ΥΟ7 – Παράλληλα Συστήματα
2011-12

26/3/2012
Προγραμματισμός συστημάτων κοινόχρηστης μνήμης (Ι)

Β. Δημακόπουλος

Από τα εισαγωγικά…

- … 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.

- Κάθε οντότητα εκτέλεσης (νήμα ή διεργασία) θα εκτελείται σε έναν επεξεργαστή, όπου «επεξεργαστής» είναι ότι το λειτουργικό σύστημα θεωρεί επεξεργαστή (επεξεργαστή, πυρήνα, hardware thread).

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 0

Νήμα T1	Νήμα T2
`A = A+1;`	`A = A-1;`

- Λεπτομέρεια εκτέλεσης της εντολής A = A±1 σε έναν επεξεργαστή:
  - **α.** Ο επεξεργαστής μεταφέρει από τη μνήμη την τιμή της A
  - **β.** Αυξάνει/μειώνει κατά 1
  - **γ.** Αποθηκεύει στην 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)          /* παιδί */
(γ)       κώδικας Α;
(δ)     else                 /* γονέας */
(ε)       κώδικας Β;
(η)     κώδικας Γ;
        ...
```

```
int x = 2;

main()
{
  int y = 0;

  y = 1;
  if (fork())
    x = x+y;
  else
    x = x-y;
  printf("%d\n", x);
}

Τι θα τυπωθεί;
```

```
main()
{
  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 (p.x. 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)
{
  ...
}
```

Πάντα

```
#include <pthread.h>
```

Μετάφραση:

```
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*

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

                          Πόσα θα φτιαχτούν;
```

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

Πώς μπορώ να περάσω > 1 παραμέτρους;

```
struct manyparams { int x; int y; double z; };

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.

- 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

```
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");
}

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

Το cast αυτό θέλει πολύ προσοχή!!

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);
...
```

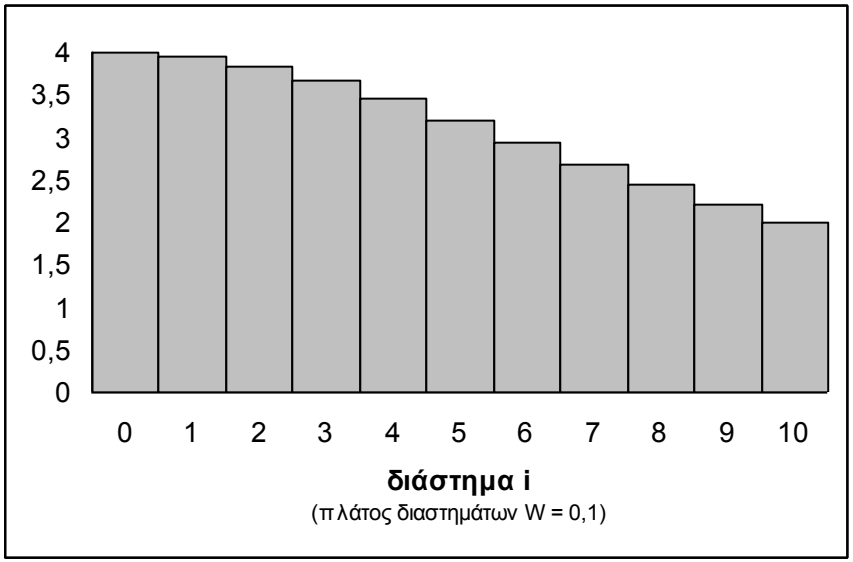
```
main() {
  pthread_mutex_t mymutex;
  pthread_mutex_init(&mymutex, NULL);
  pthread_mutex_lock(&mymutex);
    ...       /* Κρίσιμη περιοχή */
  pthread_mutex_unlock(&mymutex);
}
```

Εξασφαλίζει επίσης ότι έχουν ολοκληρωθεί οι αναφορές προς τη μνήμη.

Υπολογισμός του π = 3,14…

- Αριθμητική ολοκλήρωση

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



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

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

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);
}
```

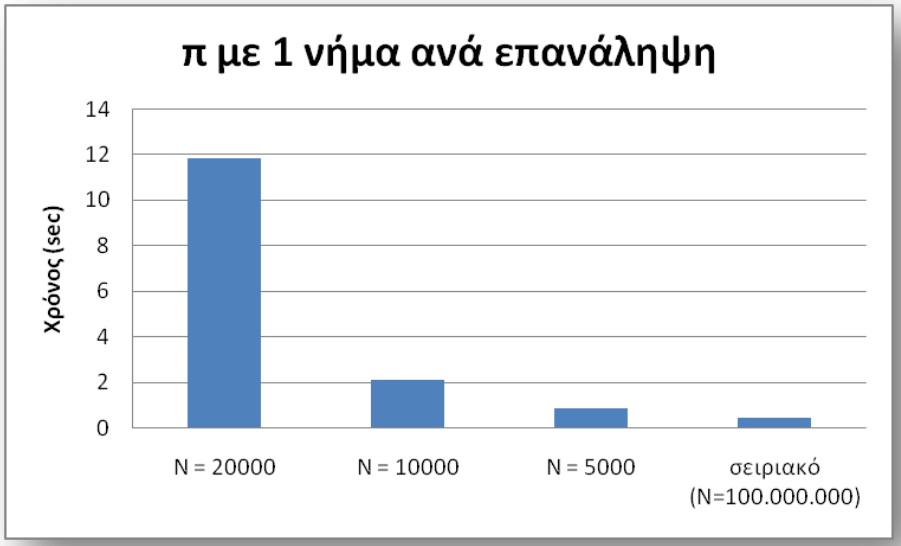
Υπολογισμός του π = 3,14... με 1 νήμα ανά επανάληψη

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

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);
}
```

```
/* 1 νήμα 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);
}

main() {
  int i;
  pthread_t tids[N];            /* Remember the tread 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);
}
```

Προβλήματα

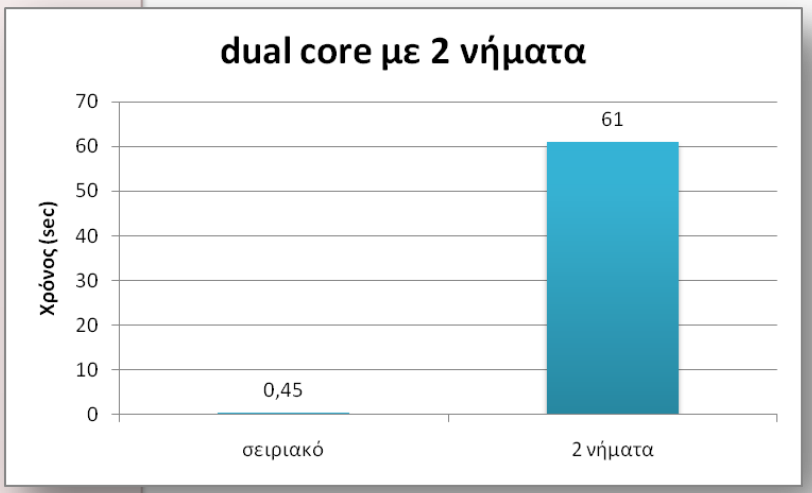
- Υπερβολικά πολλά νήματα (πολλά συστήματα δεν επιτρέπουν πάνω από λίγες χιλιάδες)
  - και επομένως υπερβολικά «λεπτόκοκκος» παραλληλισμός (fine grain)
- Ακόμα και αν επιτρέπονταν τόσα πολλά νήματα, ο συναγωνισμός για την κλειδαριά είναι τεράστιος.
- Αποτέλεσμα: αργή εκτέλεση και μόνο για μη ακριβή προσέγγιση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

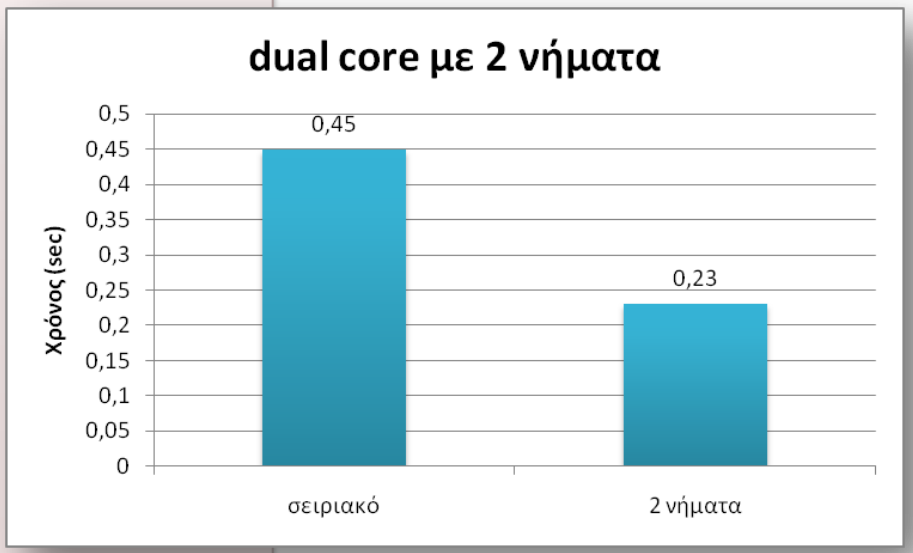
```
#define NPROCS 2          /* dual core */
#define N 10000000        /* Για ακρίβεια (ίδια με σειριακό) */
#define WORK N/NPROCS
double pi = 0.0, W = 1.0/N;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

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);
  }
}

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);
}
```



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);
}

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);
}
```

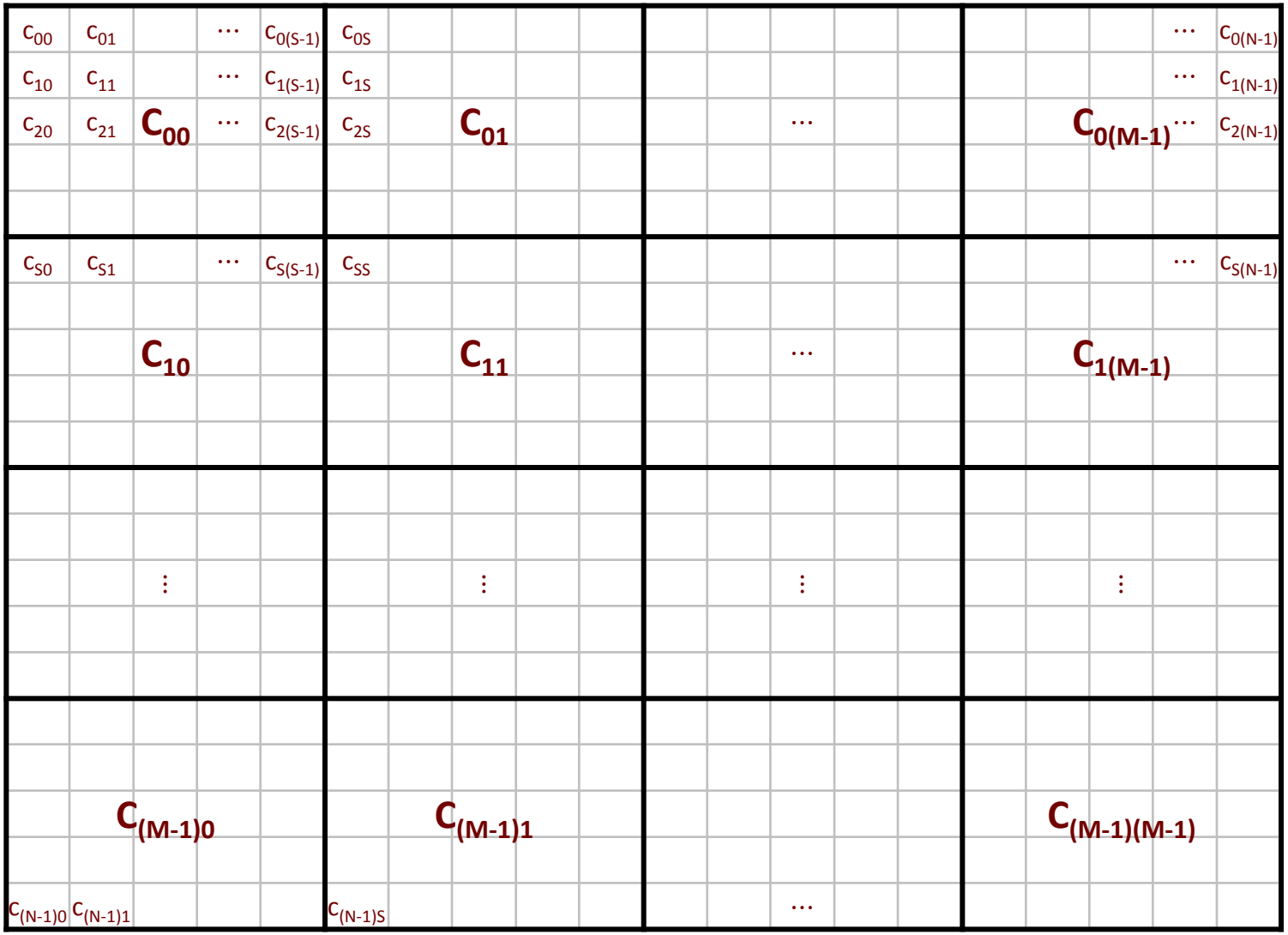


Πίνακας επί πίνακα

```
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



Υποπίνακες διάστασης S×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      50       /* Πίνακας 50x50 */
#define NPROCS 25       /* Αριθμός επεξεργαστών/διεργασιών */
#define M      5        /* Η ρίζα του 25 */
#define S      N/M      /* S = N / M */
double  A[N][N], B[N][N], C[N][N];

void *thrfunc(void *arg)
{
  int    i, j, k, x, y, myid = (int) arg;
  double sum = 0.0;
  x = myid / M;
  y = myid % M;
  for (i = x*S; i < (x+1)*S; i++)   /* υπολογίζει τα στοιχεία του Cxy  */
    for (j = y*S; j < (y+1)*S; j++)
    {
      for (k = 0; k < N; k++)
        sum += A[i][k]*B[k][j];
      C[i][j] = sum;
    };
}

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);
}
```

Δεν υπάρχει ανάγκη αμοιβαίου αποκλεισμού

Embarrassingly parallel

Πίνακας επί διάνυσμα – πιο απλό?

```
float 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)

```
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);
}
```

```
#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
#define NPROCS 200
#define RPT    N/NPROCS  /* "Rows Per Thread" */
double  A[N][N], v[N], res[N];
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *thrfunc(void *arg)
{
  int    i, j, myid = (int) arg;
  double sum = 0.0;

  i = myid / 2;               /* Γραμμή που αναλαμβάνω */
  myhalf = (N/2)*(myid % 2);  /* Οι μισές επαναλήψεις του j */
  for (j = myhalf; j < myhalf + N/2; j++)
    sum += A[i][j]*v[j];
  pthread_mutex_lock(&lock);
  res[i] += sum;
  pthread_mutex_unlock(&lock);
}
```

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

  for (i = 0; i < N; i++)
    res[i] = 0.0;
  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);
}
```

Παράλληλη αρχικοποίηση;

Κάθε ζεύγος νημάτων να αρχικοποιεί το δικό του res[i]

Π.χ. το ζυγό νήμα (myid%2 == 0) να κάνει res[i] = 0.0.

Υπάρχει κάποιο πρόβλημα;

- Απαιτείται, πλέον, αμοιβαίος αποκλεισμός.
- Προσοχή στην αρχικοποίηση του res[].

Συγχρονισμός!

- Οι οντότητες εκτέλεσης πρέπει μερικές φορές να *συγχρονίζονται*.
  - Θα πρέπει να περιμένουν μέχρι να συμβεί κάποιο γεγονός

- Συνήθης μηχανισμός: φράγματα (***barriers***)
  - 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

Σωστή παράλληλη αρχικοποίηση

```
#define N      100
#define NPROCS 200
#define RPT    N/NPROCS  /* "Rows Per Thread" */
double  A[N][N], v[N], res[N];
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_barrier_t *bar;

void *thrfunc(void *arg)
{
  int    i, j, myid = (int) arg;
  double sum = 0.0;

  i = myid / 2;               /* Γραμμή που αναλαμβάνω */
  myhalf = (N/2)*(myid % 2);  /* Οι μισές επαναλήψεις του j */
  for (j = myhalf; j < myhalf + N/2; j++)
    sum += A[i][j]*v[j];

  if (myid % 2 == 0) res[i] = 0.0;
  pthread_barrier_wait(bar);

  pthread_mutex_lock(&lock);
  res[i] += sum;
  pthread_mutex_unlock(&lock);
}
```

Όλα τα νήματα στον ίδιο barrier!

Άλλη ιδέα;

Το άρτιο νήμα να δημιουργεί το περιττό, αμέσως μετά την αρχικοποίηση (άρα => παράλληλη δημιουργία νημάτων)

Μεταβλητές συνθήκης

- Ο βασικός μηχανισμός συγχρονισμού στα νήματα Posix.
- Ένα νήμα θα περιμένει σε μια μεταβλητή συνθήκης (**`pthread_cond_wait(&condition, &mutex)`**) έως ότου η μεταβλητή το ενημερώσει ότι μπορεί να συνεχίσει
  - Κάποιο άλλο νήμα σηματοδοτεί τη μεταβλητή συνθήκης, επιτρέποντας άλλα νήματα να συνεχίσουν (**`pthread_cond_signal(&condition)`** ή **`pthread_cond_broadcast(&condition)`**)
  - 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

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

Παραδείγματα

Threads 0,1,…,N-1

```
for(...)
{
 ------------
 pthread_mutex_lock(&mut);
 count++;
 if (count== N){
  /* Wake-up thread N */
  pthread_cond_signal(&cond);
}
pthread_mutex_unlock(&mut);
------------
//Do work
}
```

Thread N

```
-----------
pthread_mutex_lock(&mut);
while(count<N) {
  pthread_cond_wait(&cond,
                    &mut);
}
pthread_mutex_unlock(&mut);
------------

//Do work
```

Thread 0

```
for(...)
{
 ------------
 pthread_mutex_lock(&mut);
 count++;
 if (count== N){
  /* Wake-up threads 1..N */
  pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&mut);
------------
//Do work
}
```

Thread 1,2,…,N

```
-----------
pthread_mutex_lock(&mut);
while(count<N) {
  pthread_cond_wait(&cond,
                    &mut);
}
pthread_mutex_unlock(&mut);
------------

//Do work
```

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 Β κάνει singal(), τότε το σήμα θα χαθεί και το νήμα Α θα περιμένει για πάντα.
- Γιατί το while;
  - Διότι μπορεί μετά το flag = TRUE να ξύπνησε το νήμα Α, 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.

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 κάποιοι επεξεργαστές
  - Είναι πιο αργοί από τους άλλους ή
  - Για το συγκεκριμένο διάστημα είναι περισσότερο φορτωμένοι από τους άλλους (διότι π.χ. εκτελούν και κάποια άλλη εφαρμογή)

- Σε αυτή την περίπτωση, η ισόποση διαμοίραση των εργασιών *ΔΕΝ ΕΙΝΑΙ ΟΤΙ ΚΑΛΥΤΕΡΟ*!

Αυτοδρομολόγηση (self-scheduling)

- Εφαρμόζετε 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-do)
{
    Work = get-the-next-work()
    execute(Work);
}
```

Αυτοδρομολόγηση – κώδικας

- Έστω ότι κάπως έχουμε χωρίσει τη δουλειά σε NUMWORKS εργασίες, από 0 έως NUMWORKS – 1.
- Έστω ότι η i-οστή εργασία εκτελείται ως execute(i)

```
#define NUMWORKS 100
int workid;
pthread_mutex_t worklock = PTHREAD_MUTEX_INITIALIZER;

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

  while (1) {    /* Για πάντα */
    pthread_mutex_lock(&worklock);
    t = workid++;
    pthread_mutex_unlock(&worklock);
    if (t >= NUMWORKS)
      break;
    execute(t);
  }
}
```

Παράδειγμα: υπολογισμός του π με αυτοδρομο/ση

```
int workid;
pthread_mutex_t worklock = PTHREAD_MUTEX_INITIALIZER;

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

  while (1) {    /* Για πάντα */
    pthread_mutex_lock(&worklock);
    t = workid++;
    pthread_mutex_unlock(&worklock);
    if (t >= NUMWORKS)
      break;
    execute(t);
  }
}
```

```
#define N 10000000       /* Για ακρίβεια (ίδια με σειριακό) */
#define NUMWORKS 10000
#define WORK N/NUMWORKS  /* Πόσες επαναλήψεις η κάθε εργασία */
double pi = 0.0, W = 1.0/N;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void execute(int iter) {
  int    i;
  double mysum = 0.0;
  for (i = iter*WORK; i < (iter+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);
}
```

Παράδειγμα: checkerboard + αυτοδρομολόγηση

```
int workid;
pthread_mutex_t worklock = PTHREAD_MUTEX_INITIALIZER;

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

  while (1) {    /* Για πάντα */
    pthread_mutex_lock(&worklock);
    t = workid++;
    pthread_mutex_unlock(&worklock);
    if (t >= NUMWORKS)
      break;
    execute(t);
  }
}
```

```
#define N 10000        /* μέγεθος πίνακα */
#define M 100          /* 100 x 100 υποπίνακες */
#define S N/M          /* Μέγεθος υποπίνακα */
#define NUMWORKS M*M   /* τόσοι υποπίνακες */

void execute(int wid) {
  int    i, j, k, x, y;
  double sum = 0.0;

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

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 1ος (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 2ος (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 */
}
```