Real Time Signalling

in Squid Performance

Prepared By
Visolve Squid Team

| Abstract | Introduction | Event Notification with poll() | POSIX Real Time Signals | Using Real-Time Signals for Network I/O | | Performance Benefits | Real-Time Signal Problems and Fixes | Signal Queue Overflow | | Signals on Closed Socket | Lost Events Before Enabling Real-Time Signals | Skipping Events | Performance Comparison | |Conclusion | References | About ViSolve.com |

Abstract

POSIX RealTime signals are potentially a better alternative to polling for scalable network I/O. RealTime signals provide a fast, scalable mechanism for signaling network events. However, they also introduce race-condition problems which application writers must deal with. We describe solutions for using RealTime signals properly. Finally we compare the performance of a cache server using both polling and RealTime signals.

Introduction

Internet servers today must handle thousands of concurrent client connections. One of the main bottlenecks for these servers is event notification for socket I/O. In the ideal scenario, the server's response time should scale linearly with the number of active socket connections.

However, the traditional poll() and select() interfaces do not scale well to large numbers of clients [1,2,3]. Servers that use poll() and select() reach 100% CPU usage quickly as the number of client connections grow. For these servers, polling is often the main performance bottleneck that limits the number of concurrent clients.

POSIX RealTime signals provide a better alternative to polling for socket I/O. Furthermore, the main obstacle to using RealTime signals, signal queue overflow, has been solved by the one-signal-per-fd patch [4].

This paper is organized as follows: Section 2 presents the poll() interface and reveals its scalability problems. Section 3 introduces RealTime signals and why it performs better. In Section 4 we describe difficulties in using RealTime signals, and fixes for dealing with these problems. Section 5 compares the performance of the Squid cache server using both polling and RealTime signals. We present our conclusions in Section 6.

Event notification with poll()

Event notification with poll()

Poll() interface

The traditional way to check for socket events on a large number of clients is by using the poll() or select() system calls. Figure 1 summarizes the poll() interface from the Linux man page.

```
/* Flags to indicate socket events. 0 indicates no event. */
```

```
#define POLLIN      0x0001    /* There is data to read */
#define POLLPRI     0x0002    /* There is urgent data to read */
#define POLLOUT     0x0004    /* Writing now will not block */
#define POLLERR     0x0008    /* Error condition */
#define POLLHUP     0x0010    /* Hung up */
#define POLLNVAL    0x00
```

Event notification with poll()

Poll() interface

The traditional way to check for socket events on a large number of clients is by using the poll() or select() system calls. Figure 1 summarizes the poll() interface from the Linux man page.

```
20    /* Invalid request: fd not open */

struct pollfd {
  int fd;
  short events;
  short revents;
};

int poll(struct pollfd *pfds, int number, int timeout);
```

Figure 1: poll() interface for getting socket events

The application passes an array of pollfd containing the sockets and the events to check for. The timeout value specifies how long the process should wait for an event to occur on any of the sockets. When the poll() function returns, it stores the actual events ready on each socket in the revents field of each pollfd.

Scalability Problems

The poll() system call performs poorly when the application passes an array with thousands of sockets. The poll() call has scalability problems for the following reasons: [1]

- For each poll() call, the kernel must copy the event list from user space to kernel space, and then copy the updated event list back from kernel space to user space.
- Both the kernel and the application must scan the entire set of sockets to check for events. The poll() running time scales linearly with the total number of client sockets open.
- In practice, only 10% to 20% of client sockets are active at a time. This is due to the high latency of data transfer over the Internet [1]. With poll(), the server wastes time polling thousands of sockets without any activity, instead of performing actual work on the few sockets with activity.

Related Work

Several new interfaces have been proposed to address the performance problems of poll() and select() for network I/O Banga and Mogul [1] propose the system calls

declare_interest() and get_next_event() to register a socket for event notification and to retrieve events. The /dev/poll interface improves upon poll() by reducing copying between user and kernel. However, /dev/poll still must poll all open sockets at the kernel level, and therefore also has scalability problems [2]. A new interface called kqueue[3] has been implemented on the FreeBSD operating system. Kqueue was designed to eliminate the performance problems of polling and avoid the difficulties found in RealTime signals. But, there is currently no implementation for Linux yet. Our main focus is on RealTime signals because it is readily available for Linux.

POSIX RealTime Signals

POSIX RealTime Signals are a general mechanism for asynchronous event delivery. For servers, RealTime signals can be used to notify socket events to the application. RealTime signals potentially offer better performance for network I/O than polling.

Comparison with standard signals

POSIX RealTime signals differ from standard UNIX signals in the following two ways:

- RealTime signals contain a data payload (siginfo_t struct) that contains information about the event. Standard signal handlers only passed the signal number as argument; the handler had no other information about the event that occurred. Figure 2 lists relevant fields found in the siginfo_t struct.
- RealTime signals are queued to a process, rather than passed to a signal handler. The application can retrieve the socket events at a convenient time, rather than interrupting normal execution with a signal handler. Unfortunately, queuing introduces race condition problems, which are discussed later in Section 4.

```
/* Siginfo struct describing event */
typedef struct siginfo {
   int si_signo;     /* Signal number */
   int si_errno;     /* Error code */
   int si_code;
   union {
      /* Skipping other fields */
      struct {
         int _band;  /* Socket event flags (similar to poll) */
         int _fd;    /* Socket fd where event occurred */
      } _sigpoll;
   } _sifields;
}  siginfo_t;

#define si_fd      _sifields._sigpoll._fd
#define si_band    _sifields._sigpoll._band
```

Figure 2: siginfo_t struct describing socket event

Using RealTime signals for network I/O

User applications perform the following steps to use RealTime signals for socket event notification. First, the user enables RealTime signals on the socket file descriptor. Figure 3 lists the flags that must be set to enable RealTime signals. Once the flags in Figure 3 are set, RealTime signals are enabled for that socket, until it is closed. Whenever an I/O event occurs on the socket, a RealTime signal will be added to the signal queue.

```
int sockfd = accept( );

/* Set socket flags to non-blocking and asynchronous */
fcntl(sockfd, F_SETFL, O_RDWR | O_NONBLOCK | O_ASYNC);

/* Set signal number >= SIGRTMIN to send a RealTime signal */
fcntl(sockfd, F_SETSIG, SIGRTMIN);

/* Set process id to send signal to */
fcntl(sockfd, F_SETOWN, getpid());

/* Allow only one signal per socket fd */
fcntl(sockfd, F_SETAUXFL, O_ONESIGFD);
```

Figure 3: Enabling RealTime signals on a socket.

Figure 4 shows how the application retrieves socket events from the signal queue. The application uses the sigwaitinfo() or sigtimedwait() system calls. The sigwaitinfo() call takes as arguments the signal set to wait for, and a siginfo structure to store the signal event. It blocks until a signal in the signal set is received in the signal queue. The socket event is stored in the siginfo_t structure, as shown in Figure 2.

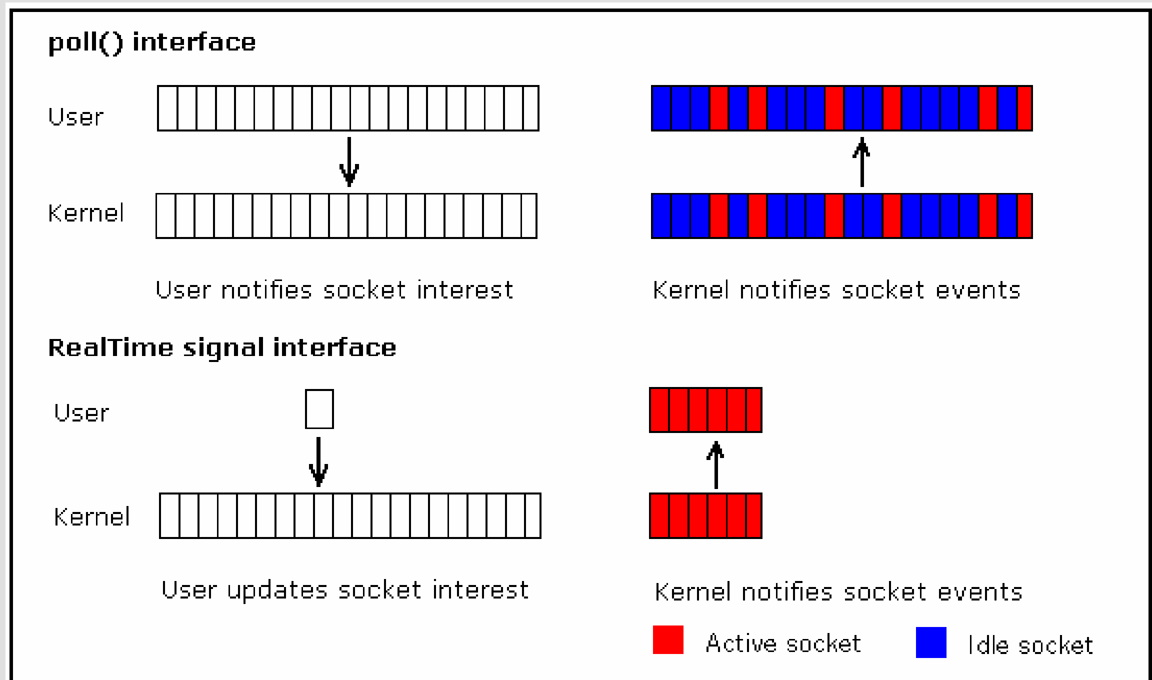
```
sigset_t signalset;
siginfo_t siginfo;
int signum, sockfd, revents;

/* Wait only for RealTime signal SIGRTMIN. */
sigemptyset(&signalset);
sigaddset(&signalset, SIGRTMIN);

/* Check for socket events */
signum = sigwaitinfo(&signalset, &siginfo);
if (signum == SIGRTMIN) {
    sockfd = siginfo.si_fd;
    revents = siginfo.si_band;
    ....
    proceed using sockfd and revents

}
```

Figure 4: Checking for socket events using RealTime signals

Performance benefits	RealTime signals perform better than polling because they scale linearly with the number of **active** sockets; poll() scales linearly with the number of **total** sockets being serviced. Figure 5 illustrates the main design differences that benefit RealTime signals: ○ With RealTime signals, the kernel keeps track of the sockets being monitored. The user updates the kernel by enabling RealTime signals on a socket. With polling, the user keeps track of the sockets being monitored, and must pass the *entire* socket list on each poll system call. ○ With RealTime signals, the kernel only returns active sockets to the user. The application uses the sigtimedwait system call to retrieve socket events. Information about idle sockets is never returned to the user. 

RealTime signal problems and fixes	We believe that RealTime signals have not achieved widespread use because of difficulties in use for application writers. To investigate the complexity of using RealTime signals in a server, we modified the Squid Web Cache Server [7] to use RealTime signals. Unfortunately, we encountered several problems that make RealTime signals difficult to use. We discuss both kernel and user level fixes to work around these problems.

Signal Queue Overflow	The number of RealTime signals sent to a process can grow infinitely. A single socket can send multiple I/O events, resulting in multiple signals in the queue. When the RealTime signal queue overflows, the application must perform complicated steps to

	recover. The application receives a SIGIO signal when the queue overflows. To resolve the signal overflow, the application must disable RealTime signals on all sockets, and remove all signals from the signal queue. Then, to retrieve the lost socket events, the application must poll all the sockets. As expected, handling signal queue overflow increases the complexity of the application, and degrades the performance under high load [2]. Queue overflow has been a major obstacle in using RealTime signals.
	
	A solution to avoid signal queue overflow is to allow only one event per socket file descriptor in the signal queue. If multiple events occur on the same socket, the event flags for the siginfo_t data will be combined (using bitwise OR). If the signal queue size is greater than the maximum number of open file descriptors, the queue will never overflow. This solution was proposed by Chandra and Mosberger [2], and implemented for the Linux 2.4.13 kernel by Luban [4]. To enable one signal per file descriptor, the application just sets a flag on the socket:
	
	`/* Allow only one signal per socket fd */`
	`fcntl(sockfd, F_SETAUXFL, O_ONESIGFD);`

Signals on closed socket	When a socket is closed, the corresponding RealTime signals in the signal queue are not removed. This forces the application to deal with expired events that no longer correspond to an open socket. A simple workaround is to poll() the socket fd when the event is removed from the signal queue. However, doing a poll for every event received will hurt performance. The kernel can prevent this problem by removing the event from the signal queue when the socket is closed. When using the one-signal-per-fd patch described in Section 4.1, the kernel only needs to remove a single event from the signal queue. The event can removed efficiently without walking through the entire signal queue. Every struct file contains a pointer to the corresponding siginfo_t in the signal queue. When the socket closes, the signal can be either removed from the queue, or the events flags can be reset to 0, indicating that no event occurred.

Lost events before enabling RealTime signals	Socket events can be lost before RealTime signals are enabled. When a new client socket is accepted, data can arrive on the socket before RealTime signals are enabled on the socket. Although data is available on the socket, no signal will be sent to the application, so the I/O event is lost. The solution we implemented to work around this problem is to poll every client socket immediately after enabling RealTime signals on that socket. This extra poll() call did not affect performance significantly, but it added complexity to the application. A kernel level solution would require a new system call for accepting client sockets. The new system call would automatically enable RealTime signals on the client socket before any data is received from the client.

Skipping events

RealTime signals are only sent once, so the application cannot skip or defer events. When using poll(), an application can ignore any events on a socket, because the event will still appear the next time the socket is polled. Applications may wish to ignore socket events to limit bandwidth or prevent overload.

This problem cannot be solved efficiently by the kernel. If the kernel constantly resends skipped events, the application will essentially be polling the socket, which leads to performance degradation.

Ultimately, the user level code must be rewritten so that socket events are never ignored. For example, in our Squid implementation, we had to abandon the bandwidth limiting features (called delay pools) because RealTime signals do not allow skipping socket events.

Performance Comparison

Test setup

To test the performance benefits of RealTime signals, we modified the Squid Web Cache server [7] to use RealTime signals for network I/O instead of poll(). The tests were performed on an HP Netserver lp1000r with an 866Mhz CPU, 512 MB memory, and two 8 GB SCSI hard disks. The machines ran Red Hat Linux 7.1 with the Luban one-sig-perfd patch. To compare performance, we used the Web Polygraph benchmark [8] for caching proxies.

Conclusion

This paper has discussed the limitations of poll() for highly concurrent network I/O. We introduced RealTime signals as a better alternative to poll. RealTime signals provide efficient event notification. The performance of RealTime signals depends only on the number of active sockets, and not number of total clients. The one-sig-perfd patch eliminates signal queue overflow, overcoming a major hurdle to using RealTime signals. Performance tests with Squid have demonstrated that porting server applications to RealTime signals dramatically improves throughput of the server.

References

[1] G. Banga and J. C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proceedings of the USENIX Annual Technical Conference*, June 1998

[2] A. Chandra and D. Mosberger. Scalability of Linux Event Dispatch Mechanisms. HP Labs Technical Report, December 2000

[3] J. Lemon. Kqueue: A generic and scalable event notification facility. In *Proceedings of the USENIX Annual Technical Conference*, June 2001

[4] V. Luban. one-sig-perfd patch http://www.luban.org/GPL/one-sig-perfd-

2.4.13.patch

[5] N. Provos and C. Lever. Scalable Network I/O in Linux. In *Proceedings of the USENIX Annual Technical Conference*, June 2000

[6] N. Provos, C. Lever, and S. Tweedie. Analyzing the Overload Behavior of a Simple Web Server. In *Proceedings of the Fourth Annual Linux Showcase and Conference*, October 2000

[7] Squid Cache Server. http://www.squid-cache.org/

[8] Web Polygraph benchmark. http://www.web-polygraph.org/

About ViSolve.com

ViSolve is an international corporation that provides technical services, for Internet based systems, for clients around the globe. ViSolve is in the business of providing software solutions since 1995. We have experience of executing several major projects and we are now completely focused on leading Internet technologies, Testing QA and support. We are committed to the Open source movement and in the same lines we provide free support for products like Linux, Apache and Squid to the user community.

Document Version : 1.0 | Created On : 28-01-02 | Updated On : 29-05-06