by Bob Japenga (USA)

Concurrency in Embedded Systems (Part 7)

Using Signals in Embedded Linux

What are software signals and how can you use them to effectively communicate between a software system's concurrent operations? This article explains how and when to use signals. It also explores using signals vs. message queues or FIFOs and the difference between standard and real-time signals.

EMBEDDED IN THIN SLICES

Early in my embedded software designer career, I sometimes used the MS-DOS operating system (OS) in my designs. (Can you really call MS-DOS an OS?) During that time, I encountered a C library function called "signal." For more years than I care to admit, much to my detriment, I didn't understand that function and avoided using it in my designs.

This article examines what software signals are and when to use them, and includes some examples of how my company uses signals in embedded Linux. Specifically, as is the pattern in this series, we will be looking at POSIX-compliant signals.

WHAT ARE SIGNALS?

In simple terms, a software signal is another means of communication between a software system's concurrent operations. In Linux, the concurrent operations used in applications are called processes and threads (see Part 4 of this article series for more information about threads and processes). Part 6 examined message queues and first-ins, first-outs (FIFOs) as a means to communicate information between processes and threads. Both messages queues and FIFOs are created by you, the designer, for inter-process communication (IPC) between your threads and processes.

Signals provide another mechanism for you to signal to your other threads that something happened. For example, you could use a signal to tell another thread that a valve has opened or that data is available in a particular device. In one of our designs, we periodically receive data via an IR receiver from a device embedded in the human body indicating aortic pressure and electrical activity from the heart. The embedded device sends five pieces of information from the heart every millisecond. We use a signal to wake up the thread that will process that data.

One other difference between signals and mechanisms such as message queues and FIFOs is that the OS can (and does) send signals to your processes and threads. You don't have to do anything and your software will receive signals from the OS. For instance, if the user kills your program, the OS will send it a SIGKILL signal. Or, if your OS is equipped to detect powerdown, it will send a SIGPWR signal.

As a designer, all signals have default behavior so you need not do anything with them (as I did with my early MS-DOS designs). But, to design robust systems, you should know what signals can be sent to your programs, what the default behavior is, and what to do when you receive these signals.

In some sense, OS-generated signals are like error interrupt vectors at the hardware level. You can choose to ignore them or reboot them, but for good robust design you need to know what they are and decide how to handle them in each case. Every instantiation of Linux is slightly different as to what signals it can generate. You should know all the signals that can come to you and how you want to handle them. Usually this information can be found in the architecture-specific documentation for your version of Linux.

For example, SIGFPE is the floating-point exception signal and would only be present if you have floating-point (software or hardware) present in your system. The default behavior in most Linux systems for this signal is to terminate your process and perform a core dump. That may not be what you want to do in your system. Perhaps you want

Listing 1—This code sends a notification to send the signal to the pump process.

```
union sigval signalval;          // This can be a pointer or an integer
signalval.sival_int = seq_num++; // This is used to keep track of sequence
                                 // and is the data sent with the signal
rc = sigqueue( pump_process_pid, SIGUSR1, signalval );
```

Listing 2—The pump process waits for the signal.

```
// Place to store information from the signal
siginfo_t sig_info;
// Setup values for the timer
long val_msec = timeout_msec % MSEC_PER_SEC;
long val_sec  = timeout_msec / MSEC_PER_SEC;
const struct timespec sig_timeout =
   {.tv_sec = val_sec, .tv_nsec = (long)val_msec * NSEC_PER_MSEC};
// Signal we are waiting for: SIGUSR1
int user_signal = SIGUSR1;
// suspend execution until SIGUSR1 is set or timeout_msec's and put the info
into sig_info
int signal_id    = sigtimedwait(&user_signal, &sig_info, &sig_timeout);
// signal_id == -1 if failed
// errno       == EAGAIN if the signal did not come in before the timeout
// errno       == EINTR if the wait was interrupt by a signal handler!
if ( signal_id == -1 && errno == EAGAIN)
{
   // Process the time out
}
else if ( signal_id == -1 &&  errno == EINTR)
{
   // Process the interruption
}
```

to notify the user, log an error, and terminate and restart your application.

Another example is the SIGSEGV signal (i.e., segment fault), which happens if your program references memory outside its permitted address space. The default behavior is to terminate your process. But that can be maddening, because it doesn't tell you where the fault occurs. Installing a handler for this signal can help you decide what happens when your program does this. (Of course, your software never exhibits such aberrant behavior!) If I was more savvy in my early days, I would have understood the "signal" function and I would have installed signal handlers in my MS-DOS systems to handle segment faults and floating-point errors.

Signals can be handled in one of three ways (technically called their "disposition"). Signals can be ignored, they can be handled with a user-defined function, or they can enable the OS to take its default action. A signal's disposition can be modified in real time.

In the POSIX standard and in Linux, there are two kinds of signals: standard signals that are typically generated by the OS and real-time signals that are typically generated by the application programmer. Real time is a bit of a misnomer, since standard signals are just as "real time" as are their real-time cousins. They might better be named "user-defined signals." Almost all standard signals (there are two user-defined ones) have a predefined meaning, whereas real-time signals are user-defined.

There are slight but important operational differences between standard and real-time signals. Identical real-time signals are queued whereas standard signals are not. For example, if three identical real-time signals are generated within microseconds of each other, all three will be received in order by the application. However, if three identical standard signals are generated before they are processed, the application will only receive one. In one of our designs, we use a standard signal (SIGUSR1) instead of a real-time signal because we don't care if we miss one signal and don't want to queue them up if they get missed.

An integer piece of data can be included only with a real-time signals. This makes real-time signals a convenient and more efficient mechanism than FIFOs or message queues for transmitting a small amount of data between concurrent operations. But you need to be aware of certain quirks about signals. Finally, real-time signals are processed in a predefined order by the application, whereas the POSIX implementation can process standard signals in any order.

WHEN TO USE SIGNALS

Obviously, you have no choice with signals that come from the OS (i.e., standard signals). You cannot choose to have the OS send you a message when you get a segment fault or illegal instruction—you receive a standard signal. Here, you need to decide when you need to write a special handler for these signals. My recommendation is to do this when you need something other than the default operation.

However, when you are faced with a decision to choose a mechanism for IPC, should you use message queues or signals? Obviously, if there is more than an integer's worth of data to be communicated, you should use a message queue, a FIFO, a socket, or shared memory. The other distinction is that message queues and FIFOs are inherently synchronous methods of communication. The receiving process only gets the information when it wants. Signals are designed for asynchronous data transfer. Thus, though simple in implementation, their asynchronous nature can create some additional complexity.

However, as designers, we also use signals when they are built into the C library's functions. For example, you can design your system to use the `pselect` function to wait on a signal generated from a change in a file descriptor (in Linux a file descriptor can apply to a regular file, a device driver, FIFOs, pipes, sockets, any serial port, etc.). The change can be that the "file" is ready to read or write or that an error has occurred. Since serial port drivers look like files under Linux, when data is available at the serial port, the "file" is ready to read and a signal is sent to your application.

For instance, we use the POSIX `pselect` function to be notified by a signal when a device has sent data over the serial port. We also use the `pselect` function to configure the system to be

notified when a file has changed. All general-purpose I/O (GPIO) looks like a file to a Linux application, so you don't need to poll the GPIO for a change. But, you can use the `pselect` function to trigger a signal when the GPIO has changed. POSIX also enables you to wait on a signal with a timeout so you can discontinue after a programmable period of time if the signal never arrives.

Timers are another common use of signals that is built into the OS. Interval timers can be set up under Linux (`setittimer` function or `timer_settime`), which will send a signal when the timer expires. In one case, it sends a fixed signal; in the other, the signal is user programmable.

HOW TO USE SIGNALS

Signals can be processed in one of two ways. You can create a signal handler that is invoked when the signal is triggered or use a `sigwait` or `sigtimedwait` function to block the software until the signal arrives in or a timeout occurs. The code snippets in Listing 1 and Listing 2 demonstrate how we use signals to notify a different process that an event has happened and how we handle that signal.

Listing 1 shows a notification that sends the signal to a pump process. In this case, the `sigqueue` function sends the SIGUSR1 signal to the pump process identified with the `pump_process_pid` process ID. A sequence number is passed to the pump process so it can record if it has dropped a signal. (Remember, the standard signals do not queue up if one is already queued.)

Listing 2 shows the pump process awaiting the signal. This code snippet sets up to wait for our signal with a timeout in milliseconds stored in the `timeout_msec` variable.

This code blocks the thread until the SIGUSR1 signal arrives or a timeout_msec has elapsed. It also illustrates one of the more nonintuitive portions of signal handlers.

In the Listing 2 code snippet, while waiting for the signal or the timeout, if another signal handler is invoked, the sigtimedwait will immediately return with a −1 return and with its errno header file set to EINTR. In this case, there is no way to know how much of the timer has expired.

The first time I discovered this, it caused me a lot of grief. For example, the delay with sleep or nanosleep functions can be terminated early when a signal handler interrupts a sleep. At least in these cases you can restart the nanosleep or sleep functions with the remaining time. There is no recourse with an interrupt sigtimedwait function.

UNDERSTANDING SIGNALS

We walk through life picking up regrets like lint in a dryer. One of my lightweight regrets in my design career was not understanding signals earlier. Even if you don't use them, your OSes are using them. We have looked at them in thin slices—just enough to whet your appetite.

Bob Japenga has been designing embedded systems since 1973. In 1988, along with his best friend, he started MicroTools, which specializes in creating a variety of real-time embedded systems. With a combined embedded systems experience base of more than 200 years, they love to tackle impossible problems together. Bob has been awarded 11 patents in many areas of embedded systems and motion control. You can reach him at rjapenga@microtoolsinc.com.

RESOURCES

B. Japenga, "Concurrency in Embedded Systems (Part 6): POSIX, FIFOs, and Message Queues," *Circuit Cellar* 273, 2013.

———, "Concurrency in Embedded Systems (Part 4): Introducing Linux and Concurrency," *Circuit Cellar* 269, 2013.