASKNUM [< INFILE] [> OUTFILE]`: executes an external command.

- Runs as a background process, meaning that the shell does not wait for the process to finish.
  - Signal handling will be necessary to detect process completion; see 2.7.5.
- The command is the external command associated with task ***`TASKNUM`***.
- If ***`INFILE`*** is specified, then the child process's input should be redirected from the file.
- If ***`OUTFILE`*** is specified, then the child process's output should be redirected to the file.
- The process status changes as it runs and eventually completes; see 2.5.
- When the process completes, the task should record the process's exit code; see 2.7.2.

Logging Requirements:

- This logging requirements are nearly identical to those of the **`run`** command.
- The exception: **`log_kitc_status_change(task_num, pid, LOG_BG, cmd, LOG_START)`**.
  - **`LOG_BG`** instead of **`LOG_FG`**.
- With background execution, process termination is logged only when the process completes.
  - This is not necessarily before returning to the shell prompt.

Implementation Hints:

- The **`bg`** built-in is like the **`run`** built-in, but without waiting for process termination.
- It's normal for a background process to print output after returning to the prompt.

Example Run (bg instruction):

```
kitc$ slow_cooker
[KITC-LOG] Adding Task #1: slow_cooker (Ready)
kitc$ bg 1
[KITC-LOG] Background Process 1805696 (Task 1): slow_cooker (Started)
kitc$ slow_cooker count down: 10 ...
slow_cooker count down: 9 ...
slow_cooker count down: 8 ...
list
[KITC-LOG] 1 Task(s)
[KITC-LOG] Task #1: slow_cooker (PID 1805696; Running)
kitc$ slow_cooker count down: 7 ...
slow_cooker count down: 6 ...
slow_cooker count down: 5 ...
slow_cooker count down: 4 ...
slow_cooker count down: 3 ...
slow_cooker count down: 2 ...
slow_cooker count down: 1 ...
slow_cooker count down: 0 ...
[KITC-LOG] Background Process 1805696 (Task 1): slow_cooker (Terminated Normally)
kitc$ list
[KITC-LOG] 1 Task(s)
[KITC-LOG] Task #1: slow_cooker (PID 1805696; Finished; exit code 0)
kitc$ _
```

2.6.4 Pipes

When we want to send output from one process to another process, we can use a **pipe**. Similar to file redirection, output of one process can be redirected to a pipe, and input to another process can be redirected from a pipe. If one process sends to a pipe, and a second process reads from the same pipe, this lets the first process send data to the second process.

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 children, then both children will inherit the pipe and be able to use it to send data from one to the other. Consider this scenario:

- Parent process **P** has two children, **A** and **B**.
- **A** wants to send data to **B**.
- **P** creates a pipe before forking **A** or **B**; thus, both **A** and **B** gain access to the pipe.
- **A** redirects its output to the *write-end* of the pipe and closes its *read-end*.
- **B** redirects its input from the *read-end* of the pipe and closes its *write-end*.
- **P** closes both its *read-end* and *write-end*, because it is not directly using either.
- Now, any output from **A** gets sent as input to **B**.

It is vitally important to close all unused pipe ends as described above. Without doing so, child **B** might never terminate, because it has no way of knowing when it has come to the end of its input (the unused pipe ends could *potentially* still be used to send input to **B**).

2.6.5 The "pipe" Built-In Instruction

`pipe` ***`TASKNUM1 TASKNUM2`***: executes two external commands connected by a pipe.

- Creates a pipe to allow output to be redirected from ***`TASKNUM1`*** and to ***`TASKNUM2`***.
- Executes ***`TASKNUM1`*** as a background process (see `bg`).
  - Signal handling will be necessary to detect process completion; see 2.7.5.
- Executes ***`TASKNUM2`*** as the foreground process (see `exec`).
- The processes' statuses change as they runs and eventually complete; see 2.5.
- When either process completes, the task should record the process's exit code; see 2.7.2.

Logging Requirements:

- If the two Task Numbers are identical, call `log_kitc_pipe_error(task_num)`.
  - Do not create a pipe or run either process.
- If either task cannot be executed due to invalid task number or incompatible state:
  - See logging requirements for the `exec` and `bg` built-ins.
  - Do not create a pipe or run either process.
- If pipe creation fails, call `log_kitc_file_error(task_num, LOG_FILE_PIPE)`.
  - `task_num` is the Task Number of the first task, ***`TASKNUM1`***.
  - Do not run either process.
- Call `log_kitc_pipe(task_num1, task_num2)` on success.
- See the logging requirements for the `exec` and `bg` commands for each process.
- With background execution, process termination is logged only when the process completes.
  - This is not necessarily before returning to the shell prompt.

Assumptions:
- Assume that neither process will use input or output file redirection.

Implementation Hints:
- The code used by the **`pipe`** built-in overlaps the **`bg`** and **`exec`** built-ins considerably.
- Do not forget to close the unused ends of the pipe as described in 2.6.4.

Example Run (pipe instruction):

```
kitc$ ls -l
[KITC-LOG] Adding Task #1: ls -l (Ready)
kitc$ grep util
[KITC-LOG] Adding Task #2: grep util (Ready)
kitc$ pipe 1 2
[KITC-LOG] Opening a pipe from Task #1 to Task #2
[KITC-LOG] Background Process 1806077 (Task 1): ls -l (Started)
[KITC-LOG] Foreground Process 1806078 (Task 2): grep util (Started)
-rw-r--r--. 1 cs367 itefacstaff  1314 Oct 11 02:17 util.c
-rw-r--r--. 1 cs367 itefacstaff   523 Oct 11 02:17 util.h
-rw-rw-r--. 1 cs367 itefacstaff  8232 Oct 13 00:55 util.o
[KITC-LOG] Background Process 1806077 (Task 1): ls -l (Terminated Normally)
[KITC-LOG] Foreground Process 1806078 (Task 2): grep util (Terminated Normally)
kitc$
```

2.7 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 kill a process before it is complete. Or we may want to suspend a running process and resume it later. This especially applies if we attempt using **`^C`** or **`^Z`** on a foreground process, from the terminal.

2.7.1 The "kill", "suspend", and "resume" Built-In Instructions

`kill` ***`TASKNUM`***: terminates the process associated with Task ***`TASKNUM`*** using **`SIGINT`**.

Logging Requirements:
- If the selected task does not exist, call **`log_kitc_task_num_error(buf_id)`**.
- If the task is currently idle, call **`log_kitc_status_error(task_num, status)`** instead.
  - An idle task is a **Ready**, **Finished** or **Killed** task; do not kill such a task.
- When signaling the kill, call **`log_kitc_sig_sent(LOG_CMD_KILL, task_num, pid)`**.
- As a side effect, the child may exit, which leads to additional logs; see 2.7.2.

Assumptions:
- **Running** or **Suspended** processes can be killed; others 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 kill to the process.
  - Termination and clean-up would be handled separately; see 2.7.2.

Example Run (kill instruction):

```
kitc$ sleep 1000
[KITC-LOG] Adding Task #1: sleep 1000 (Ready)
kitc$ bg 1
[KITC-LOG] Background Process 1806602 (Task 1): sleep 1000 (Started)
kitc$ list
[KITC-LOG] 1 Task(s)
[KITC-LOG] Task #1: sleep 1000 (PID 1806602; Running)
kitc$ kill 1
[KITC-LOG] Kill message sent to Task #1 (PID 1806602)
kitc$ [KITC-LOG] Background Process 1806602 (Task 1): sleep 1000 (Terminated by Signal)
kitc$ list
[KITC-LOG] 1 Task(s)
[KITC-LOG] Task #1: sleep 1000 (PID 1806602; Killed; exit code 0)
kitc$
```

`suspend` ***`TASKNUM`***: suspends the process associated with Task ***`TASKNUM`*** using `SIGTSTP`.

Logging Requirements:

- If the selected task does not exist, call `log_kitc_task_num_error(buf_id)`.
- If the task is currently idle, call `log_kitc_status_error(task_num, status)` instead.
  - An idle task is a **Ready**, **Finished** or **Killed** task; do not suspend such a task.
- When signaling, call `log_kitc_sig_sent(LOG_CMD_SUSPEND, task_num, pid)`.
- As a side effect, the child may be suspended, which leads to additional logs; see 2.7.2.

Assumptions:

- **Running** or **Suspended** processes can be suspended; others 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 suspending.
- The signal which we want to send is `SIGTSTP`.
- This instruction only needs to signal the need to suspend to the process.
  - The suspension event would be handled separately; see 2.7.2.

`resume` ***`BUFID`***: resumes the suspended process associated with Task ***`TASKNUM`*** using `SIGCONT`.

Logging Requirements:

- If the selected task does not exist, call `log_kitc_task_num_error(buf_id)`.
- If the task is currently idle, call `log_kitc_status_error(task_num, status)` instead.
  - An idle task is a **Ready**, **Finished** or **Killed** task; do not resume such a task.
- When signaling, call `log_kitc_sig_sent(LOG_CMD_RESUME, task_num, pid)`.
- As a side effect, the child may be resumed, which leads to additional logs; see 2.7.2.

Assumptions:

- **Running** or **Suspended** processes can be resumed; others will produce the error log.
- It is ok to resume a process which is not suspended – it is unlikely to have an effect, though.
- A resumed process should become a foreground process when it is eventually resumed.

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 2.7.2.

Example Run (suspend and resume instructions):

```
kitc$ slow_cooker 5
[KITC-LOG] Adding Task #1: slow_cooker 5 (Ready)
kitc$ bg 1
[KITC-LOG] Background Process 3480932 (Task 1): slow_cooker 5 (Started)
kitc$ slow_cooker count down: 5 ...
slow_cooker count down: 4 ...
suspend 1
[KITC-LOG] Suspend message sent to Task #1 (PID 3480932)
kitc$ [KITC-LOG] Background Process 3480932 (Task 1): slow_cooker 5 (Stopped)
kitc$ list
[KITC-LOG] 1 Task(s)
[KITC-LOG] Task #1: slow_cooker 5 (PID 3480932; Suspended)
kitc$ resume 1
[KITC-LOG] Resume message sent to Task #1 (PID 3480932)
kitc$ [KITC-LOG] Foreground Process 3480932 (Task 1): slow_cooker 5 (Continued)
slow_cooker count down: 3 ...
slow_cooker count down: 2 ...
slow_cooker count down: 1 ...
slow_cooker count down: 0 ...
[KITC-LOG] Foreground Process 3480932 (Task 1): slow_cooker 5 (Terminated Normally)
kitc$
```

2.7.2 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 running a foreground process (see 2.6.2). We may also perform waiting whenever we receive a signal from the child (see 2.7.5).

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 resuming it; section 2.7.1), 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:

- On state change: **`log_kitc_status_change(task_num, pid, type, cmd, transition)`**.
  - Make this call whenever a process's state is affected.
  - **`pid`** is the Process ID of the process (not the Task ID).
  - **`type`** is **`LOG_FG`** for foreground processes, or **`LOG_BG`** otherwise.
    - A resumed process should automatically choose **`LOG_FG`**.
  - **`cmd`** is the full user command associated with the task.
  - **`transition`** indicates how the process status was affected.
    - Can be: exited; terminated by signal; stopped; continued.
    - Uses **`LOG_TERM`**, **`LOG_TERM_SIG`**, **`LOG_SUSPEND`**, 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.
  - Requires certain options to be set; check the `man` page for details.
- A resumed process will automatically become a foreground process.

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) be used 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.
- A call to `waitpid()` allows us to record a process's exit code.
  - This is the best way to be sure to record exit codes for a terminated process!

2.7.3 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 foreground process – otherwise we would have no way to enter commands to be able to stop or suspend 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 foreground process to terminate its execution.
- `ctrl-Z`: A `SIGTSTP` (value 20) is sent to the foreground process to suspend its execution.
- If there is no foreground 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 tasks).

Logging Requirements:

- Call `log_kitc_ctrl_c()` to report the arrival of `SIGINT` triggered by `^C`.
- Call `log_kitc_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 Task's state is not **Running**, 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):

```
kitc$ sleep 1000
[KITC-LOG] Adding Task #1: sleep 1000 (Ready)
kitc$ exec 1
[KITC-LOG] Foreground Process 1806313 (Task 1): sleep 1000 (Started)
^C[KITC-LOG] Keyboard Combination control-c Received
[KITC-LOG] Foreground Process 1806313 (Task 1): sleep 1000 (Terminated by Signal)
kitc$ list
[KITC-LOG] 1 Task(s)
[KITC-LOG] Task #1: sleep 1000 (PID 1806313; Killed; exit code 0)
kitc$
```

```
kitc$ sleep 1000
[KITC-LOG] Adding Task #1: sleep 1000 (Ready)
kitc$ exec 1
[KITC-LOG] Foreground Process 1806399 (Task 1): sleep 1000 (Started)
^Z[KITC-LOG] Keyboard Combination control-z Received
[KITC-LOG] Foreground Process 1806399 (Task 1): sleep 1000 (Stopped)
kitc$ list
[KITC-LOG] 1 Task(s)
[KITC-LOG] Task #1: sleep 1000 (PID 1806399; Suspended)
kitc$
```

2.7.4 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 what 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 task 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 delete Task 2, just before the running process in Task 3 (child process) finishes;
2. The parent unlinks Task 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 task list. In fact, it is a good idea to block signals right before any update to any global data (such as the task 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 task 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()`.

2.7.5 Signal Handling with SIGCHLD

If you have already implemented the keyboard interrupts described in section 2.7.3, then you already have signal handling built into your code. If you have taken the signal blocking precautions described in 2.7.4, 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 2.7.4) 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 `kill` or `suspend` 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 2.7.1.
- 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 (`p3_handout.tar`) from the same place you got this document. Once you un-tar the handout on Zeus (using `tar xvf p3_handout.tar`), you will have the following files in the `p3_handout` directory:

- `taskctl.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!
- `taskctl.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 copying and clearing strings and argument lists.
- `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 redirection).

- `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 three times. Feel free to edit the C source code to change its behavior. This program helps to test signal handling.

- `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. This program will help you test multiple concurrent processes.

- `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. This program will help you test whether your program correctly handles exit codes.

To get started on this project, read through the provided code in `taskctl.c` and the constants and definitions in `taskctl.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`. You may use provided helper functions declared in the header files for your own purposes.

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 task 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`, `cal`, `cat`, `sort`, 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.

When it comes to debugging, `gdb` works just as well for a program which forks as it does with any other program. However, when a process forks, the debugger can only follow one of the two branches which are created. By default, `gdb` will only follow the **parent** process. However, we can choose to follow the **child** process instead. In order to change the follow mode, we would enter the command "`set follow-fork-mode child`" at the `gdb` prompt.

We always encourage the use of `valgrind` to check for memory leaks, because memory leaks are often the sign of a deeper problem, and it is a good programming habit to eliminate them. However, **we will not be testing for memory** leaks when we grade your submissions. This is because memory leaks may be tricky to track down in a program with multiple concurrent processes, and we believe that your efforts may be better spent elsewhere. Be forewarned that even if you have eliminated all memory leaks in your program, `valgrind` may still report leaks due to child processes which have terminated early.

4. Submitting & Grading

Submit this assignment electronically on Blackboard. Note that the only file that gets submitted is `taskctl.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 `taskctl.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.

5. Tips & Tricks

Global variables are allowed.

Normally, we hate global variables. They are easy to confuse and abuse, promote inflexible code, and lead to unexpected errors because they are so easy to misuse. Signal handlers are one of the rare exceptions – signal handling is inherently global, and global variables are one of the few ways to share data with a signal handler. The signal handlers essentially force us to use global variables. Thus, for this project, reasonable use of global variables is allowed. It is a good idea to keep the use to a minimum – only where necessary to share data with a signal handler.

Careless use of strcat() leads to errors.

If we try to write C code which says `strcat("/usr/bin", argv[0])`, it will typically produce an error. Why? For one, we would prefer to use `strncat` in favor of `strcat`, because the latter is not memory-safe. More notably, C does not automatically resize string arrays for us. If we use the hard-coded string `"/usr/bin"`, it has **only allocated space for that string**! Anything we append beyond that point is overwriting unknown data.

Instead, we should make sure that we create a buffer large enough to store the complete string, and we should use `strncat` to make sure that we do not overrun the buffer.

Make a copy of any strings that you plan to use.

If we link to the original string instead of duplicating it, we may have a problem if the original changes. This is especially true if we create a pointer which points directly to the `cmd` variable in `main()`. The moment that `cmd` gets updated (which happens every time we prompt for input), the contents are reset. If we did not make a copy, we would have lost our data. It also applies to `argv` or anything from `inst`.

Have a look at `util.h` for functions which will help with duplicating strings and data structures.

Trying to terminate a process which is asleep has no effect.

It is normal behavior that sending a `SIGINT` to a sleeping process will not kill it. If we want to terminate the process, wake it up first before sending the signal.

Some of the quirks of concurrency are normal.

When running commands in the background, we may see some outputs which may intuitively seem unusual, but are in fact quite normal. For example, when running a background process, we may see log messages appear in a different order than expected due to the order that events are processed. This is not necessarily a problem.

Another common result is for the command prompt to "disappear" when running a background process, as the program apparently gets stuck in an infinite loop. Fortunately, appearances can often be deceiving. What often happens is that the command prompt appears early on, then gets buried by output as the background process continues to run. Look higher up on the terminal to see if the prompt is there. Try to press enter to get a new prompt. If this works, everything is ok.

Finally, we may sometimes see a double-prompt (a second prompt appearing due to no particular action on our part). This may happen automatically if the system is waiting at the command prompt, and suddenly gets interrupted by a signal. A re-prompt in this situation is considered normal behavior.

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 num;           // the Task Number associated with the instruction,
                           // or 0 if none/default
        int num2;          // the 2nd Task Number associated with the instruction,
                           // or 0 if none/default
        char *infile;      // the input filename associated with the instruction
        char *outfile;     // the output filename associated with the instruction
} 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 **taskctl.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 **inst**. Check **parse.h** for details.
- Similar code to allocate or free **argv** can be found in **util.h**.
- The provided **taskctl.c** already includes necessary steps to make the call to **parse()**.
- The provided **taskctl.c** also has a **debug_print_parse()** function you could use to check the return of **parse()**. It's for debugging only.
- The debug messages printed by the program are fully optional, and can be disabled by changing the **debug** define to 0 instead of 1.

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