Theo Song
Prof. Hakner
ECE 357
10/30/2019

Unix: CentOS  Architecture: x86_64  Kernel: 3.10

## Problem 1 - Signal Numbers & Behavior

1) SIGTSTP

2) Ctrl + \

3) stty intr ^I

4)
```
sigset_t set;
if(sigemptyset(&set)<0)
    perror("sigemtpyset\n");
if(sigaddset(&set, 60)<0)
    perror("sigaddset\n");
if(sigprocmask(SIG_UNBLOCK, &set, NULL)<0)
    perror("sigprocmask\n");
```

For signal #60, the handler function will run 32 times. Because signal #60 is a real time signal, meaning that it could be queued, the handler function will run as much as the signal is sent.
However, if it was signal#2, the handler function will run only once. Since signal #2 is a traditional signal, when a new signal(#2) is sent, it will overwrite the older signal, but will not queue. Thus, in the case of signal #2, the handler function will run only once.

# Problem 2 - File Descriptor Tables

1) It is because some of the write() syscall are actually successful, meaning that it fully wrote 65536 bytes into the pipe buffer, which will not print the pipe short write if such is the case. In order for the write() to be successful, a signal(SIGUSR1) must not change the write() into a READY state, while the write() is in the interruptible SLEEP state. The pipe short write is written when the SIGUSR1 interrupts the write() in SLEEP state, which has written some bytes of 4K chunks. When the write() is changed to the READY state due to the signal, not actually being ready, it returns the bytes it has written so far, which is the number shown in “pipe short write”.

2) All the short write values are a multiple of 4096(4K). This is because when the requested write size exceeds 4K, the write is separated into smaller chunks of 4K, which could be interrupted by other writes or signal.

3) EINTR is shown in some parts of the stderr. While the write is in the SLEEPING state, waiting for the read() to read the pipe buffer, the SIGUSR1 signal interrupts, transitioning write() to the READY state. Once scheduled, the kernel notice that it has been interrupted by a signal. Now, if the write() has done no work, and the SA_RESTART flag is set, the write() syscall is restarted after the f() sig_handler returns. However, without the SA_RESTART flag, the write() will be considered a failure and errno will be set to EINTR, after the f() sig_handler returns. And if the write() has done partial work, it will return the partial work it has done. That is why some shows EINTR, while other lines show the short write value.

4) The program will not terminate. When a child process terminates, a SIGCHLD is sent to the parent. The signal(SIGCHLD, SIG_IGN) is saying that the parent has no interest of the child, thus not making a zombie. However, without such statement, when the child process (w & r) terminates, it will remain in the ZOMBIE state, waiting for the parent process to claim the status, using the wait family of system calls. And because the parent process does not have any such, the child process (w) will remain in the ZOMBIE state, allowing the parent process to run the infinite loop of sending SIGUSR1, without an error, to the child process(w).