KQUEUE prototype implementation for Linux.

Nikolai Joukov and Erez Zadok (instructor)

Computer Science dept., Stony Brook University
CSE 506 Final Project Report.
May 2002.

ABSTRACT - Kqueue event notification API is implemented for Linux in the form of a loadable kernel module. Existing BSD applications that use kqueue API were ported to Linux and approved the correct implementation functionality. Event requests are preserved in the kernel and thus improvement for long-living connections over standard poll mechanism was achieved. Detailed survey of existing event notification mechanisms is presented. A set of a simple kernel changes is proposed in order to enhance kernel extendibility. A scheme of a very low latency light weighted high scalability notification mechanism based on the extended kernel variant is proposed.

I. INTRODUCTION

One of the key aspects of increasing Linux servers' scalability is the improvement of different event notification mechanisms performance [1]. At the moment, there is a number of schemes available for different platforms, but all the existing Linux implementations have important drawbacks that disallow their use for efficient and reliable servers capable of handling thousands or even dozens of thousands connections simultaneously. The only thoroughly designed mechanism that is suitable for these purposes and has very broad functionality is a Kqueue interface, recently implemented for BSD UNIX. This mechanism is rapidly becoming more and more popular in the BSD world. In the current project Kqueue API is used because the Unix community already accepts it, carefully designed and documented, and allows good level of extendibility.

Alternative event notification mechanisms are not popular and are developed very slow since they are implemented in the form of kernel patches that need to change dozens of kernel source files since this part of the kernel is not extendable at all. To address this problem a

simple several line of source code modification is proposed in order to create an extension mechanism for event notification. This could boost the development and improvement of all the kernel event notification interfaces. Since this extension mechanism is not currently implemented the only reasonable form of Kqueue realization was considered a kernel module.

Current Linux Kqueue implementation solves the problem of unnecessary parameter copying of standard poll. Further improvements were not possible for the kernel module form of implementation. Necessary changes to increase the performance are described in detail later in this paper. All the source code for the Linux Kqueue was written from scratch due to the differences with the BSD file system.

To verify correct API realization dkftpbench [6] utility was used. A Poller_test program of that package performed a set of checks to verify Kqueue API correctness.

II. RELATED WORK (Existing interfaces summary)

1. Select and poll system calls.

These two notification mechanisms have the same internal implementation under Linux. Only the user interfaces are different.

For select, the input size is limited by FD_SETSIZE which is compiled into standard library and user programs. Additional overhead is introduced by the necessity to traverse the whole input/output array at least 4 times at each invocation compared to poll. The following is the pseudocode, describing the internal implementation, which is the same:

```
User input is processed
do{
  for each requested file descriptor
     call f_op->poll();
  schedule timeout}while(no signals && no events found && no
timeout)
Remove all waitqueues
Copy output to user buffer
```

f_op->poll() is different for different file descriptor types. In general, it checks if there are pending events and creates a waitqueue for the called process. In the case of event arrival process will be waken up.

Main evident drawback of this implementation is that all list of file descriptors is traversed several times (at least two) even in the case only one event is pending.

2. /dev/poll [2] - device driver that appeared in Solaris 7.

Later ported to FreeBSD and Linux but still is not very reliable. In general, /dev/poll makes the following improvements vs. poll: Input set is passed only once and then queries about the status of that set are done. (State is preserved in the kernel) In addition, a technique named hinting is implemented in order to avoid unnecessary searches for active file descriptors. Output data copying is avoided by introducing a shared memory region. In this approach data copying is reduced to a minimum.

Main drawbacks of the current Linux realization are incredibly bad scalability as the number of idle file descriptors grows [3]. It should be O(1) on idle pipes number and O(n) on active pipes number according to interface limitations (Solaris variant behavior), but it is O(n) on total number of pipes at the moment. The main difference of this approach from the above is that state is preserved inside of the kernel.

There is a less famous flavor of /dev/poll called /dev/epoll available [4]. Its main benefit is a very good handling of dead connections by avoiding the linear scan of f_op->poll() functions for events. Instead callbacks are used to notify the driver code. Actually this approach could be considerable breakthrough in terms of scalability of Linux event notification mechanisms, but general problem of hardly extensible kernel parts limits its practical use.

3. Linux 2.4 Real-time signals.

All interfaces described above are called "level-triggered" because file descriptor status is persistent and queries about it are done. Real-time signals introduce other approach called "edge-triggered" as opposed to "level-triggered". In this approach notifications are sent to the program when the state is changed only. Despite its efficiency, this approach is less forgiving to programmers' mistakes. Once an event is

retrieved and then ignored some file descriptor will be unread for a long time. This interface is very good from scalability, performance, and treating of dead connections point of view [9]. However its poor implementation makes it useless for majority of applications. In particular, all events for all processes are placed in one global queue. It is evident that one process can cause the whole queue to overflow and even cause a deadlock condition [9].

Delivering signals one at a time is very inefficient due to context switch overhead. To deal with this problem a special system call was introduced. However, this call scans the whole signal queue to get the signals requested. In the case of a busy server with highly loaded queue this operation is very inefficient and in the worst case can slow down the performance to that of a standard poll. (Even authors observed this effect at about 1500 connections [8]) In addition, more kernel work is required to handle any
event that slightly slows down the network performance. Moreover, the interface itself is very complicated for a programmer.

4. FreeBSD Kqueue system calls [5].

The most generic interface that has all the best features of the above interfaces combined. Supports both "edge-triggered" and "level-triggered" notification schemes. Preserves the state inside of the kernel as /dev/poll

The only evident drawback of this interface is that it occupies two system calls. However, I think that at least one of them can be shared with some other system call. Currently only FreeBSD variant is available, but even it has some problems with threading. Additional feature of Kqueue is that it is the only generic asynchronous interface that allows to listen for all possible asynchronous events: file descriptor can be written or read, signals, vnode changes, and some other asynchronous I/O types.

The magic of Kqueue scalability originates to bottom-up approach instead of top-down approach used by other mechanisms. In particular, number of file descriptors that can have pending events can be very high and evidently grow as O(N), where N is the number of total file descriptors. The concept of hinting used in /dev/poll can only make the list to scan to be O(A), where A is the number of active connections. Even worse, hint making introduces the same additional overhead for each event processing as pure "edge-triggered"

implementation but does not use it. On contrary, the number of event listeners per file descriptor is usually very small. (Usually one). In this situation it seems quite natural to scan the list of listeners each time an event is occurred rather then scanning the whole list of file descriptors each time events are retrieved. This approach is exploited in both /dev/epoll and real time signals, but the implementations are not thoroughly designed or limited by the kernel structure.

name	State in kernel	Edge triggered
Select	-	-
Poll	-	-
/dev/poll	+	-
/dev/epoll	+	+
r.t. signals	+	+*
Kqueue	+	+

* - real time signals are retrieved by scanning the whole queue that can be considered as checking the level if the queue size is big. Thus, it is not a pure "edge-triggered" implementation.

Table 1. Existing event notification interfaces summary.

III. DESIGN

After careful function by function, file by file investigation of existing kernel event processing code including real time signals related code and poll code it became evident that there is no way to implement "edge-triggered" mechanism with the module based implementation.

As a consequence "level-triggered" approach was employed. The full Kqueue file descriptor events functionality is implemented by partly utilizing the poll functions implemented in the existing kernel code. State of the filters that specify the file descriptors to listen to is stored in the file structure private_data field of Kqueue file descriptor. All memory allocation is done on demand only and nearly no memory is allocated at the Kqueue file descriptor creation time. All memory copying and large buffer allocation are done in the PAGE_SIZE chunks. To avoid starvation of a part of file descriptor events if the output buffer size is small events are retrieved in a circular manner. That is state of the descriptor next to last checked is verified first.

Filters are stored in a linear array with each element containing flags, describing the filter and user specific data that is returned to user together with the output if the filter is matched. Thus, this data structure has size 5*N rounded up to the multiply of PAGE_SIZE, where N is the descriptors number. Memory allocation is done in page chunks on demand. This is the only data structure permanently stored in the kernel during the lifetime of the Kqueue file descriptor. To free up space allocated for this structure special file operations structure is created with release() method implemented to free kernel memory allocated. Thus, even if the application unexpectedly terminates no kernel memory is left allocated. In general, the Kqueue interface consists of two system calls:
kqueue() and kevent(). The first one is responsible for initial file descriptor creation. The second one does the whole event related work. It processes filter modifications if any and then retrieves the events. Filter modifications include addition, deletion, enable, disable, and type of listening event change. After all filter modifications are done state of the file descriptors described by filters is retrieved by using f_op->poll() methods similar to poll. Possible errors are included in the output events list according to Kqueue standard. All the necessary macros and interfaces specific to Kqueue standard are created in the kqueue.h file.

IV. IMPLEMENTATION

The body of the kernel module is divided into the following parts:
mkqueue.c - kernel module interface and system calls registration.
sys_kqueue.c - Kqueue system call body.
sys_kevent.c - kevent system call body.
kq_poll.c - interface that resembles in-kernel poll behavior.

kqueue.h contains data structures and macroses declarations.
According to the BSD standard this file should be available as
sys/event.h to include by the user programs.

To debug the module behavior a special constant DEBUG_KQUEUE is defined in kqueue.h file. Different values of the constant define the amount of debugging code compiled into the module. Another constant DEBUG_KQUEUE_MEMORY makes a memory

allocation/deallocation control functionality to be compiled into the module. In this case memory freeing is verified to make sure that there are no kernel memory leaks.

V. EVALUATION

To verify basic correct interface functionality a small user space program called test.c was implemented. It opens the stdin and passes its handle to kevent() system call to listen for read data availability. It either returns after a user presses enter or after a two minutes timeout specified in the timespec data structure.

To verify BSD standard compliance the dkftpbench [6] package was used. In particular, Poller_test utility successfully compiled with the header file provided. Then this program performed several dozens of tests to verify the correct behavior and correct results on a set of created pipes. Even correct user data field delivery was verified.

Since the event retrieval is based on the same mechanism as poll no wonder that they perform nearly the same for short living pipes as can be concluded from Poller_bench [6] results.

However, many real servers (for example ftp) usually maintain many long-living connections that persist during a hundred or even more event availability requests. In that case filter-copying overhead of poll leads to its higher latency compared to the kqueue where filters are saved in the kernel only once.

The above reasoning was verified by the means of two benchmark programs: poll_bench.c and kqueue_bench.c based on the code from [7]. Both programs create a set of pipes that are all ready to be read. Then a set of filters is passed to the kernel for kqueue, while for the poll all filters are passed with each request. Execution times shown in table 2 and figure 1 where gained using these programs with the assumption that connection lives during 1000 event requests. This number may seem very high, but results gained with the sequence of 100 requests are very similar, however the accuracy is much lower.

pipes	kqueue	Poll
100	51	48
500	250	251
800	260	401
1000	278	527

Table 2. Averaged execution times in microseconds of Kqueue and poll interfaces in the assumption that all connections are long living.

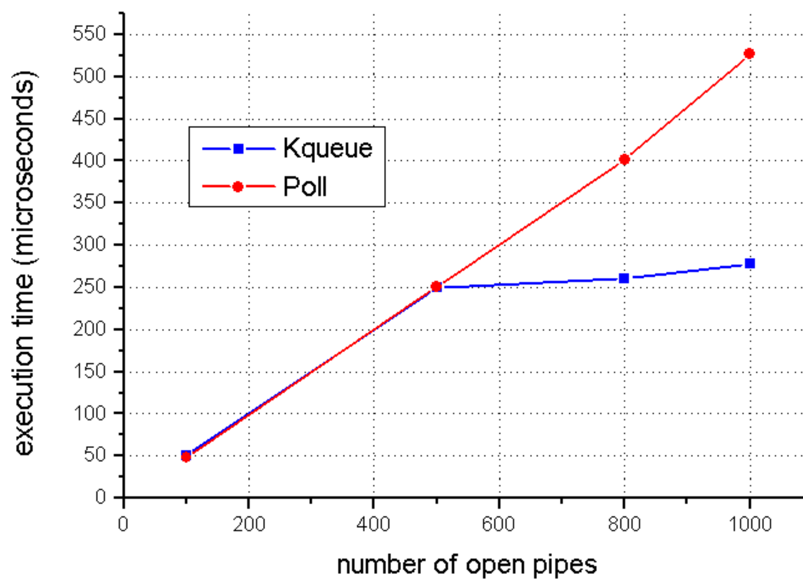

Figure 1. Averaged execution times in microseconds of Kqueue and poll interfaces in the assumption that all connections are long living.

Evidently the case above is the most favorable for Kqueue realization presented. In real circumstances the performance of a given Kqueue will be somewhere in-between the values in two columns.

VI. FURTHER WORK

General problem of all proposed Linux event delivery systems is the unavailability of any extension mechanisms related to signal and event delivery kernel mechanisms. However, recently created real time signals notification mechanism includes a function, called __kill_fasinc located in fs/fcntl.c that is called by all the asynchronous file descriptor

related event sources. In this case, adding one additional function call entry into file operations structure f_op->aio() and adding two lines to __kill_fasinc:

```
if(f_op && f_op->aio)
  f_op->aio();
```

would completely solve the extensibility problem for file related events.

From now on we assume that events are delivered to the external module by the means of f_op->aio function.

The other logical thing to do is to make function f_op->aio add the event to the output queue directly. The cost of this operation is very small since it only has to determine which of possibly several kqueue file descriptors belonging to the destined process is listening for this event. This operation can be done using a hash table for fast lookup. On contrary we can not do the same thing in conventional "level-triggered" approach since we have to traverse all the file descriptors, not just do a lookup. As soon as output queue is determined adding one element to its tail costs very little.

Using the algorithm above event retrieval consists from copying the constructed queue to user only. In this case event retrieval call time will be very close to a small constant independent on the number of file descriptors or active file descriptors.

Algorithm summary:

```
aio(process, ...)
(destined process structure is known by the caller)
{
  lookup each Kqueue filters database of this process.
  (usually one database. O(1) per lookup if hash
   or log(filter number) if sorted filter array)
  if event is in the database
     add element to the end of output array
  if process is waiting for this event
  (waitqueue of this Kqueue descriptor is not null)
     wake up the process
}

kevent(...)
{
  if(output queue is not empty)
     copy output to user
```

```
  else
    schedule_timeout();
}
```

Note that both functions run in O(1) time. The only function that will work as O(P), where P is the number of currently pending events is filter disable or filter delete operation since in that case we will have to remove all corresponding pending events from the output. Fortunately, these operations are not expected to happen frequently.

The approach described above is written for optimal Kqueue implementation under Linux. However, performance can be further increased if aio function will place pending events to a shared memory region. In that case system call to retrieve pending events can be eliminated at all. However, in that case some kind of locks should be introduced and user applications would require more careful design.

VII. CONCLUSION

Basic Kqueue API realization for Linux is implemented in the form of a loadable kernel module. Its correct functionality is verified. Further improvements resulting in necessary kernel modifications are presented. In particular, just a several lines of kernel code modifications that would allow to extend the kernel with new generation of kernel events notification mechanisms possibly in the form of modules is presented. A scheme of a high throughput low latency kernel event notification mechanism is described.

It is shown that data copying itself is responsible for only a small portion of event delivery mechanisms latency. Thus, "edge-driven" approach is proved to be necessary to create very high scalable event delivery systems.

REFERENCES

[1] Dan Kegel - "The C10K problem" http://www.kegel.com/c10k.html

[2] Niels Provos and Charles Lever - "Scalable Network I/O in Linux" http://www.citi.umich.edu/techreports/reports/citi-tr-00-4.pdf

[3] Dan Kegel - "Microbenchmark comparing poll, kqueue, and /dev/poll - 24 Oct 2000 http://www.kegel.com/dkftpbench/Poller_bench.html

[4] Davide Libenzi - "Improving (network) I/O performance" http://www.xmailserver.org/linux-patches/nio-improve.html

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

[6] http://www.kegel.com/dkftpbench/

[7] http://www.bitmover.com/lmbench/

[8] Niels Provos, Chuck Lever, Stephen Tweedie - "Analyzing the Overload Behavior of a Simple Web Server" CITI Technical Report 00-7

[9] Abhishek Chandra, David Mosberger - "Scalability of Linux Event-Dispetch Mechanisms" Hewlett Packard Technical Report PL-2000-174