Homework 4 Debugger - CSE 320 - Fall 2023

Professor Eugene Stark

Due Date: Friday 11/17/2023 @ 11:59pm

Introduction

The goal of this assignment is to become familiar with low-level Unix/POSIX system calls related to processes, signal handling, files, and I/O redirection. You will implement a simplified debugger program program, called `deet`, which is capable of managing and performing some basic debugging operations on a collection of target processes.

Takeaways

After completing this assignment, you should:

- Understand process execution: forking, executing, and reaping.
- Understand signal handling and asynchronous I/O.
- Understand the use of "dup" to perform I/O redirection.
- Have gained experience with C libraries and system calls.
- Have enhanced your C programming abilities.

Getting Started

Here is the structure of the base code:

```
.
├── .gitignore
├── .gitlab-ci.yml
└── hw4
    ├── demo
    │   └── deet
    ├── hw4.sublime-project
    ├── include
    │   ├── debug.h
    │   └── deet.h
    ├── lib
    │   └── logger.o
    ├── Makefile
    ├── src
    │   └── main.c
    ├── test_output
    │   └── .git-keep
    ├── testprog
    │   ├── tp
    │   └── tp.c
    └── tests
        ├── basecode_tests.c
        ├── rsrc
        │   ├── hello_world.err
        │   ├── hello_world.in
        │   ├── hello_world.out
        │   ├── sleep_no_wait.err
        │   ├── sleep_no_wait.in
        │   ├── sleep_no_wait.out
        │   ├── sleep_wait.err
        │   ├── sleep_wait.in
        │   ├── sleep_wait.out
        │   └── startup_quit.in
        ├── test_common.c
        └── test_common.h
```

The `include` and `src` directories contain header and source files, as usual. The `lib` directory contains headers and binaries for some library code that has been provided for you. The `demo` directory contains an executable demonstration version of the program. The `testprog` directory contains code for a very simple program that you can use as a target for debugging. The `tests` directory contains some very basic tests. All of these are discussed in more detail below.

If you run `make`, the code should compile correctly, resulting in two executables `bin/deet` and `bin/deet_tests`. The executable `bin/deet` is the main one. If you run this program, it will just abort, because the program has not yet been implemented -- you have to do it!

The executable `bin/deet_tests` runs only some very basic tests, which cover some easily overlooked cases that come up during startup and termination of the program. Most of the functionality is not exercised at all by these tests.

Hints and Tips

- Due to the fact that the kind of program you will write in this assignment will most likely be unfamiliar to you and you will need to use a number of new system calls to do it, it is almost certainly **not** a good idea for you to just set about writing the whole program at one go and trying to make it work. Instead, you should develop the program in an incremental fashion, experimenting with each of the system calls (ideally in the context of simple test programs) to develop an understanding of how they work. Then you can put the pieces together.

- We **strongly recommend** that you check the return codes of **all** system calls and library functions. This will help you catch errors.

- You should use the `debug` macro provided to you in the base code. That way, when your program is compiled without `-DDEBUG`, all of your debugging output will vanish, preventing you from losing points due to superfluous output. Your program should only produce output that has been specified in this document, and you should pay close attention to whether the output should be directed to the standard output or to the standard error output.

- Put to good use the tools that have been introduced in previous assignments. In particular, use `valgrind` to check for serious memory access errors, and `gdb` for normal debugging.

- **BEAT UP YOUR OWN CODE!** Exercise your code thoroughly with various numbers of processes, problem mix, and timing situations, to make sure that no sequence of events can occur that can crash the program.

- Your code should **NEVER** crash, and we will deduct points every time your program crashes during grading. Especially make sure that you have avoided race conditions involving process termination and reaping that might result in "flaky" behavior. If you notice odd behavior you don't understand: **INVESTIGATE**.

> *:nerd: When writing your program, try to comment as much as possible and stay consistent with code formatting. Keep your code organized, and don't be afraid to introduce new source files if/when appropriate.*

Reading Man Pages

This assignment will involve the use of many system calls and library functions that you probably haven't used before. As such, it is imperative that you become comfortable looking up function specifications using the `man` command.

The `man` command stands for "manual" and takes the name of a function or command (programs) as an argument. For example, if I didn't know how the `fork(2)` system call worked, I would type `man fork` into my terminal. This would bring up the manual for the `fork(2)` system call.

> *:nerd: Navigating through a man page once it is open can be weird if you're not familiar with these types of applications. To scroll up and down, you simply use the **up arrow key** and **down arrow key** or **j** and **k**, respectively. To exit the page, simply type **q**. That having been said, long `man` pages may look like a wall of text. So it's useful to be able to search through a page. This can be done by typing the / key, followed by your search phrase, and then hitting **enter**. Note that man pages are displayed with a program known as `less`. For more information about navigating the `man` pages with `less`, run `man less` in your terminal.*

Now, you may have noticed the `2` in `fork(2)`. This indicates the section in which the `man` page for `fork(2)` resides. Here is a list of the `man` page sections and what they are for.

Section Contents

1 User Commands (Programs)
2 System Calls
3 C Library Functions
4 Devices and Special Files
5 File Formats and Conventions
6 Games, et al
7 Miscellanea
8 System Administration Tools and Daemons

From the table above, we can see that `fork(2)` belongs to the system call section of the `man` pages. This is important because there are functions like `printf` which have multiple entries in different sections of the `man` pages. If you type `man printf` into your terminal, the `man` program will start looking for that name starting from section 1. If it can't find it, it'll go to section 2, then section 3 and so on. However, there is actually a Bash user command called `printf`, so instead of getting the `man` page for the `printf(3)` function which is located in `stdio.h`, we get the `man` page for the Bash user command `printf(1)`. If you specifically wanted the function from section 3 of the `man` pages, you would enter `man 3 printf` into your terminal.

> *:scream: Remember this: **`man` pages are your bread and butter**. Without them, you will have a very difficult time with this assignment.*

Development and Test Strategy

You will probably find it the most efficient approach to this assignment to write and test your code incrementally, a little bit at a time, to develop your understanding and verify that things are working as expected. You will probably it overwhelmingly difficult to debug your code if you try to write a lot of it first without trying it out little by little.

Putting some effort into creating useful, understandable, debugging trace output will also be very helpful.

The `killall` Command

In the course of debugging this program, you will almost certainly end up in situations where there are a number of "leftover" target processes that have survived beyond a particular test run of the program. If you allow these processes to accumulate, it can cause confusion, as well as consume resources on your computer. The `ps(1)` (process status) command can be used to determine if there are any such processes around; e.g.

```
$ ps alx | grep a.out
```

will find all processes currently running the program `a.out`. If there are a lot of them, it can be tedious to kill them all using the `kill` command. The `killall` command can be used to kill all processes running a program with a particular name; e.g.

```
$ killall a.out
```

It might be necessary to use the additional option `-s KILL` to send these processes a `SIGKILL`, which they cannot catch or ignore, as opposed to the `SIGTERM`, which is sent by default.

The `strace` Command

An extremely useful (but rather advanced) feature that Linux provides for debugging is the `strace(1)` command. When a program is run via `strace`, you can get an extremely detailed trace of all of the operating system calls made by the main process of this program, as well as child processes. This can be useful when all else fails in trying to understand what a program is doing, however the down side is that to understand the voluminous output produced by `strace` requires a fair amount of technical knowledge about Linux system calls. You might want to give it a try, though.

`Deet`

Description of Behavior

The `deet` program is a highly simplified version of a debugger, in the spirit of `gdb`. It provides the user with a command-oriented interface that permits "target" processes to be started, stopped, examined, modified, and killed. All user input is read from the standard input and output for the user is printed to the standard output. Output to the standard output should be exactly in the format described below and nothing else should be printed to standard output.

:scream Your `deet` program should recognize the command-line flag `-p`. If this flag is given, the normal interactive prompt `deet> ` printed by `deet` should be suppressed, otherwise it should be printed as discussed below. It is very important that you do this correctly, because the absence of the prompt simplifies the task of automatically analyzing the other output produced by your program.

`Deet` is able to manage multiple target processes at once. It maintains information about all the targets that are currently being debugged and their current state. Commands given by the user name the target process that is to be manipulated by specifying an integer "`deet` ID". When a target process is started, it receives the smallest unused `deet` ID (starting from 0). Later, when the target process has terminated, `deet` will forget about it and make its `deet` ID available for re-use. The rules for when `deet` will forget about a target process are given later in this document.

Here is an example of a `deet` session. First, `deet` is started from the Linux shell and it prompts the user:

```
$ bin/deet
deet>
```

The prompt consists of the string `deet> `, which contains a single space at the end, and no newline. The user enters a blank line, in which case the prompt is simply given again, without any other output. Next, the user asks for help:

```
deet> help
Available commands:
help -- Print this help message
quit (<=0 args) -- Quit the program
show (<=1 args) -- Show process info
run (>=1 args) -- Start a process
stop (1 args) -- Stop a running process
cont (1 args) -- Continue a stopped process
release (1 args) -- Stop tracing a process, allowing it to continue normally
wait (1-2 args) -- Wait for a process to enter a specified state
kill (1 args) -- Forcibly terminate a process
peek (2-3 args) -- Read from the address space of a traced process
poke (3 args) -- Write to the address space of a traced process
bt (1 args) -- Show a stack trace for a traced process
```

The commands understood by `deet` are listed, together with information about the number of arguments each takes. Each command has a minimum number of arguments that it requires and a maximum number of arguments that it permits. For some commands, the minimum and maximum are the same; for those commands, exactly that number of arguments must be given. For some other commands (such as `run`), no maximum is specified; in that case, an arbitrary number of arguments greater than or equal to the minimum may be given.

Issuing a command that is not in the list causes `deet` to report an error:

```
deet> bogus
?
```

This is how `deet` reports an error: by a single question mark `?` followed by a newline output to the standard output. This applies to all error situations. For debugging/explanatory purposes, `deet` is permitted to issue an error message, but this **must** go to the standard error output.

The user then issues the `run` command to start a process to be debugged:

```
deet> run echo a b c
0	137480	T	running		echo a b c
```

Here the program `echo` has been run, with three command-line arguments: `a`, `b`, and `c`. A process with Linux process ID `137480` has been created, and `deet` ID 0 has been assigned to it. A status line has been printed by `deet`, giving the essential information about this process. The status line contains: the `deet` ID, the Linux process ID, the character `T` indicating that the process is being traced by `deet` (a character `U` would indicate that the process is "untraced"), the current state of the process from `deet`'s point of view (here it is `running`), the exit status that was returned when the process terminated (here there is none, so this field is empty), and finally the command that was executed to start the process. The fields in the output are separated by single TAB (*i.e.* `'\t'`) characters, except within the command, which is printed as the last field on the line and shows the words from the command line separated from each other by a single space character. The status line is terminated by a single newline (`'\n'`) character.

The process that has been started then immediately stops before actually doing anything. This is to allow `deet` to take control of it:

```
deet> 0	137480	T	stopped		echo a b c
```

A status line similar to the first has been printed, with the difference that the state of the process is now reported as `stopped`, rather than `running`. Notice that the status line has appeared after the `deet>` prompt. This is because the target process started executing concurrently with `deet` and then `deet` received an asynchronous notification (via a `SIGCHLD` signal) that the target process had stopped. By this time, `deet` had already printed the prompt and was waiting for input from the user, but the asynchronous notification interrupted the reading of user input and caused `deet` to print the new status line. After that, it will go back and print another prompt and continue to wait for user input.

Next, the user started another process to be debugged; this time running the `sleep` command with an argument of `10` (`sleep` is a command that simply waits for a specified number of seconds before terminating):

```
deet> run sleep 10
1	137935	T	running		sleep 10
deet> 1	137935	T	stopped		sleep 10
```

Notice that the process has stopped immediately as before. It has been assigned `deet` ID 1 and is running a process with Linux process ID `137935`.

The user now asks for information about all the processes currently being debugged. The current status of both processes is shown:

```
deet> show
0       137480  T       stopped         echo a b c
1       137935  T       stopped         sleep 10
```

Next, the user starts a process running a test program `testprog/tp`, which has been provided for you:

```
deet> run testprog/tp
2       138055  T       running         testprog/tp
deet> 2 138055  T       stopped         testprog/tp
```

The program stops, as before. Next, the user tells the test program (which has `deet` ID 2) to continue:

```
deet> cont 2
2       138055  T       running         testprog/tp
deet> function a @ 0x5607f22551cd, argument x @ 0x7ffeac24a4ec (=666)
function b @ 0x5607f225521b: argument x @ 0x7ffeac24a4cc (=667)
function c @ 0x5607f2255269: argument x @ 0x7ffeac24a4ac (=668)
function d @ 0x5607f22552b7: argument x @ 0x7ffeac24a48c (=669)
function e @ 0x5607f2255305: argument x @ 0x7ffeac24a46c (=670)
function f @ 0x5607f225535b called
static_variable @ 0x5607f2258030 (=0)
local_variable @ 0x7ffeac24a440 (=29a)
2       138055  T       stopped         testprog/tp
```

`Deet` is informed that the test program is now running. The test program prints out some information about itself which will aid in testing the debugging features of `deet`. It prints the addresses (in its text segment) of six functions, called `a`, `b`, `c`, `d`, `e`, `f`. Each function prints the address (in the stack segment) of its argument `x`, together with the current value of that argument. The test program also prints the address and value of a variable `static_variable`, which lives in the data segment, and of a local variable `local_variable`, which lives in the stack segment, in the stack frame for function `f`. The test program then uses the `kill()` system call to send itself a `SIGSTOP` signal. As a result, it stops execution and `deet` receives an asynchronous notification that the test process has stopped. In response, `deet` prints a status line showing the changed status of this process.

The user then asks `deet` to give a stack trace for the stopped test process:

```
deet> bt 2
00007ffeac24a470	00005607f2255302
00007ffeac24a490	00005607f22552b4
00007ffeac24a4b0	00005607f2255266
00007ffeac24a4d0	00005607f2255218
00007ffeac24a4f0	00005607f22551c6
00007ffeac24a510	00007fc824f52d90
```

Each line shows two addresses in hexadecimal, separated by a single TAB character (the addresses were printed using the `printf` conversion specifier `%016lx`). The first address on each line is the address of the stack frame for a function activation. These addresses are in the stack segment of the test process. The second address on each line is the return address that is stored in that stack frame. These addresses are in the text segment of the test process, except for the last one, which is in C library code that has been mapped into a different part of the address space by the dynamic loader. The return addresses reflect the chain of calls that was made leading to the current state: `(start) -> main -> a -> b -> c -> d -> e -> f`. Here `(start)` stands for some anonymous startup code in the C library that runs first and makes the initial call to `main`. Note that the stack frames are traversed and printed in the reverse order from which the functions were called, so that the first line gives the address of the stack frame for the most-recently called function `f` and the return address reflects the point within the code for function `e` just after the call of function `f`.

The user then requests that the test process continue execution:

```
deet> cont 2
2	138055	T	running		testprog/tp
deet> function f @ 0x5607f225535b called
static_variable @ 0x5607f2258030 (=0)
local_variable @ 0x7ffeac24a440 (=29a)
2	138055	T	stopped		testprog/tp
```

A status line is printed by `deet` when it is notified that the test process has continued execution. In the test program, function `f` returns to function `e` and is then called again, resulting in the same printout as before being repeated, and then the test process uses `SIGSTOP` to stop itself once again. `Deet` is notified that the test process has stopped and it prints an updated status line.

Next, the user uses the `peek` command to examine the contents of the argument `x` stored in the stack frame for function `c`:

```
deet> peek 2 7ffeac24a4ac
00007ffeac24a4ac	ac24a4d00000029c
```

The two least-significant bytes of the word printed contains the value `029c`, which is hexadecimal for `668`, the current value of the argument `x` to function `c`.

The user then uses the `poke` command to modify the value of the global variable `static_variable` stored in the data segment:

```
deet> poke 2 5607f2258030 1023
```

The user then allows the test process to continue:

```
deet> cont 2
2       138055  T       running         testprog/tp
deet> function f @ 0x5607f225535b called
static_variable @ 0x5607f2258030 (=1023)
local_variable @ 0x7ffeac24a440 (=29a)
2       138055  T       stopped         testprog/tp
```

Once again, the test process returns from `f` to `e`, the function `f` is called again, and the current values of the variables are printed. Now the value of `static_variable` is seen to be `1023`.

At this point, the user has decided that enough has been done with the test process, and so uses the `kill` command to cause it to terminate:

```
deet> kill 2
2       138055  T       killed          testprog/tp
deet> 2 138055  T       dead    0x9     testprog/tp
```

Execution of the `kill` command causes `SIGKILL` to be sent to the test process, and at the same time `deet` reports that the state of this process is now `killed`, though the process has not yet terminated. Shortly thereafter, the test process receives the `SIGKILL` and actually does terminate. `Deet` is then notified of the termination and it reports that the process is now `dead`. The exit status (obtained via the `waitpid()` system call is also reported by `deet`). In this case, the program terminated as a result of `SIGKILL`, which is signal number 9, so the exit status is `0x9` (had the program terminated normally, the low-order byte would be 0 and the exit status passed to `exit()` would appear in the next-most-significant byte).

The user then decides to allow the process running `sleep` to continue:

```
deet> cont 1
1       137935  T       running         sleep 10
deet> 1 137935  T       dead    0x0     sleep 10
```

Initially, `deet` reports that the process is running and then (after a 10-second delay) it reports that the process has terminated with exit status `0x0` returned by `waitpid()`.

Finally, the user decides to quit `deet`:

```
deet> quit
$
```

Although there was no output to document it, before it terminated, `deet` arranged to kill the remaining process (running `echo`), and await its termination, before exiting itself.

Details of the Commands

The previous section illustrated most aspects of `deet`'s functionality; the current section fills in some more specific details about the commands. The behavior of your implementation should match these descriptions.

- The `help` command (zero or more arguments) ignores any arguments it is given and prints a message listing information about the available commands.
- The `quit` command (no arguments required or allowed) causes `deet` to terminate, after first killing any processes being debugged to be killed and then waiting for them to actually terminate before it itself exits.
- The `show` command may be given with no arguments, in which case it outputs one line of information about the status of each of the processes currently being managed by `deet`. It also may be given a single optional integer argument, in which case it gives only information about the process having that integer as its "`deet` ID". If an invocation of the `show` command causes no output to be printed (either because there are no processes currently being managed or because a process with the specified `deet` ID does not exist), then an error is reported instead.
- The `run` command serves to start a process to be debugged. At least one argument is required, which is interpreted as the name of the program to be run. There may be additional arguments, which taken together with the first argument form the `argv[]` vector for the program to be executed. A process started with `run` begins execution with tracing enabled (more on this below). Tracing remains enabled until the process terminates or a `release` command is executed on that process.

In order to explain the remaining commands, we first need to look at the states that a process may be in, from the point of view of `deet`. The following list (defined in `deet.h`) enumerates these states:

```
PSTATE_NONE,        // State of a nonexistent process.
PSTATE_RUNNING,     // State of a process immediately following successful fork().
PSTATE_STOPPING,    // State of the process while attempting to stop it,
                    // but before receiving the subsequent notification via SIGCHLD.
PSTATE_STOPPED,     // State of the process once it is known to have stopped.
PSTATE_CONTINUING,  // State of the process after having continued it,
                    // but before receiving the subsequent notification via SIGCHLD
                    // that the process has continued.
PSTATE_KILLED,      // State of the process after sending SIGKILL but before
                    // receiving the subsequent notification via SIGCHLD
                    // that the process has terminated.
PSTATE_DEAD         // State of the process once it is known to be terminated.
```

> *:nerd: Note that the `PSTATE_NONE` state does not correspond to the state of any actual process. This value may be used, for example, as a place-holder value to identify a free slot in the table of processes that `deet` maintains.*

When the user issues a `run` command, `deet` creates a child process using the `fork()` system call. Upon a successful call to `fork()`, the newly created process is considered by `deet` to be in the `PSTATE_RUNNING` state. The child process first uses the `dup2()` system call to close its original standard output and arrange for any subsequent output to STDOUT_FILENO to be redirected to the file open on STDERR_FILENO. This is so that output from the child process can be separated from the output produced by `deet` itself. The child process uses the `PSTATE_TRACEME` request to the `ptrace()` system call to request that it be traced by `deet` (more on this below). As a result of this request, the child immediately receives a `SIGSTOP` signal and stops execution. At the same time, `SIGCHLD` signal is sent by the Linux kernel to the `deet` process. When the `deet` process receives this signal, it uses the `waitpid()` system call to determine which of the potentially several processes it is managing has changed state, and what that state change was. When `deet` learns that the newly created child process has stopped, it records that that process is now in the `PSTATE_STOPPED` state and it arranges for updated status information to be printed for the user.

> *:scream: Any time `deet` changes the state of a process, it should arrange for an updated status line for that process to be printed before `deet` issues its next prompt to the user. This will not be explicitly mentioned again in the sequel.*

- A process in the `PSTATE_STOPPED` state may be caused to continue execution by the user entering a `cont` command and specifying the `deet` ID of that process. Upon seeing such a command, `deet` arranges for a `SIGCONT` signal to be sent to the process, and it records the process as being in the `PSTATE_CONTINUING` state. When the process receives the `SIGCONT` signal and actually does continue execution, the Linux kernel notifies deet by sending it a SIGCHLD signal. Upon receiving this SIGCHLD, deet again uses waitpid() to determine what has occurred, and upon finding that the child process has continued execution it records it as now being in the PSTATE_RUNNING state.

- A process in the PSTATE_RUNNING state may be caused to stop execution by the user entering a stop command and specifying the deet ID of that process. Upon seeing such a command, deet arranges for a SIGSTOP signal to be sent to the target process, and it records that process as being in the PSTATE_STOPPING state. When the target process receives the SIGSTOP signal and actually stops execution, the OS kernel once again notifies the deet process via the SIGCHLD/waitpid() mechanism. When the deet process learns that the child process has actually stopped, it records it as being in the PSTATE_STOPPED state.

- The release command, like the cont command, causes the specified target process to continue. However, in this case, further tracing is turned off for this process, using the PTRACE_DETACH request to ptrace(). The result is that although deet can still stop and continue this process (by sending it SIGSTOP and SIGCONT), it will no longer be possible to "peek" or "poke" into the address space of that process, or in general to apply any further ptrace() requests to that process.

A process in any state other than PSTATE_NONE or PSTATE_DEAD may terminate, either by exiting normally (if it was running) or as a result of receiving a signal (a SIGKILL signal will cause the process to terminate even if it was stopped, but other signals will only be received and acted on by the process while it is actually running). When termination of a target process occurs, deet is notified by the SIGCHLD/waitpid() mechanism. It then marks the state of the target process as PSTATE_DEAD and it records the exit status returned by waitpid().

- A process in any state other than PSTATE_NONE, PSTATE_KILLED, or PSTATE_DEAD may be induced by the user to terminate by the user entering a kill command and specifying the deet ID of that process. Upon seeing such a command, deet sends a SIGKILL signal to the process, and it records the process as being in the PSTATE_KILLED state. When the target process actually does terminate, the deet process is notified via the SIGCHLD/waitpid() mechanism, and deet then records the target process as being in the PSTATE_DEAD state and it saves the exit status as for any other termination event.

Once a process has entered the PSTATE_DEAD state, it no longer exists from the point of view of the Linux kernel, however deet will retain information about it until the next time the user uses the run command. As part of the processing of the run command, deet "garbage collects" its table of processes: that is, it finds all processes that are in the PSTATE_DEAD state and it deletes them from the table. This is done *before* allocating a deet ID for the new process to be run. By delaying the deletion of information about dead processes in this way, the user can see the final status of a dead process, but it avoids requiring the user to explicitly delete processes in order to avoid unbounded accumulation of information about dead processes.

- The wait command causes deet to wait until a specified target process has either entered a specified state or has terminated. The first argument is the deet ID of the process to wait for. The (optional) second argument is a string specifying the desired state. Possibilities for this argument are: running, stopping, stopped, continuing, killed, and dead. The default if this argument is not given is dead. Once this command has been entered, no further prompt will be issued until the target process has either entered the specified state or terminated. If the process never enters the specified state or terminates, then the only way for the `wait` command to terminate is if `deet` receives `SIGINT`; for example, as a result of the user typing CTRL-C.

- The user may enter the `show`" command to obtain information about the processes currently being managed by `deet`. Simply entering `show` without any arguments causes `deet` to show the status of every process it knows about, in the format described above. The `show` command may also be given an optional integer argument, which is interpreted as the `deet` ID of a particular process for which information is requested. In that case, `deet` will report on only this one process. If invocation of the `show` command does not result in information on any processes being printed, either because a specified process does not exist or because the set of processes currently being managed is empty, then an error is reported.

- A `quit` command entered by the user causes `deet` to terminate, but only after it has sent `SIGKILL` to all existing processes that it is managing and it has learned (via `SIGCHLD/waitpid()`) that those processes have actually terminated. As soon as all extant processes have entered the `PSTATE_DEAD` state, then `deet` will itself terminate without undue delay.

`Deet` provides three commands that can be used to query or manipulate a process that is in the `PSTATE_STOPPED` state:

- The `peek`" command allows the user to retrieve the value stored at a specified address in the target process' address space. The address is specified in hexadecimal. `Deet` uses the `PTRACE_PEEKDATA` request to retrieve data one word at a time (where a "word" is a 64-bit value). The address and the associated value are printed (in hexadecimal) as described earlier. The `peek` command will also accept an optional additional integer argument (specified in decimal) which is interpreted as the number of successive words to access using the specified address as a starting point. If no additional argument is specified, a single word is retrieved by default. If multiple words are retrieved, each word is printed together with its associated address on a separate line.

- The `poke` command allows the user to modify the value stored at a specified address in the target process' address space. The address is specified in hexadecimal as for the `peek` command. For `poke` the second argument (which is not optional) is interpreted as a 64-bit word of data to be written at the specified address. The `poke` command only stores one full word; there is no way to store only part of a word, and there is no option for treating multiple words as there was for the `peek` command.

- The `bt` ("backtrace") command is used to obtain a stack trace for a stopped process. A backtrace consists of the sequence of return addresses stored in the frames on the stack, starting from the top frame (for the function in which the process was executing when it was stopped), and working back toward the base of the stack. The trace is printed in the format shown previously. The `bt` command takes an optional additional argument, which if given is interpreted as a decimal number that specifies the maximum length of the stack trace. A stack trace that would end up being longer than the specified length is truncated (this also protects against `deet` going into an infinite loop in case for some reason it does not get a valid pointer to start the stack trace). If no limit argument is given, then the default limit of 10 is used.

Stack Traces

Extracting a stack trace requires a basic understanding of the format of the process' stack. The frames on the stack are chained together in a linked list, whose head is maintained in the `rbp` ("base pointer" or "frame pointer") register. Normally, the value in `rbp` is a pointer to the top frame on the stack (corresponding to the currently executing function). Stored at this address is a pointer to the next frame (the caller's frame) and so on. The chain terminates at a frame for the caller of `main()`, which is a function in the runtime startup code in the C library. The invalid pointer value `(void *)0x1` is used as a sentinel to indicate the end of the chain.

The value stored in the `rbp` register can be retrieved by `deet` using the `PTRACE_GETREGS` request to the `ptrace()` system call. This request requires that the `data` argument to the `ptrace()` call be the address of a C "struct" into which the register values are to be stored. The struct in question is `struct user_regs_struct`, which is defined in the header file `<user.h>`. You will probably want to refer to the actual header file itself, which may be found in `/usr/include/x86_64-linux-gnu/sys/user.h`. The field `rbp` is the field to read to get the value of the `rbp` register, once the registers have been stored into this structure using the system call. The `PTRACE_PEEKDATA` request to `ptrace()` can then be used to follow the chained stack frames from the starting address in `rbp`.

Within each stack frame, the return address is stored in the next word "above" (*i.e.* at the next higher address than) the place where the link to the caller's stack frame is stored. For example, the return address for the top stack frame consists of the 64-bit quantity stored starting at address `rbp+0x8`. This organization is the result of the following "preamble code" that is executed at the beginning of every function that is called:

```
        pushq   %rbp
        movq    %rsp, %rbp
```

The first instruction pushes the value of `rbp`, which is a pointer to the caller's stack frame. The second instruction sets the new value of `rbp` to be the top of the stack, which is the place where the old `rbp` was just stored. This links a new stack frame at the front of the list headed by `rbp`.

> *NOTE: Under certain conditions (such as when a process is stopped within a function in the C library, rather than in normal C code) the `rbp` register will contain the value `0x0`, rather than the address of the top frame on the stack. In this case, `ptrace()` will return an error when you attempt to access the value stored at that location. You will need to be careful to check for error returns from `ptrace()` and to terminate the backtrace as soon as there is any error, to avoid "garbage" output.*

Implementation Notes

The `deet` program makes extensive use of processes, signals, and handlers to create and manage the processes being debugged. A number of system calls are involved in this, most of which will have been discussed at least briefly in lecture. When a program to be debugged is started, `deet` uses the `fork()` system call to create a child process for it. Once the child process has started, it uses the `dup2()` system call to redirect standard output, it uses a `PTRACE_TRACEME` request to the `ptrace()` system call to request that it be traced by `deet`, and then it uses the `execvp()` system call to execute the program that was specified by the user.

`Deet` itself needs to arrange to receive asynchronous notifications about changes in state of the processes that it is managing. It does this by using the `sigaction()` system call to install handler for the `SIGCHLD` signal. Upon receipt of `SIGCHLD`, `deet` uses the `waitpid()` system call to find out what process or processes have changed state and what that state change was. You will want to use the `WNOHANG`, `WUNTRACED`, and `WCONTINUED` options to `waitpid()` to ensure: (1) that it does not block if there are no further state changes to find out about; (2) that it finds out about processes that have stopped as a result of receiving `SIGSTOP`; and (3) that it finds out about processes that have continued as a result of receiving `SIGCONT`. The `WIFxxx`, *etc.* macros should be used to extract information from the exit status returned by `waitpid()` (as discussed in class and in the man page for `waitpid()`).

`Deet` should also install a handler for `SIGINT` in order to arrange to clean up (*i.e.* kill and await termination of) any processes being debugged before exiting if CTRL-C should be typed by the user. `Deet` should also be sure to detect any EOF condition that may arise on the standard input and to treat it as if a `quit` command had been typed by the user. For testing purposes, an EOF condition may be generated on the terminal by typing CTRL-D at the beginning of a line (or typing CTRL-D twice if a partial line of input has already been entered).

For robust behavior, when implementing `deet` attention should be paid to using only async-signal-safe functions within a signal handler. The `sigprocmask()` function should be used to temporarily prevent the execution of a signal handler during periods of time when such an execution could interfere with what is going on in the main program. Examples of such situations are when examining or modifying process state information that might change as a result receiving an asynchronous signal that triggers the execution of a handler. The use of `sigprocmask()` implies the use of the associated "signal set" macros such as `sigemptyset()`, `sigaddset()`, *etc*.

`Deet` should use the `kill()` system call to send signals to processes being managed. It will be necessary to use at least `SIGSTOP`, `SIGCONT`, and `SIGKILL`. Exercise caution that you only invoke the `kill()` system call with a positive value for the process ID of the process to be signalled, as invoking it with a a negative value or zero could cause the termination of `deet` itself.

`Deet` should use the `sigsuspend()` system call in order to suspend execution while waiting for a process to transition to the specified state or to terminate.

`Deet` should use the `getline()` function to read user input (from the standard input stream). It may use `fprintf()`, `fputc()`, and the like to emit output for the user (to the standard output stream). Note that `fflush()` should be called on the standard output at any point where user interaction is required (such as after printing a prompt); otherwise the output could remain buffered in memory without the user seeing it. If the `deet` process is blocked in `getline()` waiting for user input, then the receipt of a signal will cause the `getline()` call to be interrupted. After the execution of any handler, the interrupted call to `getline()` will return. This situation can be detected by checking for an error return from `getline()` and looking for `errno` having an associated value `EINTR`. Should this occur, the `getline()` function should be called again after re-printing the user prompt.

The ptrace() System Call

Versions of the `ptrace()` system call have been provided in Unix-like systems since the early days. The name stands for "parent trace", and the system call is intended to support the implementation of a debugger such as `gdb`. The basic model is that of a parent process using this system call to manipulate a child process being debugged. In order for `ptrace()` to work at all, the parent process must first "attach" to the child process. There are various ways to do this, but for `deet` we will use what is perhaps the simplest: when a child process is forked by `deet`, before calling `execvp()` to execute another program it first invokes `ptrace()` with the `PTRACE_TRACEME` request. Note that this is the only time that `ptrace()` is used by the child process -- all the rest of the calls are made by the parent process (*i.e.* `deet` itself). Once the child has attached, then it is possible for the parent process to manipulate the child using various `ptrace()` requests. The parent process can relinquish the ability to trace the child by invoking the `PTRACE_DETACH` request. After that, the child process runs normally and it is no longer possible to manipulate it using `ptrace()`.

For `deet`, it is only necessary to use a few of the requests supported by `ptrace()`; namely, `PTRACE_TRACEME`, `PTRACE_CONT`, `PTRACE_DETACH`, `PTRACE_GETREGS`, `PTRACE_PEEKDATA`, and `PTRACE_POKEDATA`. In order to complete this assignment, you will need to refer to the man page for `ptrace()`. It is quite long, but you only have to know about this small subset of the requests described there.

Event Logging Functions

It is somewhat of a challenging problem for us to automatically evaluate for grading purposes the behavior of a program like `deet`. In order to allow us to track the behavior of your program, we **require** that you call certain functions, which we have provided in binary form, in specific situations arising during execution. These functions are listed below, together with a description of when they are required to be called. Failure to call these functions exactly as specified **will** negatively impact your grade. Note that implementations of these functions have been provided for you in binary form: the `Makefile` will automatically include the file `lib/logger.o` when linking your program. The basecode implementations of these functions will print out tracing information on `stderr` so that you can verify that they have been called properly. You may also find this information helpful for debugging. When we grade your program, we will replace the basecode implementations of these functions by different implementations that communicate automatically with a "tracker" program to check the correctness of what your program is doing.

The list of logging functions that you must call is given below. Function prototypes for these functions have been included in the header file `deet.h`.

- `void log_startup(void);` This function must be called when `deet` starts up, before it does anything else.
- `void log_shutdown(void);` This function must be called as the last action taken before the `deet` program terminates.
- `void log_prompt(void);` This function must be called **immediately before** each place where your program issues a prompt for the user. For technical reasons, even if the actual printing of the prompt has been suppressed because the `-p` command-line option was given, you still need to call this function just before the place where the prompt would otherwise have been printed.
- `void log_error(char *msg);` This function must be called every time execution of a user command results in an error.
- `void log_signal(int sig);` This function must be called as the first action in the signal handler, whenever your program handles a SIGCHLD or SIGINT.

- `void log_input(char *line);` This function must be called every time your program reads a line of input from the user. The argument should be the line of input read, including any terminating newline character.

- `void log_state_change(pid_t pid, PSTATE old, PSTATE new, int status);` This function must be called **every time** `deet` determines that the state of a process has changed, either as the direct result of some action it takes to manipulate the process (such as starting it or continuing it), or as the result of information it receives from handling a SIGCHLD signal (such as the process having stopped, continued, or terminated). The meaning of the arguments is as follows:

  - `pid` The Linux process ID of the process, as returned from `fork()`.
  - `old` The state the process was in before the change. For a newly created process, this will be `PSTATE_NONE`. For an existing process, this state **must** match the new state that was given on the immediately preceding call to `log_state_change()` for that process.
  - `new` The state the process is now in after the change.
  - `status` If the new state is PSTATE_DEAD, then this must be its exit status, as returned, for example, by `waitpid()`.

The following is an example of the type of tracing output that the basecode version of these functions will produce (all of the tracing output goes to the standard error output):

```
$ bin/deet
[00000.000000] STARTUP
[00000.000069] PROMPT
deet>
[00004.224843] INPUT
[00004.224861] PROMPT
deet> bogus
[00008.326146] INPUT bogus
[00008.326161] ERROR bogus
?
[00008.326168] PROMPT
deet> run echo a b c
[00017.982799] INPUT run echo a b c
[00017.983064] CHANGE 142211: none -> running
0       142211  T       running         echo a b c
[00017.983147] PROMPT
deet> [00017.983868] SIGNAL 17
[00017.983895] CHANGE 142211: running -> stopped
0       142211  T       stopped         echo a b c
[00017.983931] PROMPT
deet> run sleep 10
[00036.619853] INPUT run sleep 10
[00036.620021] CHANGE 142212: none -> running
1       142212  T       running         sleep 10
[00036.620088] PROMPT
deet> [00036.620490] SIGNAL 17
[00036.620506] CHANGE 142212: running -> stopped
1       142212  T       stopped         sleep 10
[00036.620527] PROMPT
deet> cont 1
[00042.853035] INPUT cont 1
[00042.853058] CHANGE 142212: stopped -> running
1       142212  T       running         sleep 10
[00042.853105] PROMPT
deet> [00052.854331] SIGNAL 17
[00052.854372] CHANGE 142212: running -> dead
1       142212  T       dead    0x0     sleep 10
[00052.854407] PROMPT
deet> show
[00059.644313] INPUT show
0       142211  T       stopped         echo a b c
1       142212  T       dead    0x0     sleep 10
[00059.644383] PROMPT
```

```
deet> quit
[00064.028144] INPUT quit
[00064.028169] CHANGE 142211: stopped -> killed
[00064.028309] SIGNAL 17
[00064.028335] CHANGE 142211: killed -> dead
[00064.028344] SHUTDOWN
$
```

Each line of tracing output includes a timestamp that gives the number of seconds and microseconds since `deet` was started, the type of event that is being logged, and associated information about this event. By default, the tracing information is enabled, as it will likely be helpful to understanding and debugging. The tracing output can be suppressed by declaring:

```
extern int silent_logging;
```

and setting this variable to a nonzero value.

Demonstration Program and Test Program

I have found that providing a demonstration version of what is to be implemented generally seems to help students to understand the assignment specifications and serves to reduce the number of questions about those specifications. To that end, a demonstration version of `deet` can be found as `demo/deet` in the basecode distribution. The behavior of this program will hopefully match the description in the present document, but in case of any discrepancy the text in this document takes precedence over the behavior of the demonstration program. By default, the demonstration version sends the tracing output produced by the logging functions to the standard error. If it is desired that the tracing output not be produced, you may invoke `demo/deet` with the `-q` option.

Please note that the demonstration program is provided only for you to execute as an aid to your understanding of what I am asking you to do. It is not intended that you should attempt to deconstruct the binary or reverse-engineer source code from it. In the end, you are to write your own code for this assignment. **Any evidence that source code you submit for this assignment has been reverse-engineered from the binary demonstration version I have provided will be considered as evidence of Academic Dishonesty and will result in charged being filed against you.**

I have also provided a very simple test program that can be used as a target for debugging. The use of this program, called `testprog/tp`, was illustrated earlier in this document. The source code for `tp` has been provided (in `testprog/tp.c`) and you should take a look at it to understand what it is doing. You are free to make any modifications you like to it for your own purposes.

Suggested Plan of Attack

Although the amount of code required to implement `deet` is not that great, the complexity of its behavior is much higher than what we have done so far, and you will need to use a lot of new, unfamiliar system calls for it. As a result, there is a substantial chance that many students will not succeed in implementing the full functionality. This is not a disaster, as long as your implementation does *some* things correctly. To have the greatest likelihood of being able to submit something that has at least some testable correct functionality, it is suggested that you attack this assignment in an incremental fashion, as described below.

- Start by implementing at least a rudimentary version of code for interacting with the user. This should be able to prompt the user, read a line of input, separate it into words, identify the command to be executed, and dispatch to a function to execute that command. Initially the command execution functions can just be "stubs", which you will flesh out as you go along. Make the user interface conform to the specifications in this document, so something still can be tested about your program even in case you don't manage to implement everything.

> *:scream: Note that the user interface is not really the main point of this assignment, so you should try not to get bogged down in minute details of it that prevent you from getting on to implementing the actual features of `deet`.*

- Experiment with signal handling, to the point where you understand how to install how to use `sigaction()` to install a handler for `SIGINT` that will be invoked in case the user types CTRL-C. You will then be prepared to implement the handler for `SIGCHLD` that is required in order to manage processes.

- Implement some of the functionality of the `run` command. Use `fork()` and `execvp()` to create a child process and verify that you can get it to execute a specified command. Verify that you know how to use `waitpid()` to retrieve the exit status of a child process.

- Implement a `SIGCHLD` handler to receive asynchronous notifications about the state of a child process. Verify that this handler gets called when a child process is stopped, continued, or terminated. Note that the Bash shell provides a `kill` command that can be used to manually send signals to a process from the terminal. You might find it helpful to do this before having `deet` send these signals programmatically.

- Implement the `stop`, and `kill` commands.

- Add the functionality provided by the `ptrace()` system call and implement the `cont` command.

> *:nerd: Note that the `PTRACE_CONT` request should be used to continue a stopped process that is being traced. To continue a stopped process that is not being traced, you should use the `kill()` system call to send it a `SIGCONT` signal.*

- Identify situations in which asynchronous execution of a signal handler could interfere with what is going on in the main program and use `sigprocmask()` to temporarily mask signals at such times.

Hand-in instructions

As usual, make sure your homework compiles before submitting. Test it carefully to be sure that doesn't crash or exhibit "flaky" behavior due to race conditions. Use `valgrind` to check for memory access errors and leaks.

Submit your work using `git submit` as usual. This homework's tag is: `hw4`.