Προγραμματισμός Δικτύων – Ε-01

7η Διάλεξη

Διδάσκων: Νίκος Ντάρμος

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

Τμήμα Πληροφορικής
Πανεπιστήμιο Ιωαννίνων

Στο σημερινό μάθημα...

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

Βασικές και προχωρημένες αρχιτεκτονικές εξυπηρετητών

Διάρθρωση

- Ταυτόχρονη εξυπηρέτηση πολλαπλών αιτήσεων.
- Ταυτόχρονη υποστήριξη πολλαπλών πρωτοκόλλων.
- Ταυτόχρονη υποστήριξη πολλαπλών υπηρεσιών.
- Σχεδιάζοντας αποδοτικούς εξυπηρετητές.

Εξυπηρέτηση πολλαπλών αιτήσεων

Ταυτόχρονη εξυπηρέτηση πολλαπλών αιτήσεων

- Ποια είναι η «καλύτερη» αρχιτεκτονική;
  - Διεργασίες;
  - Νήματα;
  - Γεγονότα;
  - Υβρίδια;
- Πως ορίζεται η «καλύτερη»;
  - Γρηγορότερη στην απόκριση;
  - Με μικρότερες απαιτήσεις εύρους ζώνης δικτύου ή/και υπολογιστικών πόρων;
  - Ευκολότερη στην υλοποίηση;
  - Σταθερότερη υπό φόρτο;
  - Καλύτερα κλιμακώσιμη;
  - ...

Ταυτόχρονη εξυπηρέτηση πολλαπλών αιτήσεων

Επαναληπτικοί (iterative) εξυπηρετητές

- Μόνο ένα νήμα/διεργασία εξυπηρέτησης.
- Οι αιτήσεις επεξεργάζονται σειριακά.

+ Πολύ απλή υλοποίηση: δεν χρειάζεται συγχρονισμός, διαχείριση σημάτων, διαχείριση διεργασιών, . . .
+ Χρήσιμοι όταν η χρονική διάρκεια επεξεργασίας των αιτήσεων είναι περιορισμένη (π.χ. υπηρεσία *daytime*).

– Προβληματικοί όταν ο ρυθμός άφιξης πελατών είναι μεγάλος σε σχέση με το χρόνο εξυπηρέτησης.
  - Ο χρόνος εξυπηρέτησης δεν εξαρτάται μόνο από τον εξυπηρετητή.
    - Καθυστερήσεις λόγω Ε/Ε, δικτύου και κλήσεων που μπορεί να κάνουν.
    - Καθυστερήσεις λόγω φόρτου συστήματος.
    - Καθυστερήσεις λόγω κακής σχεδίασης του πελάτη.

– Κάθε πελάτης «πληρώνει» την καθυστέρηση των προηγούμενων από αυτόν.
– Κίνδυνος για λιμοκτονία, livelocks ή επιθέσεις τύπου Denial of Service (DoS) αν ο τερματισμός της σύνδεσης εξαρτάται από τον πελάτη.
  - Παράδειγμα: ένας πελάτης συνδέεται, ο εξυπηρετητής περιμένει τα δεδομένα της αίτησής του, αλλά αυτός δεν τα στέλνει ποτέ ή τα στέλνει με αργό ρυθμό.

Ταυτόχρονη εξυπηρέτηση πολλαπλών αιτήσεων

- Για «ποιοτική» εξυπηρέτηση ή πιο πολύπλοκες συνθήκες, **απαιτείται ταυτοχρονισμός.**
- Διακρίνουμε τις εξής περιπτώσεις:
  - Ταυτόχρονοι εξυπηρετητές.
    - Μία διεργασία για κάθε αίτηση.
    - Ένα νήμα για κάθε αίτηση.
  - Επαναληπτικοί εξυπηρετητές (πολλές αιτήσεις εξυπηρετούνται από ένα νήμα).
    - Με χρήση non-blocking E/E και έλεγχο ετοιμότητας (level-triggered).
    - Με χρήση non-blocking E/E και έλεγχο αλλαγών (edge-triggered).
    - Με ασύγχρονη E/E και ειδοποιήσεις ολοκλήρωσης.
  - Προεκχώρηση διεργασιών (μία διεργασία ανά πελάτη).
    - Χωρίς συγχρονισμό για κλήση της `accept(2)`.
    - Με χρήση συγχρονισμού για κλήση της `accept(2)`.
    - Με μεταφορά των file descriptors ανάμεσα στις διεργασίες.
  - Προεκχώρηση νημάτων (ένα νήμα ανά πελάτη).
    - Με κλήση της `accept(2)` σε κάθε νήμα.
    - Με ένα κεντρικό νήμα να καλεί `accept(2)`.
  - Υβριδικές λύσεις με δυναμική εναλλαγή ανάμεσα σε επαναληπτική, ταυτόχρονη και προεκχωρημένη εξυπηρέτηση.

Ταυτόχρονοι εξυπηρετητές

Ταυτόχρονοι εξυπηρετητές

Μία διεργασία ανά αίτηση: Συνδεσμοστραφής επικοινωνία

- Βασική ιδέα:
  - Υπάρχει μία βασική διεργασία η οποία καλεί `accept(2)` σε έναν ατέρμονο βρόχο.
  - Για κάθε νέο πελάτη, η βασική διεργασία δημιουργεί μία νέα με `fork(2)`.
  - Η αρχική διεργασία κλείνει τον descriptor που επέστρεψε η `accept(2)`. Κατόπιν συνεχίζει στο βρόχο της.
  - Η νέα διεργασία κλείνει τον descriptor στον οποίο έρχονται νέες αιτήσεις, εξυπηρετεί τον νέο πελάτη μέσω του descriptor που επέστρεψε η `accept(2)` και τερματίζει.

```
 1  int sd, cd, pid;
 2  sd = socket(/* ... */);
 3  bind(sd, /* ... */);
 4  listen(sd, /* ... */);
 5  while (1) {
 6      cd = accept(sd, /* ... */);
 7      if ((pid = fork())) { // Father
 8          close(cd);  // Close client sd
 9          /* ... */   // Housekeeping
10      } else {        // Child
11          close(sd);  // Close listening sd
12          /* ... */   // Process request
13          close(cd);  // Close client sd
14          return 0;   // Terminate
15      }
16  }
```

Ταυτόχρονοι εξυπηρετητές

Μία διεργασία ανά αίτηση: Ασυνδεσμική επικοινωνία

- Βασική ιδέα:
  - Υπάρχει μία βασική διεργασία η οποία διαβάζει τις αιτήσεις των πελατών σε έναν ατέρμονο βρόχο.
  - Για κάθε νέο πελάτη, η βασική διεργασία δημιουργεί μία νέα με `fork(2)`.
  - Η αρχική διεργασία συνεχίζει στο βρόχο της.
  - Η νέα διεργασία κλείνει τον descriptor στον οποίο έρχονται νέες αιτήσεις, επεξεργάζεται την αίτηση του πελάτη και τερματίζει. Αν πρέπει να στείλει δεδομένα στον πελάτη δημιουργεί ένα νέο socket.

```
1  int sd, pid;
2  char buf[DATAGRAMSIZE];
3  sd = socket(/* ... */);
4  bind(sd, /* ... */);
5  while (1) {
6      recvfrom(sd, buf, /* ... */);
7      if (!(pid = fork())) { // Child
8          close(sd);
9          sd = socket(/* ... */);
10         /* ... */
11         sendto(sd, /* ... */);
12         close(sd);
13         return 0;
14     }
15 }
```

Ταυτόχρονοι εξυπηρετητές

Μία διεργασία ανά αίτηση

- Πλεονεκτήματα:
  - \+ Εύκολη υλοποίηση.
  - \+ (Ψευδο-)παράλληλη εξυπηρέτηση πελατών.
  - \+ Διαχωρισμός πελατών.
- Μειονεκτήματα:
  - – Η `fork(2)` είναι χρονοβόρα.
  - – Άνω όριο στο πλήθος διεργασιών-παιδιών.
  - – Κακή απόδοση όσο αυξάνονται οι διεργασίες λόγω context switching.
  - – Απαιτεί έλεγχο κατάστασης (`SIGCHLD`) για τις διεργασίες–παιδιά.

Ταυτόχρονοι εξυπηρετητές

Ένα νήμα ανά αίτηση

- Η ίδια βασική ιδέα όπως και με μία διεργασία ανά αίτηση.
  - Υπάρχει μία βασική διεργασία/νήμα που καλεί `accept(2)` ή διαβάζει τα δεδομένα των πελατών.
  - Για κάθε νέο πελάτη, το βασικό νήμα δημιουργεί ένα νέο νήμα το οποίο θα αναλάβει την εξυπηρέτηση της αίτησης.
  - **Προσοχή: Όλα τα νήματα μοιράζονται τον ίδιο file descriptor table και τις ίδιες καθολικές μεταβλητές!**
    - Το αρχικό νήμα δεν κλείνει το socket που επέστρεψε η `accept(2)`.
    - Το νέο νήμα δεν κλείνει το socket στο οποίο περιμένει νέους πελάτες το βασικό νήμα.
    - Απαιτείται συγχρονισμός για πρόσβαση σε κοινούς πόρους.

```
1  void* my_thr(void *arg) {
2      int cd = *((int*)arg);
3      /* ... */ // Process request
4      close(cd);
5      return NULL;
6  }
7  int main() {
8      int sd, cd, pid;
9      sd = socket(/* ... */);
10     bind(sd, /* ... */);
11     listen(sd, /* ... */);
12     while (1) {
13         cd = accept(sd, /* ... */);
14         pthread_create(/* ... */, my_thr,
                   (void *)&cd);
15     }
16 }
```

Ταυτόχρονοι εξυπηρετητές

Ένα νήμα ανά αίτηση

- Πλεονεκτήματα:
  - \+ Εύκολη υλοποίηση.
  - \+ (Ψευδο-)παράλληλη εξυπηρέτηση πελατών.
  - \+ Διαχωρισμός πελατών.
  - \+ Δεν έχει το κόστος της `fork(2)`.
  - \+ Πολλές φορές αποδοτικότερη απο την πιο αποδοτική αρχιτεκτονική προεκχώρησης διεργασιών.
- Μειονεκτήματα:
  - – Άνω όριο στο πλήθος νημάτων.
  - – Κακή απόδοση όσο αυξάνονται τα νήματα λόγω context switching.
  - – Απαιτεί έλεγχο κατάστασης νημάτων.
  - – Απαιτεί συγχρονισμό για πρόσβαση σε κοινές μεταβλητές ή κοινούς πόρους και πρόβλεψη αποφυγής λιμοκτονίας και αδιεξόδων.
  - – Πολλά συστήματα δεν έχουν πλήρη ή καλή υποστήριξη για νήματα.

Προχωρημένοι επαναληπτικοί εξυπηρετητές

Προχωρημένοι επαναληπτικοί εξυπηρετητές

Πολυπλεξία E/E

- Βασική ιδέα:
  - Υπάρχει μία διεργασία/ένα νήμα εκτέλεσης.
  - Κάθε E/E είναι ασύγχρονη/non-blocking.
  - Η διεργασία πολυπλέκει τις προσβάσεις E/E και την εξυπηρέτηση πελατών.
    - Με χρήση non-blocking E/E και έλεγχο ετοιμότητας (level-triggered): `select(2)`, `pselect(2)`, `poll(2)`, `/dev/poll`, `epoll(4)`, `kqueue(2)`.
    - Με χρήση non-blocking E/E και έλεγχο αλλαγών (edge-triggered): `epoll(4)`, `kqueue(2)`, `SIGIO`.
    - Με ασύγχρονη E/E και ειδοποιήσεις ολοκλήρωσης: `epoll(4)`, `kqueue(2)`, Solaris event ports.
- Πλεονεκτήματα:
  - \+ (Ψευδο-)παράλληλη εξυπηρέτηση πελατών.
  - \+ Δεν έχει το κόστος της `fork(2)` ή του context switching.
  - \+ Δεν απαιτεί συγχρονισμό ή έλεγχο κατάστασης νημάτων/διεργασιών–παιδιών.
  - \+ Ευκολότερη αποσφαλμάτωση.
- Μειονεκτήματα:
  - – Δεν ενδείκνυται αν η εξυπηρέτηση αιτήσεων είναι χρονοβόρα.
  - – Αυξημένη πολυπλοκότητα σχεδίασης για παραπάνω από βασικές περιπτώσεις.
  - – Εκμεταλλεύεται μόνο έναν επεξεργαστικό πυρήνα.
  - – Μη πλήρης ή καλή υποστήριξη ασύγχρονης/non-blocking E/E από κάποια συστήματα.

Ταυτόχρονοι εξυπηρετητές με προεκχώρηση διεργασιών/νημάτων

Ταυτόχρονοι εξυπηρετητές με προεκχώρηση διεργασιών

- Βασική ιδέα:
  - Μία βασική διεργασία δημιουργεί αρχικά έναν αριθμό από νέες διεργασίες.
  - Σε κάθε νεό πελάτη ανατίθεται και μία τέτοια διεργασία–παιδί.
- Πλεονεκτήματα:
  - \+ (Ψευδο-)παράλληλη εξυπηρέτηση πελατών.
  - \+ Γλιτώνει το κόστος της `fork(2)` πέρα από το αρχικό στάδιο.
- Μειονεκτήματα:
  - – Η αρχική διεργασία πρέπει να επιλέξει εξαρχής έναν αριθμό διεργασιών που θα δημιουργήσει.
  - – Διαφορετικά η αρχική διεργασία πρέπει να παρακολουθεί το φόρτο των διεργασιών–παιδιών και να δημιουργεί νέες ή να τερματίζει υπάρχουσες αν υπερφορτώνονται ή υποχρησιμοποιούνται αντίστοιχα.
- Διακρίνονται οι εξής περιπτώσεις:
  - Χωρίς συγχρονισμό για κλήση της `accept(2)`.
  - Με συγχρονισμό για κλήση της `accept(2)`.
  - Με μεταφορά των file descriptors ανάμεσα στις διεργασίες.

Ταυτόχρονοι εξυπηρετητές με προεκχώρηση διεργασιών

Χωρίς συγχρονισμό για κλήση της `accept(2)`

- Βασική ιδέα.
  - Η βασική διεργασία δημιουργεί ένα socket στο οποίο έρχονται νέες αιτήσεις, καθώς και έναν αριθμό από διεργασίες–παιδιά. Κατόπιν απλά περιμένει.
  - Κάθε διεργασία–παιδί καλεί `accept(2)` στο socket που δημιούργησε η αρχική διεργασία.
  - Αν έρθει νέος πελάτης στο socket αυτό, όλες οι διεργασίες θα «ξυπνήσουν», όμως μόνο μία θα συνεχίσει την εκτέλεση ενώ οι υπόλοιπες θα μπλοκάρουν και πάλι.

```
 1  int sd, cd, i, pid[NUMCHILDREN];
 2  sd = socket(/* ... */);
 3  bind(sd, /* ... */);
 4  listen(sd, /* ... */);
 5  for (i = 0; i < NUMCHILDREN; i++) {
 6      if (!(pid[i] = fork())) { // Child
 7          while (1) {     // Forever...
 8              cd = accept(sd, /* ... */);
 9              /* ... */   // Process request
10              close(cd);  // Close client sd
11          }
12      }
13  }
14  /* Wait doing nothing... */
```

Ταυτόχρονοι εξυπηρετητές με προεκχώρηση διεργασιών

Χωρίς συγχρονισμό για κλήση της `accept(2)`

- Πλεονεκτήματα:
  - \+ Απλούστερη υλοποίηση εξυπηρετητή με προεκχώρηση διεργασιών.
- Μειονεκτήματα:
  - – Πρόβλημα «thundering herd»: για κάθε εισερχόμενη αίτηση ο πυρήνας του λειτουργικού πρέπει να ξεμπλοκάρει όλες τις διεργασίες–παιδιά.
  - – Προσοχή: αν καλέσουμε `select(2)` για το αρχικό socket δημιουργούνται συγκρούσεις!
  - – Δεν υποστηρίζεται από όλα τα συστήματα.
- Λύση: Συγχρονισμός για κλήση της `accept(2)`.
  - Μπορεί να γίνει είτε με βάση κλασσικές τεχνικές (π.χ. file locking) είτε με σημαφόρους (απλούς ή νηματικούς).

Ταυτόχρονοι εξυπηρετητές με προεκχώρηση διεργασιών

Με μεταφορά των file descriptors ανάμεσα στις διεργασίες

- Βασική ιδέα:
  - Η βασική διεργασία δημιουργεί ένα socket στο οποίο έρχονται νέες αιτήσεις, καθώς και έναν αριθμό από διεργασίες–παιδιά και pipes ή socketpairs για να μπορεί να επικοινωνεί μαζί τους.
  - Η βασική διεργασία καλεί `accept(2)` και κατόπιν μεταφέρει τον επιστρεφόμενο file descriptor σε κάποια διεργασία–παιδί.
- Πλεονεκτήματα:
  - \+ Δεν απαιτεί συγχρονισμό για κλήση της `accept(2)`.
  - \+ Δεν υποφέρει από το πρόβλημα thundering herd.
- Μειονεκτήματα:
  - – Απαιτεί η αρχική διεργασία να διατηρεί πληροφορίες κατάστασεις για τις διεργασίες–παιδιά (π.χ. ποιες εξυπηρετούν πελάτες, ποιες είναι ανενεργές, κτλ.)
  - – Απαιτεί μία μέθοδο για αποστολή file descriptors στις διεργασίες–παιδιά.

Ταυτόχρονοι εξυπηρετητές με προεκχώρηση νημάτων

- Η ίδια σχεδιαστική αρχή προεκχώρησης διεργασιών επεξεργασίας πελατών μπορεί να εφαρμοστεί και για νήματα.
  - Με κλήση της `accept(2)` σε κάθε νήμα
    - Ίδιες σχεδιαστικές αρχές με τις αρχιτεκτονικές ταυτόχρονης εξυπηρέτησης με προεκχώρηση διεργασιών και συγχρονισμό στην κλήση της `accept(2)`, με χρήση νηματικών σημαφόρων.
  - Με ένα κεντρικό νήμα να καλεί την `accept(2)`.
    - Ίδιες σχεδιαστικές αρχές με τις αρχιτεκτονικές ταυτόχρονης εξυπηρέτησης με προεκχώρηση διεργασιών και μεταφορά των descriptors.
    - Καθώς τα νήματα μοιράζονται τον χώρο μνήμης και τον πίνακα των file descriptors της διεργασίας, δεν απαιτείται pipe ή socketpair για μεταφορά των descriptors.
- Πλεονεκτήματα:
  - \+ Σαφώς πιο γρήγορες και πιο απλές στην υλοποίηση από τις αρχιτεκτονικές πολλαπλών διεργασιών.
- Μειονεκτήματα:
  - − Απαιτεί συγχρονισμό για πρόσβαση σε κοινές μεταβλητές ή κοινούς πόρους και πρόβλεψη αποφυγής λιμοκτονίας και αδιεξόδων.
  - − Όχι πάντα κατάλληλη, ανάλογα με το μοντέλο εξυπηρέτησης.

Υβριδικές αρχιτεκτονικές

Υβριδικές αρχιτεκτονικές

- Μπορούμε να συνδυάσουμε τις προηγούμενες αρχιτεκτονικές για να καταπολεμήσουμε τα αρνητικά τους;
  - Επαναληπτικοί εξυπηρετητές ⟶ γρηγορότεροι για μονοεπεξεργαστικά συστήματα.
  - Ταυτόχρονοι εξυπηρετητές ⟶ κλιμακώσιμοι σε πολλούς επεξεργαστικούς πυρήνες.
- Και βέβαια! Με υβριδικές αρχιτεκτονικές.
  - Δυναμική εναλλαγή ανάμεσα σε διάφορες σχεδιάσεις
  - Ταυτόχρονη χρήση γεγονότων και διεργασιών/νημάτων
- Παράγοντες που επηρεάζουν την επιλογή:
  - Χρόνος επεξεργασίας αιτημάτων.
  - Χρόνος Ε/Ε (από το δίκτυο, από το δίσκο, κτλ)
  - Κόστος του context switching.
  - Όρια που επιβάλλονται από το λογισμικό ή το υλικό.
  - Γενικότερος υπολογιστικός φόρτος του εξυπηρετητή.
- Παραδείγματα υβριδικής σχεδίασης:
  - Έστω κλασσικός επαναληπτικός εξυπηρετητής με πολυπλεξία Ε/Ε. Για κάθε αίτημα πελάτη, θέτουμε ένα χρονικό όριο. Αν η επεξεργασία υπερβεί το όριο αυτό, δημιουργούμε μία διεργασία ή ένα νήμα να συνεχίσει την εξυπηρέτηση.
  - Έστω κλασσικός ταυτόχρονος εξυπηρετητής με προεκχώρηση διεργασιών/νημάτων. Διαμοιράζουμε το φόρτο στις διεργασίες/νήματα, καθεμιά από τις οποίες εκτελεί επαναληπτικό αλγόριθμο με πολυπλεξία Ε/Ε.

Εξυπηρέτηση πολλαπλών πρωτοκόλλων

Εξυπηρέτηση πολλαπλών πρωτοκόλλων

- Είδαμε ότι μπορούμε να εξυπηρετούμε πολλαπλούς πελάτες ταυτόχρονα.
- **Ερώτηση:** Μπορούμε να χρησιμοποιούμε πολλαπλά πρωτόκολλα (π.χ. TCP, UDP, SCTP, . . .) στον ίδιο εξυπηρετητή;
- **Απάντηση:** Φυσικά! Με πολλαπλά sockets να δέχονται εισερχόμενες αιτήσεις και πολυπλεξία τους με χρήση των `select(2)`,`poll(2)`,...

```
1   int tcp_sd, udp_sd, sel;
2   fd_set fds;
3   tcp_sd = socket(AF_INET, SOCK_STREAM, 0);
4   bind(tcp_sd, /* ... */);
5   listen(tcp_sd, /* ... */);
6   udp_sd = socket(AF_INET, SOCK_DGREAM, 0);
7   bind(udp_sd, /* ... */);
8   while (1) {
9       FD_ZERO(&fds);
10      FD_SET(tcp_sd, &fds);
11      FD_SET(udp_sd, &fds);
```

```
12      sel = select(max(tcp_sd, udp_sd) + 1,  &
                fds, NULL, NULL, NULL);
13      if (sel > 0) {
14          if (FD_ISSET(tcp_sd, &fds)) {
15              cd = accept(tcp_sd, /* ... */);
16              /* ... */
17          } else if (FD_ISSET(udp_sd, &fds)) {
18              recvfrom(udp_sd, buf, /* ... */);
19              /* ... */
20          }
21      }
22  }
```

Εξυπηρέτηση πολλαπλών υπηρεσιών

Εξυπηρέτηση πολλαπλών υπηρεσιών

- **Ερώτηση:** Μπορούμε να εξυπηρετούμε πολλαπλές υπηρεσίες (π.χ. HTTP, SMTP, . . .) με τον ίδιο εξυπηρετητή;
- **Απάντηση:** Φυσικά! Με πολλαπλά sockets να δέχονται εισερχόμενες αιτήσεις και πολυπλεξία τους με χρήση των `select(2)`, `poll(2)`, ...

```
1   int http_sd, smtp_sd, sel;
2   fd_set fds;
3   http_sd = socket(AF_INET, SOCK_STREAM, 0);
4   bind(http_sd, /* ... */);
5   listen(http_sd, /* ... */);
6   smtp_sd = socket(AF_INET, SOCK_STREAM, 0);
7   bind(smtp_sd, /* ... */);
8   listen(smtp_sd, /* ... */);
9   while (1) {
10      FD_ZERO(&fds);
11      FD_SET(http_sd, &fds);
12      FD_SET(smtp_sd, &fds);
13      sel = select(max(http_sd, smtp_sd) + 1,  &
                fds, NULL, NULL, NULL);
14      if (sel > 0) {
15          if (FD_ISSET(http_sd, &fds)) {
16              cd = accept(http_sd, /* ... */);
17              /* ... */
18          } else if (FD_ISSET(smtp_sd, &fds)) {
19              cd = accept(smtp_sd, /* ... */);
20              /* ... */
21          }
22      }
23  }
```

Σχεδίαση αποδοτικών εξυπηρετητών

Εισαγωγή

- Κύριοι παράγοντες που επηρεάζουν την απόδοση ενός εξυπηρετητή:
  - Context switching.
  - Καθυστέρηση στην Ε/Ε.
  - Αντιγραφή δεδομένων από/προς τον πυρήνα/συναρτήσεις/διεργασίες/νήματα.
  - Συμφόρηση συγχρονισμού.
  - Δέσμευση/απελευθέρωση μνήμης.

Σχεδίαση αποδοτικών εξυπηρετητών

Context switching

- Ο #1 λόγος κακής απόδοσης δικτυακών εξυπηρετητών και ο #1 στόχος επιθέσεων τύπου Denial of Service (DoS).
  - Πρόβλημα: το σύστημα σπαταλά περισσότερο χρόνο στην εναλλαγή διεργασιών/νημάτων απ'ότι στην εκτέλεση του κώδικά τους.
  - Ακραία περίπτωση: το σύστημα δε μπορεί να δημιουργήσει άλλες διεργασίες/νήματα, με αποτέλεσμα διάφορες υπηρεσίες να καταρρέουν!
  - Ιδανική περίπτωση: τόσες διεργασίες/νήματα όσοι και οι επεξεργαστικοί πυρήνες.

Καθυστέρηση Ε/Ε

- Γνωστό πρόβλημα απόδοσης.
- Προκύπτει τόσο στο δίκτυο (αργές συνδέσεις, απώλεια πακέτων, αργοί πελάτες/εξυπηρετητές, . . .) όσο και σε τοπικό επίπεδο (Ε/Ε από/προς το δίσκο ή περιφερειακές συσκευές).
- Στις περισσότερες περιπτώσεις μπορεί να «κρυφτεί» με χρήση non-blocking ή ασύγχρονης Ε/Ε.

Σχεδίαση αποδοτικών εξυπηρετητών

Αντιγραφή δεδομένων

- Πολλές φορές δημιουργούμε αντίγραφα δεδομένων στη μνήμη.
  - Αντιγραφή δεδομένων στο stack σε κλήσεις συναρτήσεων.
  - Αντιγραφή τιμών επιστροφής συναρτήσεων για διατήρησή τους.
  - Αντιγραφή δεδομένων από/προς τον πυρήνα του λειτουργικού.
- Η αντιγραφή δεδομένων πρέπει να αποφεύγεται.
  - Προσθέτει καθυστερήσεις.
  - Σπαταλά πόρους συστήματος.
  - Μπορεί να δημιουργήσει προβλήματα αν τα αντίγραφα δεν διατηρούνται συνεπή.
- Η καλύτερη μέθοδος αποφυγής αντιγράφων είναι η χρήση «buffer descriptors». Για κάθε τμήμα μνήμης, διατηρούμε:
  - Δείκτη στο τμήμα μνήμης και το συνολικό μέγεθος του τμήματος.
  - Δείκτη προς και μήκος του κομματιού του τμήματος που χρησιμοποιείται.
  - Δείκτες προς επόμενους/προηγούμενους buffer descriptors.
  - Μετρητή διεργασιών/νημάτων/συναρτήσεων που τον χρησιμοποιούν.
- Αντί να χρησιμοποιούμε ως όρισμα ή τιμή επιστροφής σε συναρτήσεις απλά ένα δείκτη στο τμήμα μνήμης, χρησιμοποιούμε (αλυσίδες από) buffer descriptors.
  - Δείτε για παράδειγμα την `getaddrinfo(3)`.
- Καλύτερα αποτελέσματα για μεγάλα τμήματα μνήμης παρά για μικρά.

Σχεδίαση αποδοτικών εξυπηρετητών

Συμφόρηση συγχρονισμού

- Δημιουργείται όταν πολλά τμήματα κώδικα καταλήγουν να περιμένουν για κάποιο lock (π.χ. σημαφόρο, κλειδωμένο αρχείο κτλ.)
- Από τη μία έχουμε απλά/«χοντροκομμένα» σχήματα συγχρονισμού.
  - Για παράδειγμα, ένας σημαφόρος για τα πάντα.
  - Εύκολο στην υλοποίηση αλλά καθόλου αποδοτικό.
- Από την άλλη έχουμε «λεπτομερή» σχήματα συγχρονισμού.
  - Για παράδειγμα, ένας σημαφόρος για κάθε κοινό τμήμα κώδικα ή πόρο.
  - Θεωρητικά πιο αποδοτικό, ωστόσο προσθέτει μεγάλη πολυπλοκότητα στη σχεδίαση και αυξάνει τις απαιτήσεις σε χώρο μνήμης και υπολογιστικούς πόρους.
- Και στις δύο περιπτώσεις υπάρχει κίνδυνος για λιμοκτονία, αδιέξοδα, race conditions κτλ.
- Συνήθως ξεκινάμε με ένα απλό σχήμα και προχωρούμε προς ένα λεπτομερές, έως ότου η απόδοση είναι αποδεκτή.
- **Λάθος:** απαιτείται προσεκτική σχεδίαση εξαρχής, διαχωρισμός των προσβάσεων σε κοινούς πόρους/τμήματα κώδικα και αποφυγή συμφόρησης συγχρονισμού: αν τελείως διαφορετικά τμήματα κώδικα συναγωνίζονται για τους ίδιους πόρους, κάτι έχουμε κάνει λάθος. . .

Σχεδίαση αποδοτικών εξυπηρετητών

Δέσμευση/απελευθέρωση μνήμης

- Από τις πιο κοινές πράξεις σε πολλές εφαρμογές.
  - Προσφέρει δυναμικότητα στη διαχείριση της διαθέσιμης μνήμης.
  - Ωστόσο, οι αλγόριθμοι δέσμευσης/απελευθέρωσης μνήμης εισάγουν καθυστερήσεις (κυρίως λόγω συμφόρησης συγχρονισμού).
- Λύσεις:
  - Προεκχώρηση αρκετής μνήμης στην αρχή του προγράμματος/συνάρτησης για όλη την εκτέλεσή του και διαχείρισή της από το ίδιο το πρόγραμμα.
  - Χρήση λιστών για αντικείμενα που δεσμεύονται και απελευθερώνονται συχνά, με σκοπό την επαναχρησιμοποίησή τους.

Επιλογές υποδοχών

Επιλογές υποδοχών

- Το API των BSD sockets επιτρέπει τον έλεγχο ορισμένων ιδιοτήτων/επιλογών των sockets.
- Προβλέπει τρεις μεθόδους:
  - `getsockopt(2)` και `setsockopt(2)`.
  - `fcntl(2)`.
  - `ioctl(2)`.
- Οι επιλογές διακρίνονται ανάλογα με το «επίπεδο» στο οποίο αναφέρονται:
  1. Επιλογές επιπέδου υλοποίησης των sockets.
  2. Επιλογές πρωτοκόλλων επιπέδου μεταφοράς.
  3. Επιλογές πρωτοκόλλων επιπέδου διαδικτύου.

getsockopt(2)/setsockopt(2)

Επιλογές υποδοχών

getsockopt(2)/setsockopt(2)

```
int getsockopt(int s, int level, int optname, void* optval,
     socklen_t* optlen);
int setsockopt(int s, int level, int optname, const void* optval,
     socklen_t optlen);
```

- `s`: ένας socket descriptor που αντιστοιχεί σε ένα ανοιχτό socket.
- `level`: το επίπεδο στο οποίο αναφέρεται η επιλογή.
  - `SOL_SOCKET`: επιλογές επιπέδου υλοποίησης των sockets.
  - `IPPROTO_IP`: επιλογές επιπέδου πρωτοκόλλου IPv4.
  - `IPPROTO_IPV6`: επιλογές επιπέδου πρωτοκόλλου IPv6.
  - `IPPROTO_ICMPV6`: επιλογές επιπέδου πρωτοκόλλου ICMPv6.
  - `IPPROTO_TCP`: επιλογές επιπέδου πρωτοκόλλου TCP.
- `optval`: δείκτης σε χώρο μνήμης μεγέθους `(*)optlen` στον οποίο θα αποθηκευθεί ή από τον οποίο θα τεθεί η τιμή της αντίστοιχης ιδιότητας.
- Επιστρέφει 0 για επιτυχία, -1 για αποτυχία.
- Δείτε το man page για λεπτομέρειες.

Επιλογές υποδοχών

getsockopt(2)/setsockopt(2)

- Κυριότερες επιλογές επιπέδου υλοποίησης sockets:
  - `SO_BROADCAST`: αποστολή μηνυμάτων προς όλους τους κόμβους του τοπικού δικτύου. Ορίζεται μόνο για UDP sockets και μόνο αν το υποστηρίζει το εκάστοτε δίκτυο.
  - `SO_DEBUG`: καταγράφει πληροφορίες για όλα τα πακέτα που στέλνονται ή λαμβάνονται από το socket. Ορίζεται μόνο για TCP sockets.
  - `SO_ERROR`: ανάκτηση λαθών στην επικοινωνία του socket.
  - `SO_KEEPALIVE`: διατήρηση σύνδεσης μέσω της περιοδικής ανταλλαγής πακέτων ειδικού τύπου. Ορίζεται μόνο για TCP sockets.
  - `SO_LINGER`: καθορίζει αν η `close(2)` θα επιστρέφει αμέσως ή αν και για πόσο θα περιμένει να σταλούν όλα τα δεδομένα που βρίσκονται στους buffers του λειτουργικού συστήματος και να ληφθεί τα `FIN` της άλλης πλευράς.
  - `SO_RCVBUF`/`SO_SNDBUF`: καθορίζουν το μέγεθος του buffer προσωρινής αποθήκευσης εισερχόμενων/εξερχόμενων δεδομένων.
  - `SO_RCVTIMEO`/`SO_SNDTIMEO`: καθορίζουν χρονικά όρια για την ολοκλήρωση των κλήσεων συστήματος για την λήψη/αποστολή δεδομένων.
  - `SO_REUSEADDR`: καθορίζει αν επιτρέπεται η επαναχρησιμοποίηση της διεύθυνσης και της θύρας του socket από άλλα sockets της ίδιας ή άλλης διεργασίας. Πρέπει να τεθεί πριν κληθεί η `bind(2)`.
  - `SO_TYPE`: επιτρέπει την ανάκτηση του τύπου του socket (π.χ. `SOCK_STREAM` ή `SOCK_DGRAM`.

Επιλογές υποδοχών

getsockopt(2)/setsockopt(2)

- Οι επιλογές `SO_RCVTIMEO` και `SO_SNDTIMEO` προσφέρουν δυνατότητα για απευθείας Ε/Ε με χρονικό περιορισμό. Χωρίς αυτές, θα πρέπει είτε να χρησιμοποιήσουμε μία εκ των `select(2)`/`poll(2)`/κτλ. ή το σήμα `SIGALARM` μέσω της κλήσης `alarm(3)`.
- Κυριότερες επιλογές επιπέδου πρωτοκόλλου TCP:
  - `TCP_KEEPALIVE`: καθορισμός του χρονικού ορίου πέραν του οποίου θα αρχίσουν να στέλνονται πακέτα ειδικού τύπου για την διατήρηση της σύνδεσης. Προϋποθέτει ότι έχει ενεργοποιηθεί και η επιλογή `SO_KEEPALIVE`.
  - `TCP_MAXRT`: καθορίζει το χρονικό όριο πέραν του οποίου διακόπτεται η σύνδεση αν το TCP αναγκαστεί να ξαναμεταδώσει κάποια πακέτα.
  - `TCP_MAXSEG`: καθορισμός του MSS της σύνδεσης.
  - `TCP_NODELAY`: ενεργοποίηση/απενεργοποίηση του αλγορίθμου Nagle.

fcntl(2)/ioctl(2)

Επιλογές υποδοχών

fcntl(2)/ioctl(2)

```
int fcntl(int fd, int cmd, ... /* arg */);
int ioctl(int d, unsigned long request, ... /* arg */);
```

- `fd`/`d`: ένας socket descriptor που αντιστοιχεί σε ένα ανοιχτό socket.
- `cmd`/`request`: καθορίζει την λειτουργία που θα εκτελεστεί.
  - `F_GETFL`: ανακτά και επιστρέφει τις επιλογές (flags) του `fd`.
  - `F_SETFL`: θέτει τις επιλογές του `fd` στην τιμή του `arg`.
  - Κυριότερες τιμές των flags:
    - `O_NONBLOCK`: ενεργοποίηση non-blocking E/E για τον descriptor `fd`.
      Με `ioctl(2)`: `FIONBIO`.
    - `O_ASYNC`: ενεργοποίηση ασύγχρονης E/E για τον descriptor `fd` με ειδοποίηση ολοκλήρης μέσω του σήματος `SIGIO`.
      Με `ioctl(2)`: `FIOASYNC`.
  - `F_GETOWN`/`F_SETOWN`: ανακτά/θέτει το αναγνωριστικό της διεργασίας στην οποία παραδίδονται σήματα (`SIGIO`, `SIGURG`) που αφορούν την κατάσταση του `fd`.
    Με `ioctl(2)`: `SIOCGPGRP`/`SIOCSPGRP` ή `FIOGETOWN`/`FIOSETOWN`.
  - `F_GETSIG`/`F_SETSIG`: ανακτά/θέτει τον αριθμό σήματος που θα παραδίδεται στην διεργασία ως αποτέλεσμα αλλαγών ετοιμότητας E/E του `fd`. Μόνο σε Linux.
  - `FIONREAD`: Με `ioctl(2)`, ανακτά το πλήθος των bytes στο buffer εισόδου του socket.

Προχωρημένες τεχνικές Ε/Ε

Ε/Ε διασποράς/συγκέντρωσης

Προχωρημένες τεχνικές E/E

E/E διασποράς/συγκέντρωσης (scatter-gather I/O)

```
ssize_t readv(int d, const struct iovec *iov, int iovcnt);
ssize_t writev(int d, const struct iovec *iov, int iovcnt);
struct iovec {
   void*  iov_base;
   size_t iov_len;
};
```

- Οι συναρτήσεις αυτές επιτρέπουν την E/E από/προς πολλαπλά σημεία μνήμης παράλληλα.
- Δέχονται ως όρισμα έναν file descriptor `d` και έναν πίνακα `iov`, `iovcnt` στοιχείων (μέγιστο: `IOV_MAX`), από δομές τύπου `struct iovec`.
- Κάθε δομή `struct iovec` περιέχει έναν δείκτη `iov_base` σε μία θέση μνήμης μεγέθους `iov_len`.
- Η `readv(2)` διαβάζει δεδομένα από τον `d` και τα αποθηκεύει στις θέσεις που υποδεικνύουν οι δείκτες `iov_base`. Η εγγραφή στις δομές του πίνακα γίνεται εν σειρά: για να προχωρήσει στη δομή στη θέση `i` του πίνακα, θα πρέπει πρώτα να έχουν συμπληρώθεί και οι `i` προηγούμενες δομές.
- Αντίστοιχα η `writev(2)` γράφει δεδομένα στον `d` τα οποία διαβάζει από τις θέσους που υποδεικνύουν οι δείτκες `iov_base`. Και πάλι η επεξεργασία τους γίνεται εν σειρά.
- Τιμή επιστροφής όπως και στις `read(2)`/`write(2)`.

sendmsg(2)/recvmsg(2)

Προχωρημένες τεχνικές E/E

sendmsg(2)/recvmsg(2)

```
ssize_t sendmsg(int s, const struct msghdr *msg, int flags);
ssize_t recvmsg(int s, struct msghdr *msg, int flags);
```

- Οι γενικότερες των συναρτήσεων E/E που έχουμε δει ως τώρα.
  - Κάθε κλήση στις υπόλοιπες θα μπορούσε να αντικατασταθεί από μια κλήση μιας εκ των παραπάνω δυο, με κατάλληλα ορίσματα.
  - `s`: ο socket descriptor.
  - `msg`: δείκτης σε δομή τύπου `msghdr`.
  - `flags`: επιλογές επικοινωνίας, όπως και στις `send(2)/sendto(2)` και `recv(2)/recvfrom(2)`.
  - Επιτρέπουν ταυτόχρονα μεταφορά κανονικών δεδομένων, δεδομένων σηματοδοσίας και έλεγχο E/E.

Προχωρημένες τεχνικές E/E

sendmsg(2)/recvmsg(2)

```
struct msghdr {
    void*         msg_name;       // protocol address (struct sockaddr*)
    socklen_t     msg_namelen;    // size of msg_name

    struct iovec* msg_iov;        // scatter/gather array
    int           msg_iovlen;     // #elements in msg_iov

    void*         msg_control;    // ancillary data (struct cmsghdr*)
    socklen_t     msg_controllen; // size of msg_control

    int           msg_flags;      // MSG_EOR, MSG_TRUNC,
                                  // MSG_OOB, MSG_CTRUNC
};

struct cmsghdr {
    socklen_t  cmsg_len;          // size of cmsghdr, including header
    int        cmsg_level;        // originating protocol
    int        cmsg_type;         // protocol-specific type
    /* followed by
    u_char     cmsg_data[];       // ancillary data
    */
};
```

Προχωρημένες τεχνικές E/E

sendmsg(2)

```
1  ssize_t write(int d, const void* buf,
       size_t nbytes) {
2      struct msghdr msg;
3      struct iovec iov = {buf, nbytes};
4      memset(&msg, 0, sizeof(msg));
5      msg.msg_iov = &iov;
6      msg.msg_iovlen = 1;
7      return sendmsg(d, &msg, 0);
8  }
9  ssize_t writev(int d, const struct iovec
       * iov, int iovcnt) {
10     struct msghdr msg;
11     memset(&msg, 0, sizeof(msg));
12     msg.msg_iov = iov;
13     msg.msg_iovlen = iovcnt;
14     return sendmsg(d, &msg, 0);
15 }
```

```
16 ssize_t send(int s, const void* buf,
       size_t len, int flags) {
17     struct msghdr msg;
18     struct iovec iov = { buf, len };
19     memset(&msg, 0, sizeof(msg));
20     msg.msg_iov = &iov;
21     msg.msg_iovlen = 1;
22     return sendmsg(s, &msg, flags);
23 }
24 ssize_t sendto(int s, const void* buf,
       size_t len, int flags, const struct
       sockaddr* to, socklen_t tolen) {
25     struct msghdr msg;
26     struct iovec iov = { buf, len };
27     memset(&msg, 0, sizeof(msg));
28     msg.msg_name = to;
29     msg.msg_namelen = tolen;
30     msg.msg_iov = &iov;
31     msg.msg_iovlen = 1;
32     return sendmsg(s, &msg, flags);
33 }
```

Προχωρημένες τεχνικές E/E

recvmsg(2)

```
1  ssize_t read(int d, void* buf, size_t
       nbytes) {
2      struct msghdr msg;
3      struct iovec iov = {buf, nbytes};
4      memset(&msg, 0, sizeof(msg));
5      msg.msg_iov = &iov;
6      msg.msg_iovlen = 1;
7      return recvmsg(d, &msg, 0);
8  }
9  ssize_t readv(int d, const struct iovec*
       iov, int iovcnt) {
10     struct msghdr msg;
11     memset(&msg, 0, sizeof(msg));
12     msg.msg_iov = iov;
13     msg.msg_iovlen = iovcnt;
14     return recvmsg(d, &msg, 0);
15 }
```

```
16 ssize_t recv(int s, void* buf, size_t
       len, int flags) {
17     struct msghdr msg;
18     struct iovec iov = { buf, len };
19     memset(&msg, 0, sizeof(msg));
20     msg.msg_iov = &iov;
21     msg.msg_iovlen = 1;
22     return recvmsg(s, &msg, flags);
23 }
24 ssize_t recvfrom(int s, void* buf,
       size_t len, int flags, struct
       sockaddr* from, socklen_t* fromlen)
        {
25     ssize_t ret;
26     struct msghdr msg;
27     struct iovec iov = { buf, len };
28     memset(&msg, 0, sizeof(msg));
29     msg.msg_name = from;
30     msg.msg_namelen = *fromlen;
31     msg.msg_iov = &iov;
32     msg.msg_iovlen = 1;
33     ret = recvmsg(sd, &msg, flags);
34     *fromlen = msg.msg_namelen;
35     return ret;
36 }
```

Προχωρημένες τεχνικές Ε/Ε

sendmsg(2)/recvmsg(2)

- Το πεδίο `msg_control` μπορεί να χρησιμοποιηθεί για μεταφορά δεδομένων ελέγχου.
- Για την διαχείρισή του ορίζονται οι μακροεντολές:
  - `struct cmsghdr* CMSG_FIRSTHDR(struct msghdr* mptr)`: δεδομένου ενός δείκτη σε δομή `struct msghdr`, επιστρέφει δείκτη στην πρώτη δομή `struct cmsghdr` ή `NULL` αν δεν υπάρχουν δεδομένα ελέγχου.
  - `struct cmsghdr* CMSG_NXTHDR(struct msghdr* mptr, struct cmsghdr* cmptr)`: δεδομένου ενός δείκτη σε δομή `struct msghdr` και ενός δείκτη σε δομή `struct cmsghdr`, επιστρέφει δείκτη στην επόμενη δομή `struct cmsghdr` ή `NULL` αν δεν υπάρχουν άλλα δεδομένα ελέγχου.
  - `unsigned char* CMSG_DATA(struct cmsghdr *cmptr)`: επιστρέφει δείκτη στο πρώτο byte των δεδομένων της δομής `struct cmsghdr` στην οποία δείχνει ο `cmptr`.
  - `size_t CMSG_LEN(size_t length)`: υπολογίζει την τιμή που θα βάλουμε στο `cmsg_len`, αν θεωρήσουμε ότι θα έχουμε `length` bytes δεδομένων ελέγχου.
  - `size_t CMSG_SPACE(size_t length)`: υπολογίζει το συνολικό μέγεθος του πίνακα δεδομένων ελέγχου, δεδομένου ότι θα αποθηκεύσουμε σε αυτόν `length` bytes δεδομένων.

Προχωρημένες τεχνικές E/E

Παράδειγμα sendmsg(2): Αποστολή file descriptor

```
1  int sendDescriptor(int dest, int fd) {
2      struct msghdr msg;
3      char c = '';
4      struct iovec iov = { &c, sizeof(char) };
5      union {
6          struct cmsghdr cmsg;
7          char control[CMSG_SPACE(sizeof(int))];
8      } ctrl_un;
9      struct cmsghdr *cm;
10
11     msg.msg_control = ctrl_un.control;
12     msg.msg_controllen = sizeof(ctrl_un.control);
13     cm = CMSG_FIRSTHDR(&msg);
14     cm->cmsg_len = CMSG_LEN(sizeof(int));
15     cm->cmsg_level = SOL_SOCKET;
16     cm->cmsg_type = SCM_RIGHTS;
17     *((int*)CMSG_DATA(cm)) = fd;
18
19     msg.msg_name = NULL;
20     msg.msg_namelen = 0;
21     msg.msg_flags = 0;
22     msg.msg_iov = &iov;
23     msg.msg_iovlen = 1;
24
25     return sendmsg(dest, &msg, 0) < 0 ? -1 : 0;
26 }
```

Παράδειγμα recvmsg(2): Παραλαβή file descriptor

```
1  void recvDescriptor(int src, int* fd) {
2      struct msghdr msg;
3      char c = '';
4      struct iovec iov = { &c, sizeof(char) };
5      union {
6          struct cmsghdr cmsg;
7          char control[CMSG_SPACE(sizeof(int))];
8      } ctrl_un;
9      struct cmsghdr *cm;
10
11     msg.msg_control = ctrl_un.control;
12     msg.msg_controllen = sizeof(ctrl_un.control);
13
14     msg.msg_name = NULL;
15     msg.msg_namelen = 0;
16     msg.msg_flags = 0;
17     msg.msg_iov = &iov;
18     msg.msg_iovlen = 1;
19
20     if (recvmsg(src, &msg, 0) > 0 &&
               (cm = CMSG_FIRSTHDR(&msg)) &&
               cm->cmsg_len == CMSG_LEN(sizeof(int)) &&
               cm->cmsg_level == SOL_SOCKET &&
               cm->cmsg_type == SCM_RIGHTS) {
21         *fd = *((int*)CMSG_DATA(cm));
22         return 0;
23     }
24     return -1;
25 }
```

Δεδομένα εκτός ζώνης

Δεδομένα εκτός ζώνης

- Το πρωτόκολλο TCP παρέχει υποστήριξη για μεταφορά επείγοντων δεδομένων (δεδομένα εκτός ζώνης – out-of-band data).
- Τα δεδομένα αυτά τυγχάνουν ειδικής μεταχείρισης.
  - Δεν περιμένουν σε ουρές.
  - Δεν περιορίζονται από μηχανισμούς ελέγχου ροής.
- Προβλέπεται η κλήση `sockatmark(3)` η οποία ελέγχει τον descriptor `s` για δεδομένα εκτός ζώνης. Επιστρέφει 1 αν υπάρχουν δεδομένα εκτός ζώνης, 0 αν δεν υπάρχουν, -1 για σφάλμα.
  - `int sockatmark(int s);`
- Το πρόγραμμά μας μπορεί κατόπιν να διαβάσει τα δεδομένα αυτά χρησιμοποιώντας μία εκ των `recv*(2)` με το κατάλληλο flag (`MSG_OOB`).

Στο επόμενο μάθημα...

- Διαχείριση λαθών.
- Πύλες και Σήραγγες.