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"