The new pselect() system call

March 24, 2006
This article was contributed by Michael Kerrisk.

Applications like network servers that need to monitor multiple file descriptors using `select()`, `poll()`, or (on Linux) `epoll_wait()` sometimes face a problem: how to wait until either one of the file descriptors becomes ready, or a signal (say, `SIGINT`) is delivered. These system calls, as it turns out, do not interact entirely well with signals.

A seemingly obvious solution would be to write an empty handler for the signal, so that the signal delivery interrupts the `select()` call:

```
static void handler(int sig) { /* do nothing */  }

int main(int argc, char *argv[])
{
    fd_set readfds;
    struct sigaction sa;
    int nfds, ready;

    sa.sa_handler = handler;     /* Establish signal handler */
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGINT, &sa, NULL);
    /* ... */
    ready = select(nfds, &readfds, NULL, NULL, NULL);
    /* ... */
```

After `select()` returns we can determine what happened by looking at the function result and `errno`. If `errno` comes back as `EINTR`, we know that the `select()` call was interrupted by a signal, and can act accordingly. But this solution suffers from a race condition: if the `SIGINT` signal is delivered after the call to `sigaction()`, but before the call to `select()`, it will fail to interrupt that `select()` call and will thus be lost.

We can try playing various games like setting a global flag within the signal handler and monitoring that flag in the main program, and using `sigprocmask()` to block the signal until just before the `select()` call. However, none of these techniques can entirely eliminate the race condition: there is always some interval, no matter how brief, where the signal could be handled before the `select()` call is started.

The traditional solution to this problem is the so-called self-pipe trick, often credited to D J Bernstein. Using this technique, a program establishes a signal handler that writes a byte to a specially created pipe whose read end is also monitored by the `select()`. The self-pipe trick cleverly solves the problem of safely waiting either for a file descriptor to become ready or a signal to be delivered. However, it requires a relatively large amount of code to implement a requirement that is essentially simple. (For example, a robust solution requires marking both the read and write ends of the pipe non-blocking.)

For this reason, the POSIX.1g committee devised an enhanced version of `select()`, called `pselect()`. The major difference between `select()` and `pselect()` is that the latter call has a signal mask (`sigset_t`) as an additional argument:

```
int pselect(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
            const struct timespec *timeout, const sigset_t *sigmask);
```

The `sigmask` argument specifies a set of signals that should be blocked during the `pselect()` call; it overrides the current signal mask for the duration of that call. So, when we make the following call:

```
ready = pselect(nfds, &readfds, &writefds, &exceptfds,
                timeout, &sigmask);
```

the kernel performs a sequence of steps that is equivalent to atomically performing the following system calls:

```
sigset_t sigsaved;

sigprocmask(SIG_SETMASK, &sigmask, &sigsaved);
ready = select(nfds, &readfds, &writefds, &exceptfds, timeout);
sigprocmask(SIG_SETMASK, &sigsaved, NULL);
```

For some time now, glibc has provided a library implementation of `pselect()` that actually uses the above sequence of system calls. The problem is that this implementation remains vulnerable to the very race condition that `pselect()` was designed to avoid, because the separate system calls are not executed as an atomic unit.

Using `pselect()`, we can safely wait for either a signal to be delivered or a file descriptor to become ready, by replacing the first part of our example program with the following code:

```
    sigset_t emptyset, blockset;

    sigemptyset(&blockset);         /* Block SIGINT */
    sigaddset(&blockset, SIGINT);
    sigprocmask(SIG_BLOCK, &blockset, NULL);

    sa.sa_handler = handler;        /* Establish signal handler */
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    sigaction(SIGINT, &sa, NULL);

    /* Initialize nfds and readfds, and perhaps do other work here */
    /* Unblock signal, then wait for signal or ready file descriptor */

    sigemptyset(&emptyset);
    ready = pselect(nfds, &readfds, NULL, NULL, NULL, &emptyset);
    ...
```

This code works because the `SIGINT` signal is only unblocked once control has passed to the kernel. As a result, there is no point where the signal can be delivered before `pselect()` executes. If the signal is generated while `pselect()` is blocked, then, as with `select()`, the system call is interrupted, and the signal is delivered before the system call returns.

Although `pselect()` was conceived several years ago, and was already publicized in 1998 by W. Richard Stevens in his *Unix Network Programming, vol. 1, 2nd ed.*, actual implementations have been slow to appear. Their eventual appearance in recent releases of various Unix implementations has been driven in part by the fact that the 2001 revision of the POSIX.1 standard requires a conforming system to support `pselect()`. With the 2.6.16 kernel release, and the required wrapper function that appears in the recently released glibc 2.4, `pselect()` also becomes available on Linux.

Linux 2.6.16 also includes a new (but nonstandard) `ppoll()` system call, which adds a signal mask argument to the traditional `poll()` interface:

```
int ppoll(struct pollfd *fds, nfds_t nfds, const struct timespec *timeout,
          const sigset_t *sigmask);
```

This system call adds the same functionality to `poll()` that `pselect()` adds to `select()`. Not to be left in the cold, the epoll maintainer has patches in the pipeline to add similar functionality in the form of a new `epoll_pwait()` system call.

There are a few other, minor differences between `pselect()` and `ppoll()` and their traditional counterparts. For example the type of the `timeout` is:

```
struct timespec {
    long tv_sec;        /* Seconds */
    long tv_nsec;       /* Nanoseconds */
};
```

This allows the timeout interval to be specified with greater precision than is available with the older system calls.

The glibc wrappers for `pselect()` and `ppoll()` also hide a couple of details of the underlying system calls.

First, the system calls actually expect the signal mask argument to be described by two arguments, one of which is a pointer to a `sigset_t` structure, while the other is an integer that indicates the size of that structure in bytes. This allows for the possibility of a larger `sigset_t` type in the future.

The underlying system calls also modify their `timeout` argument so that on an early return (because a file descriptor became ready, or a signal was delivered), the caller knows how much of the timeout remained. However, the respective wrapper functions hide this detail by making a local copy of the `timeout` argument and passing that copy to the underlying system calls. (The Linux `select()` system call also modifies its `timeout` argument, and this behavior is visible to applications. However, many other `select()` implementations don't modify this argument. POSIX.1 permits either behavior in a `select()` implementation.)

Further details of `pselect()` and `ppoll()` can be found in the latest versions of the `select(2)` and `poll(2)` man pages, which can be found here.

(Log in to post comments)