Syscalls: Adv. I/O 2: Multiplexed I/O

1. Overview
2. **Multiplexed I/O**
   - **poll() and select()**
   - **level-triggered vs. edge-triggered**
   - **epoll API**
   - **libevent and similar**
3. Additional Uses for I/O Muliplexing
4. Nonblocking I/O
5. Signal-Driven I/O
6. Memory-Mapped I/O
7. Zero-Copy I/O
8. Scatter-Gather I/O
9. Asynchronous I/O (AIO)

Multiplexed I/O

Multiplexed I/O (or **I/O multiplexing**), refers to techniques for handling I/O on large numbers of file descriptors, only some of which will be capable of I/O at any point.

The basic approach is to repeatedly loop, on each cycle checking which of a set of FDs are ready for I/O, and then handling some/all of those FDs on that cycle.

Having to monitor large numbers of FDs to see which might be ready for I/O is a common pattern in many programs, such as servers.

Standard **blocking mode I/O** cannot handle such situations without creating *separate processes or threads for each FD* (e.g., for each client).

Multiplexed I/O (contd.)

Nonblocking mode I/O is another approach for dealing with multiple FDs, some of which may not be ready for I/O.

However, using it (alone) will generally be very inefficient for handling large numbers of FDs, since programs would have to repeatedly try doing I/O on all the FDs.

In other words, programs would essentially have to **busy wait**, continuously **polling** all the FDs.

The special *I/O multiplexing syscalls* provide better efficiency, because the kernel identifies those FDs that are ready for I/O.

Multiplexed I/O (contd.)

The traditional UNIX/POSIX multiplexed I/O calls are `select()` and `poll()`.

Linux' *epoll API* is a newer interface for monitoring large numbers of FDs.

It can be vastly more efficient than `select()`/`poll()`, and is the API of choice for modern high-load servers such as NGINX.

The "epoll API" includes the syscalls `epoll_create()`, `epoll_ctl()`, and `epoll_wait()`.

While epoll is Linux-specific, other UNIX variants generally have similar calls these days (e.g., kqueue in the BSDs and OS X).

However, Solaris and AIX (and Windows) use a quite different mechanism, called **I/O completion ports**.

select()

`select()` is the oldest I/O multiplexing syscall:

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

- `nfds` should be the highest-numbered file descriptor in any of the three **FD sets** *plus 1*
- `readfds`/`writefds` are FD sets representing the FDs being monitored for readiness to `read()`/`write()`
- `exceptfds` is an FD set representing the FDs being monitored for *"exceptional conditions"*
- `timeout` specifies maximum time `select()` can block waiting for a ready FD, `NULL` allows indefinite blocking
- return indicates total number of "ready" FDs in the three FD sets, or 0 on timeout, or -1 on error (see `errno`)

select() (contd.)

- any of `readfds`, `writefds`, `exceptfds` may be `NULL` (if there are no relevant FDs)
- `readfds`/`writefds`:
  - **FD sets**, i.e., type `fd_set`
  - when `select()` is called, they contain the FDs to be monitored
  - on non-error return, they contain the set of "ready" FDs
  - being "ready" means `read()`/`write()` calls will not block
- `exceptfds`:
  - FD set, type `fd_set`
  - when `select()` is called, contains the FDs to be monitored
  - on non-error return, it contains the set of FDs on which "exceptional conditions" have occurred (see below)
- Note that the three FD sets are *modified in place*, so *must be reinitialized* before being reused on next iteration

select() (contd.)

See both "`man select`" and "`man select_tut`" for full information on `select()` (and `pselect()`).

The second man page says the following about *"exceptional conditions":*
"In practice, only one such exceptional condition is common: the availability of out-of-band (OOB) data for reading from a TCP socket. See recv(2), send(2), and tcp(7) for more details about OOB data. (One other less common case where select(2) indicates an exceptional condition occurs with pseudoterminals in packet mode; see tty_ioctl(4).)"

FD Sets

The type `fd_set` represents a **file descriptor set**.

There are four macros for manipulating FD sets:

- **FD_ZERO** – clears entire set:
  `void FD_ZERO(fd_set *set)`
- **FD_SET** – adds an FD to set:
  `void FD_SET(int fd, fd_set *set)`
- **FD_CLR** – removes an FD from set:
  `void FD_CLR(int fd, fd_set *set)`
- **FD_ISSET** – tests to see if an FD is part of a set (useful after select() returns):
  `int  FD_ISSET(int fd, fd_set *set)`

select() Example

```
//Setup required variables:
fd_set infds, readyfds;
FD_ZERO(&infds);
int maxfd = 0;

//Add FDs to be monitored for reading:
...
FD_SET(newclientfd, &infds);
maxfd = MAX(maxfd,newclientfd);
...

//Monitor FDs to be ready for reading, and process:
readyfds = infds;
while (select(maxfd+1, &readyfds, NULL, NULL, NULL) > 0) {
  for (int fd=0; fd<=maxfd; fd++)
    if (FD_ISSET(fd, &testfds))
      handle_readable_fd(fd);
  readyfds = infds;  //must reset for each iteration
}
```

pselect()

`pselect()` is a related call that helps deal with *signals*:

```
int pselect(int nfds, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, const struct timespec *ntimeout,
            const sigset_t *sigmask)
```

- `sigmask` is a pointer to a **signal mask** (see `sigprocmask(2)`)

`pselect()` first replaces the current signal mask by the one pointed to by `sigmask`, then does `select()`, then restores the original signal mask.

From the man page (see for more details):
"The reason that pselect() is needed is that if one wants to wait for either a signal or for a file descriptor to become ready, then an atomic test is needed to prevent race conditions."

pselect() (contd.)

Other differences between `select()` and `pselect()`:

- `select()`'s `timeout` is a `struct timeval` that can specify seconds and *microseconds*
- `pselect()`'s `ntimeout` is a `struct timespec` that can specify seconds and *nanoseconds*
- `select()` may update `timeout` to indicate how much time was left for blocking to continue
- `pselect()` does not change `ntimeout`

poll()

`poll()` performs similar functionality to `select()`:

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout)
```

- `fds` *array* of `struct pollfd` units representing the FDs to be monitored
- `nfds` is the number of units in `fds`
- `timeout` is maximum time can block:
  - positive: microseconds can block
  - 0: immediate return (even if no FDs ready)
  - negative: indefinite blocking
- return indicates total number of "ready" FDs, or 0 on timeout, or -1 on error (see `errno`)

poll() (contd.)

FDs being monitored are represented as `struct pollfd` units:

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

- `fd` contains a file descriptor to monitor (ignored if negative, allowing easy way to ignore an FD for a cycle)
- `events` is bit mask specifying the event(s) the FD is to be monitored for
- `revents` is bit mask representing event(s) that occurred

poll() (contd.)

The bits in `events` include:

- `POLLIN` – data to read
- `POLLPRI` – urgent data to read
- `POLLOUT` – writing will not block
- `POLLRDHUP` – stream socket peer closed connection, or shut down writing half of connection

`revents` can include the above bits *plus*:

- `POLLERR` – error condition (output only)
- `POLLHUP` – hang up (output only)
- `POLLNVAL` – invalid request (e.g., `fd` not open), (output only)

poll() Example

```
//Setup required variables:
struct pollfd poll_fds[MAX_FDS];
int numfds = 0;

//Add FDs to be monitored for reading:
...
poll_fds[numfds].fd = newclientfd;
poll_fds[numfds].events = POLLIN;
numfds++;
...

//Monitor FDs to be ready for reading, and process:
while (poll(poll_fds, numfds, -1) > 0) {
  for (int entry=0; entry<numfds; entry++) {
    if (poll_fds[entry].revents & POLLIN)
      handle_readable_fd(poll_fds[entry].fd);
    else if (poll_fds[entry].revents & (POLLHUP | POLLERR | POLLNVAL))
      close_fd(poll_fds[entry].fd); }
}
```

ppoll()

ppoll() is to poll() as pselect() is to select():

```
int ppoll(struct pollfd *fds, nfds_t nfds,
          const struct timespec *timeout_ts,
          const sigset_t *sigmask)
```

Multiplexed I/O and Nonblocking I/O

When the multiplexed I/O calls report an FD as *"ready"*, this is supposed to mean that the corresponding `read()`/`write()` on the FD will *not block*.

However, the man page for `select()` has the following:
"Under Linux, select() may report a socket file descriptor as ready for reading, while nevertheless a subsequent read blocks. This could for example happen when data has arrived but upon examination has wrong checksum and is discarded. There may be other circumstances in which a file descriptor is spuriously reported as ready. Thus it may be safer to use O_NONBLOCK on sockets that should not block."

The man page for `poll()` contains this:
"See the discussion of spurious readiness notifications under the BUGS section of select(2)."

epoll API

The `select()` and `poll()` calls have been available for a long time, and are standardized.

However, because of their design, they suffer from some inherent inefficiencies that prevent them from being appropriate for modern high-load servers.

Benchmarks by Michael Kerrisk, found that with 1000 monitored FDs, the *CPU time* (in seconds) required to monitor 100k events was as follows:

`select()`	`poll()`	epoll
930	990	0.66

In order to be able to build high-load servers, UNIX/POSIX systems have introduced new multiplexed I/O syscalls that are designed specifically to make it possible to build faster servers.

The **epoll API** refers to the set of such Linux calls.

epoll API (contd.)

The epoll API refers to a set of five system calls:

- **epoll_create()** – create an epoll unit and return its *file descriptor handle*
- **epoll_create1()** – like `epoll_create()` but accepts some *flags*
- **epoll_ctl()** – register a FD to be monitored or set its monitoring characteristics
- **epoll_wait()** – block, waiting for monitored I/O event(s)
- **epoll_pwait()** – like `epoll_wait()`, but deals with *signals* as with `pselect()` and `ppoll()`

epoll_create()

`epoll_create()` creates a new epoll unit:

```
int epoll_create(int size)
```

- `size` is *ignored*, but must be greater than zero for backwards compatibility
- return is *file descriptor handle* for the epoll unit

`epoll_create1()` accepts *flags*:

```
int epoll_create(int flags)
```

The currently valid `flags` value is `EPOLL_CLOEXEC`, which sets the *close-on-exec* (`FD_CLOEXEC`) flag on the epoll unit FD.

epoll_ctl()

`epoll_ctl()` registers an FD to be monitored or set its monitoring characteristics:

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)
```

- `epfd` is the epoll unit file descriptor *handle*
- `fd` is the target file descriptor (to be monitored)
- `op` is the operation to perform on `fd`:
  - `EPOLL_CTL_ADD` – register `fd`
  - `EPOLL_CTL_DEL` – deregister `fd`
  - `EPOLL_CTL_MOD` – modify `fd`'s registration
- `event` – describes `fd`'s monitoring characteristics
- returns 0 if successful, else -1 on error (see `errno`)

epoll_ctl() (contd.)

`struct epoll_event` is defined as:

```
struct epoll_event {
  uint32_t     events;  /* Epoll events */
  epoll_data_t data;    /* User data variable */
}

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

The `events` member the `epoll_event` struct is a *bit set/vector* that defines the types of events being monitored for `fd`.

epoll_ctl() (contd.)

`events` member the `epoll_event` struct is composed from the following *event types*:

- **EPOLLIN** – available for `read()` operations
- **EPOLLOUT** – available for `write()` operations
- **EPOLLRDHUP** – stream socket peer closed connection, or shut down writing half of connection
- **EPOLLPRI** – *urgent* data available for `read()` operations
- **EPOLLERR** – error occurred
- **EPOLLHUP** – hang up (HUP) occurred
- **EPOLLET** – sets **edge triggered** behavior
- **EPOLLONESHOT** – sets **one-shot** behavior

epoll_ctl() (contd.)

Notes:

- `epoll_wait()` will always wait for error and HUP events, so not necessary to explicitly set `EPOLLERR` and `EPOLLHUP` in `events`
- default behavior is **level triggered**
- **one-shot** behavior means that after `epoll_wait()` returns an FD, monitoring of that FD is automatically disable; `epoll_ctl()` with `EPOLL_CTL_MOD` must be called to *rearm* the FD (with a new event mask)

epoll_wait()

`epoll_wait()` waits for monitored event(s) to occur:

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

- `epfd` is the epoll unit FD handle
- `events` array of `struct epoll_event` units representing the "ready" FDs/events
- up to `maxevents` are returned (must be ≥ 0)
- `timeout` is maximum time can block:
  - positive: microseconds can block
  - 0: immediate return (even if no FDs ready)
  - negative: indefinite blocking
- return indicates total number of FDs in `events`, or 0 on timeout, or -1 on error (see `errno`)

epoll Example

```
//Create epoll unit
int epoll_fd;
epoll_fd = epoll_create(1));

//Add FD to be monitored for read()'ing:
struct epoll_event ev;
ev.events = EPOLLIN;
ev.data.fd = client_fd;
epoll_ctl(epoll_fd,EPOLL_CTL_ADD,client_fd,&ev);

...add additional FDs to be monitored...
```

epoll Example (contd.)

```
//Loop, epolling the FDs and handling those that are ready for I/O:
int ready;
struct epoll_event evlist[MAXEVENTS];

while ((ready = epoll_wait(epoll_fd,evlist,2,-1)) > 0) {
  //Go through the ready FDs and transfer data:
  for (int i=0; i<ready; i++) {
    int readyfd = evlist[i].data.fd;
    //Check for errors on FD before EPOLLIN:
    if (evlist[i].events & (EPOLLERR | EPOLLHUP | EPOLLRDHUP)) {
      ...handle error on readyfd...
    }
    //Check if FD is ready to read():
    else if (evlist[i].events & EPOLLIN) {
      ...process readyfd using read(readyfd,...)...
    }
  } //end for
} //end while
```

libevent and related