Polling and Signaling 6

Input/Output Polling

This chapter describes the synchronous polling mechanism and asynchronous event notification within STREAMS. Also discussed is how a Stream can be a controlling terminal.

User processes can efficiently monitor and control multiple Streams with two system calls: `poll`(2) and the `I_SETSIG` `ioctl`(2) command. These calls allow a user process to detect events that occur at the Stream head on one or more Streams, including receipt of data or messages on the read queue and cessation of flow control. Note that `poll`() is usable on any file descriptor, not just STREAMS. The precursor to `poll`() was `select`().

To monitor Streams with `poll`(2), a user process issues that system call and specifies the Streams and other files to be monitored, the events to look for, and the amount of time to wait for an event. The `poll`(2) system call will block the process until the time expires or until an event occurs. If an event occurs, it will return the type of event and the descriptor on which the event occurred.

Instead of waiting for an event to occur, a user process may want to monitor one or more Streams while processing other data. It can do so by issuing the `I_SETSIG` `ioctl`(2) command, specifying one or more Streams and events (as with `poll`(2)). This `ioctl` does not block the process and force the user process to wait for the event but returns immediately and issues a signal when an event occurs. The process must request `signal`(2) to catch the resultant `SIGPOLL` signal.

If any selected event occurs on any of the selected Streams, STREAMS will send `SIGPOLL` to all associated requesting processes. However, the process(es) will not know which event occurred, nor on what Stream the event occurred. A process that issues the `I_SETSIG` can get more detailed information by issuing a `poll` after it detects the event.

Synchronous Input/Output

The `poll`(2) system call provides a mechanism to identify those Streams over which a user can send or receive data. For each Stream of interest, users can specify one or more events about which they should be notified. The types of events that can be polled are `POLLIN`, `POLLRDNORM`, `POLLRDBAND`, `POLLPRI`, `POLLOUT`, `POLLWRNORM`, `POLLWRBAND`:

`POLLIN`	A message other than an `M_PCPROTO` is at the front of the Stream head read queue. This event is maintained for compatibility with the previous releases of Solaris.
`POLLRDNORM`	A normal (non-priority) message is at the front of the Stream head read queue.
`POLLRDBAND`	A priority message (band > 0) is at the front of the Stream head queue.
`POLLPRI`	A high priority message (`M_PCPROTO`) is at the front of the Stream head read queue.
`POLLOUT`	The normal priority band of the queue is writable (not flow controlled).
`POLLWRNORM`	The same as `POLLOUT`.
`POLLWRBAND`	A priority band greater than 0 of a queue downstream.

Some of the events may not be applicable to all file types. For example, it is not expected that the `POLLPRI` event will be generated when polling a regular file. `POLLIN`, `POLLRDNORM`, `POLLRDBAND`, and `POLLPRI` are set even if the message is of zero length.

The `poll` system call will examine each file descriptor for the requested events and, on return, will indicate which events have occurred for each file descriptor. If no event has occurred on any polled file descriptor, `poll` blocks until a requested event or timeout occurs. `poll`(2) takes the following arguments:

- an array of file descriptors and events to be polled
- the number of file descriptors to be polled
- the number of milliseconds `poll` should wait for an event if no events are pending (-1 specifies wait forever)

The following example shows the use of `poll`. Two separate minor devices of the communications driver are opened, thereby establishing two separate Streams to the driver. The *pollfd* entry is initialized for each device. Each Stream is polled for incoming data. If data arrive on either Stream, data is read and then written back to the other Stream.

```
#include <sys/stropts.h>
#include <fcntl.h>
#include <poll.h>

#define NPOLL 2       /* number of file descriptors to poll */
int
main()
{
    struct pollfd pollfds[NPOLL];
    char buf[1024];
    int count, i;

    if ((pollfds[0].fd = open("/dev/ttya",
              O_RDWR|O_NONBLOCK)) < 0) {
          perror("open failed for /dev/ttya");
          exit(1);
    }
    if ((pollfds[1].fd = open("/dev/ttyb",
              O_RDWR|O_NONBLOCK)) < 0) {
          perror("open failed for /dev/ttyb");
          exit(2);
    }
```

The variable *pollfds* is declared as an array of the `pollfd` structure that is defined in `<poll.h>` and has the following format:

```
struct pollfd {
    int fd;                     /* file descriptor */
    short events;               /* requested events */
    short revents;              /* returned events */
}
```

For each entry in the array, *fd* specifies the file descriptor to be polled and *events* is a bitmask that contains the bitwise inclusive OR of events to be polled on that file descriptor. On return, the *revents* bitmask will indicate which of the requested events has occurred.

The example continues to process incoming data as follows:

Code Example 6-1 Polling

```
    pollfds[0].events = POLLIN;/* set events to poll */
    pollfds[1].events = POLLIN;/* for incoming data */

    while (1) {
            /* poll and use -1 timeout (infinite) */
            if (poll(pollfds, NPOLL, -1) < 0) {
                perror("poll failed");
                exit(3);
            }
            for (i = 0; i < NPOLL; i++) {
                switch (pollfds[i].revents) {

                default:                /* default error case */
                        perror("error event");
                        exit(4);

                case 0:                 /* no events */
                        break;

                case POLLIN:
                        /*echo incoming data on "other" Stream*/
                        while ((count = read(pollfds[i].fd,
                         buf, 1024)) > 0)
                        /*
                         * write loses data if flow control
```

Code Example 6-1 Polling

```
                        * prevents the transmit at this time
                        */
                       if (write(pollfds[(i+1) % NPOLL].fd buf,
                           count) != count)
                           fprintf(stderr,"writer lost data");
                       break;
                   }
               }
       }
}
```

The user specifies the polled events by setting the *events* field of the pollfd structure to POLLIN. This requested event directs poll to notify the user of any incoming data on each Stream. The bulk of the example is an infinite loop, where each iteration will poll both Streams for incoming data.

The second argument to the `poll` system call specifies the number of entries in the *pollfds* array (2 in this example). The third argument is a timeout value indicating the number of milliseconds `poll` should wait for an event if none has occurred. On a system where millisecond accuracy is not available, *timeout* is rounded up to the nearest value available on that system. If the value of *timeout* is 0, `poll` returns immediately. Here, the value of *timeout* is -1, specifying that `poll` should block until a requested event occurs or until the call is interrupted.

If the `poll` call succeeds, the program looks at each entry in the *pollfds* array. If *revents* is set to 0, no event has occurred on that file descriptor. If *revents* is set to `POLLIN`, incoming data is available. In this case, all available data is read from the polled minor device and written to the other minor device.

If *revents* is set to a value other than 0 or `POLLIN`, an error event must have occurred on that Stream, because `POLLIN` was the only requested event. The following are `poll` error events:

`POLLERR` A fatal error has occurred in some module or driver on the Stream associated with the specified file descriptor. Further system calls will fail.

POLLHUP	A hangup condition exists on the Stream associated with the specified file descriptor. This event and POLLOUT are mutually exclusive; a Stream can't be writable if a hangup has occurred.
POLLNVAL	The specified file descriptor is not associated with an open Stream.

These events may not be polled for by the user, but will be reported in *revents* whenever they occur. As such, they are only valid in the *revents* bitmask.

The example attempts to process incoming data as quickly as possible. However, when writing data to a Stream, the `write` call may block if the Stream is exerting flow control. To prevent the process from blocking, the minor devices of the communications driver were opened with the `O_NDELAY` (or `O_NONBLOCK`, see note) flag set. The `write` will not be able to send all the data if flow control is exerted and `O_NDELAY` (`O_NONBLOCK`) is set. This can occur if the communications driver is unable to keep up with the user's rate of data transmission. If the Stream becomes full, the number of bytes the `write` sends will be less than the requested *count*. For simplicity, the example ignores the data if the Stream becomes full, and a warning is printed to `stderr`.

Note – For conformance with the IEEE operating system interface standard, POSIX, it is recommended that new applications use the `O_NONBLOCK` flag, whose behavior is the same as that of `O_NDELAY` unless otherwise noted.

This program continues until an error occurs on a Stream, or until the process is interrupted.

Asynchronous Input/Output

The `poll` system call enables a user to monitor multiple Streams synchronously. The `poll`(2) call normally blocks until an event occurs on any of the polled file descriptors. In some applications, however, it is desirable to process incoming data asynchronously. For example, an application may wish to do some local processing and be interrupted when a pending event occurs. Some time-critical applications cannot afford to block, but must have immediate indication of success or failure.

The I_SETSIG ioctl call (see streamio(7)) is used to request that a SIGPOLL signal be sent to a user process when a specific event occurs. Listed below are events for the ioctl I_SETSIG. These are similar to those described for poll(2).

S_INPUT	A message other than an M_PCPROTO is at the front of the Stream head read queue. This event is maintained for compatibility with the previous releases of Solaris.
S_RDNORM	A normal (non-priority) message is at the front of the Stream head read queue.
S_RDBAND	A priority message (band > 0) is at the front of the Stream head read queue.
S_HIPRI	A high priority message (M_PCPROTO) is present at the front of the Stream head read queue.
S_OUTPUT	A write queue for normal data (priority band = 0) is no longer full (not flow controlled). This notifies a user that there is room on the queue for sending or writing normal data downstream.
S_WRNORM	The same as S_OUTPUT.
S_WRBAND	A priority band greater than 0 of a queue downstream exists and is writable. This notifies a user that there is room on the queue for sending or writing priority data downstream.
S_MSG	An M_SIG or M_PCSIG message containing the SIGPOLL flag has reached the front of Stream head read queue.
S_ERROR	An M_ERROR message reaches the Stream head.
S_HANGUP	An M_HANGUP message reaches the Stream head.
S_BANDURG	When used with S_RDBAND, SIGURG is generated instead of SIGPOLL when a priority message reaches the front of the Stream head read queue.

S_INPUT, S_RDNORM, S_RDBAND, and S_HIPRI are set even if the message is of zero length. A user process may choose to handle only high priority messages by setting the *arg* to S_HIPRI.

6

Signals

STREAMS allows modules and drivers to cause a signal to be sent to user process(es) through an `M_SIG` or `M_PCSIG` message. The first byte of the message specifies the signal for the Stream head to generate. If the signal is not `SIGPOLL` (see `signal`(2)), the signal is sent to the process group associated with the Stream. If the signal is `SIGPOLL`, the signal is only sent to processes that have registered for the signal by using the `I_SETSIG ioctl`(2).

An `M_SIG` message can be used by modules or drivers that wish to insert an explicit in-band signal into a message Stream. For example, this message can be sent to the user process immediately before a particular service interface message to gain the immediate attention of the user process. When the `M_SIG` message reaches the head of the Stream head read queue, a signal is generated and the `M_SIG` message is removed. This leaves the service interface message as the next message to be processed by the user. Use of the `M_SIG` message is typically defined as part of the service interface of the driver or module.

Extended Signals

To enable a process to obtain the band and event associated with `SIGPOLL` more readily, STREAMS supports extended signals. For the given events, a special code is defined in <`sys/siginfo.h`> that describes the reason `SIGPOLL` was generated. The following table describes the data available in the `siginfo_t` structure passed to the signal handler.

Table 6-1 Data in siginfo_t structure

Event	si_signo	si_code	si_band	si_errno
S_INPUT	SIGPOLL	POLL_IN	band readable	unused
S_OUTPUT	SIGPOLL	POLL_OUT	band writable	unused
S_MSG	SIGPOLL	POLL_MSG	band signaled	unused
S_ERROR	SIGPOLL	POLL_ERR	unused	Stream error
S_HANGUP	SIGPOLL	POLL_HUP	unused	unused
S_HIPRI	SIGPOLL	POLL_PRI	unused	unused

Stream as a Controlling Terminal

Job Control

An overview of Job Control is provided here because it interacts with the STREAMS-based terminal subsystem. More information on Job Control may be obtained from the following manual pages: `exit`(2), `getpgid`(2), `getpgrp`(2), `getsid`(2), `kill`(2), `setpgid`(2), `setpgrp`(2),`setsid`(2), `sigaction`(2), `signal`(2), `sigsend`(2), `termios`(2), `waitid`(2), `waitpid`(3C), `signal`(5), and `termio`(7).

Job Control breaks a login session into smaller units called *jobs*. Each job consists of one or more related and cooperating processes. One job, the *foreground job*, is given complete access to the controlling terminal. The other jobs, *background jobs*, are denied read access to the controlling terminal and given conditional write and `ioctl` access to it. The user may stop the executing job and resume the stopped job either in the foreground or in the background.

Under Job Control, background jobs do not receive events generated by the terminal and are not informed with a hangup indication when the controlling process exits. Background jobs that linger after the login session has been dissolved are prevented from further access to the controlling terminal, and do not interfere with the creation of new login sessions.

The following list defines terms associated with Job Control:

- **Background Process group** - A process group that is a member of a session that established a connection with a controlling terminal and is not the foreground process group.
- **Controlling Process** - A session leader that established a connection to a controlling terminal.
- **Controlling Terminal** - A terminal that is associated with a session. Each session may have at most one controlling terminal associated with it and a controlling terminal may be associated with at most one session. Certain input sequences from the controlling terminal cause signals to be sent to the process groups in the session associated with the controlling terminal.

- **Foreground Process Group** - Each session that establishes a connection with a controlling terminal distinguishes one process group of the session as a foreground process group. The foreground process group has certain privileges that are denied to background process groups when accessing its controlling terminal.
- **Orphaned Process Group** - A process group in which the parent of every member in the group is either a member of the group, or is not a member of the process group's session.
- **Process Group** - Each process in the system is a member of a process group that is identified by a process group ID. Any process that is not a process group leader may create a new process group and become its leader. Any process that is not a process group leader may join an existing process group that shares the same session as the process. A newly created process joins the process group of its creator.
- **Process Group Leader** - A process whose process ID is the same as its process group ID.
- **Process Group Lifetime** - A time period that begins when a process group is created by its process group leader and ends when the last process that is a member in the group leaves the group.
- **Process ID** - A positive integer that uniquely identifies each process in the system. A process ID may not be reused by the system until the process lifetime, process group lifetime, and session lifetime ends for any process ID, process group ID, and session ID sharing that value.
- **Process Lifetime** - A time period that begins when the process is forked and ends after the process exits, when its termination has been acknowledged by its parent process.
- **Session** - Each process group is a member of a session that is identified by a session ID.
- **Session ID** - A positive integer that uniquely identifies each session in the system. It is the same as the process ID of its session leader. (POSIX)
- **Session Leader** - A process whose session ID is the same as its process and process group ID.
- **Session Lifetime** - A time period that begins when the session is created by its session leader and ends when the lifetime of the last process group that is a member of the session ends.

The following signals manage Job Control: (see also `signal`(5))

`SIGCONT`	Sent to a stopped process to continue it.
`SIGSTOP`	Sent to a process to stop it. This signal cannot be caught or ignored.
`SIGTSTP`	Sent to a process to stop it. It is typically used when a user requests to stop the foreground process.
`SIGTTIN`	Sent to a background process to stop it when it attempts to read from the controlling terminal.
`SIGTTOU`	Sent to a background process to stop it when one attempts to write to or modify the controlling terminal.

A session may be allocated a controlling terminal. For every allocated controlling terminal, Job Control elevates one process group in the controlling process's session to the status of foreground process group. The remaining process groups in the controlling process's session are background process groups. A controlling terminal gives a user the ability to control execution of jobs within the session. Controlling terminals play a central role in Job Control. A user may cause the foreground job to stop by typing a predefined key on the controlling terminal. A user may inhibit access to the controlling terminal by background jobs. Background jobs that attempt to access a terminal that has been so restricted will be sent a signal that typically will cause the job to stop. (See "Accessing the Controlling Terminal" later in this chapter.)

Job Control requires support from a line-discipline module on the controlling terminal's Stream. The `TCSETA`, `TCSETAW`, and `TCSETAF` commands of `termio`(7) allow a process to set the following line discipline values relevant to Job Control:

`SUSP` character	A user defined character that, when typed, causes the line discipline module to request that the Stream head send a `SIGTSTP` signal to the foreground process with an `M_PCSIG` message, which by default stops the members of that group. If the value of `SUSP` is zero, the `SIGTSTP` signal is not sent, and the `SUSP` character is disabled.
`TOSTOP` flag	If `TOSTOP` is set, background processes are inhibited from writing to their controlling terminal.

A line discipline module must record the SUSP suspend character and notify the Stream head when the user has typed it, and record the state of the TOSTOP bit and notify the Stream head when the user has changed it.

Allocation and Deallocation

A Stream is allocated as a controlling terminal for a session if:

- The Stream is acting as a terminal,
- The Stream is not already allocated as a controlling terminal, and
- The Stream is opened by a session leader that does not have a controlling terminal.

Drivers and modules can inform the Stream head to act as a terminal Stream by sending an M_SETOPTS message with the SO_ISTTY flag set upstream. This state may be changed by sending an M_SETOPTS message with the SO_ISNTTY flag set upstream.

Controlling terminals are allocated with the open(2) system call. A Stream head must be informed that it is acting as a terminal by an M_SETOPTS message sent upstream before or while the Stream is being opened by a potential controlling process. If the Stream head is opened before receiving this message, the Stream is not allocated as a controlling terminal.

Hungup Streams

When a Stream head receives an M_HANGUP message, it is marked as hung-up. Streams that are marked as Hungup are allowed to be reopened by their session leader if they are allocated as a controlling terminal, and by any process if they are not allocated as a controlling terminal. This way, the hangup error can be cleared without forcing all file descriptors to be closed first.

If the reopen is successful, the Hungup condition is cleared.

Hangup Signals

When the SIGHUP signal is generated via an M_HANGUP message (instead of an M_SIG or M_PCSIG message), the signal is sent to the controlling process instead of the foreground process group, since the allocation and deallocation of controlling terminals to a session is the responsibility of that process group.

Accessing the Controlling Terminal

If a process attempts to access its controlling terminal after it has been deallocated, access will be denied. If the process is not holding or ignoring `SIGHUP`, it is sent a `SIGHUP` signal. Otherwise, the access will fail with an EIO error.

Members of background process groups have limited access to their controlling terminals:

- If the background process is ignoring or holding the `SIGTTIN` signal or is a member of an orphaned process group, an attempt to read from the controlling terminal will fail with an EIO error. Otherwise, the process is sent a `SIGTTIN` signal, which by default stops the process.
- If the process is attempting to write to the terminal and if the terminal's `TOSTOP` flag is clear, the process is allowed access.

  The `TOSTOP` flag is set upon reception of an `M_SETOPTS` message with the `SO_TOSTOP` flag set in the *so_flags* field. It is cleared upon reception of an `M_SETOPTS` message with the `SO_TONSTOP` flag set.
- If the terminal's `TOSTOP` flag is set and a background process is attempting to write to the terminal, the write will succeed if the process is ignoring or holding `SIGTTOU`. Otherwise, the process will stop except when it is a member of an orphaned process group, in which case it is denied access to the terminal and it is returned an EIO error.

If a background process is attempting to perform a destructive `ioctl` (an `ioctl` that modifies terminal parameters), the `ioctl` call will succeed if the process is ignoring or holding `SIGTTOU`. Otherwise, the process will stop except when the process is a member of the orphaned process group. In that case the access to the terminal is denied and an EIO error is returned.

6