CMPSC 311

Scribe Notes Process Management – Processes and Signals (Part 2)

Lecture 16 – 02/ 19/ 2016

Dhruva Sharma, Chris Sturges, Meirui (Mary) Sha, Gabe Stanton

OVERVIEW:

In lecture 15, we learned about process termination, process groups, and were briefly introduced to signals. A signal is a software interrupt that the operating system generates to detect asynchronous/exceptional situations to the program. Lecture 16 continues on to talk more about different signal types and their functionality.

We covered the following topics on signals:

- *Signal Handler*
- *Catching Signals*
- *Waiting for Signals*
- *Signal Safety*
- *Real-time Signals*
- *Queuing Signals*

Catching Signals

Continuing a bit from the last lecture, we can actually "catch" signals in a program's process. A user may want to "catch" a signal if they want specific instructions at certain times in a program. In order to catch a signal, a signal handler is required which must then be set up in such a way that the it calls the signal handler that returns the caught signal.

What is a signal handler? And how does it work?

A *signal handler* is a "user-defined" function, which is compiled together with rest of the user's program, that is then called when a signal arrives.

Example:

```
void (*sa_handler)(int)
        SIG_DFL  : default signal handler
        SIG_IGN  : ignore given signal
```

Catching Signals using "*sigaction*" statement

Types of "*struct sigaction*":

- *sa_handler* : the signal handler
- *sa_mask* : more signals
- *sa_flags* : reserved flags and options for signals
- *sa_sigaction* : real-time signal handler (covered later)

Example:

```
int sigaction(int sig, struct sigaction *action, struct sigaction *old_action);
```

- *sig* : defines name of the signal
- *action* : calls action to be taken
- *old_action* : older action given with the signal

Waiting for Signals

Signals can be used to wait for an event without busy waiting. Busy waiting means using CPU cycles for essentially doing nothing. During busy waiting, the CPU cycles will be constantly testing for an event to occur, which usually will be happening by checking a variable in a loop. This is obviously very inefficient. There are two approaches that can use signals in order to fix this inefficiency.

Approach 1: set a signal handler for a specific signal and wait until the signal is caught. There are 2 functions that use this approach.

The *pause* function

```
#include <unistd.h>
int pause(void);
```

The *pause* function stops the calling thread and waits until a signal arrives. The signal will specify an action that will either terminate the process or execute a signal handler. The function will return when the return of the signal handler occurs. With the *pause* function, the function will give no information as to which signal caused it to return. Therefore, the signal handler will set a flag in order to check which signal arrived after *pause* returns. However, *pause* will not work if the signal is blocked. Therefore, the signal must be unblocked immediately before calling *pause* in order to be processed by the handler function.

The *sigsuspend* function

```
#include <signal.h>
int sigsuspend(const sigset_t *sigmask);
```

The *sigsuspend* function does the operations of *pause* function atomically. *Sigsuspend* suspends the process until a signal is caught, and then `sigmask` atomically unblocks the signal. When *sigsuspend* returns, *sigmask* resets.

Approach 2: Block a signal and wait for a signal to be generated.

The *sigwait* function

```
#include <signal.h>
int sigwait(sigset_t *set, int *signo)
```

The *sigwait* function waits for a set of signals instead of only one. The set of signals is stored in the address that is pointed to by `set`. The function returns when one of the signals becomes pending and removes this signal from the set. Then the signal number is assigned to the address pointed to by `signo`. The biggest difference between *sigwait* and the above two functions is that no signal handler is needed and that all signals are blocked before calling *sigwait*. Because no signal handler is needed, this is why the signals can be blocked.

Example of *sigwait* which counts the number of SIGINT signals delivered

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

int main(void) {
        int signalcount = 0;
        int signo;
        int signum = SIGINT;
        sigset_t sigset;
        if ((sigemptyset(&sigset) == -1) ||
                        (sigaddset(&sigset, signum) == -1) ||
                        (sigprocmask(SIG_BLOCK, &sigset, NULL) == -1))
                perror("Failed to block signals before sigwait");
                fprintf(stderr, "This process has ID %ld\n",
(long)getpid());
        for ( ; ; ) {
                if (sigwait(&sigset, &signo) == -1) {
                        perror("Failed to wait using sigwait");
                        return 1;
                }
                signalcount++;
                fprintf(stderr, "Number of signals so far: %d\n",
signalcount);
    }
}
```

Signal Safety

Signals are “asynchronous” - they can be sent without regard to the system clock or the program’s planned progression, even in the middle of a function. A signal handler can run concurrently with another function. Because of this, not all library functions are asynchronous-signal safe. Async-signal safe functions are those functions which are safe to call within a signal handler. Some examples of non async-signal safe functions are malloc() and free().

- int raise(int signo);
  - can be used to raise the specified signal
  - returns 0 if success, nonzero otherwise

Signals also have some limitations. In particular, if several signals occur at once, as might happen if there are several pending signals when an unblock function is called, only one will be handled. The rest are lost. Also, the user cannot pass data to signal handler functions, which limits the ability to use signals as a message passing system between processes. Finally, the user is only given two user-defined signals to work with: SIGUSR1 and SIGUSR2

For these reasons, if one is not careful with signals, they can easily cause runtime errors and/or crashes for the user.

Realtime Signal Handlers

In certain scenarios, it is most helpful to use realtime signal handlers, which, as their name implies, are sent in realtime, as a process is running. This can be helpful when processes operating in different parts of a system need to pass integer values. Realtime signal handlers can also be used to more efficiently send timer interrupts, to make I/O multiplexing easier by sending file descriptors with I/O ready signals, or to easily signal Asynchronous I/O by returning a data pointer with the signal.

- `void func(int signo, siginfo_t *info, void *context);`

Parameters:

- `signo:` provides the signal number
- `info:` Contains information about signal data
- `context:` Undefined

`siginfo_t:`

- `si_signo:` Signal number (same as signo)
- `si_code:` How signal was generated
  - The `si_code` tells how the signal was generated
- `Si_value:` Data generated with the signal
  - Takes the form of a union of int and (void *)

Realtime Signal Example: Sender

```
Int pid;
Union sigval value;

/* Set value to send with signal */
Value.sival_int = 1;

/* Send signal to be queued */
Sigqueue(pid, SIGRTMIN, value);
```

Realtime Signal Example: Receiver

```
/* Signal handler */
void myhandler(int signo, siginfo_t *info,
            void *context) {
      int val = info->si_value.sival_int;
      printf("Signal: %d, value: %d\n", signo, val);
}
struct sigaction act;
act.sa_sigaction = myhandler; /* Set RT sig handler */
sigemptyset(&act.sa_mask);
act.sa_flags = SA_SIGINFO; /* RT sigs flag */

/* Install the signal handler for SIGRTMIN */
Sigaction(SIGRTMIN, &act, NULL);
```

Waiting for RT Signals: `sigwaitinfo`

Sigwaitinfo is similar to `sigwait`, but it waits for a specific signal to become pending, specified by the info passed with the signal, before removing it from the set of pending signals and returning. This allows for more pointed and specific usage for triggering with signals between processes.

- `int sigwaitinfo(sigset_t *set, siginfo_t *info);`

Parameters:

- `set:` Set provides the set of signals to wait for
- `info:` Contains info about signal data

Queuing Signals

```
#include <signal.h>
int sigqueue(pid_t pid, int signo, const union sigval value);
```

The `sigqueue` function sends signal `signo` with value `value` to the process ID `pid`. If `SA_SIGINFO` was not set for `signo`, then the signal is sent and not queued. However, if `SA_SIGINFO` was set when the handler of `signo` was installed, then the signal is queued.