Προγραμματισμός Δικτύων – Ε-01

6η Διάλεξη

Διδάσκων: Νίκος Ντάρμος

<ntarmos@cs.uoi.gr>
[http://www.cs.uoi.gr/~ntarmos/Courses/NetworkProgramming/]

Τμήμα Πληροφορικής
Πανεπιστήμιο Ιωαννίνων

Στο σημερινό μάθημα...

- Πολυεπεξεργασία βασισμένη σε γεγονότα (συνέχεια).

Εισαγωγή

- Βασική ιδέα:
  - Ο εξυπηρετητής/πελάτης αποτελείται από μία διεργασία με ένα νήμα εκτέλεσης.
  - Ο ταυτοχρονισμός επιτυγχάνεται με χρήση non-blocking E/E και αναμονή για «γεγονότα».
- Έστω ένα σύνολο από socket descriptors. Υπάρχουν δύο κύριες κατηγορίες μεθόδων παρακολούθησης γεγονότων στο σύνολο αυτό:
  - «Level-triggered»: περιοδικά ελέγχουμε το σύνολο για αλλαγές/νέα γεγονότα.
  - «Edge-triggered»: αναφέρουμε στον πυρήνα το σύνολο των socket descriptors και κατόπιν αυτός μας ειδοποιεί **όταν γίνεται κάποια αλλαγή**.
- Πολλές υλοποιήσεις:
  - `select(2)` (POSIX.1).
  - `pselect(2)` (POSIX.1).
  - `poll(2)` (SVR4).
  - `SIGIO` (Linux).
  - `epoll(4)` (Linux).
  - `/dev/poll` (Solaris).
  - Event ports (Solaris).
  - `kqueue(2)` (*BSD, Mac OS X).
  - Grand Central Dispatch -- GCD (Mac OS X, FreeBSD).

Πολυπλεξία στην Ε/Ε

`select(2)`

- `int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);`
  - Περιμένει το πολύ `timeout` χρόνο (ή για πάντα αν `NULL`) για να γίνει κάποια αλλαγή στους πρώτους `nfds` file descriptors που βρίσκονται στα σύνολα `readfds`, `writefds` και `exceptfds`.
    - `nfds`: ο μεγαλύτερος file descriptor στα σύνολα, συν 1.
    - `readfds`: file descriptors από τους οποίους θα διαβάσουμε δεδομένα ή στους οποίους θα δεχτούμε συνδέσεις.
    - `writefds`: file descriptors στους οποίους θα γράψουμε δεδομένα.
    - `exceptfds`: file descriptors στους οποίους περιμένουμε «επείγοντα δεδομένα» (OOB) ή γενικά ειδικές καταστάσεις.
    - Όταν η `select(2)` επιστρέψει, στα σύνολα αυτά θα είναι «set» μόνο οι file descriptors που είναι έτοιμοι για Ε/Ε.
    - Επιστρέφει το πλήθος των file descriptors που είναι έτοιμοι για επεξεργασία, 0 για τέλος χρόνου, -1 για σφάλμα.
  - Για τη διαχείριση των συνόλων των file descriptors ορίζονται οι εξής μακροεντολές:
    - `FD_SET(fd, &fdset);`
    - `FD_CLR(fd, &fdset);`
    - `FD_ISSET(fd, &fdset);`
    - `FD_ZERO(&fdset);`

Πολυπλεξία στην Ε/Ε

select: Παράδειγμα

```
1  int sd, cd, sel;
2  fd_set fds;
3
4  sd = socket(/* ... */);
5  bind(sd, /* ... */);
6  listen(sd, /* ... */);
7
8  while (1) {
9      FD_ZERO(&fds);
10     FD_SET(sd, &fds);
11     sel = select(sd + 1, &fds, NULL, NULL, NULL);
12     if (sel <= 0) { /* do something */ }
13     if (sel > 0) {
14         if (FD_ISSET(sd, &fds)) {
15             cd = accept(sd, /* ... */);
16             if (cd < 0) error(/* ... */);
17             // Serve the request
18             close(cd);
19         }
20     }
21 }
```

Πολυπλεξία στην Ε/Ε

- Έστω το εξής σενάριο:

```
1   int num_events = 0;
2   void my_sigfunc(int s) {
3       num_events++;
4       signal(SIGXXX, my_sigfunc);
5   }
6   int main() {
7       // Variable declarations/initialization etc.
8       signal(SIGXXX, my_sigfunc);
9       while (1) {
10          for (; num_events > 0; num_events--) {
11              // Process signal events
12          }
13          sel = select(nfds, &rfds, &wfds, &efds, NULL);
14          // ...
15      }
16  }
```

  - Αν το `SIGXXX` έρθει όσο είμαστε στη `select(2)` τότε αυτή θα επιστρέψει με `errno = EINTR`.
  - **Πρόβλημα:** Τι γίνεται αν το `SIGXXX` έρθει μετά το `for(...)` αλλά πριν τη `select(2);`
  - **Απάντηση:** signal masking $\Rightarrow$ `pselect(2)`.

Πολυπλεξία στην Ε/Ε

pselect(2)

- `int pselect(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, const struct timespec *ntimeout, const sigset_t *sigmask);`
  - Περιμένει το πολύ `ntimeout` χρόνο (ή για πάντα αν `NULL`) για να γίνει κάποια αλλαγή στους πρώτους `nfds` file descriptors που βρίσκονται στα σύνολα `readfds`, `writefds` και `exceptfds`.
    - `nfds`, `readfds`, `writefds`, `exceptfds` και τιμή επιστροφής όπως και στη `select(2)`.
    - `sigmask`: σύνολο σημάτων που θα είναι ενεργοποιημένα κατά την εκτέλεση της κλήσης αυτής.
  - Για τη διαχείριση των συνόλων των file descriptors ορίζονται οι ίδιες μακροεντολές όπως και στη `select(2)`.
  - Για τη διαχείριση του `sigmask` ορίζονται αντίστοιχες συναρτήσεις – δείτε το `sigsetops(3)`.
  - Το `struct timespec` προσφέρει ακρίβεια νανοδευτερόλεπτου, ενώ το `struct timeval` της `select(2)` έχει «μόνο» μικροδευτερόλεπτου.
  - Το `struct timeval` της `select(2)` πρέπει να αρχικοποιείται πριν από κάθε κλήση της, ενώ η `pselect(2)` δεν αλλάζει το `struct timespec` της.

Πολυπλεξία στην Ε/Ε

pselect: Παράδειγμα

```
1   int num_events = 0;
2   void my_sigfunc(int s) {
3       num_events++;
4       signal(SIGXXX, my_sigfunc);
5   }
6   int main() {
7       // Variable declarations/initialization etc.
8       sigset_t sigmask, orig_mask;
9       sigemptyset(&sigmask);
10      sigaddset(&sigmask, SIGXXX);
11      sigprocmask(SIG_BLOCK, &sigmask, &orig_sigmask);
12      signal(SIGXXX, my_sigfunc);
13      while (1) {
14          for (; num_events > 0; num_events--) {
15              // Process signal events
16          }
17          sel = pselect(nfds, &rfds, &wfds, &efds, NULL, &orig_sigmask);
18          // ...
19      }
20  }
```

Πολυπλεξία στην Ε/Ε

- **Πρόβλημα:** Οι select(2) και pselect(2) μπορούν να παρακολουθήσουν ως το πολύ `FD_SETSIZE` file descriptors
  - 1024 σε *BSD, Solaris, Linux.
  - Πολύ μικρότερο από το μέγιστο πλήθος των file descriptors ανά διεργασία.
- **Πρόβλημα:** Ακόμα κι αν παρακολουθούμε μόνο ένα file descriptor στην `select(2)/pselect(2)`, εσωτερικά η υλοποίηση ελέγχει `nfds` file descriptors!
- **Απάντηση:** `poll(2)`.

Πολυπλεξία στην E/E

`poll(2)`

- `int poll(struct pollfd fds[], nfds_t nfds, int timeout);`
  - Αρχικά είχε οριστεί για το API των STREAMS· κατόπιν ορίστηκε και υλοποιήθηκε και για αρχεία, sockets, FIFOs, pipes κτλ.
  - Περιμένει το πολύ `timeout` milliseconds (για πάντα αν `INFTIM` (= -1) ή καθόλου αν 0) για να γίνει κάποια αλλαγή στους file descriptors που περιγράφονται στον, μεγέθους `nfds`, πίνακα `fds`.
  - Επιστρέφει το πλήθος των FDs που είναι έτοιμοι για E/E, 0 για τέλος χρόνου, -1 για σφάλμα.

Πολυπλεξία στην Ε/Ε

poll(2)

- Τα πεδία της δομής `pollfd` είναι ως εξής:
  - `fd`: ο file descriptor προς παρακολούθηση. Αν -1, αρχικοποίησε το πεδίο `revents`, και αγνόησε τη δομή αυτή.
  - `events`: Γεγονότα προς παρακολούθηση (bitmask).
  - `revents`: Γεγονότα που συνέβησαν (bitmask). Καθορίζει τι είδους Ε/Ε μπορεί να γίνει.
- Πιθανοί τύποι γεγονότων:
  - `POLLIN`: υπάρχουν δεδομένα προς ανάγνωση.
  - `POLLRDNORM`, `POLLOUT`/`POLLWRNORM`: μπορεί να γίνει λήψη ή αποστολή μη επειγόντων δεδομένων.
  - `POLLRDBAND`, `POLLWRBAND`: μπορεί να γίνει λήψη / αποστολή επειγόντων δεδομένων.
  - `POLLPRI`: υπάρχουν δεδομένα υψηλής προτεραιότητας προς ανάγνωση.
  - `POLLERR`: συνέβη κάποιο σφάλμα.
  - `POLLHUP`: η σύνδεση/το αρχείο έκλεισε.
  - `POLLNVAL`: άκυρος τύπος γεγονότος ή κλειστό αρχείο/σύνδεση.

struct pollfd

```
struct pollfd {
    int    fd;
    short  events;
    short  revents;
};
```

Πολυπλεξία στην Ε/Ε

poll: Παράδειγμα

```
1  int sd, cd, nready;
2  struct pollfd clients[1];
3
4  sd = socket(/* ... */);
5  bind(sd, /* ... */);
6  listen(sd, /* ... */);
7
8  clients[0].fd = sd;
9  clients[0].events = POLLRDNORM;
10
11 while (1) {
12     nready = poll(clients, 1, INFTIM);
13     if (nready <= 0) { /* do something */ }
14     if (nready > 0) {
15         if (clients[0].revents & POLLRDNORM) {
16             cd = accept(sd, /* ... */);
17             if (cd < 0) error(/* ... */);
18             // Serve the request
19             close(cd);
20         }
21     }
22 }
```

Πολυπλεξία στην Ε/Ε

- Οι `select(2)`, `pselect(2)` και `poll(2)` υπάρχουν σχεδόν σε όλα τα συστήματα που υλοποιούν το πρότυπο POSIX.1.
- **Πρόβλημα:** Η `poll(2)` δεν έχει τον περιορισμό του `FD_SETSIZE` των `select(2)` και `pselect(2)`, ωστόσο γίνεται πολύ αργή για μεγάλο αριθμό από παρακολουθούμενους file descriptors.
- **Απάντηση:** Κάθε λειτουργικό υλοποιεί κάποιο API που παρέχει την λειτουργικότητα της `poll(2)`, χωρίς το πρόβλημα της απόδοσης.
  - Linux: `SIGIO` (πυρήνες 2.4.x), `epoll(4)` (πυρήνες 2.6.x).
  - *BSD, Mac OS X: `kqueue(2)`.
  - FreeBSD, Mac OS X: Grand Central Dispatch.
  - Solaris: `/dev/poll` (`poll(7D)`), Event ports.

Πολυπλεξία στην Ε/Ε

SIGIO

- Η μέθοδος αυτή έγκειται στη χρήση «σημάτων πραγματικού χρόνου» και προσφέρει «edge-triggered» παρακολούθηση.
- Διαθέσιμη μόνο σε Linux, σε εκδόσεις πυρήνα 2.4.x και μεγαλύτερη.
  - Αρχικά, «μπλοκάρουμε» το σήμα που θα χρησιμοποιήσουμε, καθώς και το `SIGIO`.
  - Καθορίζουμε ότι ο πυρήνας θα μας αποστέλλει το σήμα αυτό κάθε φορά που Ε/Ε είναι δυνατή σε κάποιον file descriptor.
  - Κατόπιν, έχουμε ένα εξωτερικό βρόγχο με `poll(2)`. Σε κάθε επανάληψη:
    - Χειριζόμαστε ότι μας έχει επιστρέψει η `poll(2)`.
    - Εισερχόμαστε σε ένα δεύτερο βρόγχο στον οποίο καλούμε `sigwaitinfo(2)`. Αν έρθει το σήμα που καθορίσαμε προηγουμένως, η κλήση μας επιστρέφει τις πληροφορίες που χρειαζόμαστε για την εξυπηρέτηση του γεγονότος. Αν έρθει `SIGIO` τότε υπήρξε υπερχείλιση της ουράς σημάτων, οπότε αδειάζουμε την ουρά θέτοντας τον χειριστή του σήματος σε `SIG_IGN` και βγαίνουμε στον εξωτερικό βρόγχο.

Πολυπλεξία στην E/E

SIGIO: Παράδειγμα

```
1  int fds[XXX], flags, signum;
2  sigset_t sigset;
3  siginfo_t siginfo;
4  // ...
5  sigemptyset(&sigset);
6  sigaddset(&sigset, SIGPOLL);
7  sigaddset(&sigset, SIGIO);
8  sigprocmask(SIG_BLOCK, &sigset, NULL);
9  foreach(fd in fds) {
10     fcntl(fd, F_SETOWN, (int)getpid());
11     fcntl(fd, F_SETSIG, SIGPOLL);
12     flags = fcntl(fd, F_GETFL);
13     fcntl(fd, F_SETFL, flags | O_NONBLOCK |
             O_ASYNC);
14 }
```

```
16 while (1) {
17     nready = poll(/* ... */);
18     /* process poll outcome */
19     while (1) {
20         signum = sigwaitinfo(&sigset, &siginfo);
21         if (signum == SIGIO) {
22             signal(SIGPOLL, SIG_IGN);
23             signal(SIGPOLL, SIG_DFL);
24             // ...
25             break;
26         } else if (signum == SIGPOLL) {
27             // siginfo.si_fd: event file
                   descriptor
28             // siginfo.si_band: poll(2) events
29         }
30     }
31 }
```

Πολυπλεξία στην Ε/Ε

epoll

- Πρόκειται για ένα API το οποίο επιτρέπει τη δημιουργία και τον έλεγχο ενός συνόλου γεγονότων.
- Προσφέρει τόσο level-triggered όσο και edge-triggered παρακολούθηση.
- Διαθέσιμο μόνο σε Linux, έκδοση πυρήνα 2.6.x.
- Καθορίζονται 4 βασικές συναρτήσεις: `epoll_create(2)`, `epoll_ctl(2)`, `epoll_wait(2)` και `epoll_pwait(2)`.

```
#include <sys/epoll.h>

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

Πολυπλεξία στην E/E

`epoll_create(2)`

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

- Δημιουργεί ένα σύνολο τουλάχιστον `size` γεγονότων.
- Επιστρέφει έναν «file descriptor» που αναφέρεται στο σύνολο αυτό και τον οποίο πρέπει στο τέλος να αποδεσμεύσουμε με `close(2)`, -1 για σφάλμα.

Πολυπλεξία στην Ε/Ε

epoll_ctl(2)

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

- `epfd`: Ο file descriptor που αντιστοιχεί στο σύνολο γεγονότων.
- `op`: Η πράξη που θα γίνει στο παραπάνω σύνολο.
  - `EPOLL_CTL_ADD`: Πρόσθεσε το `fd` στο σύνολο `epfd` και αντιστοίχισε το γεγονός `event` με αυτό.
  - `EPOLL_CTL_MOD`: Αντιστοίχισε το γεγονός `event` με το `fd` στο σύνολο `epfd`.
  - `EPOLL_CTL_DEL`: Αφαίρεσε το `fd` από το σύνολο `epfd` (το `event` αγνοείται).
- `event`: Περιγράφει το σύνολο γεγονότων που θα παρακολουθούνται στο `fd`.
- Επιστρέφει 0 για επιτυχία, -1 για σφάλμα.

```
struct epoll_event {
    __uint32_t events;
    epoll_data_t data;
};
```

```
typedef union epoll_data {
    void *ptr;
    int fd;
    __uint32_t u32;
    __uint64_t u64;
} epoll_data_t;
```

Πολυπλεξία στην Ε/Ε

`epoll_ctl(2)`

- Το πεδίο `events` της δομής `struct epoll_event` είναι ένα bitmask, όπως και στην περίπτωση της `poll(2)`.
- Δυνατές τιμές:
  - `EPOLLIN`: υπάρχουν δεδομένα προς ανάγνωση.
  - `EPOLLOUT`: μπορεί να γίνει αποστολή δεδομένων.
  - `EPOLLRDHUP`: η σύνδεση έκλεισε (το άλλο άκρο κάλεσε `close(2)` ή `shutdown(2)` για αποστολή).
  - `EPOLLPRI`: υπάρχουν δεδομένα προτεραιότητας προς ανάγνωση.
  - `EPOLLERR`: συνέβη κάποιο σφάλμα. Η `epoll_wait(2)` πάντα περιμένει για τέτοιου είδους γεγονότα κι έτσι δε χρειάζεται να το προσθέτουμε.
  - `EPOLLHUP`: η σύνδεση/το αρχείο έκλεισε. Η `epoll_wait(2)` πάντα περιμένει για τέτοιου είδους γεγονότα κι έτσι δε χρειάζεται να το προσθέτουμε.
  - `EPOLLET`: θέτει τον αντίστοιχο file descriptor σε κατάσταση «edge triggered» παρακολούθησης.
  - `EPOLLONESHOT`: απενεργοποιεί την παρακολούθηση μετά το πρώτο γεγονός. Μπορούμε να την ενεργοποιήσουμε εκ νέου με χρήση της `epoll_ctl(2)` με `EPOLL_CTL_MOD`.

Πολυπλεξία στην Ε/Ε

`epoll_wait(2)` / `epoll_pwait(2)`

```
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);
```

- Περιμένει το πολύ `timeout` milliseconds (για πάντα αν -1, καθόλου αν 0) να συμβεί κάποιο γεγονός στο σύνολο που καθορίζει το `epfd`.
- `events`: Πίνακας από `maxevents` δομές `struct epoll_event` που αντιστοιχούν στα γεγονότα που συνέβησαν μετά την τελευταία κλήση της `epoll_wait(2)`/`epoll_pwait(2)`.
- `sigmask`: Χρησιμοποιείται όπως και στην `pselect(2)`.
- Επιστρέφει το πλήθος των γεγονότων (file descriptors που είναι διαθέσιμοι για Ε/Ε), 0 για τέλος χρόνου, -1 για σφάλμα.

Πολυπλεξία στην Ε/Ε

epoll: Παράδειγμα

```
1  int sd, cd, nready, epfd, i, flags;
2  struct epoll_event ev, events[100];
3
4  sd = socket(/* ... */);
5  bind(sd, /* ... */);
6  listen(sd, /* ... */);
7
8  epfd = epoll_create(50);
9  ev.events = EPOLLIN;
10 ev.data.fd = sd;
11 epoll_ctl(epfd, EPOLL_CTL_ADD, sd, &ev);
12
13 while (1) {
14     nready = epoll_wait(epfd, events, 100, -1);
15     if (nready <= 0) { /* do something */ }
16     for (i = 0; i < nready; i++) {
17         if (events[i].data.fd == sd) {
18             cd = accept(sd, /* ... */);
19             flags = fcntl(cd, F_GETFL);
20             fcntl(cd, F_SETFL, flags |
                       O_NONBLOCK);
22             ev.data.fd = cd;
23             ev.events = EPOLLIN;
24             epoll_ctl(epfd, EPOLL_CTL_ADD, cd,
                       &ev);
25         } else if (events[i].events & EPOLLIN) {
26             cd = events[i].data.fd;
27             ev.data.fd = cd;
28             ev.events = EPOLLOUT;
29             epoll_ctl(epfd, EPOLL_CTL_MOD, cd,
                       &ev);
30             recv(cd, /* ... */);
31         } else if (event[i].events & EPOLLOUT) {
32             cd = events[i].data.fd;
33             send(cd, /* ... */);
34             ev.data.fd = cd;
35             epoll_ctl(epfd, EPOLL_CTL_DEL, cd,
                       &ev);
36             close(cd);
37         }
38     }
39 }
40
41 close(epfd);
```

Πολυπλεξία στην Ε/Ε

/dev/poll

- Πρόκειται για ένα API το οποίο χρησιμοποιεί μία ιδεατή συσκευή (`/dev/poll`) και επιτρέπει το γρήγορο έλεγχο ετοιμότητας πολλαπλών file descriptors.
- Διαθέσιμο μόνο σε SunOS/Solaris.
- Για να το χρησιμοποιήσουμε:
  - Χρησιμοποιούμε κι εδώ έναν πίνακα από δομές `struct pollfd` όπως και στην `poll(2)`.
  - Ορίζεται επιπλέον το «ειδικό» γεγονός `POLLREMOVE` με το οποίο αφαιρούμε file descriptors από το σύνολο των παρακολουθούμενων.
  - Αρχικά ανοίγουμε (`open(2)`) το αρχείο `/dev/poll`.
  - Για να παρακολουθήσουμε το σύνολο των file descriptors για τα γεγονότα που περιγράφονται από τον παραπάνω πίνακα, γράφουμε τον πίνακα αυτό στο παραπάνω αρχείο. Το πεδίο `revents` δεν χρησιμοποιείται.
  - Κατόπιν ελέγχουμε για γεγονότα με κλήσεις της συνάρτησης `ioctl(2)` στο αρχείο αυτό.

```
#include <sys/devpoll.h>
int fd = open("/dev/poll", O_RDWR);
ssize_t n = write(int fd, struct pollfd buf[], int bufsize);
int n = ioctl(int fd, DP_POLL, struct dvpoll* arg);
int n = ioctl(int fd, DP_ISPOLLED, struct pollfd* pfd);
```

Πολυπλεξία στην Ε/Ε

/dev/poll

- Ο πίνακας δομών που γράφουμε στο `/dev/poll` καθορίζει ποια γεγονότα μας ενδιαφέρουν και ποιους file descriptors παρακολουθούμε.
- Η δομή `struct dvpoll` καθορίζει πόσο χρόνο θα περιμένει η κλήση στην `ioctl(2)` (`dp_timeout`) και περιέχει τα αποτελέσματα της «`poll(7D)`».
- Κλήση της `ioctl(2)` με όρισμα `DP_POLL` ελέγχει για τα γεγονότα.
- Κλήση της `ioctl(2)` με όρισμα `DP_ISPOLLED` ελέγχει αν κάποιος συγκεκριμένος file descriptor ανήκει στο σύνολο των παρακολουθούμενων.

struct dvpoll

```
struct dvpoll {
    struct pollfd* dp_fds;
    int dp_nfds;
    int dp_timeout;
};
```

Πολυπλεξία στην Ε/Ε

/dev/poll: Παράδειγμα

```
1  int sd, cd, pfd, res, npfds, i;
2  struct pollfd pfds[2];
3
4  sd = socket(/* ... */);
5  bind(sd, /* ... */);
6  listen(sd, /* ... */);
7
8  pfd = open("/dev/poll", O_RDWR);
9  pfds[0].fd = sd;
10 pfds[0].events = POLLIN;
11 pfds[0].revents = 0;
12 npfds = 1;
13 write(pfd, &pfds[0], npfds * sizeof(struct
       pollfd));
14
15 while (1) {
16     dp.dp_timeout = -1;
17     dp.dp_nfds = npfds;
18     dp.dp_fds = pfds;
19     res = ioctl(pfd, DP_POLL, &dp);
20     if (res < 0) { /* error */ }
21     for (i = 0; i < res; i ++) {
22         if (dp.dp_fds[i].fd == sd) {
23             cd = accept(sd, NULL, 0);
24             flags = fcntl(cd, F_GETFL);
25             fcntl(cd, F_SETFL, flags |
                   O_NONBLOCK);
27             pfds[1].fd = cd;
28             pfds[1].events = POLLIN;
29             pfds[1].revents = 0;
30             npfds++;
31             write(pfd, pfds, npfds * sizeof(
                   struct pollfd));
32         } else if (dp.dp_fds[i].revents &
                   POLLIN) {
33             cd = dp.dp_fds[i].fd;
34             pfds[1].events = POLLOUT;
35             write(pfd, pfds, npfds * sizeof(
                   struct pollfd));
36             recv(cd, /* ... */);
37         } else if (dp.dp_fds[i].revents &
                   POLLOUT) {
38             cd = dp.dp_fds[i].fd;
39             send(cd, /* ... */);
40             pfds[1].events = POLLREMOVE;
41             write(pfd, pfds, npfds * sizeof(
                   struct pollfd));
42             npfds--;
43             close(cd);
44         }
45     }
46 }
47
48 close(pfd);
```

Πολυπλεξία στην E/E

Event ports

- Πρόκειται για ένα API το οποίο προσφέρει γενικότερα παρακολούθηση γεγονότων (όχι μόνο έλεγχο ετοιμότητας για file descriptors).
- Υποστηρίζεται μόνο στο SunOS/Solaris, έκδοση 10 και μεγαλύτερη.
- Επιτρέπει την παρακολούθηση:
  - Αρχείων, sockets, pipes, κτλ.
  - Συναλλαγών ασύγχρονης E/E (asynchronous I/O (AIO) transactions).
  - Timers.
  - Γενικών γεγονότων που καθορίζονται από τον χρήστη.
- Για περισσότερες πληροφορίες:

  `http://developers.sun.com/solaris/articles/event_completion.html`

Πολυπλεξία στην E/E

kqueue

- Πρόκειται επίσης για API το οποίο προσφέρει παρακολούθηση γεγονότων και πέρα από απλή ετοιμότητα για file descriptors.
- Υποστηρίζεται από όλα τα BSDs (FreeBSD, NetBSD, OpenBSD, DragonflyBSD, Mac OS X).
- Επιτρέπει την παρακολούθηση:
  - Αρχείων, sockets, pipes και οτιδήποτε μπορούμε να χειριστούμε με file descriptors.
  - Συναλλαγών ασύγχρονης E/E (asynchronous I/O (AIO) transactions).
  - Παράδοσης σημάτων.
  - Γεγονότων διεργασιών (δημιουργία νέας διεργασίας, τερματισμός διεργασίας, κτλ.)
  - Κατάστασης συστήματος αρχείων.
- Καθορίζει 2 βασικές συναρτήσεις και μια μακροεντολή: `kqueue(2)`, `kevent(2)`, `EV_SET`.

```
int kqueue(void);
int kevent(int kq, const struct kevent *changelist, int nchanges,
    struct kevent *eventlist, int nevents, const struct timespec *
    timeout);
EV_SET(&kev, ident, filter, flags, fflags, data, udata);
```

Πολυπλεξία στην Ε/Ε

kqueue(2)

```
int kqueue(void);
```

- Δημιουργεί μία νέα «ουρά γεγονότων» στον πυρήνα
- Επιστρέφει τον χειριστή της ουράς, -1 για σφάλμα.
  - Οι ουρές αυτές δεν κληρονομούνται σε διεργασίες–παιδιά, εκτός και χρησιμοποιηθεί η κλήση συστήματος `rfork(2)` με κατάλληλα ορίσματα (`RFFDG`).

Πολυπλεξία στην Ε/Ε

kevent(2)

```
int kevent(int kq, const struct kevent *changelist, int nchanges,
    struct kevent *eventlist, int nevents, const struct timespec *
    timeout);
```

- Χρησιμοποιείται τόσο για να προσθέσει γεγονότα προς παρακολούθηση στην ουρά γεγονότων, όσο και για να ελέγξει για καινούρια συμβάντα.
- Αρχικά εφαρμόζει τις αλλαγές που περιγράφει ο μεγέθους `nchanges` πίνακας `changelist`.
- Κατόπιν εξετάζει για νέα συμβάντα και τα επιστρέφει στον μεγέθους `nevents` πίνακα `eventlist`. Αν το nevents είναι μηδενικό, επιστρέφει αμέσως ακόμα κι αν υπάρχει ορισμένο `timeout`.
- Περιμένει το πολύ `timeout` χρόνο για νέα συμβάντα. Αν το `timeout` είναι `NULL`, περιμένει για πάντα. Αν δείχνει σε μηδενικό `struct timespec` επιστρέφει αμέσως.
- Τα `changelist` και `eventlist` μπορεί να δείχνουν στον ίδιο πίνακα.
- Αν συμβούν πολλαπλά γεγονότα για το ίδιο «φίλτρο», επιστρέφονται σε μία δομή `struct kevent` και όχι σε πολλές.

Πολυπλεξία στην E/E

EV_SET

`EV_SET(&kev, ident, filter, flags, fflags, data, udata);`

- Χρησιμοποιείται για να αρχικοποιήσει μία δομή `struct kevent`.
- Απλά αναθέτει τα ορίσματά της στα πεδία της δομής.

struct kevent

```
struct kevent {
    uintptr_t ident;
    short     filter;
    u_short   flags;
    u_int     fflags;
    intptr_t  data;
    void*     udata;

};
```

Πολυπλεξία στην Ε/Ε

Δομή `struct kevent`

- `ident`: ταυτοποιεί το γεγονός. Εξαρτάται από τον τύπο του γεγονότος. Για ετοιμότητα Ε/Ε το θέτουμε στην τιμή του αντίστοιχου file descriptor.
- `filter`: καθορίζει τον έλεγχο που θα γίνεται για το γεγονός αυτό. Δυνατές τιμές:
  - `EVFILT_READ`: έλεγχος για ύπαρξη δεδομένων προς ανάγνωση από κάποιον file descriptor.
  - `EVFILT_WRITE`: έλεγχος για δυνατότητα εγγραφής σε κάποιον file descriptor.
  - `EVFILT_SIGNAL`: έλεγχος για παράδοση σημάτων στη διεργασία.
  - `EVFILT_AIO`, `EVFILT_VNODE`, `EVFILT_PROC`, `EVFILT_TIMER`, `EVFILT_NETDEV`.
- `udata`: δεδομένα που καθορίζει ο χρήστης και παραμένουν ανέπαφα από τον πυρήνα.
  - Παράδειγμα: δείκτης σε συνάρτηση που θα κληθεί όταν συμβεί το αντίστοιχο γεγονός.

`struct kevent`

```
struct kevent {
    uintptr_t ident;
    short     filter;
    u_short   flags;
    u_int     fflags;
    intptr_t  data;
    void*     udata;

};
```

Πολυπλεξία στην Ε/Ε

Δομή `struct kevent`

- `flags`: καθορίζει τις πράξεις που θα κάνει η συγκεκριμένη κλήση.
  - `EV_ADD`: προσθέτει/ανανεώνει μία δομή στην ουρά και την ενεργοποιεί αυτόματα, εκτός κι αν οριστεί επίσης η τιμή `EV_DISABLE`.
  - `EV_ENABLE`: ενεργοποιεί την παρακολούθηση ενός γεγονότος.
  - `EV_DISABLE`: απενεργοποιεί την παρακολούθηση ενός γεγονότος.
  - `EV_ONESHOT`: επέστρεψε το πρώτο σχετικό γεγονός από την ουρά και αφαίρεσε την αντίστοιχη δομή από την προς παρακολούθηση ουρά.
  - `EV_CLEAR`: επαναρχικοποίησε κάποιο γεγονός μετά την παράδοσή του στον χρήστη.
  - `EV_EOF`: χρησιμοποιείται από κάποια «φίλτρα» για να υποδείξει αντίστοιχη κατάσταση `EOF`.
  - `EV_ERROR`: σφάλμα.
- `fflags,data`: χρησιμοποιούνται για να δίνονται ορίσματα στις συναρτήσεις των φίλτρων, ανάλογα την περίπτωση.
  - Παράδειγμα: αν εξετάζουμε κάποιο socket με φίλτρο EVFILT_READ και η κατεύθυνση ανάγνωσης έχει κλείσει (`shutdown(2)`), επιστρέφεται ο αντίστοιχος κωδικός λάθους στο `fflags`. Αν πρόκειται για listen socket, επιστρέφεται το πλήθος των πελατών σε αναμονή στο πεδίο `data`. Αν πρόκειται για κανονικό socket, επιστρέφεται το μέγεθος των διαθέσιμων δεδομένων.

`struct kevent`

```
struct kevent {
    uintptr_t ident;
    short     filter;
    u_short   flags;
    u_int     fflags;
    intptr_t  data;
    void*     udata;

};
```

Πολυπλεξία στην Ε/Ε

kqueue: Παράδειγμα

```
 1  int sd, cd, kq, nevents, nch, i, nkv = 100 ;
 2  struct kevent kv[100];
 3
 4  sd = socket(/* ... */);
 5  bind(sd, /* ... */);
 6  listen(sd, /* ... */);
 7
 8  kq = kqueue();
 9
10  memset(&kv, 0, nkv * sizeof(struct kevent));
11  kv[0].ident = sd;
12  kv[0].filter = EVFILT_READ;
13  kv[0].flags = EV_ADD;
14  nch = 1;
15
16  while (1) {
17      nevents = kevent(kq, kv, nch, kv, nkv, NULL);
18      nch = 0;
19      if (nevents <= 0) { /* Do something */ }
20      for (i = 0; i < nevents; i++) {
21          if (kv[i].flags & EV_EOF) {
22              close(kv[i].ident);
23          } else if (kv[i].ident == sd) {
24              cd = accept(kv[i].ident, /* ... */);
25
26              kv[nch].ident = cd;
27              kv[nch].filter = EVFILT_READ;
28              kv[nch].flags = EV_ADD;
29              nch++;
30          } else if (kv[i].filter == EVFILT_READ) {
31              read(kv[i].ident, /* ... */);
32              kv[nch].ident = cd;
33              kv[nch].filter = EVFILT_READ;
34              kv[nch].flags = EV_DELETE;
35              nch++;
36              kv[nch].ident = cd;
37              kv[nch].filter = EVFILT_WRITE;
38              kv[nch].flags = EV_ADD;
39              nch++;
40          } else if (kv[i].filter == EVFILT_WRITE)
                {
41              write(kv[i].ident, /* ... */);
42              kv[nch].ident = cd;
43              kv[nch].filter = EVFILT_WRITE;
44              kv[nch].flags = EV_DELETE;
45              nch++;
46              close(kv[i].ident);
47          } else
48              /* error */
49      }
50  }
```

Πολυπλεξία στην Ε/Ε

Grand Central Dispatch (GCD)

- Πρόκειται για ένα API το οποίο επιτρέπει την παραλληλοποίηση πολλών εργασιών, συμπεριλαμβανομένης της διαχείρισης των νημάτων μίας διεργασίας.
- Υποστηρίζεται μόνο στο Mac OS X, έκδοση 10.6 και μεγαλύτερη, και στο FreeBSD, έκδοση 8.1-RELEASE και μεγαλύτερη.
- Απαιτεί υποστήριξη και από τον compiler.
- Επιτρέπει την παρακολούθηση:
  - Αρχείων, sockets, pipes, κτλ.
  - Συναλλαγών ασύγχρονης Ε/Ε (asynchronous I/O (AIO) transactions).
  - Timers.
  - Γενικών γεγονότων που καθορίζονται από τον χρήστη.
- Επιπλέον, επιτρέπει τον ορισμό *blocks*: τμήματα κώδικα που μπορούν να τρέξουν παράλληλα.
  - Το GCD καθορίζει τότε πόσα και ποια νήματα θα εκτελούνται κάθε φορά, ούτως ώστε η χρήση των πυρήνων του συστήματος να είναι η βέλτιστη.
- Για περισσότερες πληροφορίες:

  `http://developer.apple.com/mac/library/documentation/Performance/Reference/GCD_libdispatch_Ref/`

Στο επόμενο μάθημα...

- Βασικές και προχωρημένες αρχιτεκτονικές εξυπηρετητών.
- Σχεδίαση αποδοτικών εξυπηρετητών.
- Επιλογές υποδοχών.
- Προχωρημένες τεχνικές Ε/Ε.