

# Opiframe Oy





- The traditional blocking I/O model is sufficient for many applications
- However there are applications that require following:
- Check a file descriptor for possible I/O without blocking.
- Monitor multiple file descriptors to see if I/O is possible on any of them.





- There are different ways of handling both of these situations
- You can use non blocking I/O for the first case and handle possible errors.
- You can use multiple threads to handle the second case.



- Non blocking I/O usually polls whether I/O is possible.
- This is usually very undesirable behavior since it leads either to long latency on I/O or much overhead on processor time.
- This demand increases with multiple file descriptors.



- Using multiple processes or threads to handle multiple file descriptors uses loads of memory and other resources.
- In case of processes you most likely have to also use the cumbersome IPC methods to communicate between different file descriptors.
- The problem is of course expense and complexity.





- Because of the limitations of both of these approaches some other methods have been devised.
- I/O Multiplexing allows process to simultaneously monitor multiple file descriptors.
- This is handled by either select() or poll() system call.



- Signal-driven I/O is a technique where a process is warned by the kernel with a signal when I/O is possible.
- This is also historically called asynchronous I/O although there is an API for POSIX Async I/O.
- When monitoring a large number of file descriptors (hundreds+) this
  is preferable to select() or poll()



- The epoll API is a Linux-specific feature that first appeared in kernel series 2.6.
- Like I/O Multiplexing, epoll API allows process to monitor multiple file descriptors for possible I/O.
- And like signal-driven I/O epoll provides much better performance when monitoring large numbers of file descriptors.





- In effect, I/O Multiplexing, signal-driven I/O and epoll are all methods to achieve same result:
- To monitor several file descriptors simultaneously to see whether any or all of them are *ready* to perform I/O.
- To be precise, ready, here means that the I/O system call (commonly read() or write()) would not block.



- The transition of a file descriptor into a ready state is triggered by some kind of I/O event.
- These include but are not limited to arrival of input, completion of socket connection or availability of space in previously full socket send buffer.
- Note that none of these I/O techniques actually do any I/O. They
  merely tell when the descriptor is ready.







- There are two distinct ways for I/O events to happen. These are called readiness notifications.
- Level-Triggered notification: A file descriptor is considered ready if it is possible to perform I/O without blocking.
- Edge-Triggered notification: Notification is provided if there is I/O activity on file descriptor since it was last monitored.



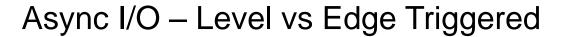


- I/O Multiplexing (select() and poll()) uses level-triggered notification.
- Signal-driven I/O uses edge-triggered notification.
- epoll uses both.
- There are a couple of major differences in handling of these readiness notifications.





- With level-triggered notification we can check the availability of the file descriptor for I/O at any time.
- So we can perform some I/O on the descriptor and check if there is still possibility for more I/O
- In other words, since we check availability at any time there is no need to read or write as much as possible when I/O notification is triggered.



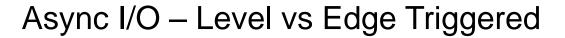


- By contrast when we employ edge-triggered notification we only receive the notification when I/O event occurs.
- Also we do not usually know the amount of I/O is possible at the time of the notification.
- Therefore, all programs that employ edge-triggered notification should be designed according to following two rules:





- After the notification for I/O happens, the program should at some point perform as much I/O as possible on corresponding file descriptor."
- "At some point" is a safety instruction. If you perform extremely large I/O operations on one file descriptor when monitoring several you might starve the other descriptors.
- So choosing the best possible point is part of the design.





- When dealing with looped I/O handling all your file descriptors should be in non blocking mode.
- Since we loop around a command to read from or write into a file descriptor at some point the I/O will end and the system call would otherwise block.
- So we perform these I/O commands until we get either EAGAIN or EWOULDBLOCK error message from read() or write().



#### Async I/O – Non Blocking I/O



- Some examples of the use of non blocking I/O
- The reason already explained before
- If multiple processes (or threads) are performing I/O on the same file descriptor then from particular processes view the file descriptor might change state between the check and actual I/O handling.
- If we are writing a large enough block into stream socket even after level-triggered notification, the call will block.





- We perform I/O multiplexing using either of two system calls which essentially do the same thing.
- The calls are **select()** and **poll()**.
- We can use these to monitor file descriptors for regular files, terminals, pseudoterminals, pipes, FIFOs, sockets and some types of character devices.
- Both allow process to block indefinitely or set a timeout.





The select() system call has following prototype

```
int select(int ndfs, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval
    *timeout);
```

- select() blocks until one or more file descriptors becomes ready or the timeout expires.
- The fd set arguments specify the sets of file descriptors to monitor.
- The ndfs must be one greater than the largest number of file descriptors in any of the three sets







- readfds is the set of file descriptors to be tested to see if input is possible
- writefds is the set of file descriptors to be tested to see if output is possible
- and exceptfds is the set of file descriptors to be tested for exceptional conditions





- The exceptional conditions are not errors!
- There are two exceptional conditions that relate to specific file descriptor types.



- The fd set type is implemented as a bit mask.
- It is opaque and should not be handled directly.
- The macros to handle fd sets are:
- FD\_ZERO(fd\_set \*set) for zeroing



- FD\_SET(int fd, fd\_set \*set) for adding into a set
- FD\_CLR(int fd, fd\_set \*set) for removing from a set
- **int FD\_ISSET**(int *fd*, fd\_set \*set) for checking if fd is in (return value 1) or not in (return value 0).
- The fd\_sets have a maximum amount of file descriptors they can handle.



- This is defined by a constant FD\_SETSIZE which in Linux is set at 1024.
- There is no easy way to change this constant.
- Changing it requires changing the header select.h.
- If you are handling more than 1024 file descriptors using epoll is preferable anyway.



- Before the call to select() all the fd\_set structures must be initialized with FD\_ZERO or set as NULL in the call.
- You add your file descriptors into the sets you are interested in.
- If we use a looping structure to loop around a select call this initialization must be done on all loops.
- This is because the initialization modifies these fd set structures.





- At least one of the file descriptors given in any of the fd\_set arguments becomes ready or the call is interrupted by a signal handler.
- If the call is interrupted by a signal handler it will fail with EINTR as usual.
- Timeout argument brings in some portability issues. Portable programs should always initialize the timeval struct and ignore possible changes to it due signal interruption.





- The return values of select() are as follows:
- Return value of -1 indicates an error. Possible errors include EBADF for bad file descriptor or EINTR for interruption by a signal handler.
- Return value of 0 means that the call timeouted. In this case all returning fd\_sets are empty.
- Positive integer is the total number of ready file descriptors in all sets.
   Same file descriptor in multiple sets is counted multiple times.





- As with select() poll() monitors a set of file descriptors for given events.
- The difference is that in select() we give out three distinct sets of file descriptors but with poll() we give out one list of file descriptors and their corresponding events.
- The prototype for poll() is:

```
int poll(struct pollfd fds[],nfds_t nfds, int timeout);
```







The struct pollfd has three members:

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

- fd is the specified file descriptor.
- events are the flags specifying the events we are interested in.
- revents are the returned event flags.





- The input events concerned with either **events** or **revents** are:
- POLLIN (events/revents) data can be read
- POLLRDNORM (events/revents) same as POLLIN
- POLLPRI(events/revents) priority data can be read.
- POLLRDHUP(events/revents) shutdown on the peer socket.



- The output events concerned with either events or revents:
- POLLOUT (events/revents) data can be written
- POLLWRNORM(events/revents) same as POLLOUT, used sometimes with sockets.
- POLLWRBAND(events/revents) priority data can be written.



- Other flags, used with revents.
- POLLERR, an error occurred
- POLLHUP, a hang up occurred

4/23/2014

POLLNVAL, file descriptor is not open.







- It is possible to define events as 0 if you are not interested in events on particular file descriptor.
- This makes it possible to temporarily not to be interested in a file descriptor without changing the whole fds list.





- The timeout argument has three distinct behaviors.
- If the value is -1, poll() blocks until one of the file descriptors listed is ready or a signal is caught.
- If the value is 0, just check whether anyone is ready, do not block.
- If value is greater than 0, block for up to timeout milliseconds, or until one file descriptor is ready or a signal is caught.



- Return values for poll() are following:
- -1 indicates an error similar to select()
- 0 indicates that the call timed out before any file descriptor became ready.
- Positive integer value indicates the one or more file descriptors are ready. The returned value is the number of **pollfd** structures in the fds array that have a nonzero **revents** field.

#### Is a file descriptor ready?



- Correctly using select() or poll() requires understanding of what it means when a file descriptor is ready.
- The gist of the idea is here: A file descriptor (in blocking mode) is considered to be ready if a call to an I/O function would not block, regardless of whether the function would actually transfer data.
- So the idea is not data transfer but whether an I/O operation would block.

## Implementation details



- Within the kernel, select() and poll() employ the same set of kernel-internal poll routines.
- These poll routines are different from poll() system call and should not be confused.
- Each poll routine returns information about readiness of a single file descriptor.

## Implementation details



- **poll()** system call calls the kernel poll routine for each file descriptor and places the resulting information in the corresponding revents field.
- For **select()** the return values of the poll routine are grouped into three distinct groups as per select calls arguments.
- For **poll()** there is the additional **POLLNVAL** which is not featured in **select()** groupings.
- For **select()** the same thing is the return value **EBADF** and -1.

#### Performance



- I/O multiplexing calls select() and poll() have similar performance on most occasions.
- If the range of monitored file descriptors is small
- or if large number of file descriptors are densely packed (ie. the most or all file descriptors from 0 to some limit are being monitored)

#### Performance



- Then the performance is similar.
- If a small number of file descriptors are monitored on a large range then poll() is superior.

#### **Problems**



- Although these are widely used, portable and common place methods for monitoring multiple file descriptors, some problems remain.
- The CPU time required for select() and poll() increases with the number of file descriptors being monitored linearly.

#### **Problems**



- Usually programs make repeated calls to monitor the same set of file descriptors but the kernel doesn't remember the list.
- So scaling is very bad.



- Establish a signal handler for the signal to be delivered by the signal driven I/O mechanism. Default signal is SIGIO.
- Set the owner of the file descriptor using fcntl() system call ie.
  fcntl(fd, F\_SETOWN, pid);
- Enable the non blocking I/O on the file descriptor.



 Enable the signal-driven I/O by turning on the O\_ASYNC flag on the file status, ie:

```
flags = fcntl(fd,F_GETFL);
fcntl(fd,F_SETFL,flags | O_ASYNC | O_NONBLOCK);
```

The calling process can now go back and do other stuff. When I/O becomes possible the kernel generates a signal for the process and invokes the signal handler.





- Since it is edge-triggered notification we need to perform as much I/O as possible.
- Assuming non blocking I/O as we do, this means looping read() or write() around until EAGAIN or EWOULDBLOCK is returned.
- Let's consider the actual requirement for signal delivery: When is I/O possible?



- For terminals and pseudoterminals signal is generated whenever new input becomes available. Signal is also sent when end-of-file condition occurs
- No output possible signaling or terminal disconnect signaling is possible.



- For pipes and FIFOs read end is generated a signal when:
- Data is written to the pipe
- The write end is closed





- The write end is generated a signal when:
- A read from pipe increases the amount of free space so that it is possible to write PIPE\_BUF amount.
- The read end is closed.





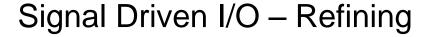
- For stream sockets signals are generated in the following circumstances:
- A new connection is received on listening socket.
- TCP **connect()** request completes.





- New input is received into the socket.
- Peer closes the socket.
- Output is possible on the socket.
- Async error occurs on the socket.





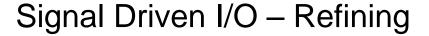


- In applications that need to simultaneously monitor very large numbers of file descriptors like certain network servers, signal-driven I/O provides significant performance gains against I/O multiplexing.
- To take full advantage of these we need to do two things.
- First we need to change the signal from SIGIO into something called real time signal.

# Signal Driven I/O – Refining



- Real time signals are the signals varying from SIGRTMIN to SIGRTMAX (32-64 in Linux).
- They can be queued when blocked unlike normal signals.
- First three real time signals are used by LinuxThreads.
- Programs should refer to these with SIGRTMIN+n for portability.





So employing fcntl() to change the signal.

```
int sig = SIGRTMIN+10;
fcntl(fd,F_SETSIG,sig);
```

You can also ask for a signal specified for a file descriptor.

```
sig = fcntl(fd,F_GETSIG);
```

4/23/2014

 You need to add #define \_GNU\_SOURCE before any includes to make F\_SETSIG work to your code.







- Specify SA\_SIGINFO flag and use the extended signal handler when using sigaction() to set the handler for the real time signal.
- The siginfo\_t struct passed on to the signal handler contains following:
- si\_signo: the number of the signal.
- si\_fd: the file descriptor in question.
- si\_code: code indicating the type of the event.
- si\_band: roughly equal to revents in poll()



## Signal Driven I/O – Refining



- Following si\_codes must match following si\_bands:
- POLL\_IN equals either POLLIN or POLLRDNORM
- POLL\_OUT equals either POLLOUT or POLLWRNORM
- POLL\_ERR equals POLLERR
- POLL\_PRI equals POLLPRI or POLLRDNORM
- POLL\_HUP equals POLLHUP or POLLERR





- First thing to note is that epoll API is strictly Linux and has been around only since 2.6 kernel series.
- The performance of epoll scales much better than **select()** or **poll()**.
- The epoll API can be used with either level-triggered or edgetriggered notification.
- The performance of epoll and signal-driven I/O is similar but epoll is much simpler to handle.





- Central to the epoll API is the epoll instance which is referred via an open file descriptor.
- This file descriptor is not used for I/O but to serve two purposes:
- Recording a list of file descriptors that this process has declared an interest in monitoring – the *interest* list.
- Maintaining a list of file descriptors that are ready for I/O the *ready* list.





- For each file descriptor monitored by epoll, we specify a bit mask indicating events that we are interested in.
- These match closely with those of poll().
- The membership in the ready list is a subset of membership in the interest list



- The epoll API consists of three system calls:
- The epoll\_create() system call creates an instance of epoll and returns a file descriptor for reference.
- The **epoll\_ctl()** system call manipulates the interest list associated with an epoll instance.
- The **epoll\_wait()** system call returns items from the ready list associated with an epoll instance.







The **epoll\_create()** system call has following prototype:

```
int epoll create(int size);
```

- The size argument specifies the number of file descriptors we except to monitor.
- This argument is not the upper limit but rather a hint to the kernel about how initially dimension internal data structures.
- The return value is the epoll file descriptor or -1 if there is an error.

## epoll – modifying the interest list



- The epoll\_ctl() system call modifies the interest list of the epoll instance.
- The prototype for epoll\_ctl() is:

```
int epoll ctl(int epfd, int op, int fd, struct epoll event *ev);
```

- The epfd argument is the epoll instance we are modifying.
- The fd argument is the actual file descriptor we are interested in. It
  might be on the interest list or we could be adding it there.



# epoll – modifying the interest list



- The op argument specifies the operation to be performed. There are three operations possible:
- EPOLL\_CTL\_ADD adds the file descriptor to the interest list. If we try add file descriptor which is already in the list, we get EEXIST error.
- **EPOLL\_CTL\_MOD** modifies the events setting for the file descriptor *fd* using the information pointed on by argument \*ev.
- EPOLL\_CTL\_DEL removes the specified file descriptor from the interest list.





 The \*ev argument is a pointer to a epoll\_event structure which is defined as follows:

```
struct epoll_event {
    uint32_t events;
    epoll_data_t data;
}
```

The events subfield is a bit mask specifying the set of events that we
are interested in. Those come when epoll\_wait() is discussed.







The data subfield is an union:

```
union epoll_data {
   void *ptr;
   int fd;
   uint32_t u32;
   uint64_t u64;
}
```

 One of these members can be used to specify information which is passed back to calling process via epoll\_wait(). So you could specify the file descriptor in question with fd subfield.





- The epoll\_wait() system call returns information about ready file descriptors from the epoll instance referred.
- The prototype for epoll\_wait() is:

```
int epoll_wait(int epfd, struct epoll_event *evlist, int maxevents, int
    timeout);
```

- The information about the ready file descriptors is returned in the \*evlist array.
- The \*evlist is allocated by the caller and contains maxevents number of elements.



- Each element in the array evlist returns information about a single ready file descriptor.
- The data subfield of the epoll\_event structure contains the only distinguishing information about the file descriptor.
- It contains the information specified by the user when **epoll\_ctl()** was called. Use this to differentiate between your file descriptors.
- The events argument contains the events for which the file descriptor is ready.





- The timeout argument determines the blocking behaviour of the epoll\_wait():
- If timeout is -1 block until an event occurs or signal is caught.
- If timeout is 0 do not block.
- If timeout is positive integer block up to timeout milliseconds, until an event occurs or a signal is caught.



- The events for both epoll\_ctl() and epoll\_wait() are:
- EPOLLIN (ctl/wait), data can be read.
- EPOLLPRI(ctl/wait), high priority data can be read.
- EPOLLRDHUP(ctl/wait), shutdown of peer socket
- EPOLLOUT(ctl/wait), data can be written
- EPOLLET(ctl) employ edge-triggered notification
- EPOLLONESHOT(ctl) disable after one use
- EPOLLERR/EPOLLHUP (wait) error or hangup has occurred.



## epoll – edge-triggered mode



- In order to use edge-triggered notification with epoll following rules must be met:
- Make all file descriptors monitored non blocking
- Build the interest list using epoll\_ctl(). Remember EPOLLET.
- Handle I/O events using loop:
- Retrieve ready list.
- For each file descriptor processs I/O until EAGAIN or EWOULDBLOCK.

