Outline

Overview

- Like *select()* and *poll()*, *epoll* can monitor multiple FDs
- *epoll* returns readiness information in similar manner to *poll()*
- Two main **advantages**:
  - *epoll* provides **much better performance** when monitoring large numbers of FDs
  - *epoll* provides two **notification modes**: **level-triggered** and **edge-triggered**
    - Default is level-triggered notification
    - *select()* and *poll()* provide only level-triggered notification
    - (Signal-driven I/O provides only edge-triggered notification)
- Linux-specific, since kernel 2.6.0

[TLPI §63.4]

epoll instances

Central data structure of *epoll* API is an ***epoll* instance**

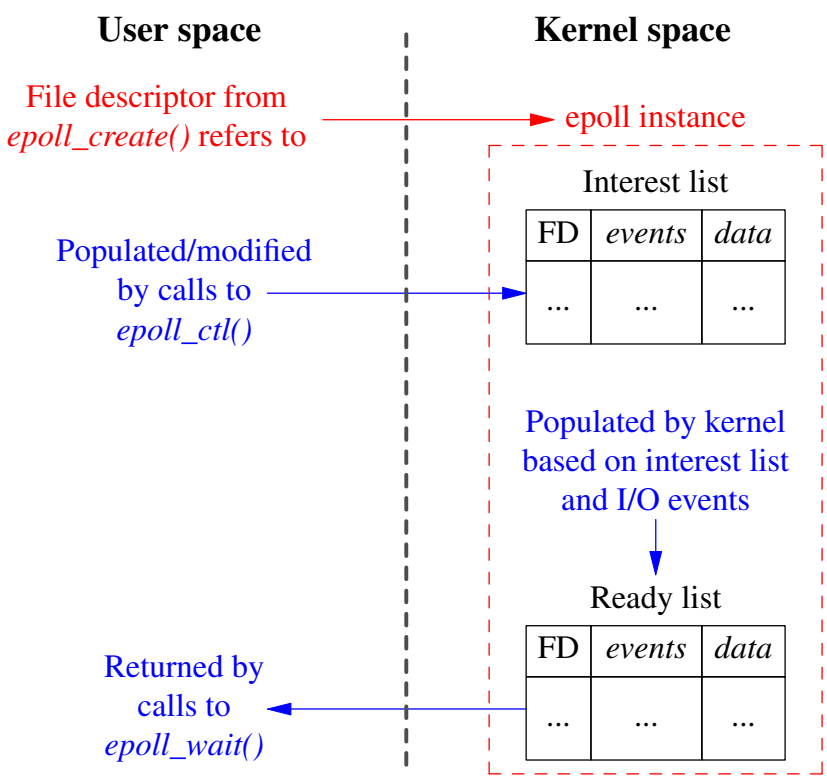
- **Persistent** data structure **maintained in kernel space**
- Referred to in user space via **file descriptor**
- Container for two information lists:
  - **Interest list**: a list of FDs that a process is interested in monitoring
  - **Ready list**: a list of FDs that are ready for I/O
  - Membership of ready list is a (dynamic) subset of interest list

epoll APIs

The key *epoll* APIs are:

- *epoll_create()*: create *epoll* instance and return FD referring to instance
  - FD is used in the calls below
- *epoll_ctl()*: modify interest list of *epoll* instance
  - Add FDs to/remove FDs from interest list
  - Modify events mask for FDs currently in interest list
- *epoll_wait()*: return items from ready list of *epoll* instance

epoll kernel data structures and APIs



Creating an *epoll* instance: *epoll_create()*

```
#include <sys/epoll.h>
int epoll_create(int size);
```

- Creates an *epoll* instance; returns FD referring to instance
- *size*:
  - Since Linux 2.6.8: serves no purpose, but must be > 0
  - Before Linux 2.6.8: an *estimate* of number of FDs to be monitored via this *epoll* instance
- Returns file descriptor on success, or -1 on error
  - When FD is no longer required it should be closed via *close()*
- Since Linux 2.6.27, there is an improved API, *epoll_create1()*
  - See the man page

[TLPI §63.4.1]

Modifying the *epoll* interest list: *epoll_ctl()*

```
#include <sys/epoll.h>
int epoll_ctl(int epfd, int op, int fd,
              struct epoll_event *ev);
```

- Modifies the interest list associated with *epoll* FD, *epfd*
- *fd*: identifies which FD in interest list is to have its settings modified
  - E.g., FD for pipe, FIFO, terminal, socket, POSIX MQ, or even another *epoll* FD
    - (Can't be FD for a regular file or directory)
- *op*: operation to perform on interest list
- *ev*: (Later)
- Returns 0 on success, or -1 on error

[TLPI §63.4.2]

epoll_ctl() op argument

The *epoll_ctl() op* argument is one of:

- `EPOLL_CTL_ADD`: add *fd* to interest list of *epfd*
  - *ev* specifies events to be monitored for *fd*
  - If *fd* is already in interest list ⇒ `EEXIST`
- `EPOLL_CTL_MOD`: modify settings of *fd* in interest list of *epfd*
  - *ev* specifies new settings to be associated with *fd*
  - If *fd* is not in interest list ⇒ `ENOENT`
- `EPOLL_CTL_DEL`: remove *fd* from interest list of *epfd*
  - *ev* is ignored
  - If *fd* is not in interest list ⇒ `ENOENT`
  - Closing an FD automatically removes it from all *epoll* interest lists

The *epoll_event* structure

epoll_ctl() ev argument is pointer to an *epoll_event* structure:

```
struct epoll_event {
    uint32_t      events;  /* epoll events (bit mask) */
    epoll_data_t  data;    /* User data */
};

typedef union epoll_data {
    void    *ptr;     /* Pointer to user-defined data */
    int      fd;      /* File descriptor */
    uint32_t u32;     /* 32-bit integer */
    uint64_t u64;     /* 64-bit integer */
} epoll_data_t;
```

- *ev.events*: bit mask of events to monitor for *fd*
  - (Similar to *events* mask given to *poll()*)
- *data*: info to be passed back to caller of *epoll_wait()* when *fd* later becomes ready
  - **Union field**: value is specified in *one* of the members

Example: using *epoll_create()* and *epoll_ctl()*

```
int epfd;
struct epoll_event ev;

epfd = epoll_create(5);

ev.data.fd = fd;
ev.events = EPOLLIN;  /* Monitor for input available */
epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
```

Outline

Waiting for events: *epoll_wait()*

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *evlist,
               int maxevents, int timeout);
```

- Returns info about ready FDs in interest list of *epoll* instance of *epfd*
- Info about ready FDs is returned in array *evlist*
  - (Caller allocates this array)
- *maxevents*: size of the *evlist* array
  - If > *maxevents* events are available, successive *epoll_wait()* calls round-robin through events

[TLPI §63.4.3]

Waiting for events: *epoll_wait()*

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *evlist,
               int maxevents, int timeout);
```

- *timeout* specifies a timeout for call:
  - -1: block until an FD in interest list becomes ready
  - 0: perform a nonblocking "poll" to see if any FDs in interest list are ready
  - > 0: block for up to *timeout* milliseconds or until an FD in interest list becomes ready
- Return value:
  - > 0: number of items placed in *evlist*
  - 0: no FDs became ready within interval specified by *timeout*
  - -1: an error occurred

[TLPI §63.4.3]

Waiting for events: *epoll_wait()*

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *evlist,
               int maxevents, int timeout);
```

- Info about **multiple** FDs can be returned in the array *evlist*
- Each element of *evlist* returns info about one file descriptor:
  - *events* is a **bit mask of events** that have occurred for FD
  - *data* is the ***ev.data* value** specified when the FD was registered with *epoll_ctl()*
- **NB**: the FD itself is **not** returned!
  - Instead, we put FD into *ev.data.fd* when calling *epoll_ctl()*, so that it is returned via *epoll_wait()*
    - (Or, put FD into a structure pointed to by *ev.data.ptr*)

epoll events

ev.events value given to *epoll_ctl()* and *evlist[].events* fields returned by *epoll_wait()* are bit masks of events

Bit	Input to *epoll_ctl()*?	Returned by *epoll_wait()*?	Description
EPOLLIN	•	•	Normal-priority data can be read
EPOLLPRI	•	•	High-priority data can be read
EPOLLRDHUP	•	•	Shutdown on peer socket
EPOLLOUT	•	•	Data can be written
EPOLLONESHOT	•		Disable monitoring after event notification
EPOLLET	•		Employ edge-triggered notification
EPOLLERR		•	An error has occurred
EPOLLHUP		•	A hangup occurred

- With the exception of EPOLLOUT and EPOLLET, these bit flags have the same meaning as the similarly named *poll()* bit flags

[TLPI §63.4.3]

Example: altio/epoll_input.c

```
./epoll_input file...
```

- Monitors one or more files using *epoll* API to see if input is possible
- Suitable files to give as arguments are:
  - FIFOs
  - Terminal device names
    - (May need to run *sleep* command in FG on the other terminal, to prevent shell stealing input)
  - Standard input
    - /dev/stdin

Example: altio/epoll_input.c (1)

```
1  #define MAX_BUF      1000     /* Max. bytes for read() */
2  #define MAX_EVENTS      5
3       /* Max. number of events to be returned from
4          a single epoll_wait() call */
5
6  int epfd, ready, fd, s, j, numOpenFds;
7  struct epoll_event ev;
8  struct epoll_event evlist[MAX_EVENTS];
9  char buf[MAX_BUF];
10
11 epfd = epoll_create(argc - 1);
```

- Declarations for various variables
- Create an *epoll* instance, obtaining *epoll* FD

Example: altio/epoll_input.c (2)

```
1  for (j = 1; j < argc; j++) {
2      fd = open(argv[j], O_RDONLY);
3      printf("Opened \"%s\" on fd %d\n", argv[j], fd);
4
5      ev.events = EPOLLIN;
6      ev.data.fd = fd;
7      epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
8  }
9
10 numOpenFds = argc - 1;
```

- Open each of the files named on command line
- Each file is monitored for input (EPOLLIN)
- *fd* placed in *ev.data*, so it is returned by *epoll_wait()*
- Add the FD to *epoll* interest list (*epoll_ctl()*)
- Track the number of open FDs

Example: altio/epoll_input.c (3)

```
while (numOpenFds > 0) {
    printf("About to epoll_wait()\n");
    ready = epoll_wait(epfd, evlist, MAX_EVENTS, -1);
    if (ready == -1) {
        if (errno == EINTR)
            continue;      /* Restart if interrupted
                              by signal */
        else
            errExit("epoll_wait");
    }
    printf("Ready: %d\n", ready);
```

- Loop, fetching *epoll* events and analyzing results
- Loop terminates when all FDs has been closed
- *epoll_wait()* call places up to MAX_EVENTS events in *evlist*
  - *timeout == -1* ⇒ infinite timeout
- Return value of *epoll_wait()* is number of ready FDs

Example: altio/epoll_input.c (4)

```
    for (j = 0; j < ready; j++) {
        printf("  fd=%d; events: %s%s%s\n", evlist[j].data.fd,
            (evlist[j].events & EPOLLIN)  ? "EPOLLIN "  : "",
            (evlist[j].events & EPOLLHUP) ? "EPOLLHUP " : "",
            (evlist[j].events & EPOLLERR) ? "EPOLLERR " : "");
        if (evlist[j].events & EPOLLIN) {
            s = read(evlist[j].data.fd, buf, MAX_BUF);
            printf("    read %d bytes: %.*s\n", s, s, buf);
        } else if (evlist[j].events & (EPOLLHUP | EPOLLERR)) {
            printf("    closing fd %d\n", evlist[j].data.fd);
            close(evlist[j].data.fd);
            numOpenFds--;
        }
    }
}
```

- Scan up to *ready* items in *evlist*
- Display *events* bits
- If EPOLLIN event occurred read some input and display it on *stdout*
- Otherwise, if error or hangup, close FD and decrements FD count
- Code correctly handles case where both EPOLLIN and EPOLLHUP are set in *evlist[j].events*