Take a look at the program `signals.c`. In `main` there is a loop that forks child processes. As this loop runs, the process is likely to receive many signals, so before the loop we need to set up some signal handlers. You will do this by completing the `setupSignalHandler` function, and calling it three times to register three signal handler functions.

1 Signal Handlers

When your single-threaded process receives a signal, its execution will jump to the start of the code for handling that signal. We want to override the default behavior, so we need to register signal our custom signal-handler functions. To do this, we will need to:

1. Create a `struct sigaction` where you will store the parameters to the `sigaction` system call that will setup the signal handler.
2. Zero all the bytes of your `struct sigaction` using the `memset` function.
3. Setup the `sa_handler` field of your `struct sigaction` to be the handler passed as a parameter to the function.
4. Use the `sigaction` system call to setup a signal handler; if the system call fails, print an error message using `perror` and exit the program.

Once you have added code to set up all three signal handlers, try compiling and running the program. If you have two terminals open on the same machine, you can run the program in one terminal, and send signals from the other. The following commands will send the signals for which we have registered handlers (replacing 999999 with the appropriate PID):

```
kill -USR1 999999
kill -CHLD 999999
kill -TERM 999999
```

2 Discussion Questions

Question 1 In `main`, we call `sleep` on every iteration of the while loop. DO NOT TRY REMOVING THIS SLEEP CALL! What is the point of calling `sleep` and why is it so important?

Question 2 We also placed the call to `sleep` in a while loop. This is less critical, but what does it accomplish?

Question 3 In the `childHandler` function, we call `waitpid` even though the only way to get into this function is if a child signal has already been received. What is the point of calling `waitpid` after a child process has exited?

Question 4 What is the purpose of calling `waitpid` in a while loop?

3 Further Information

While a signal is being handled, signals continue to be delivered to the process and they **can interrupt signal handlers as well**. Now suppose that you have two instances of a signal handler (even for the same signal): one of them is will want to print "Hello World" and the other one will want to print "Goodbye Sleep".

The following scenario is possible:

1. Your first signal handler's `printf` fills "Hello " into its internal buffer, and prints it. But then, just before it puts the rest of the sentence ...
2. Your second signal handler interrupts your first signal handler, and prints its full sentence "Goodbye Sleep".
3. When you are back to the first handler, you finish printing "World", and your final output is now: "Hello Goodbye SleepWorld". That's not what you wanted.

The problem here is that `printf` – and most other functions – are not assuming that their internal state is shared with anyone else. These functions were not designed to execute concurrently with a second instance of themselves. In other words, they are not *reentrant*.

It is important that you only call *reentrant functions* inside signal handlers. A quick manual page check or a Google search reveals whether your function is reentrant or not.