Ejercicio 4.1 Examinar prioridades de las señalas y su ejecución

Le hemos proporcionado un programa en **C** que incluye un manejador de señales el que puede manejar cualquier señal. El manejador en cuestión evita hacer llamadas al sistema (tales como las que podrían ocurrir mientras se realizan operaciones de E/S).

```
/*
 * 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)
{
        sig_count[sig]++;
        signumbuf[line] = sig;
        sigcountbuf[line] = sig_count[sig];
        line++;
}
```

Si está tomando la versión autodidacta en línea de este curso, encontrará el código fuente disponible para su descarga en la pantalla **Lab**.

Necesitará compilarlo y ejecutarlo como se muestra a continuación:

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


Al ser ejecutado, el programa realiza lo siguiente:

- No envía las señales `SIGKILL` o `SIGSTOP`, las cuales no pueden ser manejadas y siempre finalizan un programa.
- Almacena la secuencia de señales a medida en que llegan y actualiza un arreglo de contadores para cada señal que indica cuántas veces la señal ha sido manejada.
- Comienza por suspender el proceso de todas las señales y luego instala un conjunto nuevo de manejadores de señal para todas ellas.
- Envía cada señal posible múltiples veces, luego desbloquea el manejo de señales e invoca a los manipuladores de señales que estaban en espera.
- Imprime las estadísticas, incluyendo:
  - El número total de veces que cada señal fue recibida.
  - El orden en el cual se recibieron las señales, señalando cada vez el número total de veces que la señal se había recibido hasta ese momento.

Tenga en cuenta lo siguiente:

- Si una señal determinada **se emite** en varias oportunidades mientras el proceso las había bloqueado, ¿el proceso las **recibe** múltiples veces? ¿El comportamiento de señales en **tiempo real** es diferente de las señales normales?
- ¿El proceso recibe todas las señales, o algunas de ellas son manejadas antes que lleguen a él?
- ¿En qué orden se reciben las señales?

La señal `SIGCONT` (18 en **x86** puede que no logre llegar a destino, ¿se le ocurre por qué?

Nota:

En algunas distribuciones **Linux** las señales 32 y 33 no pueden ser bloqueadas y causarán que el programa falle. A pesar de que los archivos de cabecera del sistema indican `SIGRTMIN=32`, el comando `kill -l` indica `SIGRTMIN=34`.

Tenga en cuenta que **POSIX** dice que se deberían usar nombres en vez de números, los cuales están habilitados para ser completamente dependientes de la implementación.

En general se debería evitar el envío de estas señales.

