Προγραμματισμός συστημάτων UNIX/POSIX

Φετινό θέμα:

Προγραμματιστικές τεχνικές που στοχεύουν σε επιδόσεις

Βελτιστοποιήσεις με στόχο τις επιδόσεις

- Σε αρκετές περιπτώσεις δεν αρκεί να φτιάξουμε ένα πρόγραμμα που δουλεύει μόνο **σωστά** αλλά θέλουμε να δουλεύει και **όσο το δυνατόν πιο αποδοτικά / γρήγορα.**
- Για να γίνει αυτό απαιτούνται πολλές και διαφορετικές *βελτιστοποιήσεις* (optimization) του προγράμματός μας ώστε ο κώδικάς μας να είναι αποδοτικότερος
- Σε πολλές περιπτώσεις, ο μεταφραστής είναι σε θέση να κάνει τις βελτιστοποιήσεις αυτές *αυτόματα, μόνος του*, χωρίς να αλλάξουμε τίποτε στο πρόγραμμά μας – αρκεί να του το πούμε.
  - Οι σύγχρονοι μεταφραστές είναι πάρα πολύ ικανοί σε αυτή τη δουλειά.

- Π.χ. ο GCC:
  - Κατά τη μετάφραση ενός προγράμματος μπορούμε να του δώσουμε ορίσματα τύπου "-O": -O1, -O2, -O3 (-O0 είναι το default)
  - Σε πολλές περιπτώσεις έχουμε εντυπωσιακά αποτελέσματα (παρότι αργεί λίγο παραπάνω για τη μετάφραση μεγάλων προγραμμάτων)
  - Να θυμόμαστε ότι σε μερικές περιπτώσεις οι αλλαγές που γίνονται στον κώδικα είναι πολύ δραστικές και ίσως σε ακραίες περιπτώσεις οι βελτιστοποιήσεις να αλλάζουν λίγο τη συμπεριφορά του προγράμματος (και να δυσκολεύεται το debugging) – θέλει προσοχή.
  - Τι σημαίνει το κάθε όρισμα (πόσο μάλλον ***πώς*** *το κάνει αυτό που κάνει*), δεν μας απασχολεί σε αυτό το μάθημα

- Εκτός από αυτό (ή μάλλον, σε συνδυασμό με αυτό), ενδιαφερόμαστε και για *προγραμματιστικές τεχνικές* μιας και καλύπτουν και περιπτώσεις που δεν μπορεί κανένας μεταφραστής να πιάσει.

Μερικές στοιχειώδεις βελτιστοποιήσεις I

- Αποφυγή περιττών υπολογισμών μέσα σε loops

```
for (i = 0; i < N; i++) {
   s[i] = (x+3*y*z)*s[i];
}
```

```
a = x+3*y*z;
for (i = 0; i < N; i++) {
   s[i] = a*s[i];
}
```

- Οι μεταφραστές από μόνοι τους, ανακαλύπτουν αρκετές εκφράσεις (όχι όλες) που δεν εξαρτώνται από την επανάληψη του βρόχου (*loop-invariant*) και κάνουν το παραπάνω αυτόματα
  - Καλό είναι να μην τους εμπιστευόμαστε 100%

Μερικές στοιχειώδεις βελτιστοποιήσεις II

❖ Μείωση περιττών κλήσεων σε συναρτήσεις

```
void capitalize(char *s) {
  int i;

  for (i = 0; i < strlen(s); i++)
     s[i] = toupper(s[i]);
}
```

❖ Για μήκος 1 MiB

- 25.5 sec

(Στο dektop μου)

```
void capitalize(char *s) {
  int i, len = strlen(s);

  for (i = 0; i < len; i++)
     s[i] = toupper(s[i]);
}
```

❖ Για μήκος 1 MiB

- 0.01 sec

Μεταβλητές “register” I

- Μία μεταβλητή στην C μπορεί να δηλωθεί ως “`register`”, π.χ.

```
register int x;
register float y;
```

- Ο χαρακτηρισμός “`register`” δηλώνει στον μεταφραστή ότι η μεταβλητή αυτή χρησιμοποιείται πολύ συχνά και καλό είναι να μην την τοποθετήσει κάπου στην μνήμη (αργή) αλλά *σε έναν καταχωρητή μέσα στην CPU.*
- Δύο σημαντικές παρατηρήσεις:
  1. Ο μεταφραστής *δεν είναι υποχρεωμένος να το σεβαστεί* (δηλαδή μπορεί να το αγνοήσει)
  2. Δεν μπορούμε να πάρουμε τη διεύθυνση μίας μεταβλητής `register` (το &x δεν δουλεύει)

Μεταβλητές “register” II

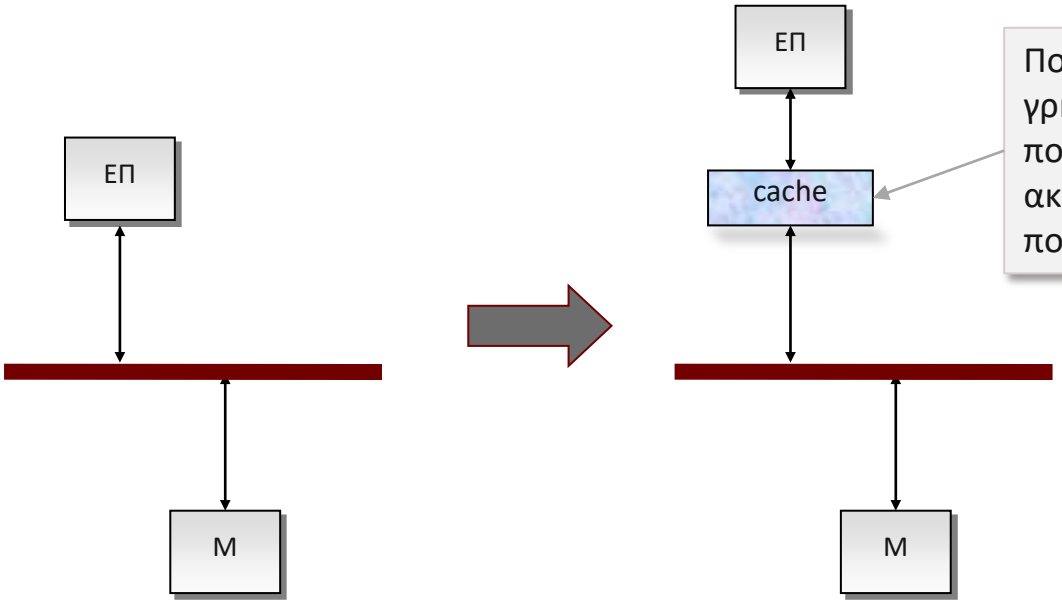
- Οι σύγχρονοι μεταφραστές κάνουν εξαιρετική διαχείριση των καταχωρητών της CPU μετά από ανάλυση του προγράμματός μας και αποφασίζουν μόνοι τους πώς και που θα αποθηκευτούν οι διάφορες μεταβλητές. Έτσι
  - Σχεδόν πάντα κοιτούν να αγνοήσουν το “`register`”
  - Αν δεν το αγνοήσουν, κάποιες φορές οδηγούμαστε σε χειρότερες επιδόσεις από αυτές που θα είχαμε αν έπαιρνε όλες τις αποφάσεις ο μεταφραστής μόνος του
  - Επομένως, καλό είναι να μην τις πολυχρησιμοποιούμε σε προγράμματα γενικού σκοπού παρά μόνο σε εξειδικευμένες περιπτώσεις και σε περιπτώσεις που ο μεταφραστής δεν έχει την ικανότητα να κάνει πολλές βελτιστοποιήσεις (π.χ. σε ενσωματωμένα συστήματα).

Προγραμματίζοντας με γνώση της ιεραρχίας μνήμης

- Οι εντολές και τα δεδομένα ενός προγράμματος βρίσκονται αποθηκευμένα στην *μνήμη*, όπως γνωρίζουμε.
- Όμως, εδώ και πολλά χρόνια, η μνήμη του υπολογιστή δεν αποτελείται μόνο από ένα «κομμάτι».
- Υπάρχει ολόκληρη ιεραρχία από διαφορετικές μνήμες, παρότι ο επεξεργαστής θεωρεί ότι μιλάει μόνο με την «κύρια μνήμη»:
  - Η βασική (κύρια) μνήμη, γνωστή δυστυχώς και ως «η RAM του συστήματος»
  - Η κρυφή μνήμη πρώτου επιπέδου (L1 cache)
  - Η κρυφή μνήμη δεύτερου επιπέδου (L2 cache)
  - Η κρυφή μνήμη τρίτου επιπέδου (L3 cache)
- Όλες αυτές αποτελούν μαζί τη «μνήμη» του συστήματος όπου αποθηκεύονται εντολές και δεδομένα.
  - Όταν μία μεταβλητή αποθηκεύεται στη μνήμη, που ακριβώς πάει;
  - Μπορώ να εκμεταλλευτώ κάπως την ιεραρχία αυτή για αυξημένες επιδόσεις;

Σύνοψη της ιεραρχίας μνήμης

- Επεξεργαστής: ταχύτατος
- Μνήμη: αργή (και μάλιστα η διαφορά ταχύτητας αυξάνεται)
- Βασική λύση: *κρυφή μνήμη (**cache**)*



Πολύ γρήγορη, πολύ ακριβή, πολύ μικρή

Στηρίζεται στο γεγονός ότι οι εφαρμογές διαθέτουν *χωρική* και *χρονική τοπικότητα* ***(locality)***

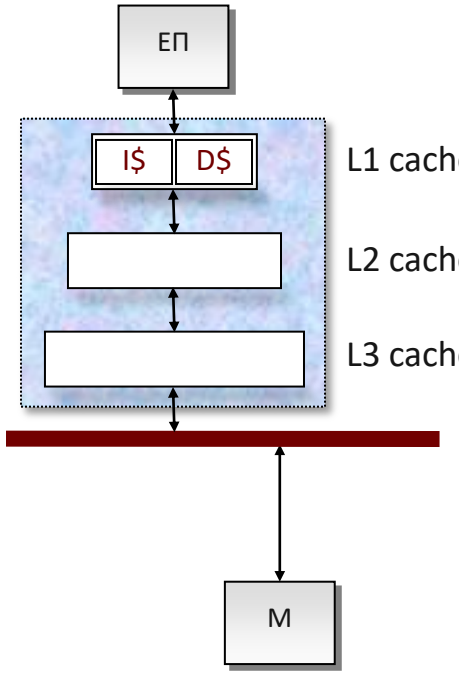
- Επικοινωνία με επεξεργαστή: ***λέξεις***
- Επικοινωνία με μνήμη: ***blocks λέξεων***
  - » τοποθετούνται στις **γραμμές** της cache ***(cache lines)***

Συμπεριφορά σε αναγνώσεις (read)

- Read **hit :** η λέξη υπάρχει ήδη σε μία γραμμή της cache
  - Η ζητούμενη λέξη ξεχωρίζει από την γραμμή της cache και παραδίδεται στον επεξεργαστή.
- Read **miss :** η λέξη δεν υπάρχει σε καμία γραμμή της cache
  - Το μπλοκ που περιλαμβάνει τη ζητούμενη λέξη προσκομίζεται από τη μνήμη, τοποθετείται σε μία γραμμή της cache και η λέξη παραδίδεται στον επεξεργαστή
  - ***Μεγάλη καθυστέρηση***
- Hit ratio = # προσπελάσεων που ήταν hit / συνολικό # προσπελάσεων
  - Ποσοστά 70 – 95% πολύ συχνά, ακόμα και με μικρές cache

Γενίκευση σε πολλαπλά επίπεδα (ιεραρχία caches)

- Για μείωση του κόστους αστοχίας (miss)
- 2-3 επίπεδα από αυξανόμενου μεγέθους και μειούμενου κόστους / ταχύτητας caches:



L1 cache ~ 16 – 64 KB, 1-4 cycles

L2 cache ~ 256 KB – 8 MB, 6-15 cycles

L3 cache ~ 4 – 32 MB, 30-50 cycles

~ Gbytes, 100+ cycles

Cache

- *Απαραίτητη* μιας και εξοικονομεί πολύ χρόνο
  - Πολύ γρηγορότερη από την κύρια μνήμη
  - Πολύ μικρότερη από την κύρια μνήμη
- Ότι δεδομένο χρησιμοποιήσει η διεργασία μας (δηλαδή ο επεξεργαστής),
  - Το φέρνει και το φυλάει
  - Φέρνει και τα γειτονικά του δεδομένα (φέρνει ένα ολόκληρο μπλοκ μνήμης)
- Εφόσον το πρόγραμμα έχει *τοπικότητα* (δηλαδή χρησιμοποιεί σχετικά λίγα και γειτονικά δεδομένα για αρκετή ώρα), έχουμε πολύ μεγάλη βελτίωση στην ταχύτητα προσπέλασης των δεδομένων.

Πώς να καταστρέψετε την cache

❖ Τι γίνεται εδώ;

```
for (i = 0; i < 10000; i++)
  for (j = 0; j < 10000; j++)
    a[i][j] = 2*b[i][j];
```

Χρόνος εκτέλεσης στο (παλιό) φορητό μου:

1.2 sec

Αυξημένη τοπικότητα (προσπέλαση γειτονικών στοιχείων το ένα μετά το άλλο λόγω της προσπέλασης *κατά γραμμές*)

```
for (j = 0; j < 10000; j++)
  for (i = 0; i < 10000; i++)
    a[i][j] = 2*b[i][j];
```

Χρόνος εκτέλεσης στο (παλιό) φορητό μου:

14 sec

Κακή τοπικότητα λόγω της προσπέλασης *κατά στήλες*. Φέρνοντας το b[0][0], έρχεται ολόκληρη γραμμή με τα b[0][1], b[0][2], …, b[0][7] (και ίσως και παραπάνω).
Όμως στην επόμενη επανάληψη του i, θέλουμε το b[1][0]! Επομένως έφερε τη γραμμή άσκοπα, πρέπει να φέρει νέα γραμμή … η οποία επίσης είναι άχρηστη για τη συνέχεια κ.ο.κ.

Οδηγίες για προγράμματα cache-friendly

- Στοχεύστε κυρίως στα loops που τρώνε τον περισσότερο χρόνο εκτέλεσης
- Αυξήστε την τοπικότητα σιγουρεύοντας ότι στο πιο συχνά χρησιμοποιούμενο εσωτερικό βρόχο δεν δημιουργούνται πολλά cache misses.

- Δείτε και τις επόμενες διαφάνειες ☺

Padding I

- Οι σύγχρονοι επεξεργαστές έχουν cache με γραμμές 32, 64 ή 128 bytes. Πρέπει να γνωρίζουμε το μέγεθος όταν βελτιστοποιούμε τις επιδόσεις.
- Για παράδειγμα, ας θεωρήσουμε ότι έχουμε cache με γραμμή μεγέθους **LINESIZE = 32** bytes και θέλουμε να δημιουργήσουμε έναν πίνακα από τις παρακάτω δομές:

```
struct data {
  int   a,b,c,d;    /* 4 bytes each */
  float z,y;        /* 4 bytes each */
}
```

- Αν και οι μεταφραστές μπορεί να κάνουν κάποιες τροποποιήσεις στον χώρο της δομής, λογικά το μέγεθος του struct είναι 24 bytes. Αυτό σημαίνει ότι πιάνει ¾ μίας γραμμής cache.
- Έτσι, π.χ. ενώ το στοιχείο 0 ενός πίνακα από τέτοια struct πέφτει σε μία γραμμή, το στοιχείο 1 πέφτει σε ΔΥΟ γραμμές (8 bytes από την μία και 16 bytes από την επόμενη)
  - Άρα η προσπέλαση του στοιχείου 1 απαιτεί προσκόμιση ΔΥΟ γραμμών, ενώ το 0 θέλει μόνο μία!!

Padding II

- Λύση: **padding** («γέμισμα») με άχρηστα bytes ώστε να πιάνω χώρο που είναι ακέραιο πολλαπλάσιο του LINESIZE:

```
struct data {
  int   a,b,c,d;
  float x,y;
  char  pad[8];  /* Άχρηστο, για να φτάσω 32 bytes */
}
```

- Κάθε στοιχείο ενός πίνακα από τέτοια structs πιάνει χώρο ακριβώς όσο και 1 γραμμή της cache.
  - Επομένως αρκεί να προσκομιστεί 1 μόνο γραμμή για κάθε στοιχείο!

- *«Πληρωμή» σε χώρο για να κερδίσουμε ταχύτητα*

Alignment

- Τα μπλοκ από την κύρια μνήμη πάνε σε γραμμές τις cache.
- OK, το padding, αλλά πώς εξασφαλίζω ότι το `malloc()` που θα κάνω για τον πίνακα με τα `structs` θα ξεκινάει ΑΚΡΙΒΩΣ στην αρχή ενός μπλοκ (δηλαδή θα ξεκινά από ακέραιο πολλαπλάσιο του LINESIZE)?
- Π.χ. αν το malloc() επιστρέψει τη θέση 32010?
  - Τότε κανένα στοιχείο του πίνακα δεν θα πέφτει στην αρχή μίας γραμμής!!
- Λύση: **alignment** (στοίχιση) διευθύνσεων σε ακέραια πολλαπλάσια του LINESIZE.
  - Είτε αυτόματα με ειδικές εντολές του μεταφραστή (μη αποδεκτή/μεταφέρσιμη C)
  - Είτε με το χέρι, παίρνοντας λίγο παραπάνω χώρο

«Χειροποίητο» alignment

- Έστω ότι θέλω να δεσμεύσω χώρο S bytes. Πώς μπορώ να ξέρω αν η `malloc()` θα μου επιστρέψει διεύθυνση η οποία είναι πολλαπλάσιο του LINESIZE?
- Επειδή δεν μπορώ να το γνωρίζω αυτό, η ιδέα είναι η εξής:
  a) Ζητάω λίγο παραπάνω χώρο, συγκεκριμένα (LINESIZE-1) παραπάνω bytes.
  b) Από αυτά που μου δόθηκαν χρησιμοποιώ τα S bytes ακριβώς, ξεκινώντας όμως από τη θέση που σίγουρα είναι πολλαπλάσιο του LINESIZE
- Δέσμευση παραπάνω μνήμης:

```
whole = malloc(S + LINESIZE – 1);
```

- Υποθέτοντας (μιας και συμβαίνει πάντα στην πράξη) ότι το LINESIZE είναι μία δύναμη του 2 (32, 64, 128 bytes), η διεύθυνση που είναι πολλαπλάσιο του LINESIZE είναι η:

```
alignedstart = (whole + LINESIZE – 1) & ~(LINESIZE – 1);
```

και αυτή πρέπει να χρησιμοποιήσω ως αρχή των S bytes.

(Γιατί;;;)

Βοήθεια: αν το LINESIZE είναι 2^k, τα πολλαπλάσιά του έχουν πάντα μηδενικά στα πρώτα/δεξιότερα k bits τους

Ερώτηση

- Τι συμπεριφορά θα έχει το παρακάτω πρόγραμμα;
  - Τι γίνεται στην εκτέλεση της `sumarray()` όταν αυξάνει το `stride`; Όταν αυξάνει το `size`;
  - Σε τι θα χρησίμευε; Μπορώ να καταλάβω κάτι από τους χρόνους;

```
#define MAX (1 << 22)      /* Up to 4 Mbytes */
int array[MAX];

for (size = MAX; size >= 0; size--) {
  for (stride = 1; stride <= MAXSTRIDE; stride++) {
    sumarray(size, stride);
  }
}

void sumarray(int size, int stride) {
  int i, sum = 0;
  for (i = 0; i < size; i += stride)
    sum += array[i];
}
```