Exercise 4.1: Examining Signal Priorities and Execution

We give you a **C** program that includes a signal handler that can handle any signal. The handler avoids making any system calls (such as those that might occur while doing I/O). This file can be extracted from your downloaded SOLUTIONS file as signals.c

signals.c

```c
/*
 * Examining Signal Priorities and Execution.
 *
 * The code herein is: Copyright the Linux Foundation, 2014
 * Author: J. Cooperstein
 *
 * This Copyright is retained for the purpose of protecting free
 * redistribution of source.
 *
 * This code is distributed under Version 2 of the GNU General Public
 * License, which you should have received with the source.
 *
 @*/

#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

#define NUMSIGS 64

/* prototypes of locally-defined signal handlers */

void (sig_handler) (int);

int sig_count[NUMSIGS + 1];         /* counter for signals received */
volatile static int line = 0;
volatile int signumbuf[6400], sigcountbuf[6400];

int main(int argc, char *argv[])
{
        sigset_t sigmask_new, sigmask_old;
        struct sigaction sigact, oldact;
        int signum, rc, i;
        pid_t pid;

        pid = getpid();

        /* block all possible signals */
        rc = sigfillset(&sigmask_new);
        rc = sigprocmask(SIG_SETMASK, &sigmask_new, &sigmask_old);

        /* Assign values to members of sigaction structures */
        memset(&sigact, 0, sizeof(struct sigaction));
        sigact.sa_handler = sig_handler;        /* we use a pointer to a handler */
        sigact.sa_flags = 0;        /* no flags */
        /* VERY IMPORTANT */
        sigact.sa_mask = sigmask_new;        /* block signals in the handler itself  */

        /*
         * Now, use sigaction to create references to local signal
         * handlers * and raise the signal to myself
         */

        printf
            ("\nInstalling signal handler and Raising signal for signal number:\n\n");
        for (signum = 1; signum <= NUMSIGS; signum++) {
                if (signum == SIGKILL || signum == SIGSTOP || signum == 32
                    || signum == 33) {
                        printf("  --");
                        continue;
                }
                sigaction(signum, &sigact, &oldact);
                /* send the signal 3 times! */
                rc = raise(signum);
                rc = raise(signum);
                rc = raise(signum);
                if (rc) {
                        printf("Failed on Signal %d\n", signum);
                } else {
                        printf("%4d", signum);
                        if (signum % 16 == 0)
                                printf("\n");
                }
        }
        fflush(stdout);

        /* restore original mask */
        rc = sigprocmask(SIG_SETMASK, &sigmask_old, NULL);

        printf("\nSignal  Number(Times Processed)\n");
        printf("--------------------------------------------\n");
        for (i = 1; i <= NUMSIGS; i++) {
                printf("%4d:%3d  ", i, sig_count[i]);
                if (i % 8 == 0)
                        printf("\n");
        }
        printf("\n");

        printf("\nHistory: Signal  Number(Count Processed)\n");
        printf("--------------------------------------------\n");
        for (i = 0; i < line; i++) {
                if (i % 8 == 0)
                        printf("\n");
                printf("%4d(%1d)", signumbuf[i], sigcountbuf[i]);
        }
        printf("\n");
        exit(EXIT_SUCCESS);
}

void sig_handler(int sig)
{
```

```c
        sig_count[sig]++;
        signumbuf[line] = sig;
        sigcountbuf[line] = sig_count[sig];
        line++;
}
```

You will need to compile it and run it as in:

```
$ gcc -o signals signals.c
$ ./signals
```

When run, the program:

- Does not send the signals `SIGKILL` or `SIGSTOP`, which can not be handled and will always terminate a program.
- Stores the sequence of signals as they come in, and updates a counter array for each signal that indicates how many times the signal has been handled.
- Begins by suspending processing of all signals and then installs a new set of signal handlers for all signals.
- Sends every possible signal to itself multiple times and then unblocks signal handling and the queued up signal handlers will be called.
- Prints out statistics including:
  - The total number of times each signal was received.
  - The order in which the signals were received, noting each time the total number of times that signal had been received up to that point.

Note the following:

- If more than one of a given signal is **raised** while the process has blocked it, does the process **receive** it multiple times? Does the behavior of **real time** signals differ from normal signals?
- Are all signals received by the process, or are some handled before they reach it?
- What order are the signals received in?

One signal, `SIGCONT` (18 on **x86**) may not get through; can you figure out why?

Please Note

On some **Linux** distributions signals 32 and 33 can not be blocked and will cause the program to fail. Even though system header files indicate `SIGRTMIN=32`, the command `kill -l` indicates `SIGRTMIN=34`.

Note that **POSIX** says one should use signal names, not numbers, which are allowed to be completely implementation dependent.

You should generally avoid sending these signals.