Lab 8: Signals

In this lab, you'll implement your own successively-enhanced versions of the program `killme`. The final version will allow you to experiment with sending and receiving signals from one process to others, including child processes.

http://www.tricity.wsu.edu/~bobl/cpts360/lab08_signals

(There is also a link to it in the schedule on the course web page.) Calling it up in a browser will allow you to download all of the files you need for this lab. `writeup.html` in that directory contains an HTML version of this page.

Every part after Part 1 depends on the part that went right before it, so be sure that part is working before you move on.

Part 1: A Single Process `killme`

The first version of `killme` will set up handlers for (almost) every possible signal and then wait to handle one of them. Here, "handle" means printing out a notice of the event.

Download `killme_pt1_tplt.c` from the lab directory and rename it to `killme_pt1.c`. It looks like this:

```
#include <stdlib.h>    // for exit()
#include <stdio.h>     // for the usual printf(), etc.
#include <getopt.h>    // for getopt()
/*
 * ASSIGNMENT
 *
 * - "#include" any other necessary headers (as indicated by "man"
 *    pages)
 */


// To get `getopt_long()` to work, you need to provide a static
// (usually) array of `struct option` structures.  There are four
// members to be filled in:

// 1. `name` is a (char *) string containing the "long" option name
// (e.g. "--help" or "--format=pdf").

// 2. `has_arg` has one of these values that describe the
// corresponding option:
enum {
    NO_ARG  = 0, // the option takes no argument
    REQ_ARG = 1, // the option must have an argument
    OPT_ARG = 2  // the option takes an optional argument
};

// 3. The "flag" is an int pointer that determines how the function
// will return its value. If it is NULL, getopt_long() will return
// "val" (the fourth member) as its function return. If it is not
```

```
// NULL, getopt_long() will return 0 and set "*flag" to "val".

// 4. "val" is an int which is either a character to denote a "short"
// (e.g. "-h" or "-f pdf") option or 0, to denote an option which does
// not have a "short" form.

// The array is terminated by an entry with a NULL name (first
// element).

static struct option options[] = {
    // elements are:
    // name       has_arg   flag   val
    { "children", OPT_ARG,  NULL,  'c'},
    { "help",     NO_ARG,   NULL,  'h'},
    { "nosync",   NO_ARG,   NULL,  'n'},
    { "pgid",     NO_ARG,   NULL,  'g'},
    { "ppid",     NO_ARG,   NULL,  'p'},
    { NULL }  // end of options table
};

/*
 * These globals are set by command line options. Here, they are set
 * to their default values.
 */
int showPpids = 0;   // show parent process IDs
int showPgids = 0;   // show process group IDs
int synchronize = 1; // synchronize outputs (don't use until Part 3)


enum { IN_PARENT = -1 }; // must be negative
/*
 * In the parent, this value is IN_PARENT. In the children, it's set
 * to the order in which they were spawned, starting at 0.
 */
int siblingIndex = IN_PARENT;


// This is a global count of signals received.
int signalCount = 0;


void writeLog(char message[], const char *fromWithin)
// print identifying information about the current process to stdout
{
    /*
     * ASSIGNMENT
     *
     * - if `siblingIndex` is IN_PARENT,
     *     + set a string buffer `processName` to "parent" (hint: strcpy())
```

```
     *     + set `colonIndent` to 20
     * - otherwise,
     *     + set a string buffer `processName` to a string of the form
     *       "child #" where "#" is `siblingIndex` (hint: snprintf())
     *     + set `colonIndent` to 30
     * - use `colonIndent` to set the indent up to the ":"
     *   using this example, which prints `processName`:
     *
     *     printf("%*s: %s\n", colonIndent, "process name", processName);
     *
     * - print the process ID (hint: getpid(2)) indented as above
     * - if `showPpids` is true,
     *     + print the parent process ID (hint: getppid(2)) indented as above
     * - if `showPgids` is true,
     *     + print the process group ID (hint: getpgrp(2)) indented as above
     * - print `signalCount` indented as above (with a "%d" format, of course)
     * - print `message` indented as above
     * - print `fromWithin` indented as above
     * - print a blank line to separate this from other log entries
     *
     * (Note: The second argument to this function, `fromWithin`, should
     *  always be `__func__` (no quotes, just an identifier `__func__`).)
     */
}


void handler(int sigNum)
// handle signal `sigNum`
{
    /*
     * ASSIGNMENT
     *
     * - increment signalCount
     * - create a message that includes `sigNum` and its string
     *   equivalent (hint: snprintf(3) and strsignal(3))
     * - add an entry to the log that includes that message (hint:
     *   writeLog())
     *
     * Note: Although you'll be doing it here, It is not a safe
     * practice to call printf() or any other standard I/O function
     * from within (or below) a signal handler, for a reason that will
     * be made clear in the lab.
     */
}


void initSignals(void)
// initialize all signals
{
```

```
    /*
     * ASSIGNMENT
     *
     * - for every signal from 1 to _NSIG (NSIG on MacOS)
     *   except SIGTRAP and SIGQUIT (to make debugging easier),
     *     - try to set its handler to handler() (hint: signal(2))
     *       if this fails,
     *         - add an entry to the log with the signal number and
     *           its string equivalent (hint: strsignal(3) and
     *           writeLog())
     */
}


static void usage(char *progname)
// issue a usage error message
{
    eprintf("usage: %s [{args}]*\n", progname);
    eprintf("%s\n", " {args} are:");
    eprintf("%s",
        "  -g or --pgid                    list process group IDs\n"
        "  -n or --nosync                  turn off synchronization\n"
        "  -p or --ppid                    list parent PIDs (default: no)\n"
        "  -h or --help                    help (this message) and exit\n"
        );
    return;
}

int main(int argc, char **argv)
{
    int ch;
    static char *progname = "**UNSET**";

    /*
     * Parse the command line arguments.
     */
    progname = argv[0];
    for (;;) {
        ch = getopt_long(argc, argv, "c::ghnp", options, NULL);
        if (ch == -1)
            break;

        switch (ch) {

        case 'c':
            break;

        case 'g':
            showPgids = 1;
```

```
            break;

        case 'h':
            usage(progname);
            exit(0);

        case 'n':
            synchronize = 0;
            break;

        case 'p':
            showPpids = 1;
            break;

        default:
            printf("?? getopt returned character code 0x%02x ??\n", ch);
            exit(1);
        }
    }
    /*
     * ASSIGNMENT
     *
     * - initialize all signals (hint: initSignals())
     * - add an entry to the log that the parent is pause()'d for a
     *   signal (hint: writeLog())
     * - await a signal (hint: pause())
     */
    return 0;
}
```

Follow the instructions in the "ASSIGNMENT" comments (and only there) to implement the pseudocode.

Create a `Makefile` for this.

Test your `killme_pt1` as we did for `killme` in class: Run it in one window and send signals to with the `kill` shell command in a different window.

Be sure to include your `killme_pt1.c` in the submission tarball.

Part 2: A Multiprocess `killme`

The second version of `killme` will create a arbitrary number of child processes which will also handle (almost) every possible signal by printing out notice of them. This time, the parent and child processes can only be killed via SIGQUIT, SIGTRAP, or SIGKILL. (Remember that you can send SIGQUIT from the keyboard via "Ctrl-\".) Once they are all forked, the parent process will wait for all of its children to do exit.

Download `killme_pt2_tplt.c` from the lab directory and rename it to `killme_pt2.c`. It looks like this:

```
#include <stdlib.h>    // for exit()
#include <stdio.h>     // for the usual printf(), etc.
#include <getopt.h>    // for getopt()
```

```
/*
 * ASSIGNMENT
 *
 * - "#include" any other necessary headers (as indicated by "man"
 *    pages)
 */


// To get `getopt_long()` to work, you need to provide a static
// (usually) array of `struct option` structures.  There are four
// members to be filled in:

// 1. `name` is a (char *) string containing the "long" option name
// (e.g. "--help" or "--format=pdf").

// 2. `has_arg` has one of these values that describe the
// corresponding option:
enum {
    NO_ARG  = 0, // the option takes no argument
    REQ_ARG = 1, // the option must have an argument
    OPT_ARG = 2  // the option takes an optional argument
};

// 3. The "flag" is an int pointer that determines how the function
// will return its value. If it is NULL, getopt_long() will return
// "val" (the fourth member) as its function return. If it is not
// NULL, getopt_long() will return 0 and set "*flag" to "val".

// 4. "val" is an int which is either a character to denote a "short"
// (e.g. "-h" or "-f pdf") option or 0, to denote an option which does
// not have a "short" form.

// The array is terminated by an entry with a NULL name (first
// element).

static struct option options[] = {
    // elements are:
    // name       has_arg   flag   val
    { "children", OPT_ARG,  NULL,  'c'},
    { "help",     NO_ARG,   NULL,  'h'},
    { "nosync",   NO_ARG,   NULL,  'n'},
    { "pgid",     NO_ARG,   NULL,  'g'},
    { "ppid",     NO_ARG,   NULL,  'p'},
    { NULL }  // end of options table
};

/*
 * These globals are set by command line options. Here, they are set
 * to their default values.
```

```
 */
int showPpids = 0;   // show parent process IDs
int showPgids = 0;   // show process group IDs
int synchronize = 1; // synchronize outputs (don't use until Part 3)


enum { IN_PARENT = -1 }; // must be negative
/*
 * In the parent, this value is IN_PARENT. In the children, it's set
 * to the order in which they were spawned, starting at 0.
 */
int siblingIndex = IN_PARENT;


// This is a global count of signals received.
int signalCount = 0;


void writeLog(char message[], const char *fromWithin)
// print identifying information about the current process to stdout
{
    /*
     * ASSIGNMENT
     *
     * - Insert your previous writeLog() code here
     */
}


void inChild(int iSibling)
// do everything that's supposed to be done in the child
{
    /*
     * ASSIGNMENT
     *
     * - set (the global) `siblingIndex` to `iSibling`
     * - add an entry to the log that includes a message
     *   that the process is "pause()d" (hint: writeLog())
     * - in an infinite loop,
     *   + call pause(2) to wait for the next signal
     */
}


void handler(int sigNum)
// handle signal `sigNum`
{
    /*
     * ASSIGNMENT
```

```
     *
     * - insert your previous handler() code here unchanged
     */
}


void initSignals(void)
// initialize all signals
{
    /*
     * ASSIGNMENT
     *
     * - insert your previous initSignals() code here unchanged
     */
}


void inParent(void)
// do everything that's supposed to be done in the parent
{
    /*
     * ASSIGNMENT
     *
     * - add an entry to the log that the parent is waiting for children
     *   to die (hint: writeLog())
     * - as long as there are child processes to wait upon (hint: wait(2)),
     *     + if the child exited normally (hint: WIFEXITED()),
     *         ~ add an entry to the log that the child exited normally and
     *           include its process ID and status (hint: writeLog())
     *     + otherwise,
     *         ~ add an entry to the log that the child exited abnormally
     *           and include its process ID (hint: writeLog())
     *     + if the child was signaled (hint: WIFSIGNALED()),
     *         ~ add an entry to the log that the child was terminated
     *           by a signal and include the signal number and its
     *           string equivalent (hint: strsignal(3) and writeLog())
     * - if wait() caused an error because there were no children to wait for,
     *     + add an entry to the log to that effect (hint: writeLog())
     * - otherwise
     *     + add an entry to the log that wait() failed for an unknown
     *       reason (hint: writeLog())
     */
}


static void usage(char *progname)
// issue a usage error message
{
    eprintf("usage: %s [{args}]*\n", progname);
```

```
    eprintf("%s\n", " {args} are:");
    eprintf("%s",
        "  -c[{arg}] or --children[={arg}]  fork {arg} children (default: 1)\n"
        "  -g or --pgid                     list process group IDs\n"
        "  -n or --nosync                   turn off synchronization\n"
        "  -p or --ppid                     list parent PIDs (default: no)\n"
        "  -h or --help                     help (this message) and exit\n"
        );
    return;
}

int main(int argc, char **argv)
{
    int ch;
    int nSiblings = 0;
    static char *progname = "**UNSET**";

    /*
     * Parse the command line arguments.
     */
    progname = argv[0];
    for (;;) {
        ch = getopt_long(argc, argv, "c::ghnp", options, NULL);
        if (ch == -1)
            break;

        switch (ch) {

        case 'c':
            if (optarg)
                nSiblings = atoi(optarg);
            else
                nSiblings = 1;
            break;

        case 'g':
            showPgids = 1;
            break;

        case 'h':
            usage(progname);
            exit(0);

        case 'n':
            synchronize = 0;
            break;

        case 'p':
            showPpids = 1;
```

```
            break;

        default:
            printf("?? getopt returned character code 0x%02x ??\n", ch);
            exit(1);
        }
    }
    /*
     * ASSIGNMENT
     *
     * (We'll use the [*] to mark steps that are unchanged from Part 1.)
     *
     * - initialize all signals (hint: initSignals()) [*]
     * - if no children are being spawned,
     *     + add an entry to the log that the parent is pause()'d for a
     *       signal (hint: writeLog()) [*]
     *     + in an infinite loop,
     *         ~ await a signal (hint: pause()) [*]
     * - otherwise,
     *     + for each of the `nSiblings` siblings whose loop index is
     *       `iSibling`,
     *         ~ fork the parent (hint: fork())
     *         ~ in the child,
     *             : invoke inChild(iSibling)
     *         ~ in the parent,
     *             : add an entry to the log that the parent forked a
     *               child, including `iSibling` and its process ID
     *               (hint: snprintf())
     *     + invoke inParent()
     */
    return 0;
}
```

Follow the instructions in the comments, copying working code from Part 1 where indicated and adding new code as indicated.

Modify your `Makefile` to compile `killme_pt2` as well `killme_pt1`.

Test your `killme_pt2` as you did `killme_pt1`, except that this time you can send signals to the children as well as the parent.

When you run your `killme_pt2` with more than a few children, you'll notice that the log messages are interleaved or even garbled. This is because all processes, parent and children, are trying to write to the same device at the same time (even though each of them has their own standard I/O buffer -- remember "copy-on-write").

This is to be expected. We'll fix it in Part 3.

Be sure to include your `killme_pt2.c` in the submission tarball.

Part 3: A Synchronized Multiprocess `killme`

The third version of `killme` fixes the interleaved output problem using a mechanism called a *critical section*. A critical section acts like a fence around a piece of code that several processes have access to and guarantees that only one process will be inside the fence at one time. One way to create a critical section is to use software tools called *semaphores*, which we'll explore in an upcoming unit, but for now we'll treat them as a black box.

Download `killme_pt3_tplt.c`, `critical_section.h`, and `critical_section.o` from the lab directory.

Rename `killme_pt3_tplt.c` to `killme_pt3.c`. It looks like this:

```
#include <stdlib.h>    // for exit()
#include <stdio.h>     // for the usual printf(), etc.
#include <getopt.h>    // for getopt()
/*
 * ASSIGNMENT
 *
 * - "#include" any other necessary headers (as indicated by "man"
 *    pages)
 */

/*
 * Note the new #include
 */
#include "critical_section.h"


// To get `getopt_long()` to work, you need to provide a static
// (usually) array of `struct option` structures.  There are four
// members to be filled in:

// 1. `name` is a (char *) string containing the "long" option name
// (e.g. "--help" or "--format=pdf").

// 2. `has_arg` has one of these values that describe the
// corresponding option:
enum {
    NO_ARG  = 0, // the option takes no argument
    REQ_ARG = 1, // the option must have an argument
    OPT_ARG = 2  // the option takes an optional argument
};

// 3. The "flag" is an int pointer that determines how the function
// will return its value. If it is NULL, getopt_long() will return
// "val" (the fourth member) as its function return. If it is not
// NULL, getopt_long() will return 0 and set "*flag" to "val".

// 4. "val" is an int which is either a character to denote a "short"
// (e.g. "-h" or "-f pdf") option or 0, to denote an option which does
// not have a "short" form.
```

```
// The array is terminated by an entry with a NULL name (first
// element).

static struct option options[] = {
    // elements are:
    // name       has_arg   flag   val
    { "children", OPT_ARG,  NULL,  'c'},
    { "help",     NO_ARG,   NULL,  'h'},
    { "nosync",   NO_ARG,   NULL,  'n'},
    { "pgid",     NO_ARG,   NULL,  'g'},
    { "ppid",     NO_ARG,   NULL,  'p'},
    { NULL }  // end of options table
};

/*
 * These globals are set by command line options. Here, they are set
 * to their default values.
 */
int showPpids = 0;   // show parent process IDs
int showPgids = 0;   // show process group IDs
int synchronize = 1; // synchronize outputs (don't use until Part 3)


enum { IN_PARENT = -1 }; // must be negative
/*
 * In the parent, this value is IN_PARENT. In the children, it's set
 * to the order in which they were spawned, starting at 0.
 */
int siblingIndex = IN_PARENT;


// This is a global count of signals received.
int signalCount = 0;


void writeLog(char message[], const char *fromWithin)
// print identifying information about the current process to stdout
{
    /*
     * ASSIGNMENT
     *
     * - Insert your previous writeLog() code here with this
     *   modification: If the global `synchronize` flag is set, call
     *   criticalSection_enter() before the first printf() call and
     *   criticalSection_leave() after the last one.
     */
}
```

```
void inChild(int iSibling)
// do everything that's supposed to be done in the child
{
    /*
     * ASSIGNMENT
     *
     * - insert your previous inChild() code here unchanged
     */
}


void handler(int sigNum)
// handle signal `sigNum`
{
    /*
     * ASSIGNMENT
     *
     * - insert your previous handler() code here unchanged
     */
}


void initSignals(void)
// initialize all signals
{
    /*
     * ASSIGNMENT
     *
     * - insert your previous initSignals() code here unchanged
     */
}


void inParent(void)
// do everything that's supposed to be done in the parent
{
    /*
     * ASSIGNMENT
     *
     * - insert your previous inParent() code here unchanged
     */
}


static void usage(char *progname)
// issue a usage error message
{
    eprintf("usage: %s [{args}]*\n", progname);
    eprintf("%s\n", " {args} are:");
```

```
    eprintf("%s",
        "  -c[{arg}] or --children[={arg}]  fork {arg} children (default: 1)\n"
        "  -g or --pgid                     list process group IDs\n"
        "  -n or --nosync                   turn off synchronization\n"
        "  -p or --ppid                     list parent PIDs (default: no)\n"
        "  -h or --help                     help (this message) and exit\n"
        );
    return;
}

int main(int argc, char **argv)
{
    int ch;
    int nSiblings = 0;
    static char *progname = "**UNSET**";

    /*
     * Parse the command line arguments.
     */
    progname = argv[0];
    for (;;) {
        ch = getopt_long(argc, argv, "c::ghnp", options, NULL);
        if (ch == -1)
            break;

        switch (ch) {

        case 'c':
            if (optarg)
                nSiblings = atoi(optarg);
            else
                nSiblings = 1;
            break;

        case 'g':
            showPgids = 1;
            break;

        case 'h':
            usage(progname);
            exit(0);

        case 'n':
            synchronize = 0;
            break;

        case 'p':
            showPpids = 1;
            break;
```

```
        default:
            printf("?? getopt returned character code 0x%02x ??\n", ch);
            exit(1);
        }
    }
    /*
     * ASSIGNMENT
     *
     * - Insert your previous main() code here unchanged, except that
     *   if the global `synchronize` flag is set, add a call to
     *   criticalSection_init() before the initializeSignals() call.
     */
    return 0;
}
```

`critical_section.h` indicates how each function should be used:

```
#ifndef INCLUDED_CRITICAL_SECTION

/*
 * Call this function once at the start of the program to initialize
 * (but not enter or leave) the critical section.
 */
extern void criticalSection_init(void);

/*
 * Call this function to enter the critical section, possibly waiting
 * to do so.
 */
extern void criticalSection_enter(void);

/*
 * Call this function to leave the critical section. Be sure you've
 * entered it first!
 */
extern void criticalSection_leave(void);

#define INCLUDED_CRITICAL_SECTION
#endif
```

Follow the instructions in the comments, copying working code from Part 2 and adding new code as indicated. It's a good idea to make critical sections as small as possible for efficiency, to minimize waiting time for processes not in them.

Modify your `Makefile` to compile and link `killme_pt3` as well as `killme_pt1` and `killme_pt2`. Note that the `killme_pt3` link will require `critical_section.o`.

Test your `killme_pt3` as you did `killme_pt2`. You should now find that no matter how many children there are, the log records are not interleaved or garbled.

Be sure to include `critical_section.h`, `critical_section.o` (both unchanged) and your `killme_pt3.c` and final `Makefile` in the submission tarball.

Submit the tarball via Canvas.