Experiences with kqueue

Ted Unangst <tedu@openbsd.org>

August 31, 2009

1 Introduction

The traditional unix polling interface is the *select* function call. Shortcomings in its design led to the *poll* interface, which is improved but very similar. Unfortunately, while poll is a little easier to use, in many cases the improvement is not that remarkable because it is built on the same underlying architecture. To truly improve things, a new interface that operated on a new principle was required.

kqueue was introduced into FreeBSD by Jonathan Lemon in 2000. It was then quickly ported to other BSD systems. For full details, see "Kqueue: A generic and scalable event notification facility".

1.1 A New Model

In constrast to *select* and *poll*, which rely on userland maintaining a list of interesting file descriptors and (as the name implies) polling the kernel as to their current state, *kqueue* pushes this work into the kernel. Userland provides the kernel with the list, then asks what objects have changed. This is an important difference. select and poll ask about the current state of the object, kqueue asks about objects that have changed states. This means that kqueue is not a drop in replacement, and can complicate usage, but it is this core change that allows kqueue to scale.

1.2 New features

While kqueue provides a different usage model, it also provides a more flexible one. The API was extended to work on more objects than just file descriptors. This allows one to use kqueue to unify application event handling. For example, a web server may need to manage a list of sockets connected to clients and a list of child processes. Traditionally, this would require using both poll for the sockets, and *wait* for the processes. kqueue can be used to manage both.

2 Usage

Unlike select or poll, kqueue maintains state in the kernel between calls. This makes it a little more challenging to use, as we have to be careful to sequence our calls correctly. First, we create a kqueue handle which we will use to communicate with the kernel. A userland process can use multiple kqueues, although I'm not aware of any reason you'd need more than one. Second, we register objects that we are interested in. Third, we ask the kernel if any state changes have occured for the registered objects. Finally, we may wish to unregister some objects after we are done with them. All the steps after the first can be repeated as necessary. A long running server will register new sockets as clients connects, process them, and finally unregister them, but this sequence of calls for one socket may be intermixed with calls for another socket. To say this another way, *poll* and *select* are level triggered, while *kqueue* is edge triggered.

2.1 pwait: A Short Example

To get a better understanding of queue, let's look at a brief example. Using kqueue on files would be too boring, so we'll look at an example that uses the process monitoring feature. PWAIT is an improved verion of your shell's wait command that can wait for any process ID, not just a child process. I've elided error handling (don't do that).

```
#include <sys/types.h>
#include <sys/event.h>
#include <sys/time.h>
#include <stdlib.h>
int
main(int argc, char **argv)
{
    int kq;
    struct kevent kev;
    pid_t pid;
    pid = strtonum(argv[1], 0, 100000, NULL);
    kq = kqueue();
    EV_SET(&kev, pid, EVFILT_PROC, EV_ADD, NOTE_EXIT, 0, NULL);
    kevent(kq, &kev, 1, NULL, 0, NULL);
    kevent(kq, NULL, 0, &kev, 1, NULL);
    return 0;
}
```

First, we include the necessary system headers. *time.h* is only really necessary if we want a timeout. We declare a few local variables, and initialize *pid* to the numeric value of the first argument. The next line creates a new *kqueue*, our handle for everything that follows. Now we initialize our kevent structure using *EV_SET*. This is the container for our message to the kernel. *pid* is the identifier for the object we are interested in. Processes are identified by their pid. Next we tell the kernel that our identifier is for a process, that we wish to add this kevent to the queue, and that we are interested in the process exit event. The last two arguments are not used in this example. Now, you will see two calls to *kevent* that look pretty similar. Remember how kqueue is used. First we must register objects, then we can query them. The same function is used for both tasks. The first call we pass in our kevent as the changelist parameter. Now the kernel will add this kevent to the monitored list. The second call we provide space for the kernel to write out any events that happened. Note that the first list is an in parameter and the second list is an out parameter. It is not necessary to use the same kevent for both calls. Also, the list (array) could be of arbitrary size, in this example, there's just one. We provide a NULL timeout to both calls; this tells kqueue to wait forever.

3 Dangers and Caveats

Now that we understand a bit about how kqueue works, we are in a position to learn some more about its sharp edges. As anyone familiar with select or poll and tell you, you may repeatedly call those functions and receive more or less the same results every time. If a socket has data available for reading, poll will return it. If you don't read from the socket and call poll again, it is still readable and poll will again return it. This is not the case with kqueue. The first call to kqueue will indicate the socket is ready for reading, but the second will not. Why not? Between the calls to kqueue, the socket's state has not changed. kqueue only records changes in state, it never actually reads the current state. To do so would mean returning to poll levels of scalability. The precise defintion of what constitutes a state change depends on the object. In the case of sockets, "available to read" is marked whenever new data arrives. So your code may work fine in testing if the other end of the socket writes more data between your kqueue calls, but once released into the variety of real world network situations the code will appear to hang. Be careful!

3.1 The librthread Reaper

Time for another example. This example is very similar to the pwait example, but it's from real code in the OpenBSD tree. librthread is an implementation of the pthreads thread library. One of the things it must do is manage the stacks of threads, creating and destroying them. Destroying the stack when a thread terminates is a bit tricky. The exiting thread can't do it, because it's still running on the thread. Another thread can do it, but needs to be sure the exiting thread is fully gone. The solution is to use kqueue.

```
void
_rthread_add_to_reaper(pid_t t, struct stack *s)
{
    struct kevent kc;
```

```
        int n;
        _rthread_debug(1, "Adding %d to reaper\n", t);
        EV_SET(&kc, t, EVFILT_PROC, EV_ADD|EV_CLEAR, NOTE_EXIT, 0, s);
        n = kevent(_rthread_kq, &kc, 1, NULL, 0, NULL);
        if (n == -1)
            _rthread_debug(0, "_rthread_add_to_reaper(): kevent %d\n", errno);
    }
```

The _rthread_add_to_reaper looks quite a bit like the pwait example. Note the addition of the last argument to EV_SET, s. We are putting a pointer to the thread's stack in the kevent. kqueue of course doesn't know what to do with this, but as we will see, it is returned to us later.

```
    void
    _rthread_reaper(void)
    {
        struct kevent ke;
        int n;
        struct timespec t;
        t.tv_sec = 0;
        t.tv_nsec = 0;
        for (;;) {
            n = kevent(_rthread_kq, NULL, 0, &ke, 1, &t);
            if (n == -1)
                _rthread_debug(0, "_rthread_reaper(): kevent %d\n", errno);
            else if (n == 0)
                break;
            else {
                _rthread_debug(1, "_rthread_reaper(): %d died\n", ke.ident);
                /* XXX check error conditions */
                _rthread_free_stack(ke.udata);
            }
        }
    }
```

The _rthread_reaper function corresponds to the last call to kevent with some error handling. As you can see, we extract the stack pointer from ke.udata and free it. In addition to just waiting on events, kqueue also provides us with a side channel to pass a little extra info when the event occurs. We use a timespec of 0 (not a NULL timespec) to indicate that we don't want to wait.

3.2 Another Caveat

So far, we haven't seen the code that calls the above two functions. I'll show you a diff instead.

```
    +_rthread_reaper();
```

```
      if (tid != _initial_thread.tid)
          _rthread_add_to_reaper(tid, stack);
     -_rthread_reaper();
      threxit(0);
```

This diff fixed an important bug in librthread that wasn't revealed until stress testing thread creation and joining. Why is the order of these two calls so important? Look closely at the argument to EV_SET and I bet you'll see it. When kqueue operates on a process, we identify it by pid. (In rthreads, threads are basically processes). But pids can be recycled, sometimes very quickly. So the following sequence was possible with the old code.

1. Thread 3456 is exiting.
2. 3456 registers itself with kqueue.
3. 3456 runs the reaper, nothing to do (nobody has exited yet).
4. 3456 exits.
5. A new thread 3456' is created.
6. 3456' is exiting.
7. 3456' registers with kqueue, updating the kevent with its own stack.
8. 3456' runs the reaper, finds the exit event from 3456 but the stack from 3456'.
9. 3456' frees its stack and dies.

The mayhem starts at step 7. The old kevent for 3456 is still in the kqueue. When 3456 exited, it was marked active, but nobody has received it yet. Because kevents are identified by pid, kqueue thinks we are talking about the same event and faithfully updates the udata pointer. Now it's possible for 3456' to receive the notification that a process has exited, but it finds its own stack in udata! The fix is to move the reaper call earlier, so that we are guaranteed that any ghosts with the same pid are cleared out before adding a new one.

Lesson learned: Get your identifiers right and make sure you know what they identify.

4 Final Remarks

kqueue is a powerful interface developers should be aware of, but the power comes at a price. kqueue does not forgive mistakes, which makes it more difficult to master. If you're interested in using kqueue, I'd recommend also looking into libevent, which provides some more structure, but is a whole topic by itself.