Lab 6 and 7 - Signals

Signal

- A **signal** is a way to notify a process that an event has occurred
- There is a list of defined signals that can be sent (or you can define your own): SIGINT, SIGSTOP, SIGKILL, SIGCONT, etc.
- A signal is really a number (e.g. SIGINT is 2)
- A program can do something in response to a type of signal being received
- Signals are sent either by the operating system, or by another process
- You can send a signal to yourself or to another process you own

Signals

Here are some examples of signals:

- SIGINT - when you type Ctl-c in the terminal, the kernel sends a SIGINT to the foreground process group. The default behavior is to terminate.
- SIGTSTP - when you type Ctl-z in the terminal, the kernel sends a SIGTSTP to the foreground process group. The default behavior is to halt it until it is told to continue.
- SIGSEGV - when your program attempts to access an invalid memory address, the kernel sends a SIGSEGV ("seg fault"). The default behavior is to terminate.

Process Lifecycle

- **Running** - a process is either executing or waiting to execute
- **Stopped** - a process is suspended due to receiving a SIGSTOP or similar signal. A process will resume if it receives a SIGCONT signal.
- **Terminated** - a process is permanently stopped, either due to finishing, or receiving a signal such as SIGSEGV or SIGKILL whose default behavior is to terminate the process.

Sending Signals

- The operating system sends many signals, but we can also send signals manually.

```
int kill(pid_t pid, int signum);
// same as kill(getpid(), signum)
int raise(int signum);
```

- kill sends the specified signal to the specified process (poorly-named; previously, default was to just terminate target process)
- pid parameter can be > 0 (specify single pid), < -1 (specify process group abs(pid)), or 0/-1 (we ignore these).
- raise sends the specified signal to yourself

Signal Handlers

- We can have a function of our choice execute when a certain signal is received.
- We must register this "signal handler" with the operating system, and then it will be called for us.

```
typedef void (*sighandler_t)(int);
...
sighandler_t signal(int signum, sighandler_t handler);
```

- *signum* is the signal (e.g. SIGCHLD) we are interested in.
- *handler* is a function pointer for the function to call when this signal is received.

Note: no handlers allowed for SIGSTOP or SIGKILL.

SIGCHLD

- When a child changes state, the kernel sends a *SIGCHLD* signal to its parent.
- This allows the parent to be notified its child has e.g. terminated while doing other work.
- we can add a SIGCHLD handler to clean up children without waiting on them in the parent!

Waiting For Signals

- Signal handlers allow us to do other work and be notified when signals arrive. But this means the notification is unpredictable.
- A more predictable approach would be to designate times in our program where we stop doing other work and handle any pending signals.
    - benefits: this allows us to control when signals are handled, avoiding concurrency issues
    - drawbacks: signals may not be handled as promptly, and our process blocks while waiting
- We will not have signal handlers; instead we will have code in our main execution that handles pending signals.

sigwait()

- **sigwait()** can be used to wait (block) on a signal to come in:

```
int sigwait(const sigset_t *set, int *sig);
```

- set: the location of the set of signals to wait on
- sig: the location where it should store the number of the signal received the return value is 0 on success, or > 0 on error.
- Note: Cannot wait on SIGKILL or SIGSTOP, nor synchronous signals like SIGSEGV or SIGFPE.

sigprocmask()

- The **sigprocmask** function lets us temporarily block signals of the specified types. Instead, they will be queued up and delivered when the block is removed.

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

- *how* is
  - SIG_BLOCK (add this to the list of signals to block)
  - SIG_UNBLOCK (remove this from the list of signals to block)
  - SIG_SETMASK (make this the list of signals to block)
- *set* is a special type that specifies the signals to add/remove/replace with *oldset* is the location of where to store the previous blocked set that we are overwriting.

Note: forked children inherit blocked signals! We may wish to remove a block in the child.

Programming Exercises

1. sigint.cpp : This program installs a SIGINT handler to catch the SIGINT (Ctrl+c) instead of the default behavior of terminating the program. (Use Ctrl+z instead to stop the program, and then `kill -9 [PID]` to terminate it (you can use SIGKILL instead of 9)).
2. sig_children.cpp : This program illustrates how a SIGCHLD handler can be used to reap background child processes (and have it work when all background processes take varying lengths of time to complete).
3. sigwait.cpp : This program is the same as sigint.cpp but instead of using a signal handler to handle SIGINT it waits for SIGTSTP signals using sigwait and then prints out a message to the user instead of the default behavior of terminating the program. (Use Ctl-c instead to stop the program).
4. sig_children_2.cpp : This program illustrates how we can wait for SIGCHLD signals using sigwait instead of a signal handler to clean up background child processes.