Syscalls: Adv. I/O 1: Overview

Basic vs. Advanced I/O

By *"basic I/O"* we mean the standard I/O techniques shown in the **Syscalls: Files** lectures:

- files/sockets use defaults:
  - `open()` called with *no* `flags` options (e.g., `O_NONBLOCK`)
  - `fcntl()` *not* used to change file descriptor **status flags**
  - I/O operations are default **blocking mode**
- I/O done via basic `read()`/`write()` syscalls
- selection among multiple file descriptors handled by program code, not syscalls
- file **caching** handled by kernel, using default behavior

Basic vs. Advanced I/O

By *"advanced I/O"* we mean use of non-default settings and/or a variety of special I/O related syscalls:

- **multiplexed I/O** (`poll()` and `select()`)
- `epoll` API
- **nonblocking mode I/O** (`O_NONBLOCK`)
- kernel file buffer advice (for regular files only)
- **signal-driven I/O** (`O_ASYNC`)
- signals as file descriptors
- **memory mapping** files
- **zero-copy I/O**
- **scatter/gather I/O**
- **AIO** (**asynchronous I/O**): POSIX, kernel, io_uring

Basic vs. Advanced I/O (contd.)

In general, these "advanced I/O" techniques have one or more of the following goals:

- simplifying code required to handle multiple I/O connections
- simplifying code required to handle asynchronous and unpredictable I/O connections
- efficiently handling large numbers of I/O connections
- safely handling malicious/errorful I/O connections
- reducing data copying for syscalls

By *"I/O connections"* we mean open regular files, pipes/FIFOs, sockets, and devices.

Basic vs. Advanced I/O (contd.)

Due to the often implementation-specific nature of efficiency, many of the relevant system calls are *Linux specific*.

Other UNIX OS's often have comparable syscalls, but not always (and not necessarily same name).

Advanced I/O Syscalls

System calls relevant to advanced I/O:

- multiplexed I/O:
  - `int poll(struct pollfd *fds, nfds_t nfds, int timeout)`
  - `int ppoll(struct pollfd *fds, nfds_t nfds, const struct timespec *timeout_ts, const sigset_t *sigmask)`
  - `int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)`
  - `int pselect(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, const struct timespec *timeout, const sigset_t *sigmask)`
- `epoll` API:
  - `int epoll_create(int size)`
  - `int epoll_create1(int flags)`
  - `int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)`
  - `int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout)`
  - `int epoll_pwait(int epfd, struct epoll_event *events, int maxevents, int timeout, const sigset_t *sigmask)`

Advanced I/O Syscalls (contd.)

System calls relevant to advanced I/O: (contd.)

- nonblocking I/O:
  - `int open(const char *pathname, int flags)`:
    as `open(pathname, flags|O_NONBLOCK)`
  - `int fcntl(int fd, int cmd, ... /* arg */ )`:
    as `fcntl(fd, F_SETFL, flags|O_NONBLOCK)`
- kernel file buffer advice:
  - `int posix_fadvise(int fd, off_t offset, off_t len, int advice)`
  - `ssize_t readahead(int fd, off64_t offset, size_t count)`

Advanced I/O Syscalls (contd.)

System calls relevant to advanced I/O: (contd.)

- signal-drven I/O:
  - `int open(const char *pathname, int flags)`:
    as `open(pathname, flags|O_ASYNC)`
  - `int fcntl(int fd, int cmd, ... /* arg */ )`:
    as `fcntl(fd, F_SETFL, flags|O_ASYNC)`
- signals as FDs:
  - `int signalfd(int fd, const sigset_t *mask, int flags)`
  - `int timerfd_create(int clockid, int flags)`
  - `int timerfd_settime(int fd, int flags, const struct itimerspec *new_value, struct itimerspec *old_value)`
  - `int timerfd_gettime(int fd, struct itimerspec *curr_value)`
  - `int eventfd(unsigned int initval, int flags)`

Advanced I/O Syscalls (contd.)

System calls relevant to advanced I/O: (contd.)

- memory mapping files:
  - `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)`
  - `int munmap(void *addr, size_t length)`
  - `int madvise(void *addr, size_t length, int advice)`
  - `int posix_madvise(void *addr, size_t len, int advice)`
- zero-copy I/O:
  - `ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count)`
  - `ssize_t splice(int fd_in, loff_t *off_in, int fd_out, loff_t *off_out, size_t len, unsigned int flags)`
  - `ssize_t tee(int fd_in, int fd_out, size_t len, unsigned int flags)`
  - `ssize_t vmsplice(int fd, const struct iovec *iov, unsigned long nr_segs, unsigned int flags)`
- scatter-gather I/O:
  - `ssize_t readv(int fd, const struct iovec *iov, int iovcnt)`
  - `ssize_t writev(int fd, const struct iovec *iov, int iovcnt)`
  - `ssize_t preadv(int fd, const struct iovec *iov, int iovcnt, off_t offset)`
  - `ssize_t pwritev(int fd, const struct iovec *iov, int iovcnt, off_t offset)`

Advanced I/O Syscalls (contd.)

System calls relevant to advanced I/O: (contd.)

- POSIX AIO:
  - `int aio_read(struct aiocb *aiocbp)`
  - `int aio_write(struct aiocb *aiocbp)`
  - `int lio_listio(int mode, struct aiocb *const aiocb_list[], int nitems, struct sigevent *sevp)`
  - `...aio_cancel(3), aio_error(3), aio_fsync(3), aio_return(3), aio_suspend(3)`
- Linux kernel AIO:
  - `int io_submit(aio_context_t ctx_id, long nr, struct iocb **iocbpp)`
  - `...io_cancel(2), io_destroy(2), io_getevents(2), io_setup(2)`
- io_uring AIO (2019):
  - `int io_uring_setup(int entries, struct io_uring_params *params)`
  - `int io_uring_enter(unsigned int fd, u32 to_submit, u32 min_complete, u32 flags)`
  - `int io_uring_register(unsigned int fd, unsigned int opcode, void *arg)`

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

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

Syscalls: Adv. I/O 3: Additional Multiplexing

1. Overview
2. Multiplexed I/O
3. **Additional Uses for I/O Muliplexing**
   - **signalfd's**
   - **timerfd's**
   - **eventfd's**
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)

FDs for signals, timers, events

The multiplexed I/O APIs `select()`, `poll()`, and epoll rely on *file descriptors* as the basis for monitoring for events.

This has meant that events based on *signals* have had to be handled differently (e.g., using `pselect()`/`ppoll()`/`epoll_pwait()` in conjunction with signal handlers).

However, syscalls have been added to Linux (and other UNIXes) that now allow *signal-based events* to be handled "normally" through multiplexed I/O calls, by creating various non-file FDs:

- SIGNALFDs
- TIMERFDs
- EVENTFDs

SIGNALFDs

A **signalfd**:
- is a monitoriable FD that is associated with a set of signals
- when one of these signals becomes pending, the FD is "ready"
- `read()`'ing from the FD *accepts* one pending signal

The syscall to create a signalfd is:
`int signalfd(int fd, const sigset_t *mask, int flags)`
- `mask` specifies the set of signals to accept via the signalfd FD
- `mask` is a **signal set** (initialized using the `sigsetops` macros)
- signals to be received should be **blocked** using `sigprocmask()` (to prevent handling according to *default dispositions*)
- not possible to receive SIGKILL or SIGSTOP via a signalfd
- `fd` is either existing valid signalfd FD, or -1 to create new FD
- return is signalfd FD

SIGNALFDs (contd.)

`read()`'ing from a signalfd returns a `signalfd_siginfo` structure:

```
struct signalfd_siginfo {
  uint32_t ssi_signo;   /* Signal number */
  int32_t  ssi_errno;   /* Error number (unused) */
  int32_t  ssi_code;    /* Signal code for why/how sent */
  uint32_t ssi_pid;     /* PID of sender */
  uint32_t ssi_uid;     /* Real UID of sender */
  int32_t  ssi_fd;      /* File descriptor (SIGIO) */
  uint32_t ssi_tid;     /* Kernel timer ID (POSIX timers)
  uint32_t ssi_band;    /* Band event (SIGIO) */
  uint32_t ssi_overrun; /* POSIX timer overrun count */
  uint32_t ssi_trapno;  /* Trap number that caused signal */
  int32_t  ssi_status;  /* Exit status or signal (SIGCHLD) */
  int32_t  ssi_int;     /* Integer sent by sigqueue(3) */
  uint64_t ssi_ptr;     /* Pointer sent by sigqueue(3) */
  uint64_t ssi_utime;   /* User CPU time consumed (SIGCHLD) */
  uint64_t ssi_stime;   /* System CPU time consumed (SIGCHLD) */
  uint64_t ssi_addr;    /* Address that generated signal
                           (for hardware-generated signals) */
  uint8_t  pad[X];      /* Pad size to 128 bytes (allow for
                           additional fields in the future) */
};
```

SIGNALFDs (contd.)

Fields in this structure are analogous to those in a `siginfo_t` structure used with signal handlers—see "`man sigaction`".

Not all fields will be valid for a specific signal; the valid fields can be determined from the `ssi_signo` and `ssi_code` fields:

- `si_signo` and `si_code` are defined for all signals
- (`si_errno` is generally unused on Linux)
- rest of struct may be a *union*, so read only meaningful fields
- see "`man sigaction`" and `siginfo_t` discussion for details

Common `ssi_code` values:

- `SI_USER` – from `kill()`
- `SI_QUEUE` – from `sigqueue()`
- `SI_KERNEL` – from kernel
- `SI_TIME` – POSIX timer expired

SIGNALFDs (contd.)

Details of using signalfd's:

- `read()` is used to *accept* one or more signals in `mask` that are currently *pending*
- each accepted signal results in a `signalfd_siginfo` struct being placed in `read()`'s buffer
- as many pending signals as will fit in `read()`'s buffer will be accepted/returned at one time
- `read()`'s buffer must be at least `sizeof(struct signalfd_siginfo)`
- `read()`'s return value will be the total number of bytes read
- `read()`'ing a *pending* signal consumes it so no longer pending (cannot be caught by *handler* or accepted using `sigwaitinfo()`)
- if no signals in `mask` are pending, `read()` *blocks* until a signal in `mask` is generated for the process, or fails with the error `EAGAIN` if the signalfd FD has been made *nonblocking*

SIGNALFD Example

Setting up a signalfd to accept `SIGINT` using epoll:

```
//Create and initialize signalfd unit:
sigset_t mask;
sigemptyset(&mask);
sigaddset(&mask,SIGINT);
sigprocmask(SIG_BLOCK,&mask,NULL);
int sigfd = signalfd(-1,&mask,0);

//Add signalfd to epoll unit for monitoring:
struct epoll_event ev;
ev.events = EPOLLIN;
ev.data.fd = sigfd;
epoll_ctl(epoll_fd,EPOLL_CTL_ADD,sigfd,&ev);
```

SIGNALFD Example

Monitoring signalfd's via epoll:

```
//Make sure all signalfd monitored signals are blocked:
sigprocmask(SIG_BLOCK,&mask,NULL);

//epoll loop for check for pending signals(s):
int ready;
struct epoll_event evlist[NUM_SIGNALS];
while ((ready = epoll_wait(epoll_fd,evlist,MAX_TIMERS,-1)) > 0) {
  for (int i=0; i<ready; i++) {
    int sigfd = evlist[i].data.fd;
    struct signalfd_siginfo siginfo;
    ssize_t sread = read(sigfd,&siginfo,sizeof(struct signalfd_siginfo));
    if (s != sizeof(struct signalfd_siginfo)) ...error...
    int sig = siginfo.ssi_signo;
    int code = siginfo.ssi_code;
    switch (sig) {
      ...
    }
} }
```

TIMERFDs

A **timerfd**:

- a timer that has an associated monitorable FD
- when the timer *expires*, the timerfd becomes "ready"
- thus, *expiration notifications* are delivered via the FD

The timerfd API is:

- `int timerfd_create(int clockid, int flags)`

-
  ```
  int timerfd_settime(int fd, int flags,
                      const struct itimerspec *new_value,
                      struct itimerspec *old_value)
  ```
- `int timerfd_gettime(int fd, struct itimerspec *curr_value)`

TIMERFD Example

Setting up a timerfd:

```
//Create and initialize timerfd unit:
int timerfd = timerfd_create(CLOCK_REALTIME);
struct itimerspec timer;
timer.it_value.tv_sec = 5;
timer.it_value.tv_nsec = 0;
timerfd_settime(timerfd,0,&timer,NULL);

//Add timerfd to epoll unit for monitoring:
struct epoll_event ev;
ev.events = EPOLLIN;
ev.data.fd = timerfd;
epoll_ctl(epoll_fd,EPOLL_CTL_ADD,timerfd,&ev);
```

TIMERFD Example (contd.)

Monitoring timerfd's via epoll:

```
//epoll loop for check for expired timer(s):
int ready;
struct epoll_event evlist[MAX_TIMERS];
while ((ready = epoll_wait(epoll_fd,evlist,MAX_TIMERS,-1)) > 0) {
  for (int i=0; i<ready; i++) {
    int timerfd = evlist[i].data.fd;
    ...do what is required given that timerfd's timer expired...
    ...may need to use FD to determine what timer expired...
    close(timerfd);  //deletes timerfd unit
} }
```

EVENTFDs

EVENTFDs – a monitorable FD can be used as "an event wait/notify mechanism by user-space applications, and by the kernel to notify user-space applications of events"

Syscalls: Adv. I/O 4: Nonblocking I/O

Blocking vs. Nonblocking I/O

By default, I/O occurs in **blocking mode**: a `read()` or `write()` call will **block** (i.e., cause the process to be *suspended*) until some data can be read or written.

In **nonblocking mode**, a `read()` or `write()` call *always returns immediately*:

- if it can immediately read/write data, then it does (just as in blocking mode)
- if it *cannot* immediately read/write data (so call would *block*), -1 is returned (error), and `ERRNO` is set to `EAGAIN` or `EWOULDBLOCK`

Nonblocking mode is useful when I/O can *block indefinitely.*

This can occur with pipes/FIFOs, sockets, terminals, and devices.

Nonblocking Mode and Regular Files

Note: *nonblocking mode is not relevant to I/O on* **regular files** (it has no effect).

Regular files are *automatically* **cached** (**buffered**) by the kernel when involved in I/O.

The purpose of file caching is to speed up reading/writing of files to secondary storage.

`write()` returns immediately because the data simply gets written to the **buffer cache** (**page cache**).

`read()` may have to wait on a disk read if the requested data is not in the buffer cache, but it will block only briefly.

(See “Kernel File Buffer Advice” below for how to address blocking in regular file `read()`’s.)

Setting Nonblocking Mode

There are two ways to configure an I/O connection (file descriptor) to be in **nonblocking mode**:

1. `open()` the file with the `flags` argument including `O_NONBLOCK`:
   `open(file, O_...|O_NONBLOCK)`
2. use `fcntl()` with `F_SETFL` to set the *file descriptor status flags* to include `O_NONBLOCK`:
   `fcntl(fd, F_SETFL, O_NONBLOCK)`

When successful, subsequent operations on the FD *cannot* cause the calling process to block.

Nonblocking I/O Errors Codes

When an FD is in nonblocking mode and a `read()`/`write()` is unable to immediately read/write data, -1 (error) is returned.

The calling program can determine that the error was due to inability to immediately read/write by checking `ERRNO`.

The following error codes indicate that the calls would have blocked if they were not in nonblocking mode:

- `EAGAIN` – when the FD refers to a *file other than a socket*
- `EAGAIN` or `EWOULDBLOCK` – when the FD refers to a *socket* (man pages say portable applications must check for both codes)

Using Nonblocking I/O

In nonblocking mode, I/O calls can immediately return—without performing any I/O.

It is up to the calling program to check if the I/O operation succeeded or not.

If it did not, it must arrange for the operation to be tried again.

One approach is for nonblocking I/O calls to be wrapped in a *loop* that keeps trying the operation until it is successful:

```
while ((nread = read(fd,buff,cnt)) == -1 &&
        (errno == EAGAIN || errno == EWOULDBLOCK));
if (nread == -1) ...handle true read error...
...process buff...
```

Using Nonblocking I/O (contd.)

Of course this is a **busy wait**, so potentially extremely inefficient!

Instead, one should generally use a **multiplexed I/O** mechanism (e.g., `poll()`) to keep checking if I/O is possible on the FD.

The multiplexed I/O syscalls allow a program to *efficiently* wait to call `read()`/`write()` again when the operation can succeed.

If one needs to use multiplexed I/O calls like `poll()`, though, then what is the value of nonblocking mode?

That is, if these syscalls can efficiently wait until `read()`/`write()` will succeed (not block), then there seems to be no reason to need to use nonblocking mode.

Using Nonblocking I/O (contd.)

It turns out that nonblocking mode is still valuable/necessary, because multiplexed I/O calls like `poll()` do not provide a 100% guarantee that an I/O operation won't block.

A key reason for this is because of the fact that a call like `poll()` checks at one instant, but a `read()`/`write()` happens later—when system state may have changed.

This means that code like the following could potentially block on the `read()`:

```
while (poll(fds,n,-1) > 0) {
  ...
  if (fds[i].revents & POLLIN) {
    if ((nread = read(fd,buff,cnt) == -1)
      ...handle read error...
    else
      ...process buff...
  }
  ...
}
```

Using Nonblocking I/O (contd.)

To be safe, when using multiplexed I/O calls, FDs should be set to *nonblocking mode*, and code such the following used:

```
while (poll(fds,n,-1) > 0) {
  ...
  if (fds[i].revents & POLLIN) {
    if ((nread = read(fd,buff,cnt)) == -1) {
      if (errno != EAGAIN && errno != EWOULDBLOCK)
        ...handle true read error... }
    else
      ...process buff...
  }
  ...
}
```

If `read()` fails with `EAGAIN`/`EWOULDBLOCK`, nothing is done with the FD on this iteration, and `poll()` simply loops again.

Nonblocking Mode and open()

Calling `open()` with `O_NONBLOCK` can alter `open()`'s behavior.

If `open()` would block due to an incompatible **file lock** or **file lease**, including `O_NONBLOCK` causes `open()` to immediately return, with error code `EWOULDBLOCK` (see "`man fcntl`").

`open()`'ing a FIFO in nonblocking mode differs from blocking mode as follows (see "`man 7 fifo`"):

- opening FIFO *read only* will succeed even if the write end has not yet been opened
- opening FIFO *write only* will fail with `ENXIO` unless the read end has already been opened

Sockets and Nonblocking Mode

Socket FDs are often set to nonblocking mode.

This is done using `fcntl()` as shown above.

Making a socket FD nonblocking has the following effects:

- `accept()` calls that would block (no pending connections), instead immediately return -1 with error `EAGAIN` or `EWOULDBLOCK`
- all input syscalls that would block (due to empty socket buffer), will return -1 with error `EAGAIN` or `EWOULDBLOCK`
- all output syscalls that would block (due to full socket buffer), will return -1 with error `EAGAIN` or `EWOULDBLOCK`
- `connect()` that cannot immediately complete connection will return -1 with error `EINPROGRESS`

Kernel File Buffer Advice

As noted earlier, I/O to *regular files* is causes the file contents to be *automatically cached/buffered* by the Linux kernel.

This is done to improve program performance and to increase disk throughput.

Linux' file buffer memory used to be called the **disk cache** but has now been merged into the general Linux **page cache**.

Nonetheless, it is common to see this memory referred to as **disk cache** or **disk buffers**.

While file caching generally works automatically, there are system calls that allow programs some control.

Kernel File Buffer Advice (contd.)

`readahead` is the simpler, Linux-specific call:
`ssize_t readahead(int fd, off64_t offset, size_t count)`

- "populates the page cache with data from a file so that subsequent reads from that file will not block on disk I/O"
- `offset` is the starting point from which data is read
- `count` specifies the number of bytes to be read
- whole pages are read, so `offset` is "rounded down" to a page boundary and bytes are read up to the next page boundary greater than or equal to `offset`+`count`

`readahead()` *blocks* until the specified data has been read, so it must be called in a separate thread/process to avoid blocking the program.

Kernel File Buffer Advice (contd.)

`posix_fadvise` is the POSIX standard call, which informs the kernel of an expected file access pattern, potentially allowing the kernel to optimize file access:
`int posix_fadvise(int fd, off_t offset, off_t len, int advice)`

- `offset` is the start of the relevant file region
- `len` is the number of bytes in the region
  (0 `len` means to end of file)
- `advice` is the "access pattern" in the region

Kernel File Buffer Advice (contd.)

Values for `advice` include:

- `POSIX_FADV_NORMAL` – no advice to give (the default value)
- `POSIX_FADV_SEQUENTIAL` - region will be accessed sequentially (from lower to higher offsets)
- `POSIX_FADV_RANDOM` – region will be accessed in random order
- `POSIX_FADV_NOREUSE` – region will be accessed only once
- `POSIX_FADV_WILLNEED` – region will be accessed soon
- `POSIX_FADV_DONTNEED` – region will not be accessed soon

Syscalls: Adv. I/O 5: Signal-Driven I/O

Signal-Driven I/O

Signal-driven I/O refers to a process configuring things so that the kernel sends it a **signal** when I/O is possible on a file descriptor.

Signal-driven I/O is enabled by doing the following steps:

1. Use `sigaction()` to register a **signal handler** for the signal that will be delivered (by default this is `SIGIO`)
2. Set the PID of the process that is to receive I/O signals (known as the FD "owner"):
   ```
   fcntl(fd, F_SETOWN, pid)
   ```
3. Enable *signal-driven I/O* for the FD and set the FD into **nonblocking mode**:
   ```
   fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_ASYNC | O_NONBLOCK)
   ```

Signal-Driven I/O (contd.)

Once the above steps are complete, the calling process can continue with other tasks.

When I/O is possible, the kernel will generate a signal for the process, leading to the signal handler being run.

(Remember that *signals* are effectively *software interrupts*, and are inherently **asynchronous**.)

The signal handler must ensure that the appropriate I/O gets carried out.

Edge-Triggered Notification

Signal-driven I/O provides **edge-triggered notification**.

This means that a signal is sent *only once*, when the FD changes state from I/O not possible to I/O possible.

Edge-triggered notification requires that the process perform *as much I/O as can be done at that time* whenever the handler is invoked.

This is why the FD was put into *nonblocking mode*.

Doing as much I/O as possible means looping, making I/O syscalls, until a call fails with the error `EAGAIN` (or `EWOULDBLOCK`).

(This is similar to what must be done when using `epoll()` in edge-triggered mode.)

Signal-Driven vs. Multiplexed I/O

Why might one be interested in *signal-driven I/O* when we have *multiplexed I/O?*

For **multiplexed I/O** (`poll()`, etc.) to provide **event-driven** behavior, there must be a *dedicated thread of execution* looping and checking for ready FDs.

Signal-driven I/O does not require a dedicated thread of execution monitoring for ready FDs.

Instead, the running thread will be *interrupted by a signal* whenever I/O is possible.

Syscalls: Adv. I/O 6: Memory-Mapped I/O

mmap()

`mmap()` creates a new **memory mapping** in the *virtual address space* of the calling process:

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset)

int munmap(void *addr, size_t length)
```

Syscalls: Adv. I/O 7: Zero-Copy I/O

Kernel Space ↔ User Space Copying

(Reminder: many objects in Linux/UNIX can be considered as *"files"* in the sense that they are accessed via *file descriptors*, including regular files, pipes, sockets, etc.)

The standard method for moving/copying data from one "file" to another is with a `read()+write()` loop such as the following:

```
ssize_t nread;
while((nread = read(fd_in,buff,buffsize)) > 0)
  write(fd_out,buff,nread)  //ignoring error checking write()
```

The "problem" with this approach is that each `read()`/`write()` syscall requires that the data being transferred is *copied twice:*

- in "file" $\xrightarrow{copy}$ *kernel-space* buffer $\xrightarrow{copy}$ `buff` (*user space*)
- `buff` (*user space*) $\xrightarrow{copy}$ *kernel-space* buffer $\xrightarrow{copy}$ out "file"

Kernel Space ↔ User Space Copying (contd.)

This means that the `read()+write()` loop requires every byte of data being transferred is copied *four times!*

Of course two of those copies (in and out from/to the files/devices) are unavoidable.

However, the kernel-space buffer ↔ user-space buffer copy operations are not necessarily required.

(Notice also, that the `read()+write()` loop requires *four context switches* on each loop iteration.)

Zero-Copy I/O

Zero-copy operations are those where the CPU is not required to move/copy data between memory locations.

Zero-copy I/O in particular refers to I/O operations that occur solely within kernel space.

I.e., no copying of data from/to user space is needed.

Linux includes several syscalls that can transfer data between file descriptors, but are more efficient, because they do the copying *solely within kernel space*.

I.e., they implement zero-copy I/O.

This avoids the two-copies problem of `read()+write()`.

Zero-Copy I/O (contd.)

The UNIX (semi)standard zero-copy I/O syscall is `sendfile()`.

Linux also includes the Linux-specific syscalls: `splice()`, `tee()`, `vmsplice()`.

The existence of these calls might seem to indicate that one should never need to use a `read()+write()` loop.

Unfortunately, `sendfile()` has limitations on what types of files can be associated with its input FD (it no longer limits the file types of its output FD).

In particular, `sendfile()` cannot be used to transfer data *in from* a socket.

So, it is not a general purpose zero-copy I/O syscall.

sendfile()

`sendfile()` copies data from an `mmap()`'able file descriptor to another file descriptor:

```
ssize_t sendfile(int fd_out, int fd_in, off_t *offset, size_t count)
```

- `fd_out` is target FD, opened for writing
- `fd_in` is source FD, opened for reading
- `offset` determines reading position for `fd_in`:
  - if `NULL`, data read starting at the current `fd_in` offset and `fd_in` offset updated by the call
  - else, points to a variable containing offset from which to start reading data from `fd_in`, variable will be updated by call but `fd_in` offset will not
- `count` is number of bytes to copy between the file descriptors
- returns number of bytes written to `fd_out`, else -1 on error

sendfile() (contd.)

Note that because `fd_in` must support `mmap()`-like operations, *it cannot be a socket!*

In Linux kernels ≥2.6.33, `fd_out` can be of *any file type*.

(If `fd_out` is associated with a *regular file*, its *offset* automatically gets updated to reflect copied data.)

sendfile() (contd.)

`sendfile()` man page notes:

- The original Linux `sendfile()` was not designed to handle large file offsets, so `sendfile64()` was added; glibc's `sendfile()` wrapper function transparently deals with this.
- Applications may wish to fall back to `read()/write()` in the case where `sendfile()` fails with `EINVAL` or `ENOSYS`.
- If `fd_out` refers to a socket or pipe with zero-copy support, callers must ensure the transferred portions of the file referred to by `fd_in` remain unmodified until the reader on the other end of `fd_out` has consumed the transferred data
- If sending files to a TCP socket, but need to send some header data first, use the `TCP_CORK` option to minimize the number of packets sent.

Linux-Specific Zero-Copy I/O Syscalls

In 2006, additional syscalls were added to the Linux kernel to support more general zero-copy I/O.

Linus Torvalds gave clear explanations of the new calls on the LKML:

"The really high-level concept is that there is now a notion of a "random kernel buffer" that is exposed to user space.

In other words, splice() and tee() work on a kernel buffer that the user has control over, where "splice()" moves data to/from the buffer from/to an arbitrary file descriptor, while "tee()" copes the data in one buffer to another.

So in a very real (but abstract) sense, "splice()" is nothing but read()/write() to a kernel buffer, and "tee()" is a memcpy() from one kernel buffer to another."

[continued on next slide]

Linux-Specific Zero-Copy I/O Syscalls (contd.)

"Now, to get slightly less abstract, there's two important practical details:

– the "buffer" implementation is nothing but a regular old-fashioned UNIX pipe. This actually makes sense on so many levels, but mostly simply because that is exactly what a UNIX pipe has always been: it's a buffer in kernel space. That's what a pipe has always been. So the splice usage isn't conceptually anything new for pipes - it's just exposing that old buffer in a new way.

– the second part of the deal is that the buffer is actually implemented as a set of reference-counted pointers, which means that you can copy them around without actually physically copy memory. So while "tee()" from a conceptual standpoint is exactly the same as a "memcpy()" on the kernel buffer, from an implementation standpoint it really just copies the pointers and increments the refcounts....

– vmsplice() system call to basically do a "write to the buffer", but using the reference counting and VM traversal to actually fill the buffer. This means that the user needs to be careful not to re-use the user-space buffer it spliced into the kernel-space one (contrast this to "write()", which copies the actual data, and you can thus re-use the buffer immediately after a successful write), but that is often easy to do."

[continued on next slide]

Linux-Specific Zero-Copy I/O Syscalls (contd.)

"Anyway, when would you actually use a kernel buffer? Normally you'd use it it you want to copy things from one source into another, and you don't actually want to see the data you are copying, so using a kernel buffer allows you to possibly do it more efficiently, and you can avoid allocating user VM space for it (with all the overhead that implies: not just the memcpy() to/from user space, but also simply the book-keeping).

It should be noted that splice() is very much not the same as sendfile(). The buffer is really the big difference, both conceptually, and in how you actually end up using it.

A "sendfile()" call (which a lot of other OS's also implement) doesn't actually need a buffer at all, because it uses the file cache directly as the buffer it works on. So sendfile() is really easy to use, and really efficient, but fundamentally limited in what it can do.

In contrast, the whole point of splice() very much is that buffer. It means that in order to copy a file, you literally do it like you would have done it traditionally in user space:"

⇒ **standard I/O `read()+write()` loop code example** (not shown here!)

[continued on next slide]

Linux-Specific Zero-Copy I/O Syscalls (contd.)

"**except you'd not have a buffer in user space, and the "read()" and "write()" system calls would instead be "splice()" system calls to/from a pipe you set up as your kernel buffer. But the construct would all be indentical - the only thing that changes is really where that "buffer" exists.**

Now, the advantage of splice()/tee() is that you can do zero-copy movement of data, and unlike sendfile() you can do it on arbitrary data (and, as shown by "tee()", it's more than just sending the data to somebody else: you can duplicate the data and choose to forward it to two or more different users - for things like logging etc).

So while sendfile() can send files (surprise surprise), splice() really is a general "read/write in user space" and then some, so you can forward data from one socket to another, without ever copying it into user space.

Or, rather than just a boring socket->socket forwarding, you could, for example, forward data that comes from a MPEG-4 hardware encoder, and tee() it to duplicate the stream, and write one of the streams to disk, and the other one to a socket for a real-time broadcast. Again, all without actually physically copying it around in memory."

[continued on next slide]

Linux-Specific Zero-Copy I/O Syscalls (contd.)

"So splice() is strictly more powerful than sendfile(), even if it's a bit more complex to use (the explicit buffer management in the middle). That said, I think we're actually going to remove sendfile() from the kernel entirely, and just leave a compatibility system call that uses splice() internally to keep legacy users happy.

Splice really is that much more powerful a concept, that having sendfile() just doesn't make any sense except as some legacy compatibility layer around the more powerful splice()."

So, we can eliminate the kernel space ↔ user space copying that occurs with a standard `read()+write()` loop, by instead doing:

```
int pfds[2]; ssize_t pipesize, nread;
pipe(pfds);
pipesize = fcntl(pfds[0],F_GETPIPE_SZ);
while((nread = splice(fd_in,NULL,pfds[1],NULL,pipesize,0)) > 0)
  splice(pfds[0],NULL,fd_out,NULL,nread,0)
```

Linux-Specific Zero-Copy I/O Syscalls (contd.)

From `splice()` man page: The three system calls...provide user-space programs with full control over an arbitrary kernel buffer, implemented within the kernel using the same type of buffer that is used for a pipe...perform the following tasks:

- `splice()` – moves data from the buffer to a file descriptor, or vice versa, or from one buffer to another.
- `tee()` – duplicates data from one buffer to another.
- `vmsplice()` – copies data from userspace memory into a buffer.

Though we talk of copying, actual copies are generally avoided. The kernel does this by implementing a pipe buffer as a set of reference-counted pointers to pages of kernel memory. The kernel creates "copies" of pages in a buffer by creating new pointers (for the output buffer) referring to the pages, and increasing the reference counts for the pages: only pointers are copied, not the pages of the buffer.

splice()

`splice()` transfers data from one file descriptor to another, where one of the FD's must refer to a *pipe:*

```
ssize_t splice(int fd_in, loff_t *offset_in, int fd_out,
               loff_t *offset_out, size_t len, unsigned int flags)
```

- `fd_in` is source FD, opened for reading
- `offset_in` determines reading position for `fd_in`
- `fd_out` is target FD, opened for writing
- `offset_out` determines writing position for `fd_out`
- `len` is number of bytes to transfer between the file descriptors
- `flags` is an options bit mask
- returns number of bytes written to `fd_out`, else -1 on error; 0 indicates no data to read and no reason to block due to no writers for `fd_in` pipe

splice() (contd.)

The following semantics apply for `fd_in` and `offset_in`:

- if `fd_in` refers to a pipe, then `offset_in` must be `NULL`;
- if `fd_in` does not refer to a pipe and `offset_in` is `NULL`, then bytes are read from `fd_in` starting from the current file offset, and the current file offset is adjusted appropriately;
- if `fd_in` does not refer to a pipe and `offset_in` is not `NULL`, then `offset_in` must point to a buffer which specifies the starting offset from which bytes will be read from `fd_in` and the file offset of `fd_in` is not changed;

Analogous statements apply for `fd_out` and `offset_out`.

tee()

`tee()` *duplicates* data from one *pipe* to another:

```
ssize_t tee(int fd_in, int fd_out, size_t len, unsigned int flags)
```

- `fd_in` is source pipe (read) FD
- `fd_out` is target pipe (write) FD
- `len` is number of bytes to duplicate
- `flags` is an options bit mask
- returns number of bytes written to `fd_out`, else -1 on error; 0 indicates no data to read and no reason to block due to no writers for `fd_in` pipe

Duplicates here means that `tee()` does not consume the data from `fd_in`, so that same data can be copied by a subsequent `splice()` call.

vmsplice()

`vmsplice()` transfers (possibly multiple) ranges of user-space memory into a *pipe:*

```
ssize_t vmsplice(int fd, const struct iovec *iov,
                 unsigned long nr_segs, unsigned int flags)
```

- `fd` is pipe (write) FD
- `iov` is an array of descriptions of range(s) of memory to copy
- `nr_segs` is number of ranges of user memory described by `iov` to be transferred into a pipe FD
- `flags` is an options bit mask
- returns number of bytes transferred to `fd`, else -1 on error

vmsplice() (contd.)

`iov` is an array of `iovec` structures:

```
struct iovec {
  void  *iov_base;           /* Starting address */
  size_t iov_len;            /* Number of bytes */
};
```

This is the same structure that is used with the *scatter-gather* (*vectorized*) I/O syscals such as `readv()` and `writev()`.

It is used to specify potentially multiple range of user-space memory to transfer with one syscall.

Zero-Copy IPC

Linux has also added (2015) zero-copy syscalls to transfer data between the address space of the calling *process* and the address space of another process identified by its PID.

Data moves directly between the two processes, without passing through kernel space.

This permits fast message passing, allowing messages to be exchanged with a single copy operation rather than the double copy that would be required when using, for example, shared memory or pipes.

The two syscalls are: `process_vm_readv()` and `process_vm_writev()`.

Zero-Copy IPC (contd.)

```
ssize_t process_vm_readv(pid_t pid,
                         const struct iovec *local_iov,
                         unsigned long liovcnt,
                         const struct iovec *remote_iov,
                         unsigned long riovcnt,
                         unsigned long flags)


ssize_t process_vm_writev(pid_t pid,
                          const struct iovec *local_iov,
                          unsigned long liovcnt,
                          const struct iovec *remote_iov,
                          unsigned long riovcnt,
                          unsigned long flags)
```

copy_file_range()

Linux 4.5 (March 2016) added `copy_file_range()`:

"Copying a file consists in reading the data from a file to user space memory, then copy that memory to the destination file. There is nothing wrong with this way of doing things, but it requires doing extra copies of the data to/from the process memory. In this release Linux adds a system call, `copy_file_range(2)`, which allows to copy a range of data from one file to another, avoiding the mentioned cost of transferring data from the kernel to user space and then back into the kernel.

copy_file_range() (contd.)

This system call is only very slightly faster than `cp`, because the costs of these memory copies are barely noticeable compared with the time it takes to do the actual I/O, but there are some cases where it can help a lot more. In networking filesystems such as NFS, copying data involves sending the copied data from the server to the client through the network, then sending it again from the client to the new file in the server. But with `copy_file_range(2)`, the NFS client can tell the NFS server to make a file copy from the origin to the destination file, without transferring the data over the network (for NFS, this also requires the server-side copy feature present in the upcoming NFS v4.2, and also supported experimentally in this Linux release). In next releases, local filesystems such as Btrfs, and especialized storage devices that provide copy offloading facilities, could also use this system call to optimize the copy of data, or remove some of the present limitations (currently, copy offloading is limited to files on the same mount and superblock, and not in the same file)."

copy_file_range() (contd.)

```
ssize_t copy_file_range(int fd_in, loff_t *off_in,
                        int fd_out, loff_t *off_out,
                        size_t len, unsigned int flags)
```

The following semantics apply for `off_in`, and similar statements apply to `off_out`:

- If `off_in` is `NULL`, then bytes are read from `fd_in` starting from the file offset, and the file offset is adjusted by the number of bytes copied.
- If `off_in` is not `NULL`, then `off_in` must point to a buffer that specifies the starting offset where bytes from `fd_in` will be read. The file offset of `fd_in` is not changed, but `off_in` is adjusted appropriately.

Syscalls: Adv. I/O 8: Scatter-Gather I/O

1. Overview
2. Nonblocking I/O
3. Multiplexed I/O
4. Additional Uses for I/O Muliplexing
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)

Scatter-Gather I/O

Syscalls: Adv. I/O 9: Asynchronous I/O

1. Overview
2. Nonblocking I/O
3. Multiplexed I/O
4. Additional Uses for I/O Multiplexing
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)**
   -
   -
   -
   -

Asynchronous I/O