

Προγραμματισμός συστημάτων κοινόχρηστης μνήμης (I)

Νήματα POSIX



ΜΥΕ023

Παράλληλα
Συστήματα &
Προγραμματισμός

«Οντότητες» εκτέλεσης κώδικα

- Σειριακό πρόγραμμα για υπολογισμό του $\pi = 3.141592\dots$

```
#define N 512
float pi = 0.0, W = 1.0/N;

int main() {
    int i;
    for (i = 0; i < N; i++)
        pi += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
    printf("π = %f\n", pi);
    return 0;
}
```

- Όταν φορτωθεί και εκτελείται το πρόγραμμα γίνεται **διεργασία (process)**.
- Κάθε διεργασία εκτελείται σε έναν επεξεργαστή
 - Το λειτουργικό σύστημα φροντίζει για αυτό

Διεργασίες & νήματα

- Μία διεργασία αποτελείται (κατ' ελάχιστο) από δεδομένα, κώδικα (εντολές), μία στοίβα (stack) και έναν μετρητή προγράμματος (program counter)
 - Ο μετρητής προγράμματος (PC):
 - δείχνει στην επόμενη εντολή του κώδικα που θα εκτελεστεί
 - Η στοίβα χρειάζεται για:
 - αποθήκευση των τοπικών μεταβλητών (τα «δεδομένα» της διεργασίας που είπαμε παραπάνω είναι οι global μεταβλητές)
- Ο συνδυασμός PC + στοίβα λέγεται **νήμα εκτέλεσης (thread)**
 - Δηλαδή η διεργασία αποτελείται από (global) δεδομένα, από κώδικα και από ένα νήμα εκτέλεσης που περνάει από τις εντολές του κώδικα
 - Το νήμα είναι αυτό που εκτελείται όταν λέμε ότι εκτελείται η διεργασία!
 - Το νήμα αυτό λέγεται *αρχικό νήμα* της διεργασίας

```
#define N 512
float pi = 0.0, W = 1.0/N;

int main() {
    int i;
    for (i = 0; i < N; i++)
        pi += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
    printf("π = %f\n", pi);
    return 0;
}
```

Διεργασίες & νήματα

- Μία διεργασία μπορεί να δημιουργήσει κι άλλα πολλά νήματα εκτέλεσης:
 - Δημιουργώντας τίποτε άλλο εκτός από πολλές στοίβες και PCs
 - Όλα θα τρέχουν εντολές από τον *ίδιο* κώδικα (δεν δημιουργείται άλλο αντίγραφο του) και τέλος
 - Όλα θα έχουν τα *ίδια* global δεδομένα! Δηλαδή οι καθολικές μεταβλητές της διεργασίας είναι ΑΥΤΟΜΑΤΩΣ ΚΟΙΝΕΣ μεταξύ των νημάτων της.
 - Στην κάθε στοίβα το κάθε νήμα θα έχει τις δικές του τοπικές μεταβλητές
 - Κάθε νήμα εκτελείται σε έναν επεξεργαστή
 - Το λειτουργικό σύστημα φροντίζει για αυτό (εκτός από μερικές περιπτώσεις)

Παράλληλα προγράμματα

- Επομένως, για να εκμεταλλευτούμε πολλαπλούς επεξεργαστές, έχουμε δύο βασικές τεχνικές:
 - **Πολλαπλές διεργασίες**
 - Η διεργασία μας «γεννά» κι άλλες διεργασίες και κάθε μία εκτελείται στον δικό της επεξεργαστή (π.χ. fork())
 - **Πολλαπλά νήματα σε μία διεργασία**
 - Το αρχικό νήμα εκτέλεσης «γεννά» κι άλλα νήματα και κάθε ένα εκτελείται στον δικό του επεξεργαστή
- Υπάρχουν διαφορές σε ταχύτητα δημιουργίας, απαιτήσεις σε πόρους (π.χ. μνήμη) κλπ αλλά η πιο σημαντική διαφορά είναι ότι:
 - Ανάμεσα στις πολλαπλές διεργασίες *ΔΕΝ ΥΠΑΡΧΕΙ* καμία κοινή μεταβλητή. Πρέπει να δημιουργηθούν «με το χέρι», ενώ μεταξύ των πολλαπλών νημάτων όλες οι global μεταβλητές είναι κοινές είτε το θέλουμε είτε όχι.

Shared address space / shared variables

- Τι χρειάζεται κανείς για να προγραμματίσει σε αυτό το μοντέλο:
 - Οντότητες εκτέλεσης (νήματα, διεργασίες)
 - Δημιουργία, διαχείριση
 - Κοινές μεταβλητές μεταξύ των οντοτήτων
 - Ορισμός μεταβλητών (τι είναι κοινό και πως ορίζεται)
 - Τις διαβάζουν και τις τροποποιούν όλες οι διεργασίες
 - Αμοιβαίος αποκλεισμός
 - Π.χ. κλειδαριές
 - Συγχρονισμός
 - Π.χ. κλήσεις φραγής (barrier calls)

Γιατί αμοιβαίος αποκλεισμός;

Αρχικά η κοινή μεταβλητή A είναι ίση με το 0

Νήμα T1	Νήμα T2
$A = A+1;$	$A = A-1;$

- Λεπτομέρεια εκτέλεσης της εντολής $A = A_{\pm 1}$ σε έναν επεξεργαστή:
 - α. Ο επεξεργαστής μεταφέρει από τη μνήμη την τιμή της A
 - β. Αυξάνει/μειώνει κατά 1
 - γ. Αποθηκεύει στη μνήμη την νέα τιμή.

Χρονική στιγμή	Τι κάνει ο επεξεργαστής 1	Τι κάνει ο επεξεργαστής 2
t	α	-
$t+1$	β	α
$t+2$	γ	β
$t+3$	-	γ
A = -1		

Χρονική στιγμή	Τι κάνει ο επεξεργαστής 1	Τι κάνει ο επεξεργαστής 2
t	-	α
$t+1$	α	β
$t+2$	β	γ
$t+3$	γ	
A = 1		

Χρονική στιγμή	Εντολή που εκτελεί ο επεξεργαστής 1	Εντολή που εκτελεί ο επεξεργαστής 2
t	-	α
$t+1$	-	β
$t+2$	-	γ
$t+3$	α	
$t+4$	β	
$t+5$	γ	
A = 0		

Δημιουργία οντοτήτων εκτέλεσης – fork() για διεργασίες

```
...  
(α) x = fork();  
(β) if (x == 0)  
(γ)     κώδικας A;  
(δ) else  
(ε)     κώδικας B;  
(η) κώδικας Γ;  
...
```

```
/* παιδί *  
/* γονέας
```

```
int x = 2;  
  
int main() {  
    int y = 0;  
  
    y = 1;  
    if (fork())  
        x = x+y;  
    else  
        x = x-y;  
    printf(“%d\n”, x);  
    return 0;  
}  
  
Τι θα τυπωθεί;
```

```
void function() {  
    fork();  
    fork();  
    fork();  
}  
  
Πόσες θα φτιαχτούν;
```



Δημιουργία διεργασιών

- Αντιγράφονται τα πάντα:
 - Καθολικές μεταβλητές
 - Σωρός (ότι έχει γίνει malloc())
 - Ο κώδικας της διεργασίας (ίσως να μη χρειαστεί αυτή η αντιγραφή)
 - Στοίβα (στην κατάσταση που βρίσκεται εκείνη τη στιγμή)
 - Καταχωρητές (π.χ. program counter)
 - Άρα, η διεργασία παιδί εκτελεί αμέσως μετά το fork()
 - Άρα, τίποτε κοινό οι διεργασίες μεταξύ τους
- Ιδιαίτερα χρονοβόρο
 - Με τεχνικές όπως το copy-on-write μπορεί το ΛΣ να μην αντιγράψει τα πάντα (π.χ. τις καθολικές μεταβλητές) παρά μόνο όταν χρειαστεί (δηλαδή πότε??), γλιτώνοντας κάποιον χρόνο

Δημιουργία οντοτήτων εκτέλεσης – νήματα (posix)

- Διαφορετική λογική στα νήματα
 - Δεν δημιουργείται αντίγραφο από τίποτε
 - Δεν ξεκινάει η εκτέλεση του νήματος από το σημείο δημιουργίας του
 - Το νήμα θα εκτελέσει μια συνάρτηση που θα του δοθεί και θα τερματίσει

```
capitalize(char *strings[100]) {  
    pthread_create(&thrid, NULL, capfunc, (void*) string[5]);  
    ...  
}  
  
void *capfunc(void *str) {  
    ...  
}
```



Μετάφραση προγραμμάτων με νήματα

- Πάντα
 - `#include <pthread.h>`
- Μετάφραση:
 - `gcc -pthread <file.c>`
 - Παλαιότερα:
 - `gcc <file.c> -D_REENTRANT -lpthread`

Δημιουργία νημάτων

pthread_create(&thrid, attributes, function_to_call, function_parameter);

- Η συνάρτηση του νήματος παίρνει πάντα μόνο ένα όρισμα
- Το νήμα «ζει» μέχρι να τελειώσει (επιστρέψει) η συνάρτηση.
 - Φυσικά η συνάρτηση αυτή μπορεί να καλεί κι άλλες συναρτήσεις
 - Μόλις επιστρέψει το νήμα καταστρέφεται
- Ο γονέας (αρχικό νήμα), αφού δημιουργήσει το νήμα *συνεχίζει κανονικά την εκτέλεσή του*

```
void capitalize(char *strings[100]) {  
    for (i = 0; i < 100; i++)  
        pthread_create(&thrid, NULL, capfunc, (void*) string[i]);  
    ...  
}
```

```
int main() {  
    pthread_create(&thrid, NULL, func, NULL);  
    pthread_create(&thrid, NULL, func, NULL);  
    pthread_create(&thrid, NULL, func, NULL);  
}
```

Πόσα θα φτιαχτούν;



Πώς μπορώ να περάσω > 1 παραμέτρους;

```
struct manyparams { int x; int y; double z; };
void *threadfunc(void *ptr);

int main() {
    struct manyparams myargs;
    ...                               /* Pass a pointer to the structure */
    pthread_create(&thrid, NULL, threadfunc, (void*) &myargs);
    ...
}

void *threadfunc(void *ptr) {
    struct manyparams *s = ptr;      /* Cast the pointer */
    ...
    s->x += s->y;
    ...
}
```

Τερματισμός νήματος

- Δύο τρόποι:
 - είτε επιστρέφει από τη συνάρτηση που του δόθηκε να εκτελέσει
 - είτε καλεί την **pthread_exit(&returnvalue)** και τερματίζει αμέσως
 - σε οποιοδήποτε σημείο.
- Τιμή επιστροφής:
 - είτε αυτό που γίνεται return από τη συνάρτηση του νήματος είτε το όρισμα της `pthread_exit()`.
 - και στις δύο περιπτώσεις, είναι δείκτης στην τιμή αυτή (void *).

Αναμονή τερματισμού νήματος: pthread_join()

```
void *thrfunc(void *x) {  
    int id = (int) x;  
    printf("Hi! I am thread %d\n", id);  
    return (NULL);  
}
```

```
int main() {  
    pthread_t tids[5];  
    int i;  
  
    for (i = 0; i < 5; i++)  
        pthread_create(&tids[i], NULL, thrfunc, (void*) i);  
    for (i = 0; i < 5; i++)  
        pthread_join(tids[i], NULL);  
    printf("All threads done.\n");  
    return 0;  
}
```

Το cast αυτό θέλει
πολύ προσοχή!!



Παράδειγμα παράλληλης εκτέλεσης

```
#include <pthread.h>
char *strings[10];           /* 10 pointers to strings (shared) */

void *capitalize(void *str) {
    ...                       /* Capitalize all letters */
}

int main() {
    int i;
    pthread_t thrid[10];

    read_strings(strings);    /* Get the strings somehow */
    for (i = 1; i < 10; i++) /* Create 9 threads to process them */
        pthread_create(&thrid[i], NULL, capitalize, (void *) string[i]);
    capitalize((void *) string[0]); /* Participate, too! */
    ...
    for (i = 1; i < 10; i++) /* Wait for the 9 threads to finish */
        pthread_join(thrid[i], NULL);
    ...
}
```



Αμοιβαίος αποκλεισμός - κλειδαριές

- Από τους διάφορους τρόπους για αμοιβαίο αποκλεισμό, ο συχνότερα χρησιμοποιούμενος είναι οι κλειδαριές (**locks**).
- Η κλειδαριά είναι είτε *ανοιχτή* είτε *κλειδωμένη*.
- Όταν ένα νήμα κλειδώσει την κλειδαριά, οποιοδήποτε άλλο νήμα προσπαθήσει να την κλειδώσει θα αποτύχει και θα αναγκαστεί να περιμένει.
 - Όταν το νήμα ολοκληρώσει τη δουλειά του, ξεκλειδώνει την κλειδαριά και ένα από τα νήματα που περιμένουν θα κατορθώσει να την κλειδώσει.
- Επομένως, μία κρίσιμη περιοχή του προγράμματος μπορεί να προστατευθεί (αμοιβαίος αποκλεισμός) με μία κλειδαριά:

```
...  
lock(kleidaria);  
<critical section>  
unlock(kleidaria);  
...
```

Κλειδαριές στα threads: *mutexes*

```
int main() {  
    pthread_mutex_t mymutex;  
  
    pthread_mutex_init(&mymutex, NULL);  
    pthread_mutex_lock(&mymutex);  
    ...                               /* Κρίσιμη περιοχή */  
    pthread_mutex_unlock(&mymutex);  
    ...  
}
```

- Αντί για `pthread_mutex_init()`, μπορούμε να κάνουμε και **στατική** αρχικοποίηση (αρχικοποίηση χρειάζεται πάντα):
`pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;`

2 νήματα συνεργάζονται για τον μέσο όρο

```
int          A[10];          /* I want to calculate the average value */
double       avg = 0.0;      /* shared variable to hold the result */
pthread_mutex_t mx = PTHREAD_MUTEX_INITIALIZER;

void *threadfunc1(void *arg) {
    int i, q1 = 0;

    for (i = 0; i < 5; i++)
        q1 += A[i];
    pthread_mutex_lock(&mx);
    avg = avg + (q1 / 10.0);    /* critical section */
    pthread_mutex_unlock(&mx);
    ...
}

void *threadfunc2(void *arg) {
    int i, q2 = 0;

    for (i = 5; i < 10; i++)
        q2 += A[i];
    pthread_mutex_lock(&mx);
    avg = avg + (q2 / 10.0);    /* critical section */
    pthread_mutex_unlock(&mx);
    printf("avg = %lf\n", avg); /* show result */
    ...
}
```

Συγχρονισμός!

- *Race conditions*: συνθήκες συναγωνισμού, όπου το τελικό αποτέλεσμα εξαρτάται από τις σχετικές ταχύτητες των νημάτων
- Οι οντότητες εκτέλεσης πρέπει μερικές φορές να *συγχρονίζονται*.
 - Θα πρέπει να περιμένουν μέχρι να συμβεί κάποιο γεγονός
- Συνήθεις μηχανισμοί στα νήματα:
 - Μεταβλητές συνθήκης (***condition variables***) – ο βασικός μηχανισμός
 - φράγματα (***barriers***) – ο απλούστερος / δημοφιλέστερος μηχανισμός
 - κάποιο νήμα που καλεί μία συνάρτηση φράγματος, μπλοκάρει και περιμένει όλα τα υπόλοιπα νήματα να φτάσουν στο φράγμα *πριν προχωρήσει παρακάτω*.
 - Μόλις φτάσει και το τελευταίο, «ξεμπλοκάρουν» όλα και συνεχίζουν την εκτέλεσή τους.
 - `pthread_barrier_wait()`
 - Δεν είναι μέρος του «κλασικού» στάνταρ. Αποτελεί επέκταση. Για να χρησιμοποιηθεί στα συστήματα που το υποστηρίζουν, πρέπει στον κώδικα να μπει `#define _XOPEN_SOURCE 600`

Μεταβλητές συνθήκης

- Ο βασικός μηχανισμός συγχρονισμού στα νήματα POSIX.
- Ένα νήμα θα περιμένει σε μια μεταβλητή συνθήκης έως ότου η μεταβλητή το ενημερώσει ότι μπορεί να συνεχίσει
 - Κάποιο άλλο νήμα σηματοδοτεί τη μεταβλητή συνθήκης, επιτρέποντας άλλα νήματα να συνεχίσουν
 - Κάθε μεταβλητή συνθήκης, ως διαμοιραζόμενο δεδομένο, συσχετίζεται και με ένα συγκεκριμένο mutex
- Με τις μεταβλητές συνθήκης αποφεύγεται ο συνεχής έλεγχος (busy waiting) της κατάστασης των δεδομένων
 - Ένα νήμα που τροποποιεί την τιμή των δεδομένων ειδοποιεί τα ενδιαφερόμενα μέσω μιας μεταβλητής συνθήκης
- Παράδειγμα: Σχήμα παραγωγού – καταναλωτή με χρήση ουράς δεδομένων
 - Το νήμα-καταναλωτής εξάγει και επεξεργάζεται στοιχεία της ουράς εφόσον αυτή δεν είναι άδεια, διαφορετικά μπλοκάρει
 - Το νήματα-παραγωγός προσθέτει στοιχεία στην ουρά και ειδοποιεί με τη μεταβλητή συνθήκης το νήμα – καταναλωτή που περιμένει

Χρήση

- Στατική αρχικοποίηση:
 - `pthread_cond_t condition = PTHREAD_COND_INITIALIZER;`
- Δυναμική αρχικοποίηση:
 - `pthread_cond_init(&condition);`
- Μια μεταβλητή συνθήκης συσχετίζεται πάντοτε με ένα mutex
- Αναμονή με:
 - `pthread_cond_wait(&condition, &mutex);`
 - Το νήμα αναστέλλει την εκτέλεση του μέχρι να σηματοδοτηθεί η μεταβλητή συνθήκης
 - Το νήμα πρέπει να έχει ήδη κλειδώσει το συσχετιζόμενο mutex
 - Το νήμα αναστέλλεται ενώ το mutex ξεκλειδώνεται αυτόματα ώστε να επιτραπεί η χρήση του από άλλα νήματα
 - Όταν το νήμα ενεργοποιηθεί με την ειδοποίηση του μέσω της μεταβλητής συνθήκης, αυτόματα το mutex είναι κλειδωμένο από το ίδιο

Σηματοδότηση

```
pthread_cond_signal(&condition);
```

```
pthread_cond_broadcast(&condition);
```

- Με την `pthread_cond_signal` ένα νήμα ειδοποιεί κάποιο άλλο νήμα που περιμένει σε κάποια μεταβλητή συνθήκης
- Με την `pthread_cond_broadcast`, ένα νήμα ειδοποιεί όλα τα νήματα που περιμένουν σε κάποια μεταβλητή συνθήκης
- Το νήμα που πρόκειται να σηματοδοτήσει μια μεταβλητή συνθήκης συνήθως έχει κλειδώσει το `mutex` που συσχετίζεται με αυτή. Στην περίπτωση αυτή πρέπει να ελευθερώσει το `mutex` ώστε να συνεχιστεί η εκτέλεση της `pthread_cond_wait`

Διαδικασία

- Υπάρχει κάποια συνθήκη ή κάποιο γεγονός που θέλουμε να ελέγξουμε (έστω π.χ. εάν `count >= N`).
 - Απαιτείται μία `condition variable` (`cond`) και μια κλειδαριά (`mut`)
- Ο έλεγχος του γεγονότος γίνεται μόνο αφού πρώτα το νήμα κλειδώσει το `mut`.
 - Αν το γεγονός/συνθήκη ισχύει (π.χ. `count >= N`) τότε
 - Το νήμα ξεκλειδώνει το `mut` και
 - Συνεχίζει την εκτέλεσή του
- Αν το γεγονός/συνθήκη δεν ισχύει τότε
 - Το νήμα αναστέλλεται καλώντας την `pthread_cond_wait(&cond, &mut)`
 - το `mut` ξεκλειδώνεται αυτόματα από το σύστημα
- Κάποιο άλλο νήμα θα καλέσει την `pthread_cond_signal(&cond)` όταν η συνθήκη γίνει αληθής
 - Το πρώτο νήμα ξυπνάει από την αναμονή έχοντας το `mutex` αυτόματα κλειδωμένο και εκτελεί τη δουλειά του
 - Το νήμα ξεκλειδώνει το `mutex` όταν έχει ολοκληρώσει

Παράδειγμα 2 νημάτων που συγχρονίζονται με μετ. συνθ.

```
pthread_cond_t  condvar = PTHREAD_COND_INITIALIZER;  
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;  
int             flag;
```

```
void *threadA(void *arg) {  
    ...  
    pthread_mutex_lock(&lock);  
    while (flag != TRUE) /* waiting for flag to become true */  
        pthread_cond_wait(&condvar, &lock);  
    pthread_mutex_unlock(&lock);  
    ...  
}
```

```
void *threadB(void *arg) {  
    ...  
    pthread_mutex_lock(&lock);  
    flag = TRUE;  
    pthread_cond_signal(&condvar);  
    pthread_mutex_unlock(&lock);  
    ...  
}
```



Παράδειγμα αναμονής 1 νήματος

```
/* N threads call this */
void *producers(void *arg) {
    ...
    pthread_mutex_lock(&lock);
    count++;
    if (count == N)
        pthread_cond_signal(&condvar);
    pthread_mutex_unlock(&lock);
    ...
}

void *consumer(void *arg) {
    ...
    pthread_mutex_lock(&lock);
    while (count < N)
        pthread_cond_wait(&condvar,&lock);
    pthread_mutex_unlock(&lock);
    ...
}
```



Παράδειγμα αναμονής N νημάτων

```
void *producer(void *arg) {
    ...
    pthread_mutex_lock(&lock);
    count++;
    if (count == N)
        pthread_cond_broadcast(&condvar);
    pthread_mutex_unlock(&lock);
    ...
}

/* N threads call this */
void *consumers(void *arg) {
    ...
    pthread_mutex_lock(&lock);
    while (count < N)
        pthread_cond_wait(&condvar,&lock);
    pthread_mutex_unlock(&lock);
    ...
}
```



Ορθός τρόπος χρήσης

```
/* THREAD A
 * Wait for the flag to become TRUE
 */
pthread_mutex_lock(&lock);
while (flag == FALSE)
    pthread_cond_wait(&condvar, &lock);
pthread_mutex_unlock(&lock);
```

```
/* THREAD B
 */
if (something_happens) {
    pthread_mutex_lock(&lock);
    flag = TRUE;
    pthread_cond_signal(&condvar);
    pthread_mutex_unlock(&lock);
}
```

- Γιατί η κλειδαριά;
 - Διότι το `pthread_cond_signal()` είναι *memoryless*: αν ένα *signal* σταλεί αλλά δεν υπάρχει κάποιο νήμα που κάνει `wait()`, τότε το σήμα χάνεται.
 - Αν αμέσως μετά το `while` και πριν προλάβει να κάνει `wait()`, το νήμα B κάνει `signal()`, τότε το σήμα θα χαθεί και το νήμα A θα περιμένει για πάντα.
- Γιατί το `while`;
 - Διότι μπορεί μετά το `flag = TRUE` να ξυπνήσει το νήμα A, αλλά να πρόλαβε ένα άλλο νήμα να το έκανε πάλι `FALSE`. Επίσης μπορεί μερικές φορές κάποιο νήμα να ξυπνήσει από το `wait()` απρόσμενα,
 - οπότε πριν προχωρήσει θα πρέπει να ξαναελέγξει αν όντως το `flag` είναι `TRUE`.



Δεύτερος μηχανισμός: barriers

- Φράγματα (*barriers*) – ο απλούστερος / δημοφιλέστερος μηχανισμός
- Κάποιο νήμα που καλεί μία συνάρτηση φράγματος, μπλοκάρει και περιμένει όλα τα υπόλοιπα νήματα να φτάσουν στο φράγμα πριν προχωρήσει παρακάτω.
- Μόλις φτάσει και το τελευταίο, «ξεμπλοκάρουν» όλα και συνεχίζουν την εκτέλεσή τους.
- **pthread_barrier_wait()**
 - Δεν είναι μέρος του «κλασικού» στάνταρ. Αποτελεί επέκταση (τα λεγόμενα real-time extensions).
 - Για να χρησιμοποιηθεί στα συστήματα που το υποστηρίζουν, παλαιότερα έπρεπε στον κώδικα του χρήστη να μπει `#define _XOPEN_SOURCE 600`
 - Στα περισσότερα συστήματα παρέχεται πλέον εγγενώς.

Χρήση barriers

- Ορισμός:
`pthread_barrier_t bar;`
- Δυναμική αρχικοποίηση:
`pthread_barrier_init(&bar, NULL, N);`
 - Στην αρχικοποίηση δίνεται το πλήθος των νημάτων (N) που θα πρέπει να συγχρονίσει
- Αναμονή με:
`pthread_barrier_wait(&bar);`
 - Το νήμα αναστέλλει την εκτέλεση του μέχρι τα υπόλοιπα $N-1$ νήματα να κάνουν την ίδια κλήση.
- Τον χρησιμοποιούμε όσες φορές θέλουμε χωρίς νέα αρχικοποίηση. Επίσης, νέα αρχικοποίηση με διαφορετικό N δεν επιτρέπεται!
 - Επομένως, αν θέλουμε να τον επαναχρησιμοποιήσουμε για να συγχρονίσουμε διαφορετικό πλήθος νημάτων, πρέπει πρώτα να τον «καταστρέψουμε» με:
`pthread_barrier_destroy(&bar);`
 - Το νήμα αναστέλλει την εκτέλεση του μέχρι τα υπόλοιπα $N-1$ νήματα να κάνουν την ίδια κλήση.

2 νήματα συνεργάζονται για τον μέσο όρο ΣΩΣΤΑ!

```
int          A[10];          /* I want to calculate the average value */
double       avg = 0.0;      /* shared variable to hold the result */
pthread_mutex_t mx = PTHREAD_MUTEX_INITIALIZER;
pthread_barrier_t bar;      /* Initialized in main() */

void *threadfunc1(void *arg) {
    int i, q1 = 0;

    for (i = 0; i < 5; i++)
        q1 += A[i];
    pthread_mutex_lock(&mx);
    avg = avg + (q1 / 10.0);    /* critical section */
    pthread_mutex_unlock(&mx);
    pthread_barrier_wait(&bar);
    ...
}

void *threadfunc2(void *arg) {
    int i, q2 = 0;

    for (i = 5; i < 10; i++)
        q2 += A[i];
    pthread_mutex_lock(&mx);
    avg = avg + (q2 / 10.0);    /* critical section */
    pthread_mutex_unlock(&mx);
    pthread_barrier_wait(&bar);
    printf("avg = %lf\n", avg); /* show result */
    ...
}
```

Θέματα υλοποίησης αμοιβαίου αποκλεισμού /συγχρονισμού

- Όταν ένα νήμα βρει την κλειδαριά κλειδωμένη τι γίνεται;
 - **Λύση 1 (queue locks)**: το σύστημα το αποσύρει σε μία ουρά και το «κοιμίζει» μέχρι να ξεκλειδωθεί η κλειδαριά και να το «ξυπνήσει»
 - Ελευθερώνεται ένας πυρήνας / επεξεργαστής (ώστε να μπορεί να εκτελέσει άλλα χρήσιμα πράγματα)
 - Ευγενική / καλή / δίκαια συμπεριφορά
 - Χρονοβόρο! Απόσυρση-κοίμισμα-προετοιμασία-επαναφορά έχει κόστος διαχείρισης!
 - **Λύση 2 (spin locks)**: ενεργός αναμονή (busy waiting)
 - «Μονοπωλείται» ο πυρήνας σε ένα while loop (ενώ θα μπορούσε να εκτελέσει άλλα χρήσιμα πράγματα)
 - Κακή συμπεριφορά, αλλά ταχύτατο 😊
 - **Το προτιμάμε, με την προϋπόθεση ότι οι κρίσιμες περιοχές είναι μικρές σε διάρκεια και ότι το σύστημα είναι «αφιερωμένο» στη δική μας εφαρμογή.**
- Αντίστοιχα και ο συγχρονισμός – μπορεί να υλοποιηθεί με ουρές αναμονής ή ενεργό αναμονή
- Κλειδαριές + συγχρονισμός = βασικές αιτίες καθυστέρησης στην παράλληλη εκτέλεση



... βάζοντάς τα όλα σε εφαρμογή

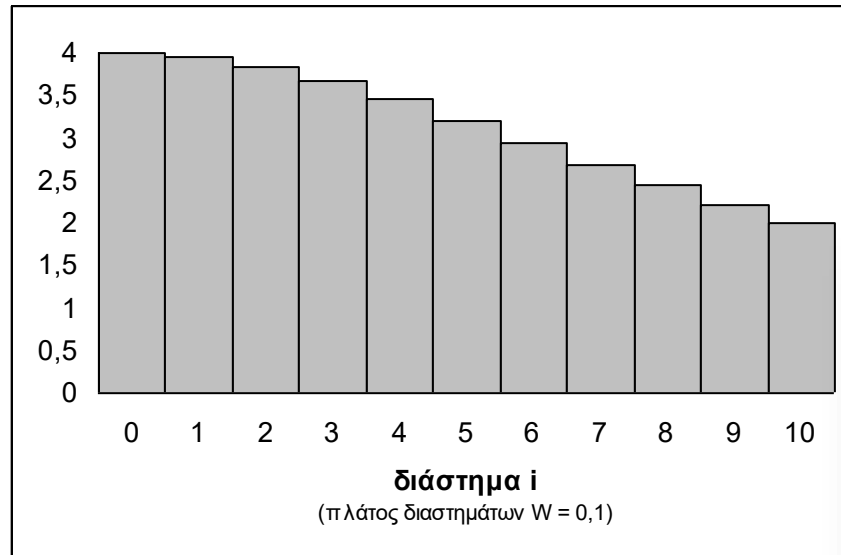


ΜΥΕΟ23
Παράλληλα
Συστήματα &
Προγραμματισμός

Υπολογισμός του
 $\pi = 3,14\dots$

- Αριθμητική ολοκλήρωση

$$\pi = \int_0^1 \frac{4}{1+x^2}$$



$$\approx \sum_{i=0}^{N-1} \frac{4W}{1 + [(i + \frac{1}{2})W]^2}$$

```
#define N 1000000
double pi = 0.0, W = 1.0/N;

int main() {
    int i;
    for (i = 0; i < N; i++)
        pi += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
    printf("pi = %.10lf\n", pi);
    return 0;
}
```

Υπολογισμός του $\pi = 3,14\dots$ με 1 βήμα ανά επανάληψη

```
#define N 100000
double pi = 0.0, W = 1.0/N;

int main() {
    int i;
    for (i = 0; i < N; i++)
        pi += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
    printf("pi = %.10lf\n", pi);
    return 0;
}
```

```
/* 1 βήμα ανά επανάληψη */
#define N 100000
double pi = 0.0, W = 1.0/N;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *thrfunc(void *iter) {
    int i = (int) iter;
    pthread_mutex_lock(&lock);
    pi += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
    pthread_mutex_unlock(&lock);
}

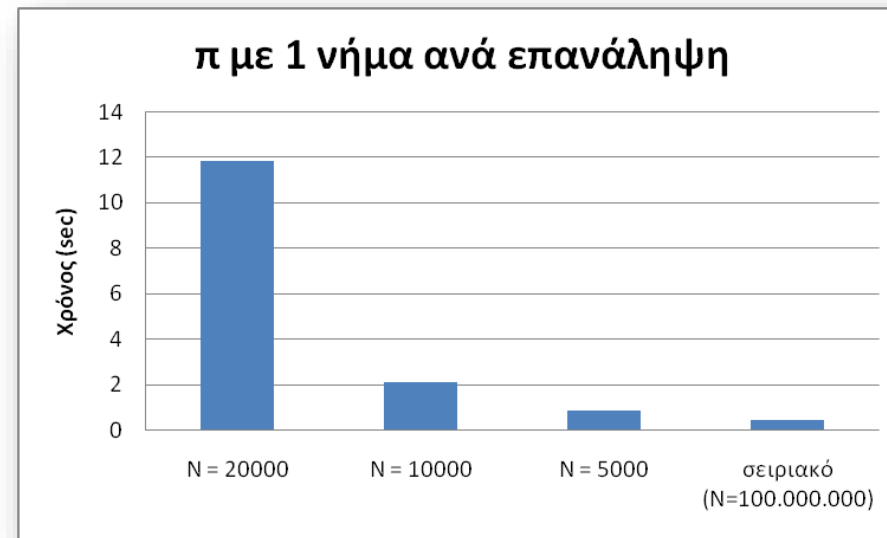
int main() {
    int i;
    pthread_t tids[N];          /* Remember the thread ids */

    for (i = 0; i < N; i++)
        pthread_create(&tids[i], NULL, thrfunc, (void *) i);
    for (i = 0; i < N; i++)
        pthread_join(tids[i], NULL);
    printf("pi = %.10lf\n", pi);
    return 0;
}
```



Προβλήματα

- Υπερβολικά πολλά νήματα (πολλά συστήματα δεν επιτρέπουν πάνω από λίγες χιλιάδες)
 - και επομένως υπερβολικά «λεπτόκοκκος» παραλληλισμός (fine grain)
- Ακόμα και αν επιτρέπονταν τόσα πολλά νήματα, ο συναγωνισμός για την κλειδαριά είναι τεράστιος.
- Αποτέλεσμα: αργή εκτέλεση και μόνο για μη ακριβή προσέγγιση του π.



- Μάθημα: **η καλύτερη τακτική είναι συνήθως να δημιουργούμε τόσα νήματα όσοι είναι και οι επεξεργαστές του συστήματος**

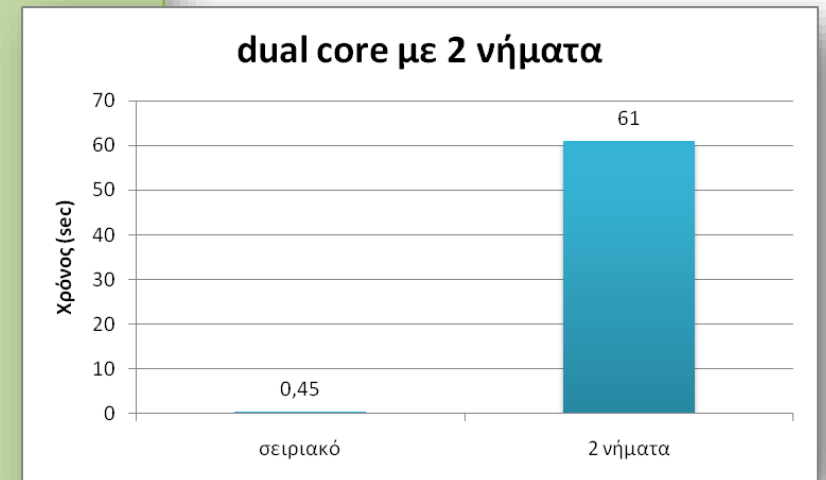
Μοίρασμα της δουλειάς σε λίγα νήματα

```
#define NPROCS 2          /* dual core */
#define N 10000000      /* Για ακρίβεια (ίδια με σειριακό) */
#define WORK N/NPROCS
double pi = 0.0, W = 1.0/N;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *thrfunc(void *iter) {
    int i, me = (int) iter;
    for (i = me*WORK; i < (me+1)*WORK; i++) {
        pthread_mutex_lock(&lock);
        pi += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
        pthread_mutex_unlock(&lock);
    }
}

int main() {
    int i;
    pthread_t tids[NPROCS];

    for (i = 0; i < NPROCS; i++) /* νήματα = # επεξεργατών */
        pthread_create(&tids[i], NULL, thrfunc, (void *) i);
    for (i = 0; i < NPROCS; i++)
        pthread_join(tids[i], NULL);
    printf("pi = %.10lf\n", pi);
    return 0;
}
```



Τι πάει στραβά;

Λίγα νήματα – αποφυγή συναγωνισμού

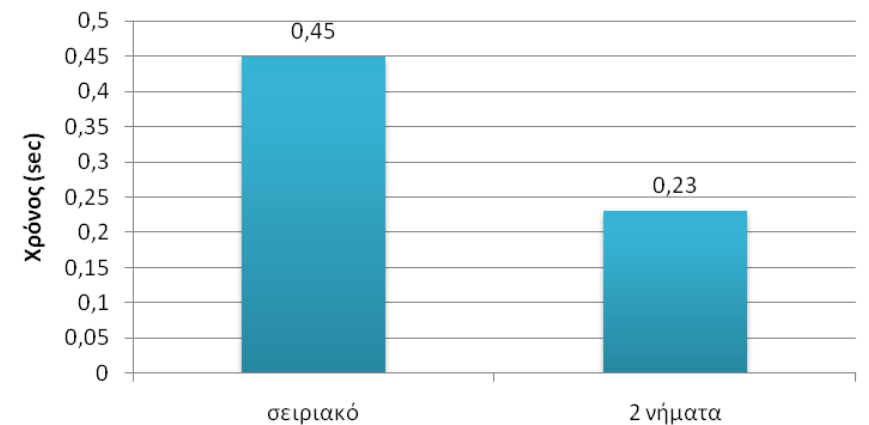
```
#define NPROCS 2          /* dual core */
#define N 10000000      /* Για ακρίβεια (ίδια με σειριακό) */
#define WORK N/NPROCS
double pi = 0.0, W = 1.0/N;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *thrfunc(void *iter) {
    int i, me = (int) iter;
    double mysum = 0.0;
    for (i = me*WORK; i < (me+1)*WORK; i++)
        mysum += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
    pthread_mutex_lock(&lock);
    pi += mysum;
    pthread_mutex_unlock(&lock);
}

int main() {
    int i;
    pthread_t tids[NPROCS];

    for (i = 0; i < NPROCS; i++) /* νήματα = # επεξεργατών */
        pthread_create(&tids[i], NULL, thrfunc, (void *) i);
    for (i = 0; i < NPROCS; i++)
        pthread_join(tids[i], NULL);
    printf("pi = %.10lf\n", pi);
    return 0;
}
```

dual core με 2 νήματα



Δρομολόγηση επαναλήψεων με άλλατα

```
#define NPROCS 2          /* dual core */
#define N 10000000      /* Για ακρίβεια (ίδια με σειριακό) */
#define WORK N/NPROCS
double pi = 0.0, W = 1.0/N;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *thrfunc(void *iter) {
    int i, me = (int) iter;
    double mysum = 0.0;
    for (i = me*WORK; i < (me+1)*WORK; i++)
        mysum += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
    pthread_mutex_lock(&lock);
    pi += mysum;
    pthread_mutex_unlock(&lock);
}

int main() {
    int i;
    pthread_t tids[NPROCS];

    for (i = 0; i < NPROCS; i++) /* νήματα = # επεξεργατών */
        pthread_create(&tids[i], NULL, thrfunc, (void *) i);
    for (i = 0; i < NPROCS; i++)
        pthread_join(tids[i], NULL);
    printf("pi = %.10lf\n", pi);
    return 0;
}
```

```
void *thrfunc(void *iter) {
    int i, me = (int) iter;
    double mysum = 0.0;
    for (i = me; i < N; i += NPROC)
        mysum += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
    pthread_mutex_lock(&lock);
    pi += mysum;
    pthread_mutex_unlock(&lock);
}
```

Γνωστό και ως “loop splitting” (ατυχές)
Πλεονεκτήματα / μειονεκτήματα;

Πίνακας επί πίνακα (τετραγωνικοί)

```
for (i = 0; i < N; i++) {  
  for (j = 0; j < N; j++) {  
    for (k = sum = 0; k < N; k++)  
      sum += A[i][k]*B[k][j];  
    C[i][j] = sum;  
  }  
}
```

- Παραλληλοποίηση:
 - Ενός βρόχου (i)
 - μοίρασμα των επαναλήψεων του πρώτου for σε νήματα
 - Δεν δουλεύει καλά πάντα, π.χ. τι γίνεται αν # επεξεργαστών > N ??
 - Δύο βρόχων (i και j)
 - *Checkerboard partitioning*: παραλληλοποίηση και των δύο βρόχων
 - Μπορώ εναλλακτικά να παραλληλοποιήσω τον βρόχο j και k?

Διαχωρισμός σκακιέρας

c_{00}	c_{01}	...	$c_{0(S-1)}$	c_{0S}						...	$c_{0(N-1)}$	
c_{10}	c_{11}	...	$c_{1(S-1)}$	c_{1S}						...	$c_{1(N-1)}$	
c_{20}	c_{21}	C_{00}	...	$c_{2(S-1)}$	c_{2S}	C_{01}		...		$C_{0(M-1)}$...	$c_{2(N-1)}$
c_{S0}	c_{S1}	...	$c_{S(S-1)}$	c_{SS}						...	$c_{S(N-1)}$	
		C_{10}				C_{11}		...		$C_{1(M-1)}$		
		⋮				⋮		⋮		⋮		
		$C_{(M-1)0}$				$C_{(M-1)1}$				$C_{(M-1)(M-1)}$		
$c_{(N-1)0}$	$c_{(N-1)1}$				$c_{(N-1)S}$...				

Υποπίνακες
διάστασης $S \times S$

Άρα
 $M = N/S$
υποπίνακες
οριζόντια /
κάθετα:
 M^2 υποπίνακες



Διαχωρισμός σκακιέρας – αρίθμηση υποπινάκων

C_{00}	C_{01}		$C_{0(M-1)}$
C_{10}	C_{11}		$C_{1(M-1)}$
		C_{xy}	
$C_{(M-1)0}$	$C_{(M-1)1}$		$C_{(M-1)(M-1)}$

Αν τους
αριθμήσουμε
από 0 έως M^2-1 ,
ο i -οστός
υποπίνακας
είναι ο C_{xy} όπου:

$$x = i / M$$

$$y = i \% M$$



Διαχωρισμός σκακιέρας – το πρόγραμμα

```
#define N    1000    /* matrices 1000x1000 */
#define NTHR 25     /* # threads */
#define M    5      /* M*M submatrices in total */
#define S    N/M    /* size of each submatrix */
double A[N][N], B[N][N], C[N][N];

void *checker(void *arg) {
    int    i, j, k, x, y, me = (int) arg;
    double sum;

    x = me / M;
    y = me % M;
    for (i = x*S; i < (x+1)*S; i++) { /* calculate Cxy */
        for (j = y*S; j < (y+1)*S; j++) {
            for (k = 0, sum = 0.0; k < N; k++)
                sum += A[i][k]*B[k][j];
            C[i][j] = sum;
        }
    }
}

int main() {
    int i;
    pthread_t thr[NTHR];

    for (i = 0; i < NTHR; i++)
        pthread_create(&thr[i], NULL, checker, (void *) i);
    for (i = 0; i < NTHR; i++)
        pthread_join(thr[i], NULL);
    show_result(C, N);
    return 0;
}
```

Δεν υπάρχει ανάγκη αμοιβαίου αποκλεισμού

Embarrassingly parallel

Πίνακας επί διάνυσμα – πιο απλό?

```
double A[N][N], v[N], res[N];

for (i = 0; i < N; i++) {
    res[i] = 0.0;
    for (j = 0; j < N; j++)
        res[i] += A[i][j]*v[j];
}
```

- Παραλληλοποίηση του βρόχου i , μοιράζοντας τις επαναλήψεις στα νήματα

Πίνακας επί διάνυσμα – παράλληλα (I)

```
int main() {
    int i;
    pthread_t tids[NPROCS];

    for (i = 0; i < NPROCS; i++)
        pthread_create(&tids[i], NULL, thrfunc, (void *) i);
    for (i = 0; i < NPROCS; i++)
        pthread_join(tids[i], NULL);
    return 0;
}
```

```
#define N      8000
#define NPROCS 8
#define RPT    N/NPROCS /* "Rows Per Thread" */
double  A[N][N], v[N], res[N];

void *thrfunc(void *arg) {
    int    i, j, myid = (int) arg;

    for (i = myid*RPT; i < (myid+1)*RPT; i++) {
        res[i] = 0.0;
        for (j = 0; j < N; j++)
            res[i] += A[i][j]*v[j];
    }
}
```

- Καλός κώδικας, όταν $NPROC < N$.
 - Τι γίνεται όταν, όμως, έχουμε $NPROC > N$?
- Η παραλληλοποίηση του βρόχου, τότε, δεν θα δουλεύει καλά.
 - Θα υπάρχουν διεργασίες που δεν κάνουν τίποτε!
 - παραλληλοποίηση και των δύο βρόχων (i και j) μαζί

Πίνακας επί διάνυσμα – παράλληλα (II)

```
#define N      100      /* vector size */
#define NPROC 200      /* #cores = #threads */
#define TPR   NPROC/N  /* threads-per-row */
double  A[N][N], v[N], res[N];
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *matvec(void *arg) {
    int    i, j, me = (int) arg;
    double sum = 0.0;

    i = me / TPR;      /* my row */
    for (j = me % TPR; j < N; j += TPR)
        sum += A[i][j]*v[j];
    pthread_mutex_lock(&lock);
    res[i] += sum;
    pthread_mutex_unlock(&lock);
}
```

```
int main() {
    int i;
    pthread_t thr[NPROC];

    for (i = 0; i < N; i++)
        res[i] = 0.0;
    for (i = 0; i < NPROC; i++)
        pthread_create(&thr[i], NULL, matvec, (void *) i);
    for (i = 0; i < NPROC; i++)
        pthread_join(thr[i], NULL);
    show_result(res, N);
    return 0;
}
```

- Υποθέτουμε ότι το N διαιρεί το NPROC
- Απαιτείται, πλέον, αμοιβαίος αποκλεισμός.
- Προσοχή στην αρχικοποίηση του res[].

Παράλληλη αρχικοποίηση;;

Κάθε ομάδα νημάτων να αρχικοποιεί το δικό της res[i]

Π.χ. το πρώτο νήμα (me%TPR==0) να κάνει res[i] = 0.0.

Υπάρχει κάποιο πρόβλημα;

Συγχρονισμός: σωστή παράλληλη αρχικοποίηση

```
#define N      100      /* vector size */
#define NPROC 200      /* #cores = #threads */
#define TPR   NPROC/N  /* threads-per-row */
double  A[N][N], v[N], res[N];
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_barrier_t bar;

void *matvec(void *arg) {
    int    i, j, me = (int) arg;
    double sum = 0.0;

    i = me / TPR;          /* my row */
    if (me % TPR == 0)    /* i will initialize */
        res[i] = 0;
    pthread_barrier_wait(&bar); /* make sure all wait here */
    for (j = me % TPR; j < N; j += TPR)
        sum += A[i][j]*v[j];
    pthread_mutex_lock(&lock);
    res[i] += sum;
    pthread_mutex_unlock(&lock);
}
```

```
int main() {
    int i;
    pthread_t thr[NPROC];

    pthread_barrier_init(&bar, NULL, NPROC);
    for (i = 0; i < NPROC; i++)
        pthread_create(&thr[i], NULL, matvec, (void *) i);
    for (i = 0; i < NPROC; i++)
        pthread_join(thr[i], NULL);
    show_result(res, N);
    return 0;
}
```

Όλα τα νήματα στον ίδιο barrier! ☹️

Άλλη ιδέα;

Το πρώτο νήμα μιας ομάδας να δημιουργεί τα υπόλοιπα, αμέσως μετά την αρχικοποίηση (άρα => παράλληλη δημιουργία νημάτων)

Δύο ακόμα θέματα με τα νήματα

- Πώς μπορώ να εξασφαλίσω ότι κάτι θα γίνει από μόνο ένα νήμα;
 - Π.χ. Έχουν ήδη δημιουργηθεί τα νήματα και θέλω να αρχικοποιήσω μία κοινή μεταβλητή.
- Πώς μπορώ να έχω ιδιωτικές global μεταβλητές στα νήματα;
 - Π.χ. Θέλω να έχω μία μεταβλητή global ώστε να μην την περνάω από συνάρτηση σε συνάρτηση, αλλά δεν θέλω να είναι κοινή μεταξύ των νημάτων.
 - “thread local storage” - TLS

pthread_once: Εκτέλεση μιας φοράς

```
pthread_once_t once_block = PTHREAD_ONCE_INIT
pthread_mutex_t mutex;

/* It is guaranteed that this will be called only once */
void routine(void) {
    pthread_mutex_init(&mutex, NULL);
}

/* Thread */
void *threadfunc(void *arg) {
    pthread_once(&once_block, routine);    /* Κλήση ρουτίνας */
    ...
}

int main() {
    pthread_create(&t, NULL, threadfunc, NULL); /* Νέο νήμα */
    pthread_once(&once_block, routine);    /* Κλήση ρουτίνας */
    ...
}
```

pthread specifics: Ιδιωτικά (Καθολικά) Δεδομένα Νημάτων

- Τρόπος 1^{ος} (εύκολος, γρήγορος, όμως δουλεύει μόνο με ορισμένους compilers και μόνο σε συγκεκριμένα λειτουργικά συστήματα κλπ)

```
__thread int x; /* Global, but each thread has its own copy of the variable */
```

- Τρόπος 2^{ος} (POSIX, portable): thread-specifics

```
pthread_key_t key;
```

```
main() {
```

```
    ...
```

```
    pthread_key_create(&key, NULL); /* Initialize the key -- should be done  
    once! */
```

```
    pthread_create(&t, NULL, thread_routine, ...);
```

```
}
```

```
void *thread_routine(void *) {
```

```
    long *value;
```

```
    mydata = malloc(sizeof(long)); /* Allocate space */
```

```
    *mydata = ...;
```

```
    pthread_setspecific(key, value); /* Store *my* data at this key */
```

```
    ...
```

```
    mydata = pthread_getspecific(key); /* Get *my* data */
```

```
}
```



Δυναμική συμπεριφορά

- Η μέχρι τώρα διάσπαση σε «εργασίες» ήταν *στατική* δηλαδή είχαμε προκαθορίσει *ακριβώς* τι θα εκτελέσει το κάθε νήμα.
 - π.χ. στο π, ή ο διαχωρισμός σκακιέρας στον πολ/μο πινάκων
- Ως γνώμονα είχαμε την *ισοκατανομή φόρτου* ώστε όλα τα νήματα να εκτελέσουν παρόμοιο έργο και άρα να δουλέψουν για ίδιο χρονικό διάστημα (που είναι το ιδανικό).
- Τι γίνεται όμως αν κάποιος επεξεργαστής
 - Είναι πιο αργός από τους άλλους ή
 - Για το συγκεκριμένο διάστημα είναι περισσότερο φορτωμένος από τους άλλους (διότι π.χ. εκτελούν και κάποια άλλη εφαρμογή)
- Σε αυτή την περίπτωση, η ισόποση διαμοίραση των εργασιών **ΔΕΝ ΕΙΝΑΙ ΟΤΙ ΚΑΛΥΤΕΡΟ!**

Αυτοδρομολόγησ η (self-scheduling)

- Εφαρμόζεται σε αυτές τις περιπτώσεις.
- Δυναμικά (δηλαδή κατά την ώρα της εκτέλεσης), τα πιο «αργά» νήματα θα εκτελέσουν λιγότερο έργο.
- Πώς;
 - Δεν προκαθορίζουμε τι θα εκτελέσει το κάθε νήμα
 - Αφήνουμε κάθε νήμα να ζητάει δουλειά να κάνει
 - Όσο πιο γρήγορα τελειώσει μία δουλειά, τόσες περισσότερες δουλειές θα πάρει να εκτελέσει

```
while (there-are-things-to-calculate)
{
    Task = get-the-next-task()
    execute(Task);
}
```

Αυτοδρομολόγηση η – κώδικας

- Έστω ότι κάπως έχουμε χωρίσει τη δουλειά σε NTASK εργασίες, από 0 έως NTASK-1.
- Έστω ότι η t-οστή εργασία εκτελείται ως `taskexecute(t)`

```
int taskid = 0;    /* the next task id to execute */
pthread_mutex_t tlock = PTHREAD_MUTEX_INITIALIZER;

void *thrfunc(void *arg) {
    int t;

    while (1) {    /* forever */
        pthread_mutex_lock(&tlock);
        t = taskid++; /* get next task */
        pthread_mutex_unlock(&tlock);
        if (t >= NTASK)
            break;    /* all tasks done */
        taskexecute(t);
    }
}
```



Αυτοδρομολόγηση, συνέχεια

- Για οποιαδήποτε εφαρμογή, ΑΚΡΙΒΩΣ το ίδιο πρόγραμμα
- Αρκεί μόνο να θέσουμε το NTASK σε σωστή τιμή και να υλοποιήσουμε τη κατάλληλη `taskexecute()`
- Τα ταχύτερα νήματα «κλέβουν» τις πιο πολλές εργασίες και άρα καλύτερη κατανομή φόρτου
- Στη γενικότερη μορφή της, υπάρχει μία *ουρά* από εργασίες που πρέπει να εκτελεστούν.
 - Τα νήματα «τραβούν» εργασίες από την κεφαλή της ουράς
- Αν δεν υπάρχει μεγάλη διαφορά στις ταχύτητες (π.χ. το σύστημα εκτελεί μόνο τη δική μας εφαρμογή), η αυτοδρομολόγηση δεν είναι πάντα ότι καλύτερο μιας και έχει συναγωνισμό για την κλειδαριά (ή την πρόσβαση στην ουρά), για κάθε εργασία.

Παράδειγμα: υπολογισμός του π με αυτοδρομο/ση

```
int taskid = 0;    /* the next task id to execute */
pthread_mutex_t tlock = PTHREAD_MUTEX_INITIALIZER;

void *thrfunc(void *arg) {
    int t;

    while (1) {    /* forever */
        pthread_mutex_lock(&tlock);
        t = taskid++; /* get next task */
        pthread_mutex_unlock(&tlock);
        if (t >= NTASK)
            break;    /* all tasks done */
        taskexecute(t);
    }
}
```

```
#define N 10000000    /* # iterations */
#define K 100        /* iterations per task */
#define NTASK N/K    /* total # tasks */
double pi = 0.0, W = 1.0/N;
pthread_mutex_t pilock = PTHREAD_MUTEX_INITIALIZER;

void taskexecute(int t) {
    int i;
    double mysum = 0.0;

    for (i = t*K; i < (t+1)*K; i++)
        mysum += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
    pthread_mutex_lock(&pilock);
    pi += mysum;
    pthread_mutex_unlock(&pilock);
}
```



Παράδειγμα: checkerboard + αυτοδρομολόγηση

```
int taskid = 0;    /* the next task id to execute */
pthread_mutex_t tlock = PTHREAD_MUTEX_INITIALIZER;

void *thrfunc(void *arg) {
    int t;

    while (1) {    /* forever */
        pthread_mutex_lock(&tlock);
        t = taskid++; /* get next task */
        pthread_mutex_unlock(&tlock);
        if (t >= NTASK)
            break; /* all tasks done */
        taskexecute(t);
    }
}
```

```
#define N 10000    /* μέγεθος πίνακα */
#define M 100     /* 100 x 100 υποπίνακες */
#define S N/M     /* μέγεθος υποπίνακα */
#define NTASK M*M /* τόσους υποπίνακες */

void taskexecute(int wid) {
    int i, j, k, x, y;
    double sum;

    x = wid / M;
    y = wid % M;
    for (i = x*S; i < (x+1)*S; i++) /* στοιχεία του Cxy */
        for (j = y*S; j < (y+1)*S; j++)
        {
            for (k = 0, sum = 0.0; k < N; k++)
                sum += A[i][k]*B[k][j];
            C[i][j] = sum;
        };
}
```



Διεργασίες ως οντότητες εκτέλεσης



ΜΥΕΟ23
Παράλληλα
Συστήματα &
Προγραμματισμός

Προγραμματισμός με διεργασίες στο UNIX

- Sys-V Shared Memory IPC

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>          /* Για shared memory */
#include <sys/sem.h>         /* Για semaphores */
```

- Δημιουργία διεργασιών με `fork()` (returns 0 to the child).
- Δεν υπάρχουν κοινές μεταβλητές (μόνο κοινός χώρος στη μνήμη, δεσμευόμενος δυναμικά)
- Δεν υπάρχουν κλειδαριές, μόνο semaphores
- Δεν υπάρχουν μηχανισμοί συγχρονισμού

... αλλά μπορούμε να κάνουμε πολλά πράγματα

Υλοποίηση της join()

- Αναμονή τερματισμού παιδιών με χρήση της `wait()` / `waitpid()`.
- Η παρακάτω κάνει το γονέα να περιμένει μέχρι να τερματίσουν N παιδιά του:

```
void join(int N) {  
    for ( ; N > 0; N--)  
        wait(NULL);    /* Αναμονή για οποιοδήποτε παιδί */  
}
```

Semaphores

- `semget()` για ορισμό
 - `semid = semget(IPC_PRIVATE, 1, 0600 | IPC_CREAT);`
- `semctl()` για αρχικοποίηση / καταστροφή / έλεγχο
 - `semctl(semid, 0, SETVAL, arg);`
 - `semctl(semid, IPC_RMID, 0);`
- `semop()` για αύξηση / μείωση
 - `semop(semid, &opr, 1);`

```
union semun {
    int          val;
    struct semid_ds *buf;
    ushort_t     *array;
};

union semun arg;
arg.val = k;    /* αρχικοποίηση */
```

Κλειδαριές από semaphores

```
initlock(int *l) {  
    union semun arg;  
  
    *l = semget(IPC_PRIVATE, 1, 0600 | IPC_CREAT);  
    arg.val = 1;          /* Binary semaphore */  
    semctl(*l, 0, SETVAL, arg);  
}
```

```
lock(int *l) {  
    struct sembuf opr;  
    opr.num = opr.flag = 0;  
    opr.op = -1;          /* Μείωση κατά 1 */  
    semop(*l, &opr, 1);  
}
```

```
unlock(int *l) {  
    struct sembuf opr;  
    opr.num = opr.flag = 0;  
    opr.op = 1;          /* Αύξηση κατά 1 */  
    semop(*l, &opr, 1);  
}
```

Καταστροφή:
`semctl(*l, IPC_RMID, 0);`

Κοινόχρηστες «μεταβλητές»

```
int memid = shmget(IPC_PRIVATE, size, 0600|IPC_CREAT);  
char *p = shmat(memid, 0, 0);  
shmctl(memid, ICP_RMID, 0);      /* Απελευθέρωση μνήμης */
```

- Στα νήματα οι μεταβλητές (καθολικές) είναι κοινές και δεν χρειάζεται τίποτε, ούτε για τον ορισμό τους, ούτε για τον χειρισμό τους
- Με διεργασίες (πολύπλοκοι χειρισμοί με pointers):

```
memid = shmget(IPC_PRIVATE, 2*sizeof(int), 0600|IPC_CREAT);  
p = shmat(memid, 0, 0);  
*((int *) p) = 2;  
*(((int *) p) + 1) = *((int *) p);
```

.. ΠΙΟ ΕΥΚΟΛΟΣ
ΧΕΙΡΙΣΜΟΣ

- Τέχνασμα (δομή με τις κοινόχρηστες μεταβλητές):

```
struct mine {  
    int i;  
    int j;  
} *myvars;
```

```
memid = shmget(IPC_PRIVATE, sizeof(struct mine), 0600|IPC_CREAT);  
myvars = (struct whatever *) shmat(memid, 0, 0);
```

```
myvars->i = 2;
```

```
myvars->j = myvars->i;
```

