Linux IO

Liran Ben Haim
liran@discoversdk.com

DiscoverSDK
Tools for Developers

Rights to Copy

creative commons
COMMONS DEED

- **Attribution – ShareAlike 2.0**
- **You are free**
  - to copy, distribute, display, and perform the work
  - to make derivative works
  - to make commercial use of the work

Under the following conditions

Attribution. You must give the original author credit.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

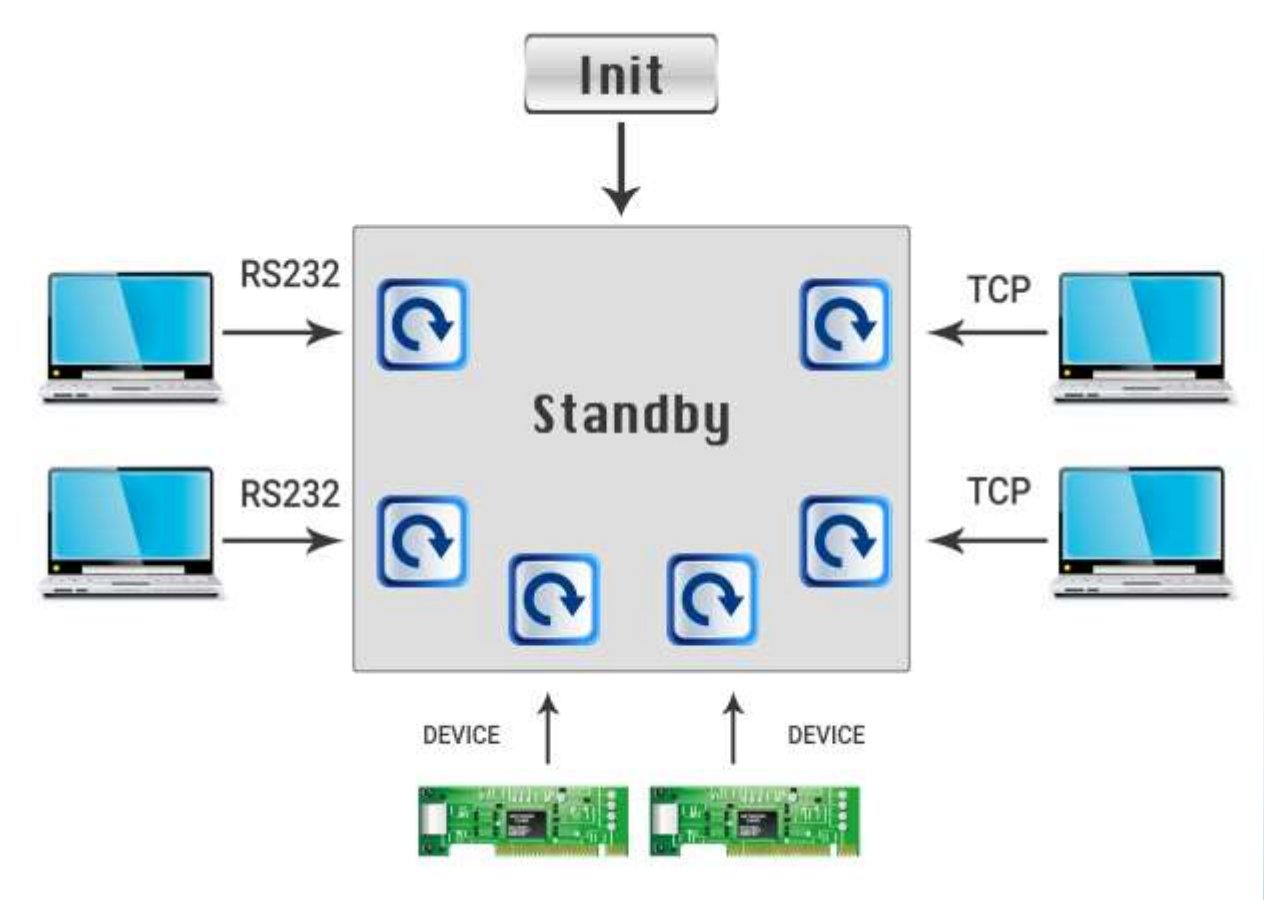
Your fair use and other rights are in no way affected by the above.

License text: http://creativecommons.org/licenses/by-sa/2.0/legalcode

- **This kit contains work by the following authors:**
- © Copyright 2004-2009
  Michael Opdenacker /Free Electrons
  michael@free-electrons.com
  http://www.free-electrons.com
- © Copyright 2003-2006
  Oron Peled
  oron@actcom.co.il
  http://www.actcom.co.il/~oron
- © Copyright 2004–2008
  Codefidence ltd.
  info@codefidence.com
  http://www.codefidence.com
- © Copyright 2009–2010
  Bina ltd.
  info@bna.co.il
  http://www.bna.co.il

DiscoverSDK
Tools for Developers

Typical System



DiscoverSDK
Tools for Developers

IO Models

- Blocking I/O
- Non-blocking I/O
- I/O multiplexing (select/poll/…)
- Signal driven I/O (SIGIO)
- Asynchronous I/O
  - aio_* functions
  - io_* functions

Blocking IO

- Default mode
- Makes the calling thread to block in the kernel in case no data is available
- Can block forever
  - To block with timeout, use select

DiscoverSDK
Tools for Developers

Non Blocking IO

- If there is no data, calling thread returns with EAGAIN or EWOULDBLOCK

flags = fcntl(fd, F_GETFL, 0);

fcntl(fd, F_SETFL, flags | O_NONBLOCK);

- To use in your own driver

```
if (filep->f_flags & O_NONBLOCK) {
        ret = -EWOULDBLOCK;
        break;
}
```

IO Multiplexing

- With **I/O multiplexing**, we call select/poll/epoll* and block in one of these system calls, instead of blocking in the actual I/O system call
- Disadvantage: using select requires at least two system calls (select and recvfrom) instead of one
- Advantage: we can wait for more than one descriptor to be ready

DiscoverSDK
Tools for Developers

Signal driven I/O

- The **signal-driven I/O model** uses signals, telling the kernel to notify us with the SIGIO signal when the descriptor is ready

```
fd=open("name", O_NONBLOCK);

fcntl(fd, F_SETSIG, SIGRTMIN + 1);
```

- Its better to use RT signals (queued)
- To use in your own driver
  - if (file->f_flags & O_NONBLOCK)
  - Send signal to file->f_owner (fown_struct)
    - signum
    - pid

Asynchronous IO

- The POSIX asynchronous I/O interface
  - Allows applications to initiate one or more I/O operations that are performed asynchronously.
  - The application can select to be notified of completion of the I/O operation in a variety of ways:
    - Delivery of a signal
    - Instantiation of a thread
    - No notification at all
  - User space asynchronous implementation
    - Call the driver read/write callbacks

Kernel Async Support

- file_operations callbacks
  - aio_read, aio_write
    - Initialize the data processing
    - Create workqueue item/completion/timer/…
  - On completion
    - aio_complete
- Kernel 3.16
  - read_iter
  - write_iter
- To use from user space call io_submit

I/O Multiplexing

- select, pselect
- poll, ppoll
- epoll
  - epoll_create, epoll_create1
  - epoll_ctl
  - epoll_wait, epoll_pwait

select(2) system call

- The select( ) system call provides a mechanism for implementing synchronous multiplexing I/O
- A call to select( ) will block until the given file descriptors are ready to perform I/O, or until an optionally specified timeout has elapsed
- The watched file descriptors are broken into three sets
  - File descriptors listed in the readfds set are watched to see if data is available for reading.
  - File descriptors listed in the writefds set are watched to see if a write operation will complete without blocking.
  - File descriptors in the exceptfds set are watched to see if an exception has occurred, or if out-of-band data is available (these states apply only to sockets).
- A given set may be NULL, in which case select( ) does not watch for that event.
- On successful return, each set is modified such that it contains only the file descriptors that are ready for I/O of the type delineated by that set

Blocking for events

- You can use select(2) to block for events
- int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
  - **nfds**: number of highest file descriptor + 1
  - **fd_sets**: sets of file descriptors to block for events on, one for read, one for write, one for exceptions.
  - **timeout**: NULL or structure describing how long to block.
  - Linux updates the timeout structure according to how much time was left to the time out.

File Descriptor Sets

- fd_set is a group of file descriptor for actions or reports:

```
void FD_CLR(int fd, fd_set *set);
```

  - Remove fd from this set.

```
int  FD_ISSET(int fd, fd_set *set);
```

  - Is fd in the set?

```
void FD_SET(int fd, fd_set *set);
```

  - Add fd to this set.

```
void FD_ZERO(fd_set *set);
```

  - Zero the set.

Driver notification example

```
int fd1, fd2;

fd_set fds_set;

int ret;

fd1 = open(“/dev/drv_ctl0”, O_RDWR);

fd2 = open(“/dev/drv_ctl1”, O_RDWR);

FD_ZERO(&fds_set);

FD_SET(fd1, &fds_set);

FD_SET(fd2, &fds_set);

do {

ret = select(fd1 + fd2 + 1, &fds_set, NULL, NULL,
NULL);


} while(errno == EINTR);
```

- Open two file descriptors fd1 and fd2.
- Create an fd set that holds them.
- Block for events on them.
  - We try again if we interrupted by a signal.

DiscoverSDK
Tools for Developers

Driver notification example 2

```
if(ret == -1) {

perror("Select failed.");

exit(1);

}



if(FD_ISSET(fd1, &fds_set)) {

printf("event at FD 1... “);

ioctl(fd1, IOCTL_CLR, 1);

printf("clear\n);

}
```

- If we have an error bail out.
- Check if fd1 one has a pending read even.
- If so, use ioctl(2) to notify driver to clear status

DiscoverSDK
Tools for Developers

pselect

- POSIX implementation
- Does not modify the timeout parameter
- Can set signal mask before entering to sleep
- Uses the timespec structure (seconds, nanoseconds)

DiscoverSDK
Tools for Developers

Poll

- **int poll (struct pollfd *fds, unsigned int nfds, int timeout);**

- Unlike select(), with its inefficient three bitmask-based sets of file descriptors, poll( ) employs a single array of nfds pollfd structures

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

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

- POSIX implementation – ppoll (with signal mask)

DiscoverSDK
Tools for Developers

```
// The structure for two events
struct pollfd fds[2];

// Monitor sock1 for input
fds[0].fd = sock1;
fds[0].events = POLLIN;

// Monitor sock2 for output
fds[1].fd = sock2;
fds[1].events = POLLOUT;

// Wait 10 seconds
int ret = poll( &fds, 2, 10000 );

     // Check if poll actually succeed
     if ( ret == -1 )
         // report error and abort
     else if ( ret == 0 )
         // timeout; no event detected
     else
     {
         // If we detect the event, zero it out so we can reuse the structure
         if ( pfd[0].revents & POLLIN )
             pfd[0].revents = 0;
             // input event on sock1

         if ( pfd[1].revents & POLLOUT )
             pfd[1].revents = 0;
             // output event on sock2
     }
```

https://www.ulduzsoft.com/2014/01/select-poll-epoll-practical-difference-for-system-architects/

Poll vs. Select

- poll( ) does not require that the user calculate the value of the highest-numbered file descriptor +1
- poll( ) is more efficient for large-valued file descriptors. Imagine watching a single file descriptor with the value 900 via select()—the kernel would have to check each bit of each passed-in set, up to the 900th bit.
- select( )'s file descriptor sets are statically sized.
- With select( ), the file descriptor sets are reconstructed on return, so each subsequent call must reinitialize them. The poll( ) system call separates the input (events field) from the output (revents field), allowing the array to be reused without change.
- The timeout parameter to select( ) is undefined on return. Portable code needs to reinitialize it. This is not an issue with pselect( )
- select( ) is more portable, as some Unix systems do not support poll( ).

DiscoverSDK
Tools for Developers

Event Poll

- Has state in the kernel
  - O(1) instead of O(n)
- epoll_create(2) – initializes epoll context
- epoll_ctl(2) – adds/removes file descriptors from the context
- epoll_wait(2) – performs the actual event wait
- Can behave as
  - Edge triggered
  - Level triggered

Example

```
int epfd = epoll_create(0);
int client_sock = socket(…..);
static struct epoll_event ev;
ev.events = EPOLLIN | EPOLLOUT | EPOLLERR;
ev.data.fd = client_sock;
int res = epoll_ctl(epfd, EPOLL_CTL_ADD, client_sock, &ev);
struct epoll_event *events = malloc(SIZE);
while (1) {
        int nfds = epoll_wait(epfd, events, MAX_EVENTS, TIMEOUT);
        for(int i = 0; i < nfds; i++) {
                int fd = events[i].data.fd;
                handle_io_on_socket(fd);
         }
}
```

Driver Implementation

- Implement poll callback on file_operations

```
unsigned int example_poll(struct file * file, poll_table * pt)
{
    unsigned int mask = 0;
    if (data_avail_to_read) mask |= POLLIN | POLLRDNORM;
    if (data_avail_to_write) mask |= POLLOUT | POLLWRNORM;
    poll_wait(file, &hr, pt);
    poll_wait(file, &hw, pt);
    return mask;
}
```

- The driver adds a wait queue to the poll_table structure by calling the function poll_wait

DiscoverSDK
Tools for Developers

Thank You

Code examples and more

http://www.discoversdk.com/blog