A Fast Polling I/O Implementation with Real-time Signals

Eiji Kawai, Youki Kadobayashi, and Suguru Yamaguchi
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara, 630-0192 JAPAN
{eiji-ka, youki-k, suguru}@is.naist.jp

Abstract

This study revisits the scalability issue of polling I/O, i.e., select() and poll(). Although polling I/O was an efficient and hence popular I/O multiplexing mechanism, it is believed inadequate today to handle tens of thousands of concurrent TCP connections because the scanning cost of such a large connection list is overwhelmingly high. However, the real problem of polling I/O is not in the semantics itself, but in various implementation factors such as memory allocations and copies to process a system call, pointer operations and function calls through VFS, and wait channel management to handle events. To mitigate these overheads and reinstate polling I/O as an efficient I/O multiplexing mechanism, we have developed a fast polling I/O library based on POSIX real-time signals. This library implements the full functionality of polling I/O by managing the state transition of each connection notified with real-time signals. Traditional polling I/O has a weakness especially when it polls a small number of active connections together with a huge number of idle ones. Our polling I/O library is proved to achieve high performance in that situation.

1 Introduction

Today’s heavily-loaded network servers manage tens of thousands of concurrent TCP connections because of the increasing Internet users and emerging sophisticated network service implementations such as HTTP/1.1 persistent connections. This phenomenon escalates the load of network servers and makes the programming model for high performance network servers important.

Many traditional network servers have utilized so far polling I/O [4], which is implemented in select() and/or poll(). Polling I/O enables a single process/thread to manipulate more than one connection concurrently by checking their I/O-readiness through a single invocation. Since polling I/O can eliminate the process/thread management cost inevitable in the multi-process/multi-thread programming model, server developers can implement an efficient communication management mechanism with it. However, as the number of the concurrent connections a today’s heavily-loaded network server handles increases, polling I/O is now conceived as a performance bottleneck [1] because of its large processing cost in socket list scanning.

To solve this issue, several research groups have developed more efficient communication management techniques with explicit event notifications [1, 3, 5, 6, 2]. Among them, POSIX real-time signals are a promising technique that was standardized in the POSIX real-time extension (IEEE 1004.1b-1993) and has been already implemented in most Unix operating systems. However, a disadvantage common to those explicit event notification techniques is their completely different programming model from the traditional polling I/O model. To make matters worse, real-time signals peculiarly require a complicated mechanism to handle exceptional conditions such as signal queue overflows. Hence, the POSIX real-time signal-driven model is currently adopted by few network servers.

In this paper, we propose a fast polling I/O mechanism as a replacement for the traditional one. It records state transitions of each connection notified by POSIX real-time signals and actual I/O, i.e., read and write I/O, and returns the recorded state at polling I/O invocations. This mechanism mitigates the processing overhead in polling I/O and enhances the server performance especially when it has to manage a small number of active connections together with a huge number of idle ones. It also has an advantage that it requires only minor modification in existing server implementations.

*This paper is a draft version of the paper presented at the 3rd IEEE International Symposium on Network Computing and Applications (IEEE NCA04).

**This research was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Young Scientist (B), 16700068, 2004.
2 Design of Fast Polling I/O

We propose a communication management mechanism that implements the full functionality of polling I/O utilizing POSIX real-time signals. It configures all the sockets non-blocking to prevent I/O blocking and obtain kernel buffer state at each I/O. In this section, we describe the basic design of the mechanism.

2.1 Management of Event Information

The semantics of polling I/O is to return the state of each connection when it is invoked. On the other hand, the semantics of real-time signals is to notify a process of I/O-readiness of a socket. In addition, I/O-unreadiness of a socket can be detected with non-blocking I/O. Hence, we can implement the polling I/O semantics by recording the event information given by real-time signals and non-blocking I/O in a table and returning the recorded information at a polling I/O invocation.

The format of the data delivered with a real-time signal is the same as that returned by traditional poll(). It is a short integer data that contains various flag bits including POLLIN, POLLOUT, POLLERR, POLLHUP, and POLLNVAL.

To achieve high efficiency, when our polling I/O library function is invoked, it is desirable to avoid much computation on the data stored in the table. We therefore prepare two types of the tables. One is an array of short integers for ufdPoll(), our library version of poll(). Only a few bit operations such as “|” and “&” in C-language notation are required on each element of this array to process ufdPoll(). Figure 1 depicts the data structure and the processing flow. The other type of the table is a set of fd_set data structures, which are used by ufdSelect().

Figure 1: Basic structure of the socket state management table for ufdPoll().

![Socket State Management Table](image)

Figure 2: Socket state transitions triggered by a signal delivery and an I/O execution.

![Socket State Transitions](image)

Figure 2 summarizes the state transition of a socket triggered by a signal delivery and a normal I/O execution. The proposed mechanism can manage the state of the read and write buffer as well as an error occurrence on each socket.

2.1.1 Read Buffer

A new socket obtained by socket() is not readable at the creation time because there is no communication counterpart configured. Hence, the state of a socket created by socket() is initialized as “not readable.” Meanwhile, we have another way to obtain a new socket, i.e., accept(). With accept(), the situation is different because the communication counterpart of the new socket is already configured when the socket is returned by accept(). Hence, there is a possibility that some data arrive in the kernel read buffer before the fast polling I/O library sets up the newly accepted socket in the real-time signal-driven I/O mode. To sidestep this issue, we

---

1 More precisely, when the client operating system receives the second SYN packet of the TCP 3-way handshake from the server, it sends an ACK packet and notifies the client process that the connection is established. Therefore, from a viewpoint of the server operating system, there is a strong possibility that it receives some data immediately after the connection is established.
initialize the socket state as “readable” when it is obtained by `accept()`. Although this initialization may incur an unnecessary read I/O request, the correct state of the kernel read buffer can be obtained by the initial read I/O. This is an overhead of the proposed mechanism to guarantee the semantics of polling I/O.

When a process invokes a read I/O function in the library, it invokes the corresponding read I/O in the operating system only if the socket is marked “readable.” If the socket is marked “not readable”, the read I/O just returns to the caller with an EAGAIN error without an invocation of the system read I/O. When read I/O returns data smaller than the prepared application buffer, we can judge the kernel read buffer of the socket gets empty. In this situation, the library marks the socket “not readable.”

### 2.2 Setting Up Non-blocking I/O

Our library utilizes non-blocking I/O on all the sockets by default. There are three reasons to use non-blocking I/O.

First, non-blocking I/O can avoid all the I/O blocking that a socket is writable, there is a possibility of I/O blocking because the process may try to write a data chunk larger than the available kernel buffer size. With non-blocking I/O, the write I/O returns to the caller process immediately after it copies the data into the kernel buffer as large as possible.

Second, as we mentioned in Section 2.1, non-blocking I/O can inform the caller process of the buffer state. When a process uses blocking I/O, each I/O usually completes successfully even when it tries to write data larger than the available kernel buffer size by blocking the process and retry to write the remaining data. This feature hides the kernel buffer state from the caller process.

Last, there is an ambiguity problem of the socket state transition. The ambiguity problem is described as follows. When a process receives full-size data in the application read buffer on a read I/O, we generally judge there are still some data left in the kernel buffer. However, there is a possibility that the size of the data in the kernel buffer is exactly the same as that of the application read buffer. With real-time signal-driven I/O, there is no signal issued when the state of a socket moves from “readable” to “not readable.” The process therefore cannot judge the correct state of the kernel buffer unless it issues another non-blocking read I/O request. If there is no data in the kernel buffer, this I/O invocation is just a waste of processing power from a viewpoint of I/O throughput although it informs the process that the kernel buffer is empty. A similar scenario can be provided on a write I/O, i.e., there is a possibility that the kernel buffer gets full when a non-blocking write I/O completes successfully. To obtain the correct buffer state, we need another non-blocking write I/O on the socket.

### 2.3 False Notification on Closed Sockets

There is a false notification problem on closed sockets in the real-time signal-driven I/O model. When a socket is closed, all the signals for the socket existing in the signal queue turn false because the descriptor of the socket may be recycled. The literature [3] discussed the same issue in the kqueue of FreeBSD. These false signals can be discriminated if they are retrieved from the signal queue before the descriptor is recycled. However, it is almost impossible because a socket descriptor may be recycled just after the socket is closed.

A straightforward solution of this problem is to scan the signal queue to remove or invalidate all the signals for a socket being closed. This solution requires modification in the operating system and it is therefore not suitable for the proposed mechanism.

We propose the following steps to solve the false notifi-
cation problem. Our approach enables the library to detect the false notification.

1. When the library receives a request to close a socket, the library invokes `shutdown()` to close the connection and keeps the descriptor unclosed in the closed connection socket list.

2. When the library detects the empty signal queue after it tries to retrieve a signal from the queue, it closes all the descriptors in the closed connection socket list.

To make this solution works correctly, the signals in the signal queue should be retrieved as fast as possible. We therefore prepare a special thread whose only task is the signal retrieval and give a high execution priority to this thread. This architecture is important also to prevent the signal queue overflow problem mentioned in the following subsection.

### 2.4 Signal Queue Overflow

When we use POSIX real-time signals, we have to be prepared for a signal queue overflow. Once a signal queue overflow occurs, we have to recover the lost signal information by switching the I/O model to some other one such as the traditional polling I/O model. However, switching the I/O model is not practical because other I/O models are generally less efficient than the real-time signal-driven I/O model. The signal queue overflow indicates a highly-loaded state of the server, and switching the I/O model can make the situation worse.

Although we give a high execution priority to the signal information retrieval to prevent a signal queue overflow, we still need some recovery mechanism from the signal queue overflow. As a solution, we simply mark all the sockets I/O-ready (readable and writable) when a signal queue overflow occurs. Although this overwrites the state of each socket incorrectly, the following non-blocking I/O execution modifies the state table correctly. This mechanism has the following advantages.

- When a signal queue overflow occurs, the process can concentrate on the I/O execution and improve the I/O throughput, which removes the temporal overload.
- The server program is not required to move on other I/O models and can continue the utilization of the real-time signal-driven I/O model.

### 2.5 Blocking I/O

The library also supports blocking I/O. Although the library sets up all the sockets in the non-blocking I/O mode, we may need blocking I/O functions. The library prepares a conditional variable on each socket, and the threads can wait for a state change on a socket. When a signal is delivered on a socket, the library unblocks all the threads waiting for the event. This mechanism imitates blocking I/O.

### 3 Implementation

We have implemented the fast polling I/O mechanism as a middleware library. The C language is used for the implementation, and its code size is about 4,000 lines including comments and blank lines. In this section, we describe the basic architecture and the APIs.

#### 3.1 Architecture

This library uses two cooperative threads in addition to the program threads that link the library. The reason we adopt a multi-threaded architecture is that we need processing entities asynchronous to the application process/threads. The threads generated in the library are the signal retriever thread and the timer thread. The basic relationship among the threads including the application threads is depicted in Figure 3.

The signal retriever thread prepares the memory area for the socket state management table and configures itself as a signal handler. Then, it enters the main loop in which it retrieves a real-time signal from the signal queue and processes the information stored in the signal. The signal retrieval is done in a non-blocking mode by invoking `sigtimedwait()` with a zero timeout argument. When it detects the signal queue is empty, it marks the descriptors in the closed connection socket list as we mentioned in Section 2.3.

The timer thread is woke up periodically (with a 100-millisecond interval). It scans the closed connection socket list, and invokes `close()` for each descriptor in the list marked by the signal retriever thread.

#### 3.2 Programming Interfaces

A program that uses this communication management library has to issue I/O requests through the library. The library provides `ufdSocket()`, `ufdAccept()`, `ufdPoll()`, `ufdRead()`, `ufdWrite()`, and `ufdClose()`, which substitute for `socket()`, `accept()`, `poll()`, `read()`, `write()`, and `close()` respectively. It also implements blocking I/O.

---

2 This thread unblocking strategy may incur a thundering herd problem [7]. The library can be configured to unblock only a single thread among those waiting for the event to avoid the problem.
version of the functions such as ufdAcceptBlock(), ufdReadBlock(), and ufdWriteBlock().

4 Benchmark Tests

To evaluate the performance of the fast polling I/O mechanism, we conducted benchmark tests. In this section, we describe the results observed from the tests.

4.1 Micro-Benchmark Program

The major target of this communication management library is a network server that handles tens of thousands of concurrent connections. In these years, several sophisticated communication mechanisms have been introduced into application network protocols such as the persistent connection mechanism in HTTP/1.1. Today’s highly-loaded network servers thus handle a small number of active connections together with a huge number of idle ones.

The benchmark test environment consisted of five client hosts and one server host connected to each other via a network switch. The server host was connected with a gigabit ethernet link, and each of the clients was with a fast ethernet link.

In this environment, each client established a great number of connections to the server and then sent pseudo requests of 1 KB data in parallel and received pseudo responses of 10 KB data. The server can switch between traditional poll() and our ufdPoll() as the polling I/O function. To focus on the performance evaluation of the fast polling I/O mechanism, the clients and the server did not process any content data in the requests and the responses. In addition, to emulate the situation that most of the connections were idle, we restricted the number of the connections in active on the clients. An active connection is defined as a connection in the period between the beginning of a request transfer and the end of a response receipt. In the benchmark tests, we parametrized the total number of the concurrent connections and the ratio of the active connections to all the connections. We set the total connection number at 5,000, 10,000, 20,000, and 40,000, and the active connection ratio at 0.5%, 1.0%, 2.0%, and 4.0%.

4.2 Processing Cost and Throughput

Figure 4 and 5 depict the mean processing time of a polling I/O invocation and the service throughput achieved in the benchmark tests respectively. The “ACR” in these figures stands for “active communication ratio.” From Figure 4, we can first observe that the fast polling I/O mechanism largely decreased the processing time of a polling I/O invocation. In all cases, the processing time was reduced by over 80%. The large processing time of traditional poll() is caused by various factors such as memory allocations and copies required to process a system call, a large number of pointer operations and function calls through VFS (Virtual File System) to check the state of each socket, and wait channel management to handle events on every socket. In the fast polling I/O library, most of these overheads are removed.

We can also observe from Figure 5 that our fast polling I/O improved the performance especially when the active connection ratio was low. The major reason of this performance improvement is the reduced processing time of polling I/O. When the active connection ratio is low and the network bandwidth for the service is provided sufficiently, the service throughput is decided mainly by the intervals of polling I/O invocations. Longer intervals
Figure 4: Mean processing time of a single polling I/O invocation (left half: `poll()`, right half: `ufdPoll()`).

Figure 5: Service throughput of the server (left half: `poll()`, right half: `ufdPoll()`).

of polling I/O invocations make lower service throughput. With traditional polling I/O, its large processing time makes the invocation intervals long and thus the service throughput low. This also explains the smaller performance gap between traditional polling I/O and our proposed one in case of higher active connection ratios. In this case, more clients can send a request to the server, and the server thus can process more requests between the polling I/O invocations.

5 Discussion

In this section, we discuss two remaining issues on the fast polling I/O mechanism. We also describe related work.

5.1 Implicit Utilization of Non-blocking I/O

The proposed mechanism is based on non-blocking I/O. Hence, server developers have to pay attention to the I/O semantics when they introduce the fast polling I/O mechanism into an existing network server. There are two points to be tackled.

One is `EAGAIN` errors (or `EWOULDBLOCK` errors on some systems). For instance, this error is returned when a process invokes a read I/O function in the non-blocking I/O mode when there is no data in the kernel buffer to be read. In this situation, the server program can simply defer the I/O execution preserving the I/O state and move on the next descriptor. The `EAGAIN` error does not occur consecutively on a socket with our mechanism since the error makes the I/O state of the socket “not readable.”

The other point is careful manipulation of the returned value by write I/O. When non-blocking I/O is utilized, write I/O can return a value smaller than the data size given as an argument. The server program has to increment the pointer into the I/O buffer and retry the remaining I/O afterward.

Incidentally, the fast polling I/O mechanism provides both non-blocking I/O and blocking I/O. When blocking I/O is invoked, it is emulated using non-blocking I/O and the POSIX thread synchronization mechanism. Our library just blocks the caller of blocking I/O until the I/O completes.

5.2 Restriction on Programming Interface

The proposed mechanism manages the state of each socket not only with real-time signals but with the results of normal (read and write) I/O execution. All the sockets handled through this library are set non-blocking. Therefore, it cannot work correctly if a server program invokes I/O functions directly in the operating system kernel or the system libraries. Server developers have to modify all the I/O execution points in an existing program to utilize this library. Meanwhile, modification of such I/O points is not a tough work in many cases because the number of the I/O functions provided by traditional Unix operating systems is small enough to check the entire server program. If the server program utilizes an external or internal communication middleware that aggregates the communication functions\(^3\), we can easily integrate our mechanism into that middleware.

\(^3\)Communication middleware that provides unified programming interfaces for both Microsoft Windows API and Unix API is a good example.
5.3 Related Work

We mentioned that the multi-thread model involves an issue of a high thread management cost. To solve this issue, Behren et al. proposed a multi-thread model with considerably low thread management cost [8]. The light-weight threads are implemented in the userland (pure user-thread model) using a coroutine library on Linux. I/O is executed in the non-blocking mode, and I/O multiplexing is managed by the epoll function set of Linux.

The motivation of our fast polling I/O library is similar to that of this multi-thread model; utilization of a popular programming model makes development of a server program simple and easy. On the other hand, our approach is the polling I/O model, whose functions are implemented with POSIX real-time signals.

6 Conclusion

In this paper, we proposed a communication management mechanism with the API of polling I/O, whose functionality is implemented with POSIX real-time signal-driven I/O. The benefit of this mechanism is summarized as follows.

- Server developers can utilize polling I/O, which is one of the most popular I/O multiplexing models for high performance network servers.
- Performance is enhanced especially when a server handles a small number of active connections mixed in a huge number of idle connections.
- POSIX real-time signals are available on most Unix operating systems today and thus the implementation is highly portable.

References