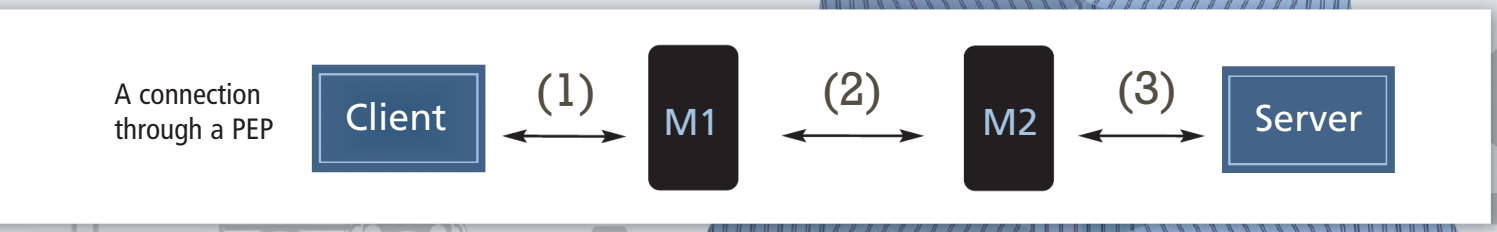
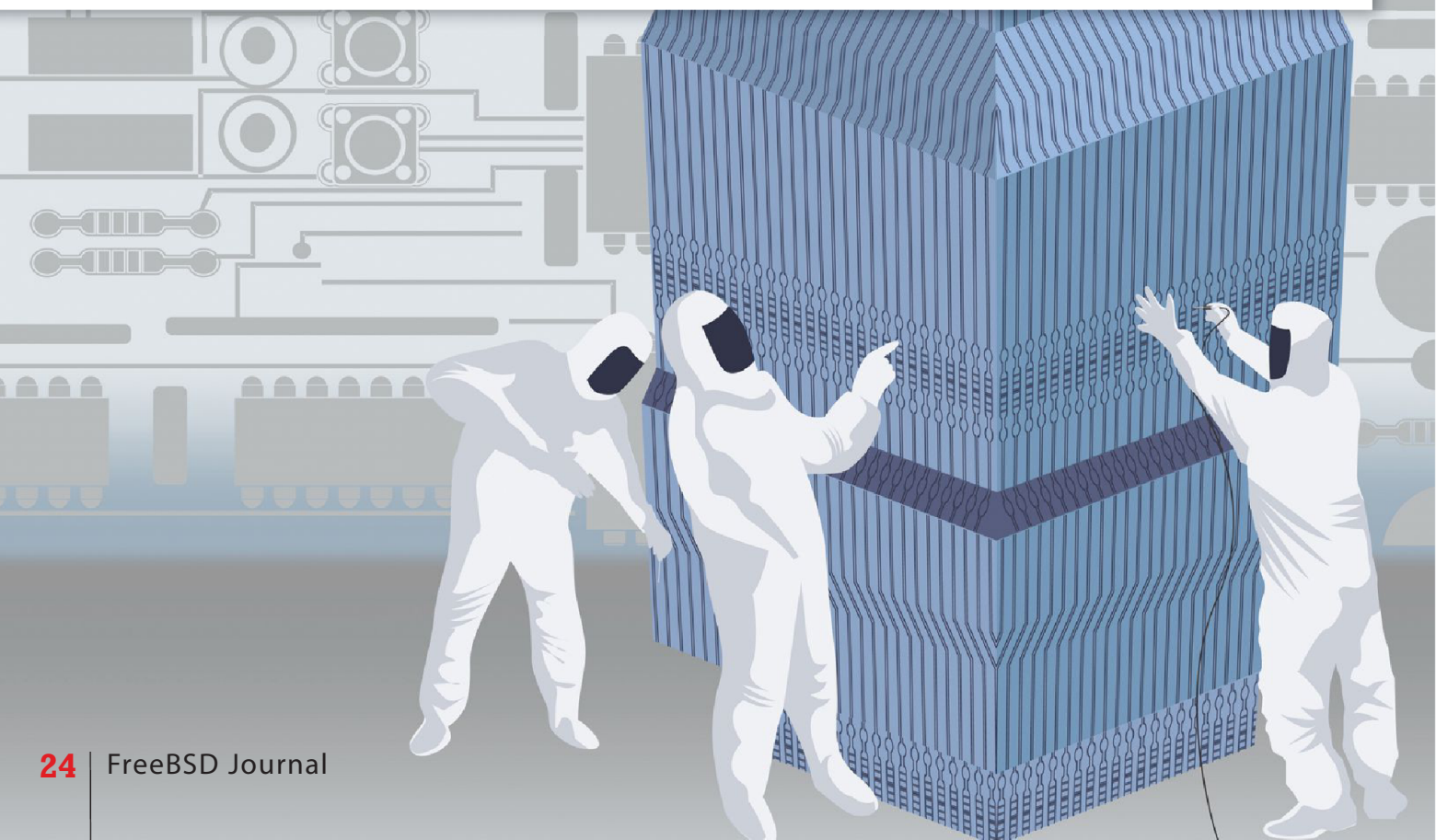
Kqueue madness

by Randall Stewart

Some time ago I was asked to participate in the creation of a Performance Enhancing Proxy (PEP) for TCP. The concept behind a PEP is to split a TCP connection into three separate connections. The first connection (1) is the normal TCP connection that goes from the client towards the server (the client is usually unaware that its connection is not going to the end server). The next connection (2) goes between two middle boxes (M1 and M2), the first middle box (M1) terminates the connection of the client pretending to be the server and uses a different connection to talk to the tail middle box (M2). This middle connection provides the "enhanced" service to the end-to-end connection. The final connection (3) goes between the tail middle box (M2) and the actual server. The figure below shows a diagram of such a connection.



A connection through a PEP



Now, as you can imagine, if you have a very busy PEP you could end up with thousands of TCP connections being managed by M1 and M2. In such an environment using poll(2) or select(2) comes with an extreme penalty. Each time a I/O event completes, every one of those thousands of connections would need to be looked at to see if an event occurred on them, and then the appropriate structure would need to be reset to look for an event next time. A process using such a setup can find itself quickly spending most of its time looking at and fiddling with the data structures involved in polling or selecting and has very little time for anything else (like doing the real work of handling the connections).

I arrived on this project late, but the team that was working on it had chosen to use a kqueue(2) instead of select(2) or poll(2). This, of course, was a very wise choice since they wanted to optimize the process to handle thousands of connections. The team used multiple kqueues and multiple threads to accomplish their end goal, but this in itself was another problem (when it came time to debug the process) since each kqueue had multiple threads and there were a **lot** of kqueues in the architecture. Depending on the number of cores in the hardware, the number of threads would increase or decrease (one thread for each kqueue for each core). This meant that in the first version of the software, a process might have over one hundred threads. After helping stabilize and ship the product I knew we had to improve things and this is where my dive into kqueue madness began.

In the re-architecture, I decided that I wanted to:

1. Keep the proxy untouched as far as the underlying work it was doing.
2. Reduce the number of threads to match the number of cores on the machine.
3. Insert under the proxy a framework that could later be reused for other purposes.

This led me to carefully craft a "dispatcher" framework that would help me accomplish all three goals.

In the middle of the rewrite, after I had gotten the framework pretty well complete and the proxy running on it, I began having issues with long-term stress testing. After deep debugging, I realized one of my problems was that I really did not fully understand the interactions that kqueues and sockets have. I found myself asking questions like:

a) What happens when I delete the kqueue event for a socket descriptor, yet do not close the socket?
b) Could I possibly see stale queued events that were yet to be read?
c) How does connect interact with kqueue?
d) What about listen?
e) What is the difference between all of the kqueue flags that I can add on to events and when do I use them properly?
f) What is the meaning of some of the error returns (EV_ERROR) and does that cover all my error cases?
g) Will I always get an end of file condition when the TCP connection closes gracefully directly from the kqueue?

So the first thing I did is access my friend Google and look for articles on kqueue (after of course reading the man page multiple times closely). Google found for me two articles, one by Jonathan Lemon [1] and the other by Ted Unangst [2]. Both articles are well worth reading but did not answer my questions. In fact the Ted Unangst article left me wondering if I did not use EV_CLEAR and failed to read all the data on a socket, I might miss ever seeing an event again, could this be the source of some of the problems I was seeing?

Since none of my questions were really answered, the only thing that made sense for me to do was write a special test application that I could use to test out various features of kqueues to answer my questions on our Operating System release based on FreeBSD 9.2. I decided to write this article so that you won't have to ponder these questions or write a kqueue test application. (You can find my test program at http://people.freebsd.org/~rrs/kqueue_test.c if you are interested.)

Kqueue Basics

In this section we will discuss the basic kqueue and kevent calls with all the "filters," "flags," and other goodies you need to know in order to transition your socket program into a kqueue based one. A lot of this information is in the man page kqueue(2) (a highly recommended source of information).

So the first thing you have to do is to actually create a kqueue. A kqueue is yet another file descriptor that you open (just like you do a socket) with a special call **kqueue().** There is nothing special about this call, just like creating a socket it returns a file descriptor index that you will use for all future calls. Unlike sockets it has no arguments whatsoever. It can fail, like any system

call, and returns a negative 1, if it does fail, errno will be set to the reason. Once you have a descriptor returned from a successful kqueue call you use this with the kevent() system call to make all the magic happen.

The **kevent()** call has six arguments that it accepts which are:

- *Kq* – The actual kq that you created via the **kqueue()** call.
- *Changelist* – This is a pointer to an array of structures of type kevent that describes changes you are asking for in the kqueue.
- *Nchanges* – This is the number in the array of changes in the *Changelist*.
- *Eventlist* – This is another pointer to an array of structures of type kevent that can be filled in by the O/S to tell you what events have been triggered.
- *Nevents* – This is the bounding size of the *Eventlist* argument so the kernel knows how many events it can tell you about.
- *Timeout* – This is a struct timespec that represents a timeout value.

Like a **poll()** or **select()** the *Timeout* field allows you to control how "blocking" the call will be. If you set the argument to NULL, it will wait forever. If the *Timeout* field is non-NULL it is interpreted to be a timespec (tv_sec and tv_nsec) on the maximum time to delay before returning. One thing to note, if you specify zero in the *Nevents* field then the **kevent()** call will not delay even if a *Timeout* is specified.

Now the **kevent()** call can be used in one of three ways:

- As input to tell the kernel what events you are interested in (for example when you are setting up a bunch of socket descriptors, but you are not yet interested in handling the events). For this you would set *Changelist* to an array of at least one (possibly more) kevents and *Nchanges* to the number in the array. The *Eventlist* field would be set to NULL and the *Nevents* field would be set to 0.
- As only output, so you can find out what events have happened (for example when you are running in an event loop processing events, but possibly not setting any new ones in). To do this you would have the *Changelist* pointer set to NULL, the *Nchanges* set to 0, but *Eventlist* set to a pointer of an array of kevents you want filled and *Nevents* to the length of that array.
- The final way you can use it is to fill both sets, the in and the out, so that as you go in and wait for an event (or events) you can change what is being waited upon. This is useful in an event loop to minimize the number of kernel system calls you are making.

So that, in a nutshell, describes the calls and their use, but what is this kevent structure that you keep talking about? Well a kevent structure looks as follows:

```
struct kevent {
      uintptr_t ident;
      short filter;
      u_short flags;
      u_int fflags;
      intptr_t data;
      void *udata;
};
```

There is also a macro that is part of the event system that is a utility function that helps you setup this structure called **EV_SET()**. But first let's go through each field and describe what you put into it to get the results that you want:

- ***ident*** – This field is the identifier that you wish to have watched. In the case of a socket it would be the socket descriptor. For other kqueue calls it may be something else besides a descriptor. Each type of event to watch for specifies what the ident is made up of (generally it is some form of file descriptor, but in some cases its not e.g. a process id is used in one instance).
- ***filter*** – This field is the actual request that you are asking the kernel to watch for. The following list is the current filters that you can setup:

EVFILT_READ – A read filter looking for read events on a file or socket. This will be one of the filters we cover in a lot more detail.

EVFILT_WRITE – A write filter looking for when we can write to a file or socket descriptor. Again this will be one of the filter types that we take a closer look at.

EVFILT_AIO – asynchronous input output filter used in conjunction with the aio calls **(aio_read()/aio_write()).** For this article we will not discuss this kqueue event.

EFILT_VNODE – A file system change on a particular file. This is a very useful event (e.g. the **tail()** utility uses this when you are doing tail –f) but we won't be discussing this in this article.

EVFILT_PROC – A process event, such as a child's death or a fork of a child. Again a very useful event to watch for, especially if you are writing a process manager, but we won't cover it here.

EVFILT_SIGNAL – This event would come in when a signal occurs (after the signal arrives). This is again something for the reader to explore or maybe a future article ;-)

EVFILT_USER –A user generated event, which can be used to signal between threads or to wake a **kqueue()** for some other specific user defined reason (I use it for shutting down my framework actually). Again this is not something I will cover in more detail but I encourage the curious to investigate on their own.

• ***flags*** – This field, on input (I), tells the kernel what you want performed and on output (O) tells you what happened. The flags that are currently defined are:

EV_ADD(I) – Used when you wish to add your event (for us a socket descriptor) to the kqueue.

EV_DELETE(I) – Delete a previously added event from the kqueue.

EV_ENABLE(I) – Turn on a kqueue event. This may seem redundant but its not, since when you add an event, unless you specify this enable flag, it will not be watched for. You also use this after an event has triggered and you wish to re-enable it (if it does not automatically re-enable itself).

EV_DISABLE(I) – This is the opposite of enable, so you can send this down with a previously enabled filter and have the event disabled, but the internal kqueue structure inside the kernel will **not** be removed (thus its ready to be re-enabled with EV_ENABLE when you want).

EV_DISPATCH(I) – This flag tells the kernel that after it sends you an event, disable it until such time as you re-enable it.

EV_ONESHOT(I) – This flag tells the kernel that when the event occurs and the user retrieves the event, automatically delete (EV_DELETE) the kqueue entry from the kernel.

EV_CLEAR(I) – This toggles the state of the filter right after you retrieve the event so it "re-enables" itself automatically. Note that this can cause a "lot" of events to happen rapidly so this flag should be used very carefully.

EV_EOF(O) – This flag indicates the socket or descriptor has hit the end-of-file condition. For TCP this would be equivalent to reading 0 from the socket (i.e. the peer will send no more data and has sent a FIN).

EV_ERROR(O) – This flag indicates an error has occurred usually the data field will have more information. For example when a peer has "reset" the TCP connection with a RST, you will get an EV_ERROR with the data set to ECONNRESET or ECONNABORTED (or possibly some other errno).

• ***fflags*** – are filter specific flags on the way in and out. For example in the case of sockets you can use this as a way to change the SO_RCVLOWAT value (the *data* field would hold the new value). On the way out, if EV_ERROR were set, the *fflags* would hold the error just like our favorite error field errno.

• ***data*** – This field is used on a per filter basis to supply added information (e.g. the SO_RCVLOWAT mark).

• ***udata*** – This is a handy little pointer that is sent in to the kqueue call and will come up with the event. It is very useful to associate state with a particular event.

Now when sending kevents down to the kernel to be watched for, it's important to realize that to the kernel, the combination of a filter and ident make up a unique entry. So, for example, if I send down a EV_READ for socket descriptor 10 and a EV_WRITE for socket descriptor 10, these end up being two separate filters inside the kernel.

With these basics in mind one can write a simple event loop that would look something like:

```
int watch_for_reading(int fd)
{
	int kq, not_done, ret;
	struct kevent event;

	ret = -1;
	kq = kqueue();
	if (kq == -1) {
		return(ret);
	}
	not_done = 1;
	EV_SET(&event, EVFILT_READ, fd, EV_ADD|EV_ENABLE), 0, NULL);
	if (kevent(kq, &event, 1, NULL, 0, NULL) == -1) {
		close(kq);
		return(ret);
	}
	while(not_done) {
		ret = kevent(kq, NULL, 0, &event, 1, NULL);
		if (ret == 1) {
			/* got the event */
			not_done = 0;
			ret = 1;
			continue;
		}
	}
	return(ret);
}
```

So using a kqueue seems pretty straightforward, but let's now look a bit deeper into how sockets and kqueue interacts as well as conside-r ations we must make for a multi-threaded event loop.

Kqueue, Socket Calls, Multi-threading and Other Mysteries

One of the first things you will want to do with a socket program is connect to a server using the connect(2) call. This, in the normal case, is a blocking call, which can take some time before it times out and gives you an error if the peer is somewhere out on the internet and a long way from you. Now to avoid this one would normally set non-blocking I/O and then do a select or poll for it. For our kqueue, we can do the same thing. The trick to remember on the connect call, however, is that you are **not** selecting on an EV_READ, but instead you are selecting on an EV_WRITE.

Now that I know this little fact seems obvious, you connect so you can write to the server some request right? Of course, when I first started playing with kqueues it was not quite that obvious and it was not until I had read a second (or was it a third) time through the man page that I stumbled onto that little bit of wisdom.

Another little socket trick is the placement of the kqueue call when on the server side you setup for a non-blocking listen. You **must** place the kqueue(2) call **before** you do the listen. If you do not do this, your listen will never wake up on your kqueue—even when it is full of pending sockets waiting to be accepted. This again was a, gee, ah-hah moment for me on a third read of the manual.

Another mystery for me was in my reading of Unangst there seemed to be an implication that if I got a read event and failed to read all I was told to read (the event .data field has the number of bytes ready to read when a socket wakes up a kqueue for reading), I might not be told to wake up again, unless some more data arrived. Of course, I worked real hard to make sure I always read the entire amount so that I would not get "stranded" by some slow sending socket. I even tried setting EV_CLEAR in the hopes of not having to worry about this "race condition." Setting EV_CLEAR actually turned into a disaster, however. This meant that **many** threads would wake up on that same event due to scheduling and context switching delay. Basically what most socket applications need is the EV_DISPATCH. Read what you want (or can) read and then when you want do a re-enable. In my testing with my little app (mentioned earlier) I proved that:

* Adding EV_CLEAR will stream in events until you either get the data read or shut off the event.
* Using EV_DISPATCH and reading only part of the message will result in another kqueue wake-up once the event is re-enabled.

EV_ERROR and EV_EOF were another set of mysteries. When did these wonderful flags get applied? Well, as it turns out, the EV_EOF will be returned once a FIN comes in from the other side. To be more succinct its really every time a kqueue event transpires on a socket and the socket has the SBS_CANTRCVMORE flag set against the receive socket buffer. What this means is that as long as you have data to read, you will continue to see the EV_EOF flag with every subsequent kevent. The EV_ERROR however is a bit different. You usually get these when the socket goes into an error state (ECONNRESET or ETIMEDOUT). Basically if the EV_ERROR gets set, the socket is pretty well washed up.

Now one other oddity, what I began seeing was an appearance of a wakeup on an event I had disabled. How could this occur? Was it that queued up events that had not been read might still be having an event that I would read later. This could cause me major issues since that little pointer in event.udata was being used to carry a pointer that I may have done a free upon. After continuing to see this happen under certain load conditions, I had to dig to the bottom of it. With my little test program I again proved that when you remove an event **all** unread events are removed for that socket (whew). So the oddity I had been seeing had to be something else, or did it?

If you remember the assignment I was given, a tcp proxy, any of the proxys will have two TCP descriptors for one flow. This is where my headache came from. Since it was entirely possible (but rare) that both socket descriptors would wake up at the same time and one of them would have a "reset" on it, if that socket descriptor got processed ahead of the other one, due to locking order of the two threads, disaster could strike. In effect, Thread1 would be destroying the flow, while Thread2 was patiently waiting to get the lock that Thread1 was holding on that flow. Thus, when the lock was released before destruction, Thread2 would wakeup and start

accessing freed memory. This one was solved in a very interesting way, but I will leave it to the reader to puzzle out a solution.

Conclusion

So what conclusion can we draw from all of this madness?

- First and foremost, if you are going to play with kqueues its best to get completely familiar with them before diving in (either that or find better reading on the internet than I did on how to use them).
- Multi-threading with kqueues, especially those that have more than one file descriptor referencing the same object can provide some interesting challenges.
- There are some subtleties with kqueues in their interactions with sockets that need to be accounted for when writing the software (order is very important as well as which type of event you are interested in).
- The proper use of a kqueue can provide you with a very efficient program that can handle tasks with thousands of file descriptors with a minimal set of overhead.
- Understanding and testing the functionality of kqueues (with the before mentioned program) can surely help you from going mad.

References

[1] Kqueue: A generic and scalable event notification facility – Jonathan Lemon. http://people.freebsd.org/~jlemon/papers/kqueue.pdf

[2] Experiences with kqueue – Ted Unangst, August 2009. http://www.tedunangst.com/kqueue.pdf

Randall Stewart currently works for Adara Networks Inc. as a Distinguished Engineer. His current duties include architecting, designing and prototyping Adara's next generation routing and switching platform. Previously Mr. Stewart was a Distinguished Engineer at Cisco Systems. In other lives he has also worked for Motorola, NYNEX S&T, Nortel and AT&T Communication.

Throughout his career he has focused on Operating System Development, fault tolerance, and call control signaling protocols. Mr. Stewart is also a FreeBSD committer having responsibility for the SCTP reference implementation within FreeBSD.