The Need for Multiplexing

Suppose: Two file descriptors (FDs) to read from (e.g., terminals, pipes, sockets), their data arrive intermittently. E.g., a server that serves many clients concurrently, one FD per client.

(Could also be wanting to write, but may block. E.g., a chat server writes to multiple clients, some are not reading, your buffers are clogging up.)

If blocking I/O: Blocked indefinitely on one, missing out if the other is ready.

If non-blocking I/O: Busying-loop over trying both?! Are you programming or frying eggs?!

Unix provides a system call ‘select’ to help. Linux adds ‘epoll’.

select

```
int select(int n, fd_set *r, fd_set *w, fd_set *e,
           struct timeval *timeout);
```

Blocks until specified FDs are ready for read/write, or timeout, or signal (discussed later). Returns 0 if timeout, positive count if some FDs are ready, -1 if error (e.g., invalid FD) or signal.

- `fd_set` type: Holds a set of FDs.
  (Next slide: How to insert, delete, query.)
- `n`: 1 + highest FD you specify.
- `r`: Those you want to read from. NULL if not needed.
- `w`: Those you want to write to. NULL if not needed.
- `e`: Not used in this course. Use NULL.
- `timeout`: Max wait time. NULL if not needed.

'select' modifies your fd_sets to those ready.

fd_set, struct timeval

```
void FD_ZERO(fd_set *s);          // make empty
void FD_SET(int fd, fd_set *s);   // add
void FD_CLR(int fd, fd_set *s);   // delete
int FD_ISSET(int fd, fd_set *s);  // query

struct timeval {
  time_t      tv_sec;
  suseconds_t tv_usec;  // microseconds
};
```

(Real hardware and OSes are unlikely to be accurate down to actual microseconds. Reasonable to expect milliseconds though.)

select Tips

- ‘select’ usually modifies your fd_sets. Always set them again before the next ‘select’ call.
- Ditto for the timeout struct.
- “Ready to read/write” does not mean there is data/room.
  Could mean the other end is closed,
  so EOF if read, broken pipe if write.
  “Ready” means won’t block if you try now.
  Except. . .
- May still block anyway if
  - Another process/thread beats you to it (if shared).
  - Large write, clogs buffer again.
  - Corner-case race condition.

  Pros set the FDs to non-blocking mode just in case.

Limitations of select

Cap on maximum fd_set size (1024 on most platforms—keep in mind typical servers want five times as many).

Slow when many FDs: You loop through them to `FD_SET`, then kernel loops through them, then you loop through them again to `FD_ISSET`.

Linux's 'epoll' supports more FDs and more efficiently. But only available on Linux. (Other OSes offer similar but incompatiable syscalls.)

epoll

API overview:

```
// create epoll instance
// return FD of new epoll instance (epfd below)
int epoll_create1(int flags);

// add/del/change what to monitor
int epoll_ctl(int epfd, int op, int fd,
              struct epoll_event *ev);

// wait
int epoll_wait(int epfd, struct epoll_event *evs,
               int n, int timeout);

// close(epfd) when done
```

epoll_create1

```
int epoll_create1(int flags);
```

flags can be 0 or `FD_CLOEXEC`. Returns an FD referring to a new “epoll instance”.

Like normal FDs in: has an entry in FD table, so everything about close, dup, fork, exec applies.

Unlike normal FDs in: not meaningful for read/write, only meaningful for other epoll functions.

epoll_ctl

Specify what to wait for.

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

Returns 0 on success.

epfd: epoll instance.

op: EPOLL_CTL_ADD (monitor fd), EPOLL_CTL_DEL (don't monitor fd), EPOLL_CTL_MOD (change what to monitor for fd).

fd cannot be regular file or directory. But can be another epoll instance (can has hierarchy of epolling)!

ev: events to wait for. Unused for EPOLL_CTL_DEL.

struct epoll_event

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

Bits for `events` field (may bitwise-or together):

`EPOLLIN`: ready to read

`EPOLLOUT`: ready to write

`EPOLLONESHOT`: monitor only once, non-recurring

`EPOLLET`: Edge-triggered

Others.

Edge-triggered = notify when change from not-ready to ready.
Default = level-triggered = notify whenever ready (like select).
Difference: E.g., data arrives, you read some but not all, then wait again. Level: notify again. Edge: won't notify again.

struct epoll_event

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

When epoll_ctl, you store anything you like.

When epoll_wait later, you get back what you stored.

What people usually store:

- the FD in question
- pointer to your own book-keeping struct

epoll_wait

```
int epoll_wait(int epfd, struct epoll_event *evs,
               int n, int timeout);
```

evs: Array to receive events.
n: Array length.
timeout: milliseconds. (-1 if no timeout.)
Returns count of ready FDs, i.e., how many entries in evs used.

events field has bits set to tell you which events occurred. Two more possible bits even if you didn't ask:

EPOLLHUP: The other end (e.g., pipe, socket) closed.
EPOLLERR: Error condition (e.g., broken pipe).

data field has what you stored with epoll_ctl add/mod.