Kqueue: A generic and scalable event notification facility for FreeBSD

Jonathan Lemon jlemon@FreeBSD.org

Abstract

Applications running on a UNIX platform need to be notified when some activity occurs on a socket or other descriptor, and this is traditionally done with the select() or poll() system calls. However, it has been shown that the performance of these calls does not scale well with an increasing number of descriptors. These interfaces are also limited in the respect that they are unable to handle other potentially interesting activities that an application might be interested in, these might include signals, file system changes, and AIO completions. This paper presents a generic event delivery mechanism, which allows an application to select from a wide range of event sources, and be notified of activity on these sources in a scalable and efficient manner. The mechanism may be extended to cover future event sources without changing the application interface.

1. Introduction

Applications are often event driven, in that they perform their work in response to events or activity external to the application and which are subsequently delivered in some fashion. Thus the performance of an application often comes to depend on how efficiently it is able to detect and respond to these events.

FreeBSD provides two system calls for detecting activity on file descriptors, these are poll() and select(). However, neither of these calls scale very well as the number of descriptors being monitored for events becomes large. A high volume server that intends to handle several thousand descriptors quickly finds these calls becoming a bottleneck, leading to poor performance.

The set of events that the application may be interested in is not limited to activity on an open file descriptor. An application may also want to know when an asynchronous I/O (aio) request completes, when a signal is delivered to the application, when a file in the filesystem changes in some fashion, or when a process exits. None of these are handled efficiently at the moment; signal delivery is limited and expensive, and the other events listed require an inefficient polling model. In addition, neither poll() nor select() can be used to collect these events, leading to increased code complexity due to use of multiple notification interfaces.

This paper presents a new mechanism that allows the application to register its interest in a specific event, and then efficiently collect the notification of the event at a later time. The set of events that this mechanism covers is shown to include not only those described above, but may also be extended to unforeseen event sources with no modification to the API.

The rest of this paper is structured as follows: Section 2 examines where the central bottleneck of poll() and select() is, Section 3 explains the design goals, and Section 4 presents the API of new mechanism. Section 5 details how to use the new API and provides some programming examples, while the kernel implementation is discussed in Section 6. Performance measurements for some applications are found in Section 7. Section 8 discusses related work, and the paper concludes with a summary in Section 9.

2. Problem

The poll() and select() interfaces suffer from the deficiency that the application must pass in an entire list of descriptors to be monitored, for every call. This has an immediate consequence of forcing the system to perform two memory copies across the user/kernel boundary, reducing the amount of memory bandwidth available for other activities. For large lists containing many thousands of descriptors, practical experience has shown that typically only a few hundred actually have any activity, making 95% of the copies unnecessary.

Upon return, the application must walk the entire list to find the descriptors that the kernel marked as having activity. Since the kernel knew which descriptors were active, this results in a duplication of work; the application must recalculate the information that the system was already aware of. It would appear to be more efficient to have the kernel simply pass back a list of descriptors that it knows is active. Walk-

ing the list is an O(N) activity, which does not scale well as N gets large.

Within the kernel, the situation is also not ideal. Space must be found to hold the descriptor list; for large lists, this is done by calling malloc(), and the area must in turn be freed before returning. After the copy is performed, the kernel must examine every entry to determine whether there is pending activity on the descriptor. If the kernel has not found any active descriptors in the current scan, it will then update the descriptor's selinfo entry; this information is used to perform a wakeup on the process in the event that it calls tsleep() while waiting for activity on the descriptor. After the process is woken up, it scans the list again, looking for descriptors that are now active.

This leads to 3 passes over the descriptor list in the case where poll or select actually sleep; once to walk the list in order to look for pending events and record the select information, a second time to find the descriptors whose activity caused a wakeup, and a third time in user space where the user walks the list to find the descriptors which were marked active by the kernel.

These problems stem from the fact that poll() and select() are stateless by design; that is, the kernel does not keep any record of what the application is interested in between system calls and must recalculate it every time. This design decision leads to main inefficiency in the current implementation. If the kernel was able to keep track of exactly which descriptors the application was interested in, and only return a subset of these activated descriptors, much of the overhead could be eliminated.

3. Design Goals

When designing a replacement facility, the primary goal was to create a system that would be efficient and scalable to a large number of descriptors. The secondary goal was to make the system flexible. UNIX based machines have traditionally lacked a robust facility for event notification. The poll and select interfaces are limited to socket and pipe descriptors; the user is unable to wait for other types of events, like file creation or deletion. Other events require the user to use a different interface; notably siginfo and family must be used to obtain notification of signal events, and calls to aiowait are needed to discover if an AIO call has completed.

Another goal was to keep the interface simple enough that it could be easily understood, and also possible to convert poll() or select() based applications to the new API with a minimum of changes. It was recognized that if the new interface was radically different, then it would essentially preclude modification of legacy applications which might otherwise take advantage of the new API.

Expanding the amount information returned to the application to more than just the fact that an event occurred was also considered desirable. For readable sockets, the user may want to know how many bytes are actually pending in the socket buffer in order to avoid multiple read() calls. For listening sockets, the application might check the size of the listen backlog in order to adapt to the offered load. The goal of providing more information was kept in mind when designing the new facility.

The mechanism should also be *reliable*, in that it should never silently fail or return an inconsistent state to the user. This goal implies that there should not be any fixed size lists, as they might overflow, and that any memory allocation must be done at the time of the system call, rather when activity occurs, to avoid losing events due to low memory conditions.

As an example, consider the case where several network packets arrive for a socket. We could consider each incoming packet as a discrete event, recording one event for each packet. However, the number of incoming packets is essentially unbounded, while the amount of memory in the system is finite; we would be unable to provide a guarantee that no events would be lost.

The result of the above scenario is that multiple packets are coalesced into a single event. Events that are delivered to the application may correspond to multiple occurrences of activity on the event source being monitored.

In addition, suppose a packet arrives containing N bytes, and the application, after receiving notification of the event, reads R bytes from the socket, where $R < N$. The next time the event API is called, there would be no notification of the (N-R) bytes still pending in the socket buffer, because events would be defined in terms of arriving packets. This forces the application to perform extra bookkeeping in order to insure that it does not mistakenly lose data. This additional burden imposed on the application conflicts with the goal of providing a simple interface, and so leads to the following design decision.

Events will normally considered to be "level-triggered", as opposed to "edge-triggered". Another way of putting this is to say that an event is be reported as long as a specified

condition holds, rather than when activity is actually detected from the event source. The given condition could be as simple as "there is unread data in the buffer", or it could be more complex. This approach handles the scenario described above, and allows the application to perform a partial read on a buffer, yet still be notified of an event the next time it calls the API. This corresponds to the existing semantics provided by poll() and select().

A final design criteria was that the API should be *correct*, in that events should only be reported if they are applicable. Consider the case where a packet arrives on a socket, in turn generating an event. However, before the application is notified of this pending event, it performs a close() on the socket. Since the socket is no longer open, the event should not be delivered to the application, as it is no longer relevant. Furthermore, if the event happens to be identified by the file descriptor, and another descriptor is created with the same identity, the event should be removed, to preclude the possibility of false notification on the wrong descriptor.

The correctness requirement should also extend to preexisting conditions, where the event source generates an event prior to the application registering its interest with the API. This eliminates the race condition where data could be pending in a socket buffer at the time that the application registers its interest in the socket. The mechanism should recognize that the pending data satisfies the "level-trigger" requirement and create an event based on this information.

Finally, the last design goal for the API is that it should be possible for a library to use the mechanism without fear of conflicts with the main program. This allows 3rd party code that uses the API to be linked into the application without conflict. While on the surface this appears to be obvious, several counter examples exist. Within a process, a signal may only have a single signal handler registered, so library code typically can not use signals. X-window applications only allow for a single event loop. The existing select() and poll() calls do not have this problem, since they are stateless, but our new API, which moves some state into the kernel, must be able to have multiple event notification channels per process.

4. Kqueue API

The kqueue API introduces two new system calls outlined below. The first creates a new kqueue, which is a notification channel, or queue, where the application registers which events it is interested in, and where it retrieves the events from the kernel. The returned value from kqueue() is treated as an ordinary descriptor, and can in turn be passed to poll(), select(), or even registered in another kqueue.

```
int
kqueue(void)

int
kevent(int kq,
        const struct kevent *changelist, int nchanges,
        struct kevent *eventlist, int nevents,
        const struct timespec *timeout)

struct kevent {
        uintptr_t   ident;    // identifier for event
        short       filter;   // filter for event
        u_short     flags;    // action flags for kq
        u_int       fflags;   // filter flag value
        intptr_t    data;     // filter data value
        void        *udata;   // opaque identifier
}
```

The second call is used by the application both to register new events with the kqueue, and to retrieve any pending events. By combining the registration and retrieval process, the number of system calls needed is reduced. Changes that should be applied to the kqueue are given in the *changelist*, and any returned events are placed in the *eventlist*, up to the maximum size allowed by *nevents*. The number of entries actually placed in the *eventlist* is returned by the kevent() call. The *timeout* parameter behaves in the same way as poll(); a zero-valued structure will check for pending events without sleeping, while a NULL value will block until woken up or an event is ready. An application may choose to separate the registration and retrieval calls by passing in a value of zero for *nchanges* or *nevents*, as appropriate.

Events are registered with the system by the application via a *struct kevent*, and an event is uniquely identified within the system by a <*kq*, *ident*, *filter*> tuple. In practical terms, this means that there can be only one <*ident*, *filter*> pair for a given kqueue.

The *filter* parameter is an identifier for a small piece of kernel code which is executed when there is activity from an event source, and is responsible for determining whether an event should be returned to the application or not. The interpretation of the *ident*, *fflags*, and *data* fields depend on which filter is being used to express the event. The current list of filters and their

Input flags:

EV_ADD	Adds the event to the kqueue
EV_ENABLE	Permit kevent() to return the event if it is triggered.
EV_DISABLE	Disable the event so kevent() will not return it. The filter itself is not disabled.
EV_DELETE	Removes the event from the kqueue. Events which are attached to file descriptors are automatically deleted on the last close of the descriptor.
EV_CLEAR	After the event is retrieved by the user, its state is reset. This is useful for filters which report state transitions instead of the current state. Note that some filters may automatically set this flag internally.
EV_ONESHOT	Causes the event to return only the first occurrence of the filter being triggered. After the user retrieves the event from the kqueue, it is deleted.

Output flags:

EV_EOF	Filters may set this flag to indicate filter-specific EOF condition.
EV_ERROR	If an error occurs when processing the changelist, this flag will be set.

Possible flag values for struct kevent

arguments are presented in the kqueue filter section.

The *flags* field is used to express what action should be taken on the kevent when it is registered with the system, and is also used to return filter-independent status information upon return. The valid flag bits are given in the table above.

The *udata* field is passed in and out of the kernel unchanged, and is not used in any way. The usage of this field is entirely application dependent, and is provided as a way to efficiently implement a function dispatch routine, or otherwise add an application identifier to the kevent structure.

Kqueue filters

The design of the kqueue system is based on the notion of filters, which are responsible for determining whether an event has occurred or not, and may also record extra information to be passed back to the user. The interpretation of certain fields in the kevent structure depends on which filter is being used. The current implementation comes with a few general purpose event filters, which are suitable for most purposes. These filters include:

EVFILT_READ
EVFILT_WRITE
EVFILT_AIO
EVFILT_VNODE
EVFILT_PROC
EVFILT_SIGNAL

The READ and WRITE filters are intended to work on any file descriptor, and the *ident* field contains the descriptor number. These filters closely mirror the behavior of poll() or select(), in that they are intended to return whenever there is data ready to read, or if the application can write without blocking. The kernel function corresponding to the filter depends on the descriptor type, so the implementation is tailored for the requirements of each type of descriptor in use. In general, the amount of data that is ready to read (or able to be written) will be returned in the *data* field within the kevent structure, where the application is free to use this information in whatever manner it desires. If the underlying descriptor supports a concept of EOF, then the EV_EOF flag will be set in the *flags* word structure as soon as it is detected, regardless of whether there is still data left for the application to read.

For example, the read filter for socket descriptors is triggered as long as there is data in the socket buffer greater than the SO_LOWAT mark, or when the socket has shutdown and is unable to receive any more data. The filter will return the number of bytes pending in the socket buffer, as well as set an EOF flag for the shutdown case. This provides more information that the application can use while processing the event. As EOF is explicitly returned when the socket is shutdown, the application no longer needs to make an additional call to read() in order to discover an EOF condition.

A non kqueue-aware application using the asynchronous I/O (aio) facility starts an I/O request by issuing aio_read() or aio_write() The request then proceeds independently of the ap-

plication, which must call aio_error() repeatedly to check whether the request has completed, and then eventually call aio_return() to collect the completion status of the request. The AIO filter replaces this polling model by allowing the user to register the aio request with a specified kqueue at the time the I/O request is issued, and an event is returned under the same conditions when aio_error() would successfully return. This allows the application to issue an aio_read() call, proceed with the main event loop, and then call aio_return() when the kevent corresponding to the aio is returned from the kqueue, saving several system calls in the process.

The SIGNAL filter is intended to work alongside the normal signal handling machinery, providing an alternate method of signal delivery. The *ident* field is interpreted as a signal number, and on return, the *data* field contains a count of how often the signal was sent to the application. This filter makes use of the EV_CLEAR flag internally, by clearing its state (count of signal occurrence) after the application receives the event notification.

The VNODE filter is intended to allow the user to register an interest in changes that happen within the filesystem. Accordingly, the *ident* field should contain a descriptor corresponding to an open file or directory. The *fflags* field is used to specify which actions on the descriptor the application is interested in on registration, and upon return, which actions have occurred. The possible actions are:

NOTE_DELETE
NOTE_WRITE
NOTE_EXTEND
NOTE_ATTRIB
NOTE_LINK
NOTE_RENAME

These correspond to the actions that the filesystem performs on the file and thus will not be explained here.

The final general purpose filter is the PROC filter, which detects process changes. For this filter, the *ident* field is interpreted as a process identifier. This filter can watch for several types of events, and the *fflags* that control this filter are outlined at the bottom of the page.

5. Usage and Examples

Kqueue is designed to reduce the overhead incurred by poll() and select(), by efficiently notifying the user of an event that needs attention, while also providing as much information about that event as possible. However, kqueue is not designed to be a drop in replacement for poll; in order to get the greatest benefits from the system, existing applications will need to be rewritten to take advantage of the unique interface that kqueue provides.

A traditional application built around poll will have a single structure containing all active descriptors, which is passed to the kernel every time the applications goes through the central event loop. A kqueue-aware application will need to notify the kernel of any changes to the list of active descriptors, instead of passing in the entire list. This can be done either by calling kevent() for each update to the active descriptor list, or by building up a list of descriptor changes and then passing this list to the kernel the next time the event loop is called. The latter approach offers better performance, as it reduces the number of system calls made.

While the previous API section for kqueue may appear to be complex at first, much of the complexity stems from the fact that there are multiple event sources and multiple filters. A program which only wants READ/WRITE events is actually fairly simple. Examples on the following pages illustrate how a program using poll() can be easily converted to use kqueue() and also presents several code fragments illustrating the use of the other filters.

Input/Output Flags:

NOTE_EXIT	process exited.
NOTE_FORK	process called fork()
NOTE_EXEC	process executed a new process via execve(2) or similar call.
NOTE_TRACK	Follow a process across fork() calls. The parent process will return with NOTE_TRACK set in the flags field, while the child process will return with NOTE_CHILD set in fflags and the parent PID in data.

Output Flags only:

NOTE_CHILD	This is the child process of a TRACKed process which called fork().
NOTE_TRACKERR	This flag is returned if the system was unable to attach an event to the child process, usually due to resource limitations.

```
handle_events()
{
    int n, timeout = TIMEOUT;



    n = poll(pfd, nfds, timeout);

    if (n <= 0)
        goto error_or_timeout;
    for (i = 0; n != 0; i++) {
        if (pfd[i].revents == 0)
            continue;
        n--;
        if (pfd[i].revents &
            (POLLERR | POLLNVAL))
        /* error */
        if (pfd[i].revents & POLLIN)
            readable_fd(pfd[i].fd);
        if (pfd[i].revents & POLLOUT)
            writeable_fd(pfd[i].fd);
    }
    . . .


update_fd(int fd, int action,
    int events)
{
    if (action == ADD) {
        pfd[fd].fd = fd;
        pfd[fd].events = events;
    } else
        pfd[fd].fd = -1;
}
```

Original poll() code

```
handle_events()
{
    int n;
    struct timespec timeout =
        { TMOUT_SEC, TMOUT_NSEC };

    n = kevent(kq, ch, nchanges,
        ev, nevents, &timeout);
    if (n <= 0)
        goto error_or_timeout;
    for (i = 0; i < n; i++) {



        if (ev[i].flags & EV_ERROR)
            /* error */

        if (ev[i].filter == EVFILT_READ)
            readable_fd(ev[i].ident);
        if (ev[i].filter == EVFILT_WRITE)
            writeable_fd(ev[i].ident);
    }
    . . .


update_fd(int fd, int action,
    int filter)
{
    ch[nchanges].ident = fd;
    ch[nchanges].filter = filter;
    ch[nchanges].flags =
        action == ADD ? EV_ADD
                      : EV_DELETE;
    nchanges++;
}
```

Direct conversion to kevent()

The code in the upper left column illustrates typical usage of the poll() system call, while the code in the upper right is a near verbatim conversion of the same code to use kqueue. While admittedly this is a simplified example, the mapping between the two calls is fairly straightforward. The main stumbling block to a conversion may be the lack of a function equivalent to update_fd, which makes changes to the array containing the pollfd or kevent structures.

If the *udata* field is initialized to the correct function prior to registering a new kevent, it is possible to simplify the dispatch loop even more, as shown by code on the right.

```
handle_events()
{
   int n;
   struct timespec timeout =
    { TMOUT_SEC, TMOUT_NSEC };

   n = kevent(kq, ch, nchanges,
    ev, nevents, &timeout);
   if (n <= 0)
    goto error_or_timeout;
   for (i = 0; i < n; i++) {
    if (ev[i].flags & EV_ERROR)
          /* error */
       ev[i].udata(ev[i].ident);
    }
. . .
```

Here is a fragment of code that illustrates how to have a signal event delivered to the application. Note the call to signal() which establishes a NULL signal handler. Prior to this call, the default action for the signal is to terminate the process. Ignoring the signal simply means that no signal handler will be called after the signal is delivered to the process.

```
struct kevent ev;
struct timespec nullts = { 0, 0 };

ev.ident = SIGHUP;
ev.filter = EVFILT_SIGNAL;
ev.flags = EV_ADD | EV_ENABLE;
kevent(kq, &ev, 1, NULL, 0, &nullts);

signal(SIGHUP, SIG_IGN);
for (;;) {
    n = kevent(kq, NULL, 0,
               &ev, 1, NULL);
    if (n > 0)
        printf("signal %d delivered"
            " %d times\n",
            ev.ident, ev.data);
}
```

The next fragment of code monitors a descriptor corresponding to a file on an ufs filesystem for specified changes. Note the use of EV_CLEAR, which resets the event after it is returned; without this flag, the event would be repeatedly returned.

```
struct kevent ev;
struct timespec nullts = { 0, 0 };

ev.ident = fd;
ev.filter = EVFILT_VNODE;
ev.flags = EV_ADD |
    EV_ENABLE | EV_CLEAR;
ev.fflags =
    NOTE_RENAME | NOTE_WRITE |
    NOTE_DELETE | NOTE_ATTRIB;
kevent(kq, &ev, 1, NULL, 0, &nullts);

for (;;) {
    n = kevent(kq, NULL, 0,
               &ev, 1, NULL);
    if (n > 0){
        printf("The file was");
        if (ev.fflags & NOTE_RENAME)
            printf(" renamed");
        if (ev.fflags & NOTE_WRITE)
            printf(" written");
        if (ev.fflags & NOTE_DELETE)
            printf(" deleted");
        if (ev.fflags & NOTE_ATTRIB)
            printf(" chmod/chowned");
        printf("\n");
}
```

The behavior of the PROC filter is best illustrated with the example below. A PROC filter may be attached to any process in the system that the application can see, it is not limited to its descendants.

There is single notification for each fork(), if the FORK flag is set in the process filter. If the TRACK flag is set, then the filter actually creates and registers a new knote, which is in turn attached to the new process. This new knote is immediately activated, with the CHILD flag set.

The fork functionality was added in order to trace the process's execution. For example, suppose that an EVFILT_PROC filter with the flags (FORK, TRACK, EXEC, EXIT) is registered for process A, which then forks off two children, processes B & C. Process C then immediately forks off another process D, which calls exec() to run another program, which in turn exits. If the application was to call kevent() at this point, it would find 4 kevents waiting:

ident: A,	fflags: FORK	
ident: B,	fflags: CHILD	data: A
ident: C,	fflags: CHILD, FORK	data: A
ident: D,	fflags: CHILD, EXEC, EXIT	data: C

The knote attached to the child is responsible for returning mapping between the parent and child process ids.

6. Implementation

The focus of activity in the Kqueue system centers on a data structure called a knote, which directly corresponds to the kevent structure seen by the application. The knote ties together the data structure being monitored, the filter used to evaluate the activity, the kqueue that it is on, and links to other knotes. The other main data structure is the kqueue itself, which serves a twofold purpose: to provide a queue containing knotes which are ready to deliver to the application, and to keep track of the knotes which correspond to the kevents the application has registered its interest in. These goals are accomplished by the use of three sub data structures attached to the kqueue:

1. A list for the queue itself, containing knotes that have previously been marked active.
2. A small hash table used to look up knotes whose ident field does not correspond to a descriptor.
3. A linear array of singly linked lists indexed by descriptor, in the same fashion as a process' open file table.

The hash table and array are lazily allocated, and the array expands as needed according to the largest file descriptor seen. The kqueue must record all knotes that have been registered with it in order to destroy them when the kq is closed by the application. In addition, the descriptor array is used when the application closes a specific file descriptor, in order to delete any knotes corresponding with the descriptor. An example of the links between the data structures is show below.

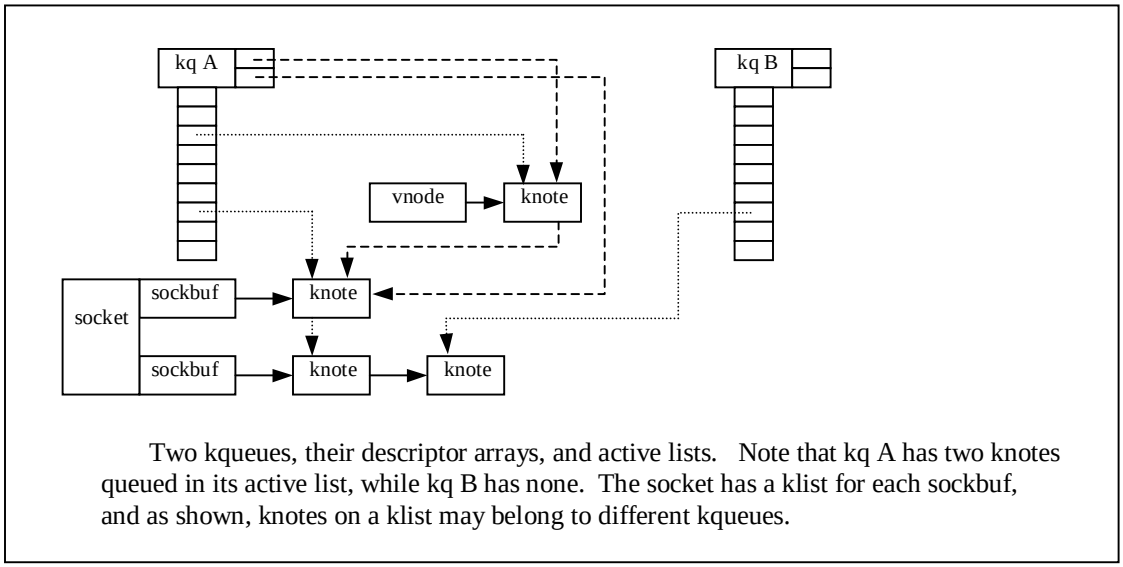
Registration

Initially, the application calls kqueue() to allocate a new kqueue (henceforth referred to as kq). This involves allocation of a new descriptor, a struct kqueue, and entry for this structure in the open file table. Space for the array and hash tables are not initialized at this time.

The application then calls kevent(), passing in a pointer to the changelist that should be applied. The kevents in the changelist are copied into the kernel in chunks, and then each one is passed to kqueue_register() for entry into the kq. The kqueue_register() function uses the <ident, filter> pair to lookup a matching knote attached to the kq. If no knote is found, a new one may be allocated if the EV_ADD flag is set. The knote is initialized from the kevent structure passed in, then the filter attach routine (detailed below) is called to attach the knote to the event source. Afterwards, the new knote is linked to either the array or hash table within the kq. If an error occurs while processing the changelist, the kevent that caused the error is copied over to the eventlist for return to the application. Only after the entire changelist is processed does is kqueue_scan() called in order to dequeue events for the application. The operation of this routine is detailed in the Delivery section.

Filters

Each filter provides a vector consisting of three functions: {attach, detach, filter}. The attach routine is responsible for attaching the knote to a linked list within the structure which receives the events being monitored, while the detach routine



Two kqueues, their descriptor arrays, and active lists. Note that kq A has two knotes queued in its active list, while kq B has none. The socket has a klist for each sockbuf, and as shown, knotes on a klist may belong to different kqueues.

is used to remove the knote this list. These routines are needed because the locking requirements and location of the attachment point are different for each data structure.

The filter routine is called when there is any activity from the event source, and is responsible for deciding whether the activity satisfies a condition that would cause an event to be reported to the application. The specifics of the condition are encoded within the filter, and thus are dependent on which filter is used, but normally correspond to specific states, such as whether there is data in the buffer, or if an error has been observed. The filter must return a boolean value indicating whether an event should be delivered to the application. It may also perform some "side effects" if it chooses by manipulating the *fflag* and *data* values within the knote. These side effects may range from merely recording the number of times the filter routine was called, or having the filter copy extra information out to user space.

All three routines completely encapsulate the information required to manipulate the event source. No other code in the kqueue system is aware of where the activity comes from or what an event represents, other than asking the filter whether this knote should be activated or not. This simple encapsulation is what allows the system to be extended to other event sources simply by adding new filters.

Activity on Event Source

When activity occurs (a packet arrives, a file is modified, a process exits), a data structure is typically modified in response. Within the code path where this happens, a hook is placed for the kqueue system, this takes the form of a knote() call. This function takes a singly linked list of knotes (unimaginatively referred to here as a klist) as an argument, along with an optional hint for the filter. The knote() function then walks the klist making calls to the filter routine for each knote. As the knote contains a reference to the data structure that it is attached to, the filter may choose to examine the data structure in deciding whether an event should be reported. The hint is used to pass in additional information, which may not be present in the data structure the filter examines.

If the filter decides the event should be returned, it returns a truth value and the knote() routine links the knote onto the tail end of the active list in its corresponding kqueue, for the application to retrieve. If the knote is already on the active list, no action is taken, but the call to the filter occurs in order to provide an opportunity for the filter to record the activity.

Delivery

When kqueue_scan() is called, it appends a special knote marker at the end of the active list, which bounds the amount of work that should be done; if this marker is dequeued while walking the list, it indicates that the scan is complete. A knote is then removed from the active list, and the flags field is checked for the EV_ONESHOT flag. If this is not set, then the filter is called again with a query hint; this gives the filter a chance to confirm that the event is still valid, and insures correctness. The rationale for this is the case where data arrives for a socket, which causes the knote to be queued, but the application happens to call read() and empty the socket buffer before calling kevent. If the knote was still queued, then an event would be returned telling the application to read an empty buffer. Checking with the filter at the time the event is dequeued, assures us that the information is up to date. It may also be worth noting that if a pending event is deactivated via EV_DISABLE, its removal from the active queue is delayed until this point.

Information from the knote is then copied into a kevent structure within the event list for return to the application. If EV_ONESHOT is set, then the knote is deleted and removed from the kq. Otherwise if the filter indicates that the event is still active and EV_CLEAR is not set, then the knote is placed back at the tail of the active list. The knote will not be examined again until the next scan, since it is now behind the marker which will terminate the scan. Operation continues until either the marker is dequeued, or there is no more space in the eventlist, at which time the marker is forcibly dequeued, and the routine returns.

Miscellaneous Notes

Since an ordinary file descriptor references the kqueue, it can take part in any operations that normally can performed on a descriptor. The application may select(), poll(), or even create a kevent referencing a kqueue; in these cases, an event is delivered when there is a knote queued on the active list. The ability to monitor a kqueue from another kqueue allows an application to implement a priority hierarchy by choosing which kqueue to service first.

Knotes are attached to the data structure they are monitoring via a linked list, contrasting with

the behavior of poll() and select(), which record a single pid within the selinfo structure. While this may be a natural outcome from the way knotes are implemented, it also means that the kqueue system is not susceptible to select collisions. As each knote is queued in the active list, only processes sleeping on that kqueue are woken up.

As hints are passed to all filters on a klist, regardless of type, when a single klist contains multiple event types, care must be taken to insure that the hint uniquely identifies the activity to the filters. An example of this may be seen in the PROC and SIGNAL filters. These share the same klist, hung off of the process structure, where the hint value is used to determine whether the activity is signal or process related.

Each kevent that is submitted to the system is copied into kernel space, and events that are dequeued are copied back out to the eventlist in user space. While adding slightly more copy overhead, this approach was preferred over an AIO style solution where the kernel directly updates the status of a control block that is kept in user space. The rationale for this was that it would be easier for the user to find and resolve bugs in the application if the kernel is not allowed to write directly to locations in user space which the user could possibly have freed and reused by accident. This has turned out to have an additional benefit, as applications may choose to "fire and forget" by submitting an event to the kernel and not keeping additional state around.

7. Performance

(Please see http://www.flugsvamp.com for performance measurements which were not available at the time of publication)

8. Related Work

This section presents some of the other work done in this area.

POSIX signal queues

For every event that should be delivered to the application, a signal (typically SIGIO) is generated, and a structure containing the file descriptor is allocated and placed on the queue for the signal handler to retrieve. No attempt at event aggregation is performed, so there is no fixed bound on the queue length for returned events. Most implementations silently bound the queue length to a fixed size, dropping events when the queue becomes too large. The structure allocation is performed when the event is delivered, opening up the possibility of losing events during a resource shortage.

The signal queues are stateless, so the application must handle the bookkeeping required to determine whether there is residual information left from the initial event. The application must also be prepared to handle stale events as well. As an example, consider what happens when a packet arrives, causing an event to be placed on a signal queue, and then dequeued by the signal handler. Before any additional processing can happen, a second packet arrives and a second event is in turn placed on the signal queue. The application may, in the course of processing the first event, close the descriptor corresponding to the network channel that the packets are associated with. When the second event is retrieved from the signal queue, it is now "stale" in the sense that it no longer corresponds to an open file descriptor. What is worse, the descriptor may have been reused for another open file, resulting in a false reporting of activity on the new descriptor. A further drawback to the signal queue approach is that the use of signals as a notification mechanism precludes having multiple event handlers, making it unsuitable for use in library code.

get_next_event (Banga, Mogul, Druschel)

This paper motivated the author to implement a similar system, from which practical experience and usage inspired the concept for kqueue. While the original system described by Banga, et.al., performs event coalescing, it also suffers from "stale" events, in the same fashion of POSIX signal queues. Their implementation is restricted to socket descriptors, and also uses a list of fixed size to hold hints, falling back to the behavior of a normal select() upon list overflow.

SGI's /dev/imon

/dev/imon is an inode monitor, and where events within the filesystem are sent back to userspace. This is the only other interface that the author is aware of that is capable of performing similar operations as the VNODE filter. However, only a single process can read the device node; SGI handles this by creating a daemon process called fmon that the application may contact to request information from.

Sun's /dev/poll

This system appears to come closest to the design outlined in this paper, but has some limitations as compared to kqueue. Applications are able to open /dev/poll to obtain a filedescriptor that behaves similarly to a kq descriptor. Events are passed to the kernel by performing a write() on the descriptor, and are read back via an ioctl() call. The returned information is limited to an revent field, similarly to that found in poll(), and the interface restricted to sockets; it cannot handle FIFO descriptors or other event sources (signals, filesystem events).

The interface also does not automatically handle the case where a descriptor is closed by the application, but instead keeps returning POLLNVAL for that descriptor until removed from the interest set or reused by the application. The descriptor obtained by opening /dev/poll can not in turn be selected on, precluding construction of hierarchical or prioritized queues. There is no equivalent to kqueue's filters for extending the behavior of the system, nor support for direct function dispatch as there is with kqueue.

9. Summary

Applications handling a large number of events are dependent on the efficiency of event notification and delivery. This paper has presented the design criteria for a generic and scalable event notification facility, as well as an alternate API. This API was implemented in FreeBSD and committed to the main CVS tree in April 2000.

The author is not aware of any other UNIX system which is capable of handling multiple event sources, nor one that can be trivially extended to handle additional sources.

While, unfortunately there are no performance measurements included with this paper at the moment, the system is in active use in a number of production sites, which have reported remarkable speedups from the code.