CS 367 Project 3 – Spring 2023:
Task Manager
Due: Friday, April 14th, 11:59pm

**This is to be an individual effort. <u>No partners</u>. No Internet code/collaboration.
<u>Protect your code</u> from anyone accessing it. <u>Do not post code on public</u> repositories.
No late work allowed after 48 hours; each day late automatically uses one of your tokens.**

Core Topics: Processes, Signals, Unix I/O, C Programming Design (Helper Functions)

1 Project Overview

For this assignment, you are going to use C to implement a Task Manager system called the **ANAV**
Task Manager. Once running, **ANAV** 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-I/O 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 **shell** (eg. the **bash** shell you enter commands into in Linux) receives instructions from
the user in a terminal. In this project, the shell is the interface to our **Task Manager**. 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 to create a new child process and execute the program
in the context of the child.

The Task Manager 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 to **pipe** the output of one task to another. The Task Manager 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 to execute and control running commands.

2.1 Features for ANAV

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.
  - **ANAV** can either wait for a process 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 4 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.

3 Implementation Responsibilities

Your project handout consists of several files:
- The starting template code in **src/anav.c**.
- Helper and logging functions are in the **src/** directory: **logging.c**, **parse.c**, and **util.c**.
- Headers are in the **inc/** directory: **logging.h**, **parse.h**, **util.h**, and **anav.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 **src/anav.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.

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.

3.1 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, which enables us to keep track of the activities of our shell.

> **The generated output from the log calls are what we use 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.

3.2 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_anav_prompt()**.
    - The call to **log_anav_prompt()** is already present in the starting template code.

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 **src/anav.c** has already included the code which calls **parse()** inside of **main()**. Feel free to use the provided **parse()** as is, or to implement your own parsing facility.

> **Appendix A has a detailed description of the input, output, and examples of using 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;
- 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 `anav.h` for these 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 `src/anav.c` leaves the design and implementation up to you as an **open-ended** project for you to solve. You are encouraged to write **many** functions and helper functions as well and you may add additional code in to `main()` as needed.

3.2.1 Basic Shell Instructions

A typical shell program supports a set of **built-in instructions** (internal functions that you design and implement in the shell itself). If a built-in instruction is received, the shell process must execute that directly **without** forking any additional process.

These following built-in instructions supported by our ANAV system can be executed directly without the need to interact with any other parts of the system:

3.2.1.1 The "help" Built-In Instruction

`help`: when called, your shell should call the appropriate logging function to print a short description of the system, including a list of built-in instructions and their usage.

Logging Requirements:
- You must call `log_anav_help()` to print out the predefined information.

Example Run (help instruction):

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

3.2.1.2 The "quit" Built-In Instruction

`quit`: when called, your shell will exit normally on its own (this is built-in to main already).

Logging Requirements:
- You must add a call **log_anav_quit()** to print out the predefined information.
- The shell will exit normally on its own, you just need to add this log message.

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):
```
ANAV$ quit
[ANAV-LOG] Thanks for using the ANAV Task Manager! Good-bye!
[username@zeus-2 handout]$
```

3.3 Basic Task Management Instructions

This program is a task management system, so it has the ability to maintain several **tasks** (processes in execution) in various states at any time. Each task is assigned a **task ID**, which is a positive integer value. In addition, every task must also track the **PID** of the process that is currently running and the **exit code** if that process has terminated. Each task must also store a copy of the **command** for the process that is running, so we can print out to the user.

```
ANAV$ list
[ANAV-LOG] 5 Task(s)
[ANAV-LOG] Task #1: slow_cooker 10 (PID 997892; Suspended)
[ANAV-LOG] Task #2: cal (PID 997871; Finished; exit code 0)
[ANAV-LOG] Task #3: ls -a -l (Ready)
[ANAV-LOG] Task #4: slow_cooker 50 (PID 998896; Killed; exit code 0)
[ANAV-LOG] Task #5: slow_cooker 42 (PID 998900; Running)
ANAV$
```

> Any task is in one of five states: **Ready**, **Running**, **Suspended**, **Finished**, or **Killed**.

All tasks will start in the **Ready** state. However, running the program will cause it to change states as appropriate to indicate the current state of that task.

Each task has both a **Task Number** and a **Process ID (PID)**, so take care to keep from confusing them. The **Task Number** is the number we assign to our task in order to allow us to keep track of it in a list of our tasks in ANAV. This is an internal number we use to quickly identify which task we wish to perform an operation on.

The **Process ID (PID)**, on the other hand, is a number assigned by the operating system to uniquely define and identify each process that is running. All processes on the operating system are guaranteed to have a unique PID. We must know the PID for each task to manage the process.

3.3.1 Task Number Assignment

Whenever a new task is created or deleted, we must follow certain rules about **Task Numbers**:
- Any new task is assigned a Task Number which is one greater than the largest Task Number.
  - For example, if current Task Numbers are 1, 3, and 5, the next Task Number will be 6.
- If there are no tasks, then the next new tasks will be assigned Task Number 1.
- If a Task is deleted, the remaining Task Numbers are not renumbered.

3.3.2 States and Logging

There are five states that each task can be in: **Ready**, **Running**, **Suspended**, **Finished**, or **Killed**. When you use certain Logging functions, you will need to pass in the current state for a task.

To make this easier, we have provided you with five #defined constants that are already defined. These defined constants are all integer values (ranging from 0 - 4), which you can use to pass in the correct state to these logging functions:

Ready State:	`LOG_STATE_READY`
Running State:	`LOG_STATE_RUNNING`
Suspended State:	`LOG_STATE_SUSPENDED`
Finished State:	`LOG_STATE_FINISHED`
Killed State:	`LOG_STATE_KILLED`

You are free to track the state of your task in any way you like, but when you need to pass the state into a logging function, you will need to use one of these five pre-defined constants.

3.3.3 Running a User Command to create a new Task

Any command which is not a built-in instruction (eg. **quit**, **help**, etc.) should be interpreted as a user command. When a user command is entered into ANAV, this will create and initialize a new task entry which describes the command.

> Entering the command **will not execute the command**, but merely creates a task entry.

Once the task is created with the information from the command that was entered (eg. Instruction's **instruct** member, arguments (**argv**), and original command line entry (cmd)), then it will be in the **Ready** State, ready to be executed. Executing the task to run the process is handled by a separate built-in instruction, **exec**. Entering the command simply creates a Task.

Implement your own data structure for the Task and to maintain a list of any number of tasks.

Logging Requirements (when a Command is entered and a new Task is created):
- Call **log_anav_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.
    - As an example, if the user enters **ls -a -l** as the command, **cmd** will be "ls -a -l"
  - PIDs and Exit Codes are only relevant once we **exec** the task; initialize these to 0.

Assumptions:
- The Task Number of the new task should be assigned consistent with the rules from 3.3.1.
- 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 arbitrary tasks may be deleted (purged), consider using a data structure which lets you add an unlimited number of tasks, and to add or remove arbitrary elements easily.
- The command and its arguments are parsed for you by the code we have already written.
  - Check **Appendix A** for a detailed description of the input, output, and examples of the provided parse() functions.
    - The parse() function gives you a filled-in Instruction struct and the argv array.
  - You are not required to use our parse() function and you may modify main to call your own code to parse the user input, if you wish, but the given starter code does all of this string-processing work for you.
  - **Note:** The end of the main loop in main will free the Instruction, argv, and cmd!
    - If you want your tasks to keep these, copy them into your data structures.
- Check out **util.c**, you may use any of the functions you find in there, if you find them useful.

```
ANAV$ ls -al
[ANAV-LOG] Adding Task #1: ls -al (Ready)
ANAV$ cat fox.txt
[ANAV-LOG] Adding Task #2: cat fox.txt (Ready)
ANAV$ cal -3
[ANAV-LOG] Adding Task #3: cal -3 (Ready)
ANAV$ slow_cooker
[ANAV-LOG] Adding Task #4: slow_cooker (Ready)
ANAV$
```

3.3.4 Built-In Instructions

ANAV supports a large number of built-in instructions. These are internal features you will be designing and writing in ANAV, unlike commands which are programs that your shell will run.

You will need to support the following built-in instructions:
> **help, quit, list, purge, exec, bg, pipe, kill, suspend,** and **resume**

The details for each of these are covered throughout this document.

list

This built-in command lists all the currently existing tasks.

- Includes the total number of tasks.

Logging Requirements:
- First call `log_anav_num_tasks(num)` to indicate the number of tasks.
- Call `log_anav_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. (See 3.3.2)
  - 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. (See 3.3.3 for command info)

Assumptions:
- Tasks will begin in the **Ready State** (see 3.3.2 for states)
- `PID` and `Exit Code` are 0 for now (see 3.4 for how this changes during process execution).
- The number of tasks begins as 0 but changes as new tasks are opened or purged.

Example Run (Command Entries and the list built-in instruction):

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

3.3.5 The "purge" Built-In Instruction

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

Logging Requirements:
- On a successful delete, call `log_anav_purge(task_num)`.
- If the selected task does not exist, call `log_anav_task_num_error(task_num)` instead.
- If the task is currently busy, call `log_anav_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 task 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 kept.

Example Run (active instruction):

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

3.4 Process Execution Instructions

Our system would not be a Task Manager 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 (eg. **execv()**) 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.

3.4.1 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, which will not have any path to begin with.

For example, if the user enters the command "**ls -al**", we need to try both "**./**" and "**/usr/bin/**" as the path argument to **execv**, **in that order**. We would first try to execute "**./ls**", and failing in that, we then execute the 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** needs to be modified from the 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.

3.4.2 The "exec" Built-In Instruction

This built-in command is how we run a non-interactive program (eg. **ls**, **wc**, **cat**, **cal**, **slow_cooker**). This will involve creating a child process to run the program in. You will need to execute the program with arguments, that are stored in your Task structure. You may have to try multiple paths (see 3.4.1) when executing the Program.

3.4.2.1 exec *TASKNUM [< INFILE] [> OUTFILE]*: *executes the 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 3.3.
    - o   In this case, it will be set to **Running**.
- When the process completes, the task should record the process's exit code; see 3.5.2.

Logging Requirements:
- If the selected task does not exist, call **log_anav_task_num_error(task_num)**.
- If the task is currently busy, call **log_anav_status_error(task_num, status)** instead.
    - o   A busy task is a **Running** or **Suspended** task; do not execute the task in this case.
    - o   The status would be **LOG_STATE_RUNNING** or **LOG_STATE_SUSPENDED**.
- Call **log_anav_status_change(task_num, pid, LOG_FG, cmd, LOG_START)**.
    - o   This should be called by the parent after forking a new process.
    - o   Use the task's full command line as **cmd**.
- If the command cannot be executed (**exec** failed), call **log_anav_exec_error(cmd)**.
    - o   Use the task's full command line as **cmd**.
    - o   Terminate the child with exit code 1.
- If a redirection is performed, call **log_anav_redir(task_num, redir_type, filename)**.
    - o   **redir_type** is **LOG_REDIR_IN** or **LOG_REDIR_OUT** depending on which type.
- If a redirect file cannot be opened, call **log_anav_file_error(task_num, filename)**.
    - o   Terminate the child with exit code 1.
- When the process terminates, there should be a log message; see 3.5.2.

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 will be added by your code).
    - o   eg. The user enters **ls -a**, which has an instruction (argv[0]) that will be **"ls"**
        - ▪   You will first try to run the instruction with **execv** using a path of **"./ls"**
        - ▪   If that fails, you will then try using a path of **"/usr/bin/ls"**

Implementation Hints:

- We can use **fork()** to create a new child process.
    - You will need to give the child a new Process Group ID (PGID) after forking.
    - See **Appendix C** for information about how to do this.
- 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 3.5.5.
- **Yes, running a completed task would result in re-running the command.**

Example Run (exec instruction):

```
ANAV$ ls -a
[ANAV-LOG] Adding Task #1: ls -a (Ready)
ANAV$ cyberpunk2077_multiplayer
[ANAV-LOG] Adding Task #2: cyberpunk2077_multiplayer (Ready)
ANAV$ grep the
[ANAV-LOG] Adding Task #3: grep the (Ready)
ANAV$ exec 1
[ANAV-LOG] Foreground Process 1789862 (Task 1): ls -a (Started)
.  ..  anav  fox.txt  inc  Makefile  my_echo  my_pause   obj  slow_cooker  src
[ANAV-LOG] Foreground Process 1789862 (Task 1): ls -a (Terminated Normally)
ANAV$ exec 2
[ANAV-LOG] Foreground Process 1789865 (Task 2): cyberpunk2077_multiplayer
(Started)
[ANAV-LOG] Error: cyberpunk2077_multiplayer: Command Cannot Load
[ANAV-LOG] Foreground Process 1789865 (Task 2): cyberpunk2077_multiplayer
(Terminated Normally)
ANAV$ exec 3 < fox.txt
[ANAV-LOG] Foreground Process 1789901 (Task 3): grep the (Started)
[ANAV-LOG] Redirecting input from fox.txt for Task #3
the quick
the lazy
[ANAV-LOG] Foreground Process 1789901 (Task 3): grep the (Terminated Normally)
ANAV$ list
[ANAV-LOG] 3 Task(s)
[ANAV-LOG] Task #1: ls -a (PID 1789862; Finished; exit code 0)
[ANAV-LOG] Task #2: cyberpunk2077_multiplayer (PID 1789865; Finished; exit code 1)
[ANAV-LOG] Task #3: grep the (PID 1789901; Finished; exit code 0)
ANAV$ exec 1 > out.txt
[ANAV-LOG] Foreground Process 1798401 (Task 1): ls -a (Started)
[ANAV-LOG] Redirecting output to out.txt for Task #1
[ANAV-LOG] Foreground Process 1798401 (Task 1): ls -a (Terminated Normally)
ANAV$
```

3.4.3 The "bg" Built-In Instruction

This built-in command is how we run a non-interactive program (eg. `ls`, `wc`, `cat`, `cal`, `slow_cooker`) in a way where we do not wait for the program to finish before prompting the user for more input.

Like with `exec`, this will involve creating a child process to run the program in. You will need to execute the program with arguments, that are stored in your Task structure. You may have to try multiple paths (see 3.4.1) when executing the Program.

3.4.3.1 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.
  - o   Signal handling will be necessary to detect process completion; see 3.5.5.
- 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 3.3.
- When the process completes, the task should record the process's exit code; see 3.5.2.

Logging Requirements:
- This logging requirements are nearly identical to those of the **exec** command.
- The exception: `log_anav_status_change(task_num, pid, LOG_BG, cmd, LOG_START)`.
- With background execution, process termination is logged only when the process completes.

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.**
- **It's normal to not see the prompt.** You can hit enter at any time to get the prompt back.

Example Run (bg instruction):

```
ANAV$ ls -a
[ANAV-LOG] Adding Task #1: ls -a (Ready)
ANAV$ slow_cooker 5
[ANAV-LOG] Adding Task #2: slow_cooker 5 (Ready)
ANAV$ bg 2
[ANAV-LOG] Background Process 1823785 (Task 2): slow_cooker 5 (Started)
ANAV$ [PID: 1823785] slow_cooker count down: 5 ...
[PID: 1823785] slow_cooker count down: 4 ...
[PID: 1823785] slow_cooker count down: 3 ...
exec 1
[ANAV-LOG] Foreground Process 1823788 (Task 1): ls -a (Started)
.  ..   anav  fox.txt  inc  Makefile  my_echo  my_pause   obj  slow_cooker   src
[ANAV-LOG] Foreground Process 1823788 (Task 1): ls -a (Terminated Normally)
ANAV$ [PID: 1823785] slow_cooker count down: 2 ...
[PID: 1823785] slow_cooker count down: 1 ...
[ANAV-LOG] Background Process 1823785 (Task 2): slow_cooker 5 (Terminated Normally)
list
[ANAV-LOG] 2 Task(s)
[ANAV-LOG] Task #1: ls -a (PID 1823788; Finished; exit code 0)
[ANAV-LOG] Task #2: slow_cooker 5 (PID 1823785; Finished; exit code 5)
ANAV$
```

3.4.4 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.

Each pipe is **one-direction** only, meaning it has a **read-end** and a **write-end**. If you want two processes to talk to each other, you will need two pipes! For this program, we'll only need one, which will let process A send its output to process B.

Consider this scenario:
- Parent process **P** has two children, **A** and **B**.
  - **A** will be sending its output 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** will get sent as input to **B**.

<div style="border:1px solid black;text-align:center;">

It is vitally important to close all unused pipe ends as described above.

</div>

Without doing so, **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**).

3.4.5 The "pipe" Built-In Instruction

In ANAV, we can create two tasks and then use the `pipe` instruction to have the first task run and redirect all of its output to the second task, which will use that as its input.

For example, if TASKNUM1 was **"cat fox.txt"** and TASKNUM2 was **"wc -l"**, then running the pipe command will run the first task, which outputs all the lines of text in fox.txt, sending them as the input into the second task, which will count the number of lines it received. The ultimate output on the screen will be the number 5, representing the 5 lines in fox.txt.

3.4.5.1 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 3.5.5.
- Executes ***TASKNUM2*** as the foreground process (see **exec**).
- The processes' statuses change as they run and eventually complete; see 3.3.
- When either process completes, the task should record the process's exit code; see 3.5.2.

Logging Requirements:
- If the two Task Numbers are identical, call **log_anav_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_anav_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_anav_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 3.4.4.

Example Run (pipe instruction):

```
ANAV$ cat fox.txt
[ANAV-LOG] Adding Task #1: cat fox.txt (Ready)
ANAV$ grep the
[ANAV-LOG] Adding Task #2: grep the (Ready)
ANAV$ pipe 1 2
[ANAV-LOG] Opening a pipe from Task #1 to Task #2
[ANAV-LOG] Background Process 1946170 (Task 1): cat fox.txt (Started)
[ANAV-LOG] Foreground Process 1946171 (Task 2): grep the (Started)
the quick
the lazy
[ANAV-LOG] Background Process 1946170 (Task 1): cat fox.txt (Terminated Normally)
[ANAV-LOG] Foreground Process 1946171 (Task 2): grep the (Terminated Normally)
ANAV$
```

3.5 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.

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

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

Logging Requirements:
- If the selected task does not exist, call **log_anav_task_num_error(task_num)**.
- If the task is currently idle, call **log_anav_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_anav_sig_sent(LOG_CMD_KILL, task_num, pid)**.
- As a side effect, the child may exit, which leads to additional logs; see 3.5.2.

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 3.5.2.

Example Run (kill instruction):
```
ANAV$ sleep 1000
[ANAV-LOG] Adding Task #1: sleep 1000 (Ready)
ANAV$ bg 1
[ANAV-LOG] Background Process 1983555 (Task 1): sleep 1000 (Started)
ANAV$ list
[ANAV-LOG] 1 Task(s)
[ANAV-LOG] Task #1: sleep 1000 (PID 1983555; Running)
ANAV$ kill 1
[ANAV-LOG] Kill message sent to Task #1 (PID 1983555)
ANAV$ [ANAV-LOG] Background Process 1983555 (Task 1): sleep 1000 (Terminated by
Signal)
ANAV$ list
[ANAV-LOG] 1 Task(s)
[ANAV-LOG] Task #1: sleep 1000 (PID 1983555; Killed; exit code 0)
ANAV$
```

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

This will **suspend** the process associate with the task number. A suspended process is taken off of the Ready Queue and placed in a special Suspened Queue, meaning it won't run again on the CPU until it's been **resumed**.

Logging Requirements:
- If the selected task does not exist, call **log_anav_task_num_error(task_num)**.
- If the task is currently idle, call **log_anav_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_anav_sig_sent(LOG_CMD_SUSPEND, task_num, pid)**.
- As a side effect, the child may be suspended, which leads to additional logs; see 3.5.2.

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 3.5.2.

3.5.1.3 *resume* TASKNUM*: resumes a suspended process with* TASKNUM *using* SIGCONT.
This will take a process and resume it.

Logging Requirements:
- If the selected task does not exist, call **log_anav_task_num_error(task_num)**.
- If the task is currently idle, call **log_anav_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_anav_sig_sent(LOG_CMD_RESUME, task_num, pid)**.
- As a side effect, the child may be resumed, which leads to additional logs; see 3.5.2.

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 3.5.2.

Example Run (suspend and resume instructions):

```
ANAV$ slow_cooker 5
[ANAV-LOG] Adding Task #1: slow_cooker 5 (Ready)
ANAV$ bg 1
[ANAV-LOG] Background Process 917461 (Task 1): slow_cooker 5 (Started)
ANAV$ [PID: 917461] slow_cooker count down: 5 ...
[PID: 917461] slow_cooker count down: 4 ...
suspend 1
[ANAV-LOG] Suspend message sent to Task #1 (PID 917461)
ANAV$ [ANAV-LOG] Background Process 917461 (Task 1): slow_cooker 5 (Stopped)
ANAV$ list
[ANAV-LOG] 1 Task(s)
[ANAV-LOG] Task #1: slow_cooker 5 (PID 917461; Suspended)
```

```
ANAV$ resume 1
[ANAV-LOG] Resume message sent to Task #1 (PID 917461)
[ANAV-LOG] Foreground Process 917461 (Task 1): slow_cooker 5 (Continued)
[PID: 917461] slow_cooker count down: 3 ...
[PID: 917461] slow_cooker count down: 2 ...
[PID: 917461] slow_cooker count down: 1 ...
[ANAV-LOG] Foreground Process 917461 (Task 1): slow_cooker 5 (Terminated Normally)
ANAV$
```

3.5.2 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 3.4.2). We may also perform waiting whenever we receive a signal from the child (see 3.5.5).

No matter how 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 states (e.g. by suspending and resuming it; section 3.5), we only need to send the signal to the process; later, when we have a chance to use **waitpid()**, we can update the task/process state in our program. This means that our **waitpid()** calls come with additional logging requirements:

Logging Requirements:
- On state change: **log_anav_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.
    o 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.
    o 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.
    o This is the best way to be sure to record exit codes for a terminated process!

3.5.3 Keyboard Interaction

Several 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.

By default, your shell is the target for the keyboard interactions, which is actually one of the reasons why we do not support interactive programs being run. So, when you enter ctrl-C, it will go to your shell! Our command shell will have to use signal handling to detect keyboard inputs, and to forward those signals to the appropriate child process.

One other note is that, by default, all child processes belong to the same **process group** as their parent process. A process group is just a logical collection of processes, but the important part is that ctrl-C or ctrl-Z be sent to your shell, but they also go to every other process in the same group, including all of your child processes! We can fix this by putting each child process you create in their own group, which will separate them from your shell's group.   See **Appendix C** for information on how to do this.

For this project, we only 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_anav_ctrl_c()** to report the arrival of **SIGINT** triggered by **^C**.
- Call **log_anav_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:
- Remember, keyboard-triggered signals are first sent to your shell process.
    o If you want the foreground child process to receive this signal, you will need to handle that signal and forward that to the appropriate child process.
- By default, a keyboard-triggered signal also gets sent to *all* processes, including children.

-     o   We can avoid this by putting different processes in different groups using **setpgid()**.
-     o   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 Run (Keyboard ctrl-c / ctrl-z):

```
ANAV$ sleep 1000
[ANAV-LOG] Adding Task #1: sleep 1000 (Ready)
ANAV$ exec 1
[ANAV-LOG] Foreground Process 826993 (Task 1): sleep 1000 (Started)
^C[ANAV-LOG] Keyboard Combination control-c Received
[ANAV-LOG] Foreground Process 826993 (Task 1): sleep 1000 (Terminated by Signal)
ANAV$ list
[ANAV-LOG] 1 Task(s)
[ANAV-LOG] Task #1: sleep 1000 (PID 826993; Killed; exit code 0)
ANAV$ exec 1
[ANAV-LOG] Foreground Process 827072 (Task 1): sleep 1000 (Started)
^Z[ANAV-LOG] Keyboard Combination control-z Received
[ANAV-LOG] Foreground Process 827072 (Task 1): sleep 1000 (Stopped)
ANAV$ list
[ANAV-LOG] 1 Task(s)
[ANAV-LOG] Task #1: sleep 1000 (PID 827072; Suspended)
ANAV$
```

3.5.4 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 task 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 parent requests to delete Task 2, just before the running process in Task 3 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 task 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()`.

3.5.5 Signal Handling with SIGCHLD

If you have already implemented the keyboard interrupts described in section 3.5.3, then you already have signal handling built into your code. If you have taken the signal blocking precautions described in 3.5.4, 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.

When a child process changes state, it automatically sends out a `SIGCHLD` signal to the parent. This includes when the process terminates, or is killed, or suspends, or resumes. To find out when our processes change state, we should set up a signal handler to 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 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 3.5.4) 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 signal the process, the action can take place in the signal handler. (see 3.5)
- 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.

4 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:

In the src/ Subdirectory:
- `anav.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!
- `logging.c` – This has a list of provided logging functions that you need to call at the appropriate times.
- `parse.c` – This has a list of provided parsing functions that you could use to divide the user command line into useful pieces.
- `util.c` – This includes utility functions which you may call to automate the process of copying and clearing strings and argument lists.
- `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 by exiting with the original count value. You can specify a different starting value 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.

In the inc/ Subdirectory:
- `anav.h` – This has some basic definitions and includes necessary header files.
- `logging.h` – This has the prototypes of logging functions implemented in `logging.c`.
- `parse.h` – This has the prototypes of parsing functions implemented in `parse.c`.
- `util.h` – This has the prototypes of parsing functions implemented in `util.c`.

In the main handout Directory:
- `Makefile` – to build the assignment (and clean up).
- `fox.txt` – a simple text file for convenient testing (of file redirection).

To get started on this project, read through the provided code in `anav.c` and the constants and definitions in `anav.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.

5 Submitting & Grading

Submit this assignment electronically on Blackboard. <u>Note that the only file that gets submitted is</u> `anav.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 `anav.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.

6 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 enough space to store the complete string, and we should use **strncat** to make sure that we do not overrun the space.

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.

```c
typedef struct instruction_struct{
        char *instruct;    // the instruction we're running
        int id1;           // the Task Number associated with the instruction,
                           // or 0 if none/default
        int id2;           // 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 **anav.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 **anav.c** already includes necessary steps to make the call to **parse()**.
- The provided **anav.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.
    - **`how`** is one of **`SIG_BLOCK`**, **`SIG_UNBLOCK`**, or **`SIG_SETMASK`**
    - 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>
```

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 these 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

7   Changelog

v1.0

      Mar 22, 8:00pm: Initial Release

v1.1

      Mar 23, 12:00pm: Removed a partial sentence ("It doesn't") on the `resume` command.

                                    Replaced all occurrences with `buf_id` to `task_num`.

                                    Replaced all references to buffers with tasks.

v1.2

      Mar 25, 8:30pm: **Updated Example Run for 3.5.1.3**

                                    - This had "Background Process" instead of "Foreground Process"

                                    Updated Example Run for 3.5.3

                                    - This had "Background Process" instead of "Foreground Process"