

# Προγραμματισμός συστημάτων UNIX/POSIX

---

*Χρονομέτρηση, καθυστέρηση και ανάλυση  
επιδόσεων*



**ΜΥΥ 502**

- ❖ Πάρα πολύ συχνά υπάρχει η ανάγκη να χρονομετρήσουμε κάτι, π.χ.
  - Πόσο χρειάστηκε ένα πρόγραμμα για να εκτελεστεί
  - Πόσος χρόνος απαιτείται για συγκεκριμένα κομβικά τμήματα του κώδικά μας
  - Σύγκριση προγραμμάτων, benchmarking για επιδόσεις υλικού / λογισμικού
  - Profiling (στατιστικά χρονομέτρησης για να βρούμε που ένα πρόγραμμα καταναλώνει τον χρόνο του).
- Η χρονομέτρηση δεν είναι κάτι απλό και χρειάζεται μεγάλη προσοχή.
- ❖ Ξεκινώντας από το τέλος (profiling), μπορούμε να βρούμε (μεταξύ άλλων) πόσο χρόνο σπαταλάει η εφαρμογή μας σε κάθε συνάρτηση.
- ❖ Το εργαλείο ονομάζεται **gprof** (GNU profiler)
  - Για τη χρήση του απαιτείται να δοθεί το flag `-pg` κατά τη μετάφραση με τον `gcc`.
  - Κατά την εκτέλεση του `a.out` (η οποία είναι αρκετά πιο αργή από ότι θα περιμέναμε) δημιουργείται ένα αρχείο `"gmon.out"`.
  - Εκτελούμε το `gprof`, το οποίο με βάση το αρχείο αυτό μας δείχνει ενδιαφέρουσες πληροφορίες.

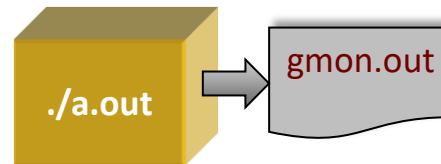
# Profiling με το gprof

1. Μετάφραση με `-pg`



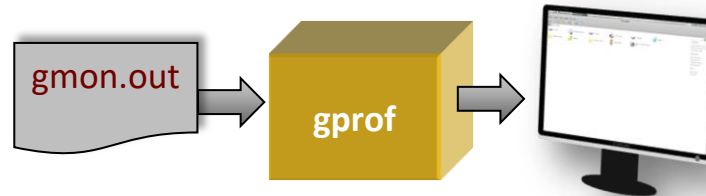
```
gcc -pg prog.c
```

2. Εκτέλεση προγράμματος



```
./a.out
```

3. Εκτέλεση gprof



```
gprof a.out gmon.out ...
```

# Παράδειγμα

testprof.c

```
int a() {
    int i, sum=0;
    for (i = 0; i < 100000; i++)
        sum += i;
    return sum;
}
int b(void) {
    int i, sum=0;
    for (i = 0; i < 400000; i++)    /* should be x4 the time of a() */
        sum += i;
    return sum;
}
int main() {
    int iterations = 1000;
    printf("profiling example.\n");
    for ( ; iterations > 0; iterations--) {
        a();
        b();
    }
    return 0;
}
```

ΤΕΡΜΑΤΙΚΟ

```
$ gcc -pg testprof.c
$ ./a.out
$
```

❖ Μπορούμε να δούμε τα αποτελέσματα με 3 τρόπους:

**1. Flat profile**

Δείχνει πόσος χρόνος σπαταλήθηκε στην κάθε συνάρτηση και πόσες φορές έγινε κλήση στη συνάρτηση αυτή.

**2. Call graph**

Για κάθε συνάρτηση, δείχνει ποιες συναρτήσεις την κάλεσαν, ποιες κάλεσε αυτή και πόσες φορές έγιναν αυτά.

**3. Annotated source**

Δείχνει τον κώδικα και πόσες φορές εκτελέστηκαν διάφορα τμήματά του. Για τη συγκεκριμένη προβολή, *πρέπει να έχουμε δώσει και το όρισμα  $-g$  κατά τη μετάφραση.*

# Flat profile

Περίοδος "δειγματοληψίας".

Όποιο αποτέλεσμα είναι μικρότερο από αυτόν το χρόνο, θεωρείται ανακριβές.

```
$ gprof a.out gmon.out -p
```

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
81.38	0.83	0.83	1000	830.12	830.12	b
19.85	1.03	0.20	1000	202.47	202.47	a
...						

Ποσοστό του χρόνου εκτέλεσης που σπαταλήθηκε στην κάθε συνάρτηση.

Πόσος χρόνος (sec) αφιερώθηκε στην κάθε συνάρτηση

Και οι δύο κλήθηκαν 1000 φορές

Χρόνος (μsec) για την κάθε κλήση

# Call graph

```
$ gprof a.out gmon.out -q
```

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.97% of 1.03 seconds

index	% time	self	children	called	name	
<spontaneous>						
[1]	100.0	0.00	1.03		main [1]	← Η συνάρτηση
		0.83	0.00	1000/1000	b [2]	← Ακολουθούν
		0.20	0.00	1000/1000	a [3]	← αυτές που καλεί
-----						
		0.83	0.00	1000/1000	main [1]	
[2]	80.4	0.83	0.00	1000	b [2]	
-----						
		0.20	0.00	1000/1000	main [1]	← Προηγούνται
		0.20	0.00	1000	a [3]	← αυτές που την
[3]	19.6	0.20	0.00	1000	a [3]	← καλούν
-----						
...						← Η συνάρτηση

Πόσος χρόνος (sec) αφιερώθηκε  
στις συναρτήσεις που κάλεσε (τις  
ονομάζει "παιδιά" της)

Η printf() πού είναι;

Πρέπει όλες οι συναρτήσεις να  
έχουν μεταφραστεί με -pg

# Annotated source

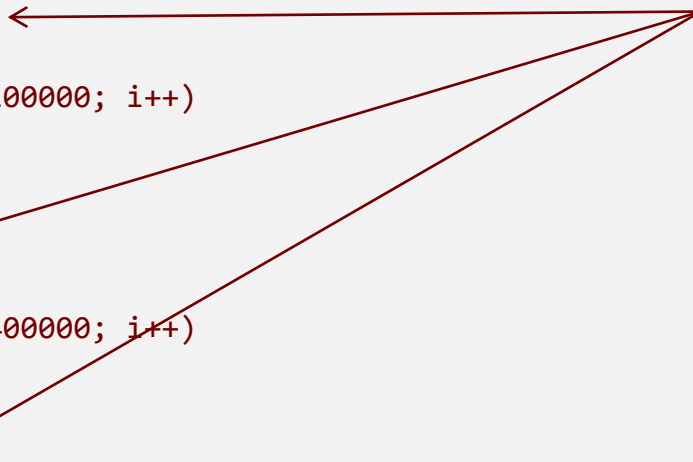
```
$ gcc -pg -g testprof.c
$ ./a.out
$ gprof a.out gmon.out -A
```

```
*** File /home/dimako/testprof.c:
    #include <stdio.h>
1000 -> int a() {
        int i, sum=0;
        for (i = 0; i < 100000; i++)
            sum += i;
        return sum;
    }
1000 -> int b(void) {
        int i, sum=0;
        for (i = 0; i < 400000; i++)
            sum += i;
        return sum;
    }
##### -> int main() {
        int iterations = 1000;
        printf("profiling example\n");
        for ( ; iterations > 0; iterations--) {
            a();
            b();
        }
    }

...

```

Κάθε block του  
κώδικα πόσες  
φορές  
εκτελέστηκε





# Χρονομέτρηση (I)

- ❖ Πρώτα από όλα, **τι ακριβώς θέλουμε να μετρήσουμε;**
  - Μας ενδιαφέρει ο χρόνος που ο επεξεργαστής αφιέρωσε σε ένα πρόγραμμα ή σε ένα συγκεκριμένο τμήμα του;
    - » *Χρόνος επεξεργαστή (χρόνος καθαρών υπολογισμών).*
  - Ή μας ενδιαφέρει πόση ώρα περνάει από ένα συγκεκριμένο γεγονός;
    - » *Πραγματικός χρόνος που παρήλθε.*
- ❖ Στην πρώτη περίπτωση, μας ενδιαφέρει μόνο πόσο χρόνο χρειάστηκε ο επεξεργαστής για καθαρούς υπολογισμούς, όχι πόσος χρόνος πέρασε από την ώρα που ξεκίνησε η χρονομέτρηση.
  - Ο χρόνος είναι ανεξάρτητος του φόρτου του συστήματος – θα είναι πάντα ο ίδιος, ακόμα και όταν εκτελούνται πολλές διεργασίες "ταυτόχρονα".
  - Μετρά "χρόνο επεξεργαστή" (CPU time) όχι πραγματικό χρόνο που παρήλθε (real/elapsed time).
  - Ίσως η πιο χρήσιμη περίπτωση όταν ενδιαφερόμαστε για εκτίμηση της επίδοσης του κώδικα.
- ❖ Στη δεύτερη περίπτωση μας ενδιαφέρει πόσος πραγματικός χρόνος παρήλθε.
  - Εξαρτάται από τον φόρτο του συστήματος! Όταν εκτελείται μόνο η εφαρμογή μας, η χρονομέτρηση θα είναι ακριβής, όταν εκτελούνται κι άλλες θα έχει διακυμάνσεις. Η χρονομέτρηση πρέπει να γίνεται σε ελεγχόμενο περιβάλλον.
  - Είναι χρήσιμο όταν μας ενδιαφέρει η συνολική συμπεριφορά, σε πραγματικές συνθήκες.

## ❖ Τι τάξη μεγέθους είναι οι χρόνοι που μας ενδιαφέρουν;

- Είναι χρόνοι της τάξης των λεπτών/ωρών/ημερών κλπ ;
  - ❖ Εδώ δεν μπαίνει θέμα ακρίβειας.
- Είναι χρόνοι τάξης δευτερολέπτων / δεκάτων ή εκατοστών του δευτερολέπτου;
  - ❖ Κι εδώ η ακρίβεια δεν είναι (συνήθως) ουσιαστικό θέμα, οι περισσότεροι μηχανισμοί χρονομέτρησης είναι εφαρμόσιμοι.
- Είναι χρόνοι της τάξης του msec / msec / nsec?
  - ❖ Εδώ απαιτείται χρονομέτρηση με κατάλληλη υποστήριξη από το υλικό. Οι χρονομετρητές πρέπει να έχουν αυξημένη ακρίβεια και ανάλυση (resolution).
- Στην τελευταία περίπτωση δεν υπάρχουν πάντα λύσεις τυποποιημένες που δουλεύουν σε όλα τα συστήματα.
- Για τις υπόλοιπες περιπτώσεις μπορούν να χρησιμοποιηθούν αρκετοί και διαφορετικοί μηχανισμοί.

## ❖ Τι μηχανισμούς χρονομέτρησης διαθέτουμε;

- Μετρούν χρόνο ("wall-clock" timers);
- Μετρούν διαστήματα (interval timers);

❖ Στην πρώτη περίπτωση, οι χρόνοι που παίρνουμε είναι «έτοιμοι» για χρήση.

❖ Στη δεύτερη περίπτωση, θα πρέπει να ξέρουμε τη *διάρκεια του κάθε διαστήματος* προκειμένου να βρούμε τον πραγματικό χρόνο:

- Π.χ. ρολόγια που αυξάνουν έναν μετρητή ανά τακτά διαστήματα.
- Π.χ. μετρητές υψηλής ακρίβειας της CPU που μετρούν το πλήθος των παλμών του ρολογιού (clock cycles): *θα πρέπει να γνωρίζουμε τη συχνότητα του επεξεργαστή* για να βρούμε ποια είναι η διάρκεια του κάθε παλμού και άρα σε πόσο χρόνο αντιστοιχούν οι παλμοί που μετρήσαμε.

# Δύο τρόποι χρονομέτρησης

## 1. Από το τερματικό

- Εντολή time: μετρά την εκτέλεση ενός ολόκληρου προγράμματος

```
$ time ./a.out
```

...

```
real      0m5.316s
```

```
user      0m2.304s
```

```
sys       0m0.004s
```

Real: πραγματικός χρόνος που παρήλθε (εμπεριέχει ότι καθυστερήσεις υπήρχαν, π.χ. αναμονή να πληκτρολογήσει κάτι ο χρήστης)

User: χρόνος καθαρών υπολογισμών (CPU time) χωρίς τις κλήσεις συστήματος

Sys: χρόνος καθαρών υπολογισμών (CPU time) που δαπανήθηκαν σε κλήσεις συστήματος

## 2. Προγραμματιστικά

- Για χρονομέτρηση ενός τμήματος του κώδικά μας:

```
<timing_call 1>    /* Record time t1 */
```

```
    <κώδικας>      /* The code we want to measure */
```

```
<timing_call 2>    /* Record time t2 */
```

- Η διαφορά των δύο χρόνων t2, t1 θα μας δώσει την επιθυμητή μέτρηση

# Χρονομέτρηση με την clock()

- ❖ Η **clock()** μετρά καθαρό χρόνο εκτέλεσης (CPU time)
  - Συνήθως ακρίβεια εκατοστού του δευτερολέπτου
- ❖ Η **clock()** είναι interval-based και επιστρέφει τον χρόνο που αφιέρωσε η CPU από τη στιγμή που ξεκίνησε το πρόγραμμά σας, μετρημένο σε «κύκλους» (clocks).
  - Για να βρείτε το χρόνο σε δευτερόλεπτα θα πρέπει να διαιρέσετε με τη σταθερά **CLOCKS\_PER\_SEC**.
  - Προσοχή στη διαίρεση γιατί και το **CLOCKS\_PER\_SEC** και η τιμή επιστροφής της **clock()** είναι ακέραιοι.
- ❖ Θα πρέπει να κάνετε `#include <time.h>`.
- ❖ Προσοχή: επειδή η **clock()** επιστρέφει ακέραιο, αν το πρόγραμμά σας χρειάζεται πολλή ώρα να τρέξει υπάρχει κίνδυνος να μηδενιστεί ο χρονομετρητής και να λάβετε λάθος μετρήσεις. Π.χ. στο solaris αναφέρεται ότι ο χρονομετρητής μηδενίζεται και μετρά πάλι από την αρχή κάθε 36 λεπτά καθαρού χρόνου εκτέλεσης!

\$ man clock

# Παράδειγμα χρονομέτρησης με την clock()

```
#include <stdio.h>
#include <time.h> /* Για την clock() */

int main() {
    double t1, t2; /* Για αποφυγή ακέραιας διαίρεσης */
    int i, sum = 0;

    t1 = (double) clock(); /* επιστρέφει clock_t (συνήθως int ή long) */
    for (i = 0; i < 100000000; i++)
        sum += i;
    t2 = (double) clock();

    printf("Added 100000000 numbers in %lf sec (CPU time).\n",
           (t2 - t1) / CLOCKS_PER_SEC );
    return 0;
}
```

# Χρονομέτρηση με την times()

- ❖ Η **times()** μετρά και πραγματικό χρόνο αλλά και καθαρό χρόνο εκτέλεσης (CPU time).
  - Ο πραγματικός χρόνος που επιστρέφει είναι το χρονικό διάστημα που παρήλθε από κάποιο απροσδιόριστο σημείο στο παρελθόν (π.χ. system boot time).
  - `#include <sys/times.h>`.
- ❖ Και η **times()** είναι interval-based. Όμως επιστρέφει χρόνους μετρημένους σε «χτύπους ρολογιού» (clock ticks).
  - Για να βρείτε το χρόνο σε δευτερόλεπτα θα πρέπει να διαιρέσετε με το πλήθος των χτύπων ρολογιού ανά δευτερόλεπτο, το οποίο το βρίσκεται μόνο προγραμματιστικά ως εξής:  

```
ticspersec = sysconf(_SC_CLK_TCK);    /* unistd.h */
```
  - Προσοχή πάλι στις διαιρέσεις γιατί και οι χτύποι ανά δευτερόλεπτο και η τιμή επιστροφής της **times()** είναι ακέραιοι.
- ❖ Επιστρέφει τον *πραγματικό χρόνο που παρήλθε*.
- ❖ Παίρνει ως παράμετρο ένα **struct tms** από όπου μπορούμε να μάθουμε για τους καθαρούς χρόνους εκτέλεσης:

```
struct tms {  
    clock_t tms_utime, tms_stime    /* for me */  
    clock_t tms_cutime, tms_cstime; /* for my child processes */  
};
```

`$ man -s 3 times`

# Παράδειγμα χρονομέτρησης με την times()

```
#include <stdio.h>
#include <sys/times.h>      /* Για την times() */
#include <unistd.h>         /* Για την sysconf() */

int main() {
    double t1, t2, cpu_time; /* Για αποφυγή ακεραίας διαίρεσης */
    struct tms tb1, tb2;     /* Το χρειάζεται η times() */
    long   ticspersec;
    int    i, sum = 0;

    t1 = (double) times(&tb1); /* Η times() επιστρέφει (long) int */
    for (i = 0; i < 100000000; i++)
        sum += i;
    t2 = (double) times(&tb2);

    cpu_time = (double) ((tb2.tms_utime + tb2.tms_stime) -
                        (tb1.tms_utime + tb1.tms_stime));
    ticspersec = sysconf(_SC_CLK_TCK); /* # clock ticks / sec */

    printf("Real time: %lf sec; CPU time: %lf sec.\n",
           (t2 - t1) / ticspersec, cpu_time / ticspersec);
    return 0;
}
```



# Χρονομέτρηση με την `gettimeofday()`

- ❖ Η `gettimeofday()` μετρά τον *πραγματικό* χρόνο που παρήλθε...
  - ... από την 1/1/1970, ώρα 00:00 (το λεγόμενο «**Epoch**»).
  - `#include <sys/time.h> /* Όχι το sys/times.h !! */`
  - Πολύ συχνή η χρήση της στην πράξη.
- ❖ Η `gettimeofday()` επιστρέφει χρόνο (wall-clock time).
  - Υλοποιείται συνήθως (όχι πάντα) με αρκετά μεγάλη ανάλυση (της τάξης του 1μsec).
- ❖ Παίρνει δύο παραμέτρους, με τη δεύτερη συνήθως NULL. Η πρώτη παράμετρος είναι δείκτης σε ένα **struct timeval** με τα εξής πεδία:

```
struct timeval {  
    time_t tv_sec;           /* seconds */  
    unsigned int tv_usec;    /* microseconds */  
};
```

To `time_t` ήταν μέχρι πρότινος ένας ακέραιος 32bit. Σε 68 χρόνια γίνεται overflow!  
-- βλ. "year 2038 problem"

```
$ man gettimeofday
```

# Παράδειγμα χρονομέτρησης με την gettimeofday()

```
#include <stdio.h>
#include <sys/time.h>          /* Για την gettimeofday() */

int main() {
    struct timeval tv1, tv2;
    int    i, sum = 0;
    double t;

    gettimeofday(&tv1, NULL);
    for (i = 0; i < 100000000; i++)
        sum += i;
    gettimeofday(&tv2, NULL);

    t = (tv2.tv_sec - tv1.tv_sec) +          /* seconds */
        (tv2.tv_usec - tv1.tv_usec)*1.0E-6; /* convert μsec to sec */

    printf("real time: %lf sec.\n", t);
    return 0;
}
```

# Τεχνική: εύρεση της ανάλυσης της gettimeofday()

❖ Πώς μπορώ να βρω τι ανάλυση (resolution) έχει η `gettimeofday()`;

➤ Δηλαδή, ποιος είναι ο μικρότερος χρόνος που μπορεί να μετρήσει;

```
struct timeval tv1, tv2;
```

```
int resolution;
```

```
gettimeofday(&tv1, NULL);
```

```
do {
```

```
    gettimeofday(&tv2, NULL);
```

```
}
```

```
while (tv1.tv_usec == tv2.tv_usec);    /* Μέχρι να αλλάξει! */
```

```
resolution = tv2.tv_usec - tv1.tv_usec; /* Σε μsec */
```

*(θεωρώντας ότι ο χρόνος για την κλήση της `gettimeofday()` είναι αμελητέος σε σχέση με την ανάλυση)*

# Χρονομέτρηση με την clock\_gettime()

## ❖ Η **clock\_gettime()** είναι η πλέον σύγχρονη κλήση:

- Μετρά με βάση κάποιο από τα παρεχόμενα **ρολόγια**.
- Σε όλα τα συστήματα POSIX εγγυημένα υπάρχει ένα ρολόι που μετρά πραγματικό χρόνο (**CLOCK\_REALTIME**).
- Διάφορα συστήματα παρέχουν επιπλέον ρολόγια.
  - ✧ Π.χ. στο Solaris υπήρχε το **CLOCK\_HIGHRES** (πραγματικού χρόνου με υπερυψηλή ανάλυση)
  - ✧ Σε πρόσφατες εκδόσεις του Linux υπάρχει το **CLOCK\_PROCESS\_CPUTIME\_ID** (για χρόνους καθαρών υπολογισμών στη CPU).
- `#include <time.h>`.

## ❖ Κλήση:

**clock\_gettime**(clockid\_t clk, struct timespec \*tp);

```
struct timespec {  
    time_t tv_sec;           /* seconds */  
    long   tv_nsec;         /* nanoseconds */  
};  
(στο tv_nsec μπορούμε να βάλουμε από 0 μέχρι 999.999.999).
```

## ❖ Επιπλέον ευκολία:

**clock\_getres**(clockid\_t clk, struct timespec \*tp);

- Στο tp λαμβάνουμε την ανάλυση (resolution) του ρολογιού clk.

`$ man clock_gettime`

# Παράδειγμα χρονομέτρησης με την clock\_gettime()

```
#include <stdio.h>
#include <time.h>          /* Για την clock_gettime() */

int main() {
    struct timespec ts1, ts2;
    int    i, sum = 0;
    double t;

    clock_gettime(CLOCK_REALTIME, &ts1);
    for (i = 0; i < 100000000; i++)
        sum += i;
    clock_gettime(CLOCK_REALTIME, &ts2);

    t = (ts2.tv_sec - ts1.tv_sec) +          /* seconds */
        (ts2.tv_nsec - ts1.tv_nsec)*1.0E-9; /* convert nsec to sec */
    printf("real time: %lf sec.\n", t);

    clock_getres(CLOCK_REALTIME, &ts1);
    printf("clock resolution: %lf nsec.\n", ts1.tv_sec*1.0E9 + ts1.tv_nsec);
    return 0;
}
```

- ❖ Μια πρακτική αναγκαιότητα είναι η τεχνητή *καθυστέρηση*.
  - Είτε ενδιαφερόμαστε να καθυστερήσουμε τα πρόγραμμά μας για λίγο (π.χ. για να προλάβει να γίνει κάποιο γεγονός)
  - Είτε θέλουμε να αφήσουμε να περάσει ένα προκαθορισμένο διάστημα προκειμένου να χρονομετρήσουμε κάτι.
- ❖ Και στις δύο περιπτώσεις, μπορούμε να πούμε στο σύστημα να "κοιμίσει" τη διεργασία μας για ένα συγκεκριμένο χρονικό διάστημα.
  - Η διεργασία σταματά προσωρινά να εκτελείται
  - Το σύστημα εκτελεί ότι άλλες διεργασίες έχει (για να μην κάθετε)
  - Όταν παρέλθει το χρονικό διάστημα που ορίσαμε, συνεχίζει την εκτέλεση της διεργασίας μας
- ❖ Συναρτήσεις τύπου "sleep()"

# Καθυστέρηση – αναμονή (II)

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
int usleep(unsigned int microsecs); /* μsec */
```

- ❖ Η `sleep()` είναι η πιο κλασική κλήση, αλλά είναι για σχετικά μεγάλα διαστήματα ( $\geq 1$  sec)
- ❖ Η `usleep()` είναι για διαστήματα *μέχρι 1 sec* (δεν δουλεύει για μεγαλύτερα διαστήματα, το όρισμα πρέπει να είναι  $\leq 1.000.000$ ).

```
#include <time.h>
int nanosleep(struct timespec *req, struct timespec *rem);
```

- ❖ Το `req` προδιορίζει το πλήθος των nanoseconds για το διάστημα αναμονής (όπως στην `clock_gettime()`)

```
struct timespec {
    time_t tv_sec;           /* seconds */
    long   tv_nsec;         /* nanoseconds */
};
```
- ❖ Ναι μεν προσδιορίζουμε `nsec`, αλλά αν το διάστημα αναμονής είναι μικρότερο από την ανάλυση του ρολογιού του συστήματος, ο χρόνος αναμονής θα είναι μεγαλύτερος από αυτόν που ζητήσαμε.
- ❖ Το `rem` είναι συνήθως `NULL`.

\$ `man nanosleep`

# Χρονόμετρο – αντίστροφη μέτρηση

- ❖ Πολλές φορές υπάρχει η ανάγκη να γνωρίζουμε πότε παρέρχεται ένα συγκεκριμένο χρονικό διάστημα.
  - Παραδείγματα:
    - ✧ Θέλουμε να υλοποιήσουμε αντίστροφη μέτρηση
    - ✧ Θέλουμε κάθε 15 λεπτά να εκτυπώνεται ένα ενημερωτικό μήνυμα προς τον χρήστη.
    - ✧ Θέλουμε σε ένα παιχνίδι να μετρήσουμε πόσες φορές ο χρήστης πάτησε ένα πλήκτρο μέσα σε 10 δευτερόλεπτα.
    - ✧ κλπ
  - Κατά τη διάρκεια αυτού του διαστήματος, θέλουμε η εφαρμογή μας να συνεχίζει την εκτέλεσή της
  - Επομένως οι συναρτήσεις τύπου `sleep()` δεν μπορούν να χρησιμοποιηθούν
- ❖ Οι συναρτήσεις που παρέχονται για τέτοιες περιπτώσεις είναι οι συναρτήσεις «ξυπνητηριών»
  - `alarm()`
  - `setitimer()`
- ❖ Η βασική ιδέα πίσω από αυτές είναι:
  - Ενεργοποίηση ενός ξυπνητηριού / χρονομετρητή (timer) που μετράει αντίστροφα
  - Όταν παρέλθει το δοθέν χρονικό διάστημα, προκαλείται διακοπή (interrupt) στη διεργασία
  - Έχουμε φροντίσει να έχουμε δική μας συνάρτηση για την εξυπηρέτηση της διακοπής



# alarm()

- ❖ Η «κλασική» κλήση είναι η `alarm()`, η οποία όμως μπορεί να μετρήσει μόνο δευτερόλεπτα:

```
#include <unistd.h>
unsigned int alarm(unsigned int sec);
```

- ❖ Η παράμετρος `sec` καθορίζει το χρονικό διάστημα σε δευτερόλεπτα.
- ❖ Όταν ολοκληρωθεί το χρονικό διάστημα προκαλείται σήμα / διακοπή τύπου `SIGALRM`.
- ❖ Παρατηρήσεις:
  1. Αν η συνάρτηση κληθεί πριν τελειώσει το προηγούμενο χρονικό διάστημα που είχε τεθεί, ακυρώνεται το παλιό και ορίζεται νέο διάστημα.
  2. Καλώντας την με παράμετρο 0, **ακυρώνεται** το χρονόμετρο.
  3. Επιστρέφει το χρόνο που έμενε μέχρι να ολοκληρωθεί το προηγούμενο διάστημα.
  4. Επειδή και η `sleep()` μπορεί να υλοποιηθεί με χρήση των ίδιων χρονομέτρων με την `alarm()`, δεν πρέπει να γίνεται `sleep()` πριν την ολοκλήρωση του χρονικού διαστήματος από την `alarm()`.

# setitimer() - I

- ❖ Η συνιστώμενη κλήση είναι η `setitimer()` :

```
#include <sys/time.h>
int setitimer(int which, /* ποιο χρονόμετρο να χρησιμοποιηθεί */
              struct itimerval *new,
              struct itimerval *old);
```

- ❖ Επιστρέφει 0 αν πέτυχε, ή  $< 0$  σε περίπτωση αποτυχίας.
- ❖ Για κάθε διεργασία ορίζονται 3 διαφορετικά χρονόμετρα
  - **ITIMER\_REAL**, για αντίστροφη χρονομέτρηση σε πραγματικό χρόνο.
  - **ITIMER\_VIRTUAL**, για αντίστροφη χρονομέτρηση σε χρόνο εκτέλεσης (CPU user time)
  - **ITIMER\_PROF**, για αντίστροφη χρονομέτρηση σε χρόνο εκτέλεσης που περιλαμβάνει και κλήσεις συστήματος (CPU user + system time, κυρίως για profiling και debugging).
  - *Μόνο ένα χρονόμετρο μπορεί να έχετε ενεργό σε οποιαδήποτε χρονική στιγμή.*
- ❖ Το **ITIMER\_REAL** είναι το ίδιο με αυτό που χρησιμοποιεί και η `alarm()`, οπότε δεν πρέπει να μπλέκονται οι δύο κλήσεις.

# setitimer() - II

```
int setitimer(int which,
              struct itimerval *new,
              struct itimerval *old);
```

❖ Το old είναι συνήθως NULL, αλλιώς εκεί επιστρέφεται ο χρόνος που απέμενε από το προηγούμενο χρονόμετρο.

❖ Το χρονικό διάστημα προσδιορίζεται στο new που έχει τύπο:

```
struct itimerval {
    struct timeval it_interval; /* next value */
    struct timeval it_value;    /* current value */
};
```

Στο it\_value δίνουμε το χρονικό διάστημα που πρέπει να παρέλθει. Όταν ολοκληρωθεί, **ΤΟ ΧΡΟΝΟΜΕΤΡΟ ΞΑΝΑΡΧΙΖΕΙ ΝΑ ΜΕΤΡΑΕΙ ΑΝΤΙΣΤΡΟΦΑ** για διάστημα ίσο με it\_interval.

- Αν μας ενδιαφέρει μόνο μία χρονομέτρηση, θα πρέπει το it\_interval να το θέσουμε σε μηδενική τιμή,
- αλλιώς θα έχουμε συνεχώς ξυπνήματα κάθε it\_interval χρόνο.

❖ Όταν ολοκληρωθεί το διάστημα, παράγεται διαφορετικό signal ανάλογα με το χρονόμετρο που χρησιμοποιήθηκε:

- SIGALRM για το ITIMER\_REAL.
- SIGVTALRM για το ITIMER\_VIRTUAL
- SIGPROF για το ITIMER\_PROF

```
To struct timeval είναι γνωστό
από την gettimeofday():
struct timeval {
    time_t tv_sec;
    unsigned int tv_usec;
};
```

# Πιθανή υλοποίηση της alarm() μέσω setitimer()

```
unsigned int myalarm (unsigned int sec) {  
    struct itimerval old, new;  
  
    new.it_value.tv_sec      = (long int) sec;  
    new.it_value.tv_usec     = 0;  
    new.it_interval.tv_sec   = 0; /* do not repeat */  
    new.it_interval.tv_usec  = 0;  
    if (setitimer(ITIMER_REAL, &new, &old) < 0)  
        return 0;  
    else  
        return old.it_value.tv_sec;  
}
```

# Απλό παράδειγμα χρήσης

```
#include <stdio.h>      /* for printf */
#include <sys/time.h>    /* for setitimer */
#include <signal.h>      /* for signal */

void handleAlarm(int);

int main() {
    struct itimerval it_val;                /* for setting itimer */

    /* sigaction() should actually be used - I use signal() due to lack of slides space */
    if (signal(SIGALRM, handleAlarm) == SIG_ERR) { /* Set SIGALRM handler */
        perror("Unable to catch SIGALRM");
        exit(1);
    }
    it_val.it_value.tv_sec = 0;              /* Set the timer */
    it_val.it_value.tv_usec = 500000;       /* 0.5 sec */
    it_val.it_interval = it_val.it_value;   /* repeat every 0.5 sec */
    if (setitimer(ITIMER_REAL, &it_val, NULL) == -1) {
        perror("error calling setitimer()");
        exit(1);
    }
    while (1)
        ;
    return (1);
}

void handleAlarm(int ignore) {
    printf("0.5 sec passed..\n");
}
```