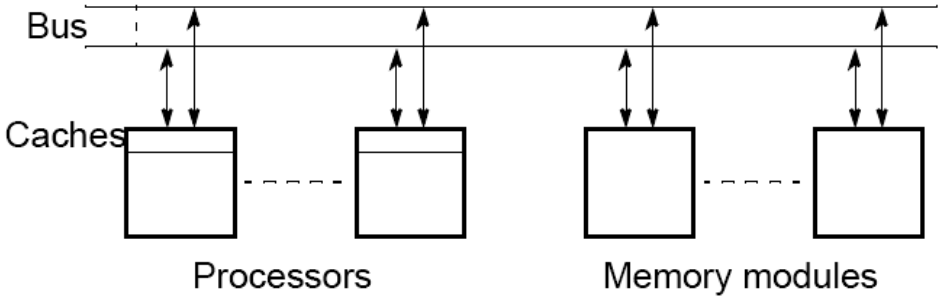
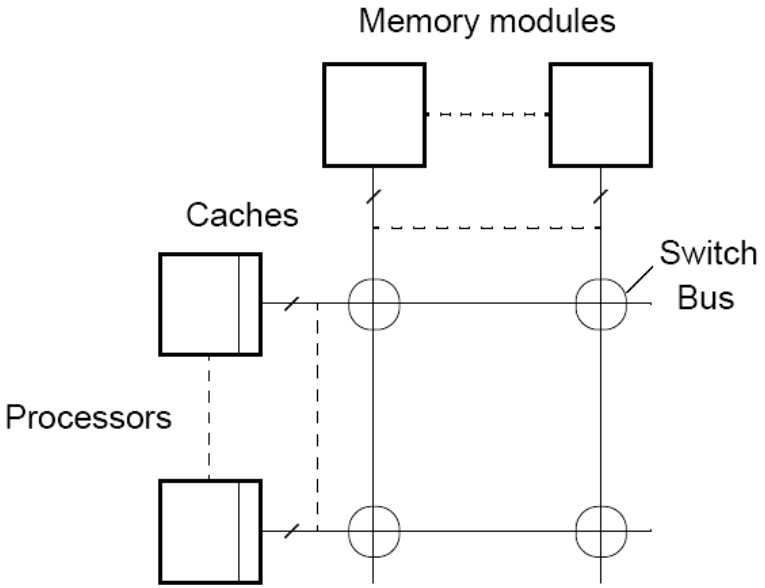
Προγραμματισμός σε συστήματα διαμοιραζόμενης μνήμης

Πολυεπεξεργαστικά συστήματα διαμοιραζόμενης μνήμης

• Σε ένα σύστημα διαμοιραζόμενης μνήμης, κάθε επεξεργαστής μπορεί να προσπελάσει οποιαδήποτε θέση της κοινά διαμοιραζόμενης μνήμης.
• Ο χώρος διευθύνσεων είναι ίδιος για όλους τους επεξεργαστές, πράγμα που σημαίνει ότι κάθε θέση μνήμης έχει μία μοναδική διεύθυνση.
•Γενικά, ο προγραμματισμός παράλληλων εφαρμογών σε συστήματα διαμοιραζόμενης μνήμης θεωρείται πιο προσιτός σε σχέση με το προγραμματισμό σε συστήματα κατανεμημένης μνήμης.
• Στις περισσότερες περιπτώσεις, η πρόσβαση στα κοινά δεδομένα θα πρέπει να ελέγχεται από τον προγραμματιστή (χρησιμοποιώντας π.χ. κρίσιμές περιοχές)
• Όταν το πλήθος των επεξεργαστών είναι μικρό, συνήθως χρησιμοποιείται μία αρχιτεκτονική μονού διαύλου όπου όλοι οι επεξεργαστές και τα τμήματα της μνήμης συνδέονται στον ίδιο δίαυλο.
• Όταν όμως αυξάνεται το πλήθος των επεξεργαστών, δημιουργείται συμφόρεση στο κοινό δίαυλο
•Η ύπαρξη cache μνήμης σε κάθε επεξεργαστή μειώνει τις προσβάσεις στη κύρια μνήμη.
• Ακόμα όμως και με αυτή τη βελτίωση, το εύρος ζώνης του διαύλου παραμένει περιορισμένο.

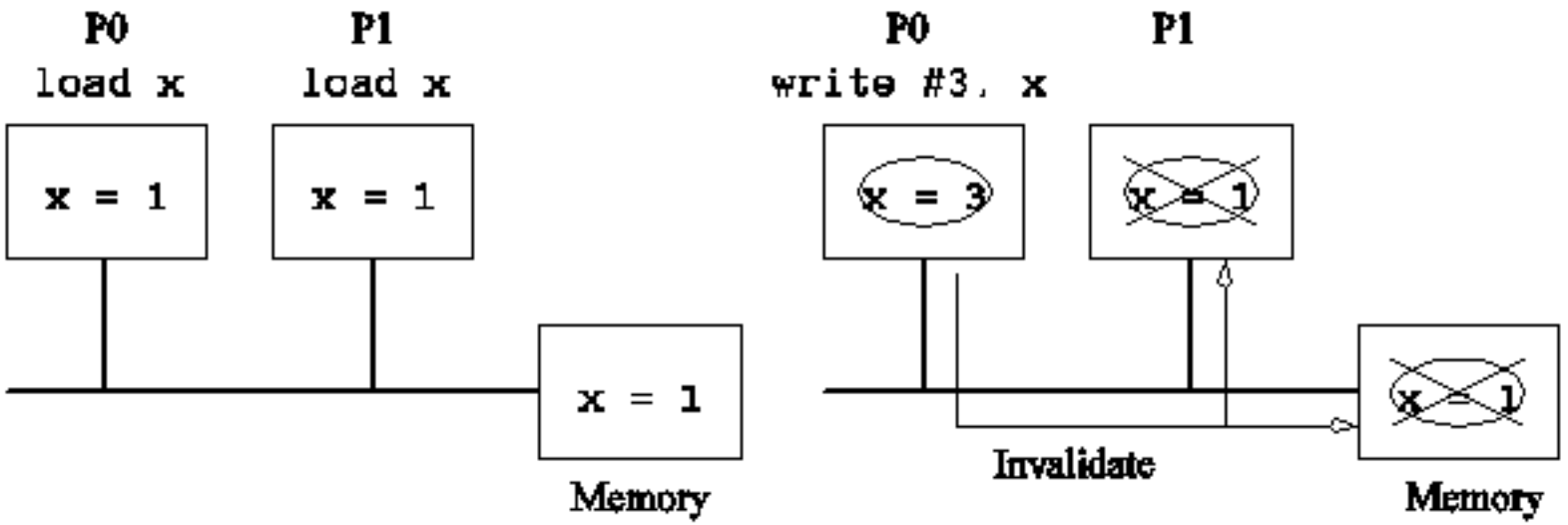




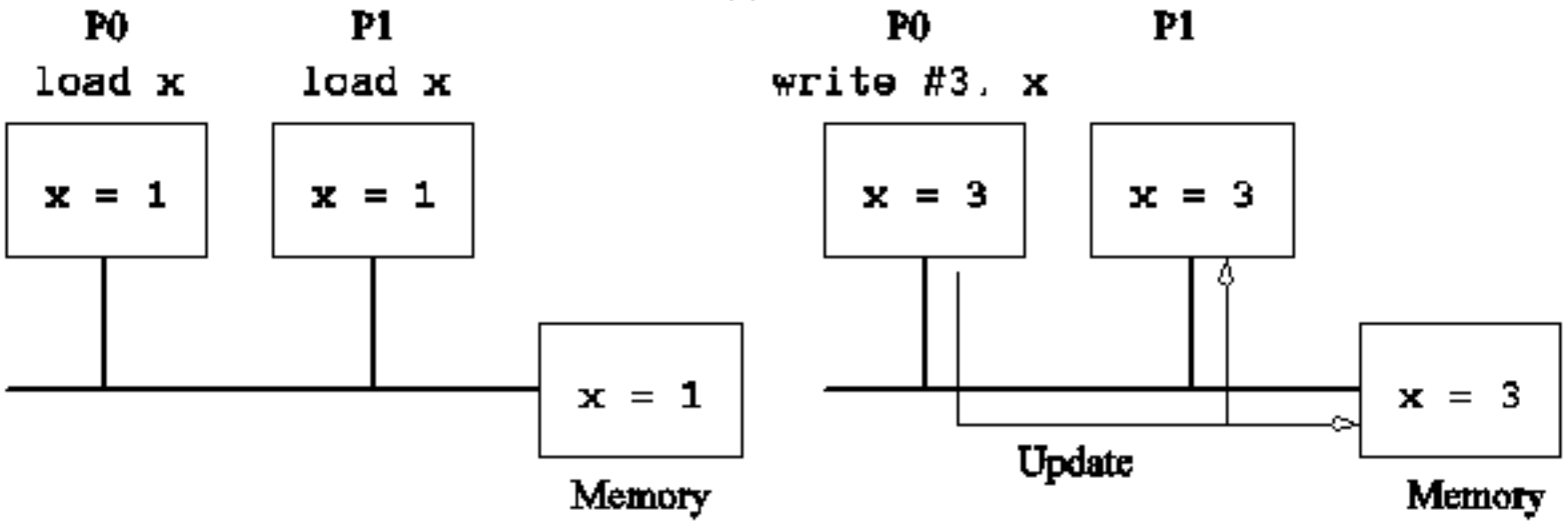
• Όταν υπάρχουν πολύ επεξεργαστές στο σύστημα, άλλα διασυνδετικά δίκτυα προτιμώνται, όπως το crossbar το οποίο παρέχει πλήρη διασύνδεση μεταξύ των επεξεργαστών και των τμημάτων μνήμης.
• Μεταξύ οποιουδήποτε επεξεργαστή και μνήμης υπάρχει μία ξεχωριστή διασύνδεση και έτσι η συμφόρεση αποφεύγεται.
• Το μειονέκτημα αυτού του δικτύου είναι το υψηλό του κόστος.

Διαμοιραζόμενα δεδομένα σε συστήματα με cache μνήμες

- Όπως αναφέρθηκε σε όλα τα σύγχρονα υπολογιστικά συστήματα, οι επεξεργαστές διαθέτουν μνήμες cache.
- Οι μνήμες αυτές είναι υψηλών επιδόσεων και χρησιμοποιούνται για να κρατούν δεδομένα στα οποία έγινε προσπέλαση πρόσφατα.
- Αυτό έχει σαν αποτέλεσμα, πολλαπλά αντίγραφα των περιεχομένων της ίδιας θέσης μνήμης να είναι, πιθανόν, κατανεμημένα στις μνήμες cache διαφορετικών επεξεργαστών.
- Αν κάποιος από αυτούς τους επεξεργαστές αλλάξει το περιεχόμενο ενός από αυτά τα αντίγραφα, αυτό έχει ως συνέπεια όλα τα αντίγραφα της ίδιας θέσης μνήμης να μην έχουν τα ίδια περιεχόμενα.
- Σε αυτή την περίπτωση, θα πρέπει να αποκατασταθεί η συνοχή μεταξύ των διαφόρων αντιγράφων
- Υπάρχουν δύο κατηγορίες πρωτοκόλλων συνοχής μνημών cache (cache coherence protocols)
    - *Πρωτόκολλα ενημέρωσης* – τα αντίγραφα σε όλες τις μνήμες cache ενημερώνονται κάθε φορά που ένα αντίγραφο ενημερώνεται.
    - Πρωτόκολλα ακύρωσης - όταν ένα αντίγραφο αλλάξει, τα αντίγραφα στις μνήμες cache ακυρώνονται (θέτοντας ένα σχετικό δυαδικό ψηφίο στη μνήμη cache). Αυτά τα αντίγραφα ενημερώνονται όταν ο σχετικός επεξεργαστής προσπαθήσει να προσπελάσει ένα άκυρο αντίγραφο.



Πρωτόκολλο ακύρωσης

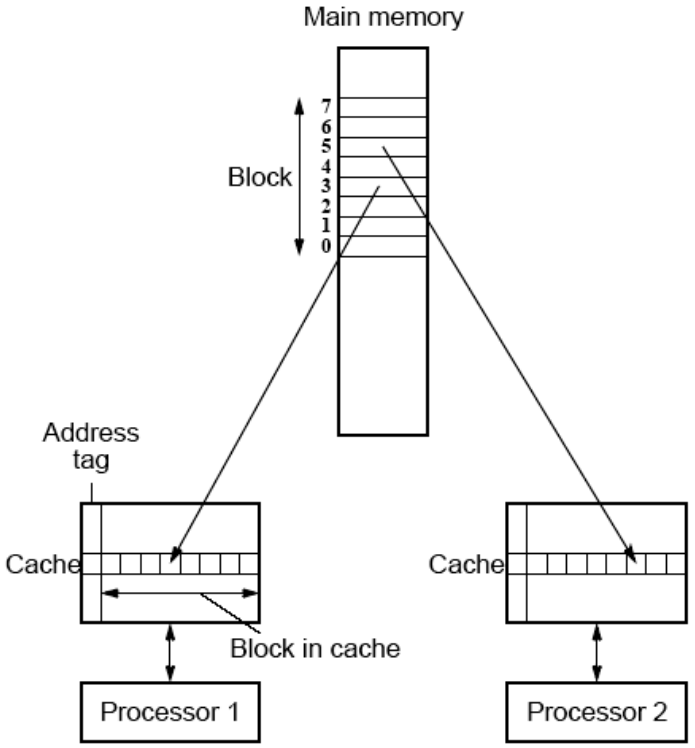


Πρωτόκολλο ενημέρωσης

Ψευδής Διαμοιρασμός (False Sharing)

Οι μνήμες cache δεν αποθηκεύουν μεμονωμένες θέσεις μνήμης αλλά blocks από συνεχόμενες θέσεις μνήμης. Αν ένας επεξεργαστής διαβάσει μία θέση μνήμης, δεν θα μεταφερθεί στη μνήμη cache το περιεχόμενο αυτής της θέσης αλλά όλο το block που περιέχει αύτη τη θέση μνήμης.

Αν τώρα διαφορετικοί επεξεργαστές τροποποιούν τα περιεχόμενα διαφορετικών θέσεων μνημών του ίδιου block θα πρέπει να εκτελεσθεί το πρωτόκολλο συνοχής της μνήμης cache (ενημέρωση ή ακύρωση όλου του block) για κάθε τροποποίηση παρόλο που οι επεξεργαστές δεν αλλάζουν την ίδια θέση μνήμης.



- Στο παράδειγμα αυτό, κάθε block αποτελείται από 8 συνεχόμενες λέξεις.
- Ο επεξεργαστής 1 διαβάζει τη λέξη 4 του block ενώ ο επεξεργαστής 2 διαβάζει τη λέξη 6.
- Η λύση στο πρόβλημα αυτό είναι η τοποθέτηση των δεδομένων στη κύρια μνήμη κατά τέτοιο τρόπο ώστε τα δεδομένα που τροποποιούνται από διαφορετικούς επεξεργαστές να είναι σε διαφορετικά blocks.

Προγραμματισμός σε Συστήματα Διαμοιραζόμενης Μνήμης

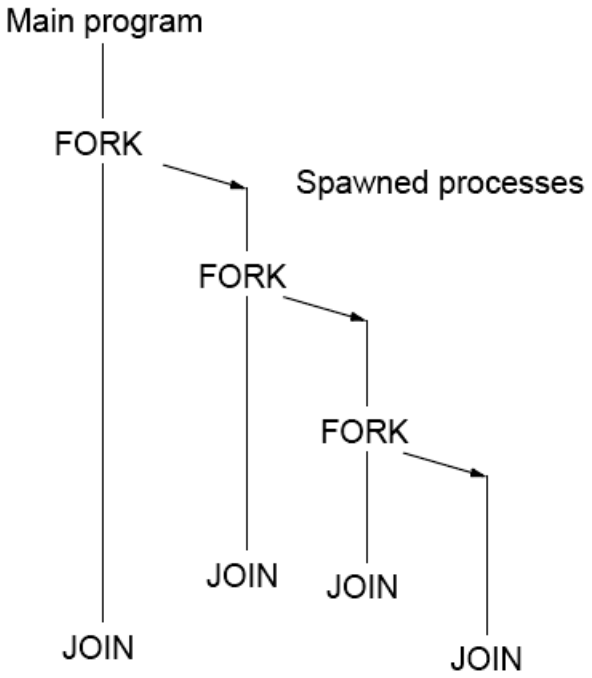
• Υπάρχουν πολλές εναλλακτικές για το προγραμματισμό πολυεπεξεργαστών διαμοιραζόμενης μνήμης:

- Χρήση «βαριών» (heavy weight) διεργασιών.
- Χρήση νημάτων π.χ. χρήση της βιβλιοθήκης Pthreads
- Χρήση συναρτήσεων βιβλιοθηκών που υλοποιούν παράλληλα διάφορους υπολογισμούς μαζί με υπάρχουσες γλώσσες ακολουθιακού προγραμματισμού.
- Χρήση γλωσσών ακολουθιακού προγραμματισμού συμπληρωμένες με οδηγίες μεταγλωττιστή που περιγράφουν λεπτομέρειες της παράλληλης υλοποίησης. Παράδειγμα είναι το πρότυπο OpenMP

Χρήση «βαριών» διεργασιών

Τα σύγχρονα λειτουργικά συστήματα βασίζονται στην έννοια της διεργασίας.
Σε ένα σύστημα ενός επεξεργαστή, ο χρόνος εκτέλεσης του επεξεργαστή μοιράζεται μεταξύ των διεργασιών, με εναλλαγή εκτέλεσης των διεργασιών. Αυτή η εναλλαγή μπορεί να συμβαίνει ανά τακτά χρονικά διαστήματα ή όταν η διεργασία που έχει τον επεξεργαστή υπό τον έλεγχο της καθυστερήσει για κάποιο λόγο (I/O).
Αν και οι διεργασίες μπορούν να χρησιμοποιηθούν στο παράλληλο προγραμματισμό εντούτοις αποφεύγονται λόγω της μεγάλης επιβάρυνσης που έχει ο χειρισμός τους.
Πάντως παραλλαγές του βασικού μηχανισμόυ fork/join για τη δημιουργία διεργασιών έχει υιοθετηθεί από πολλές βιβλιοθήκες παράλληλου προγραμματισμού (π.χ. βιβλιοθήκη Pthreads)

Τεχνική FORK-JOIN



- Η συνάρτηση συστήματος fork() δημιουργεί μία νέα διεργασία.
- Η νέα διεργασία (διεργασία παιδί) είναι ακριβές αντίγραφο της διεργασίας που καλεί τη fork (διεργασία πατέρας). Η μόνη διαφορά είναι ότι το αναγνωριστικό της διαδικασίας παιδί είναι διαφορετικό από αυτό της διεργασίας πατέρας.
- Επίσης έχει ξεχωριστό αντίγραφο των μεταβλητών του πατέρα.
- Η συνάρτηση fork() επιστρέφει 0 στη διαδικασία παιδί και επιστρέφει το αναγνωριστικό της διαδικασίας παιδί στη διαδικασία πατέρας.
- Οι διεργασίες «ενώνονται» (joined) χρησιμοποιώντας τις συναρτήσεις συστήματος wait() και exit()

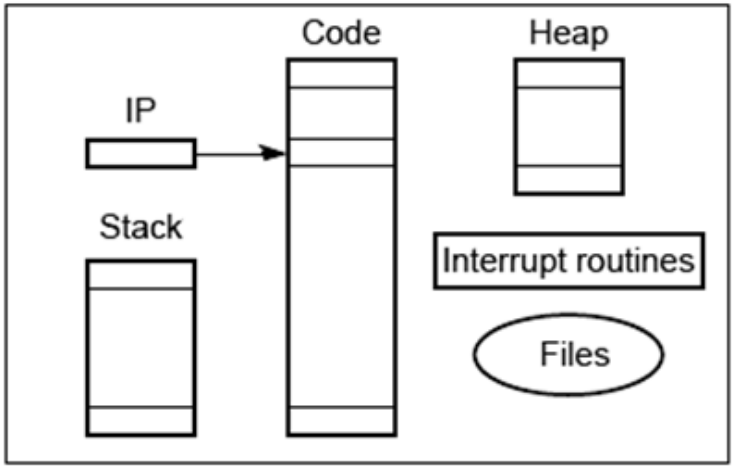
Fork/join τεχνική

```
    :
pid = fork();                          /* fork */
  Code to be executed by both child and parent
if (pid == 0) exit(0); else wait(0);/* join */
    :
```

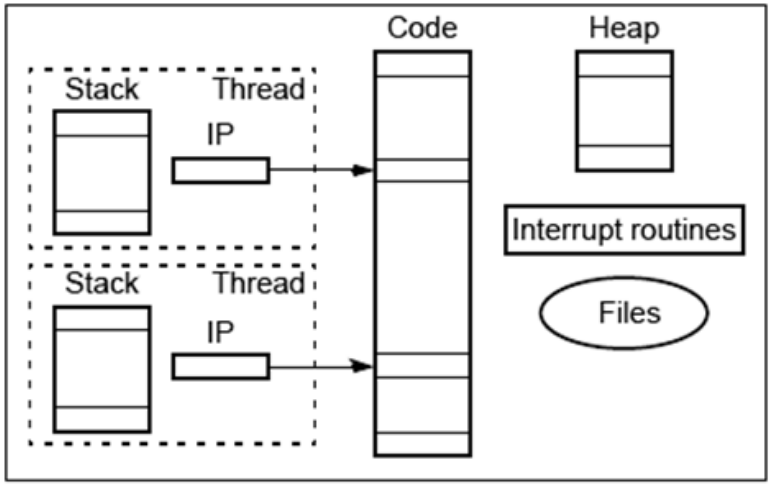
• Στο παραπάνω παράδειγμα, η διεργασία πατέρας και η διεργασία παιδί εκτελούν το ίδιο τμήμα κώδικα.
• Ο πατέρας εκτελεί την εντολή wait (pid ≠0 στον πατέρα) και περιμένει μέχρι η διαδικασία παιδί να εκτελέσει την εντολή exit (pid=0 στο παιδί).
• Στο επόμενο τμήμα κώδικα η διαδικασία παιδί εκτελεί διαφορετικό κώδικα από ότι ο πατέρας.

```
pid = fork();
if (pid == 0) {
  code to be executed by slave
} else {
  Code to be executed by parent
}
if (pid == 0) exit(0); else wait(0);
    :
    :
```

Διεργασίες και νήματα (threads)

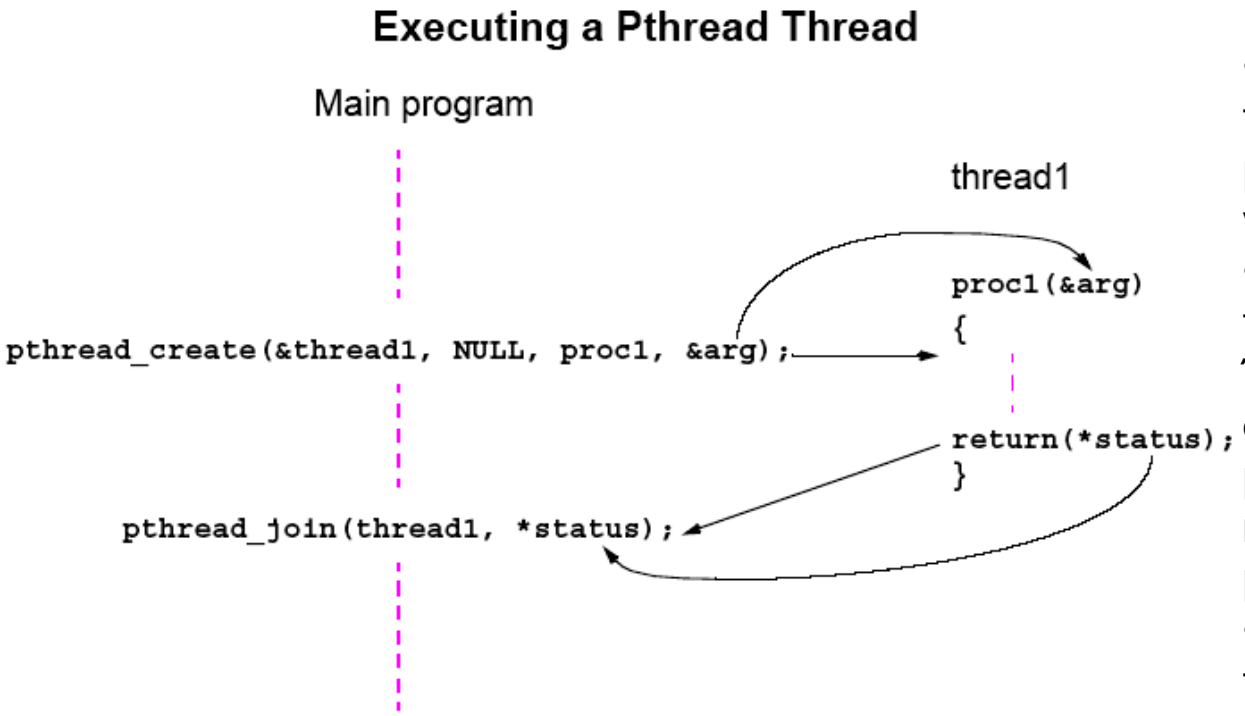


Διεργασία



Νήμα

- Οι διεργασίες είναι τελείως ξεχωριστά προγράμματα με τις δικές τους μεταβλητές, στοίβα και ιδιωτικό χώρο μνήμης.
- Η μνήμη μπορεί να διαμοιρασθεί μεταξύ των διεργασιών με τη χρήση συναρτήσεων συστήματος αλλά γενικά ο χειρισμός των διεργασιών έχει σημαντική επιβάρυνση στο λειτουργικό σύστημα.
- Πολύ λιγότερη επιβάρυνση έχουμε με τη χρήση νημάτων. Τα νήματα ανήκουν στην ίδια διεργασία και μοιράζονται το ίδιο κώδικα και σφαιρικές μεταβλητές (heap).
- Κάθε νήμα έχει ξεχωριστό δείκτη εντολών και επομένως κάθε νήμα έχει ξεχωριστή ροή εκτέλεσης.
- Επίσης, κάθε νήμα έχει το δικό του αντίγραφο τοπικών μεταβλητών (stack).
- Η δημιουργία ενός νήματος είναι πολύ γρηγορότερη σε σχέση με τη δημιουργία μίας διεργασίας
- Επίσης ο συγχρονισμός μεταξύ των νημάτων έχει πολύ μικρότερη επιβάρυνση σε σχέση με το συγχρονισμό μεταξύ των διεργασιών.



•Η βιβλιοθηκη Pthreads είναι ένα πρότυπο της IEEE που υλοποιεί βασικές λειτουργίες χειρισμών νημάτων.
• Στο πρότυπο αυτό, το κύριο πρόγραμμα είναι επίσης νήμα. Ένα νέο νήμα μπορεί να δημιουργηθεί με τη συνάρτηση pthread_create και να καταστραφεί με τη συνάρτηση pthread_join
• Το νέο νήμα (thread1) εκτελεί την συνάρτηση proc1 της οποίας το όρισμα arg περνάει μέσω της κλήσης της pthread_create.

• Μέσω του πρώτου ορίσματος της pthread_create επιστρέφεται στη μεταβλητή thread1 ένα αναγνωριστικό για το νέο νήμα. Αυτό το αναγνωριστικό χρησιμοποιείται στη συνάρτηση pthread_join
• Στο δεύτερο όρισμα της pthread_create, περνούν ως είσοδο χαρακτηριστικά που πρέπει να έχει το νέο νήμα. Στο συγκεκριμένο παράδειγμα, το δεύτερο όρισμα έχει τη τιμή NULL που σημαίνει ότι το νέο νήμα θα έχει προκαθορισμένα χαρακτηριστικά.
• Το κύριο πρόγραμμα εκτελεί την pthread_join και επιστρέφει μόνο όταν το νέο νήμα έχει ολοκληρωθεί. Η τιμή που επιστρέφει η συνάρτηση proc1 που εκτελείται από το νέο νήμα επιστρέφεται τελικά μέσω της pthread_join στο κύριο πρόγραμμα. Αν δεν απαιτείται επιστροφή τιμής από τη διεργασία thread1, το δεύτερο όρισμα της pthread_join τίθεται στη τιμή NULL.

• Σύνταξη βασικών εντολών χειρισμού νημάτων

```
#include <pthread.h>
int pthread_create (
    pthread_t *thread_handle, const pthread_attr_t *attribute,
    void * (*thread_function)(void *),
    void *arg);
int pthread_join (
    pthread_t thread,
    void **ptr);
```

•Στο προηγούμενο παράδειγμα, μόνο ένα νέο νήμα δημιουργείται. Στο παράδειγμα που ακολουθεί p νέα νήματα δημιουργούνται.

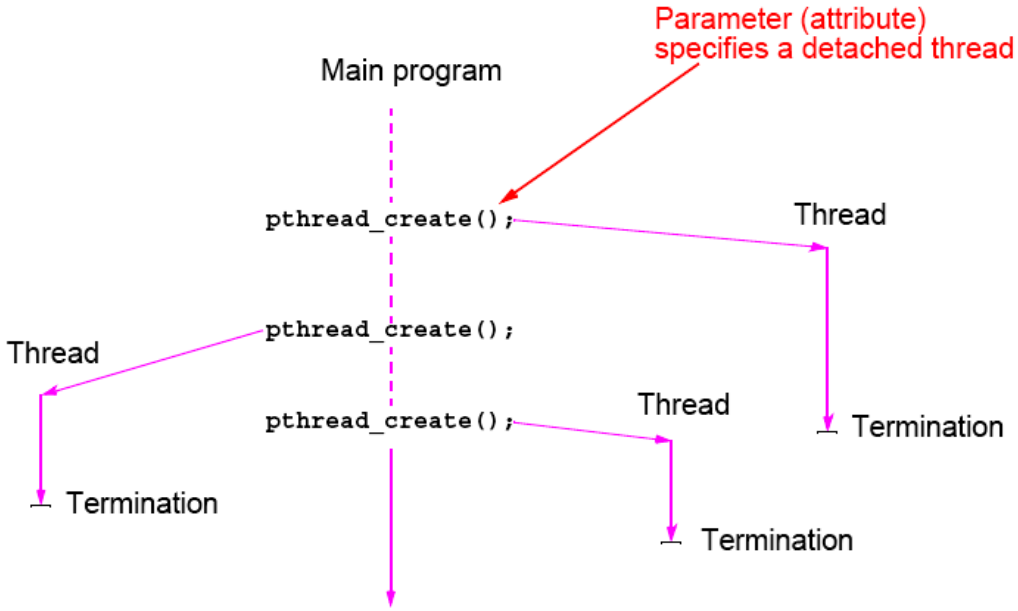
• Το κύριο νήμα (κύριο πρόγραμμα) περιμένει την ολοκλήρωση όλων των νημάτων με τη εκτέλεση της συνάρτησης pthread_join p φορές. Παρατηρείστε ότι οι πολλαπλές κλήσεις της join λειτουργούν ως φράγμα αφού το κύριο νήμα δεν προχωράει την εκτέλεσή του περαιτέρω πριν ολοκληρωθούν όλα τα νέα νήματα.

```
                                    .....
for (i=0; i<p; i++)
    pthread_create(&thread[i], NULL, (void * slave), (void *) &arg);
                                  .......
for (i=0; i<p; i++)
    pthread_join(&thread[i], NULL);
```

• Αξίζει επίσης να σημειωθεί ότι ένα νήμα μπορεί να μάθει το αναγνωριστικό του με τη κλήση της συνάρτησης pthread_self().

Απομονωμένα (detached) Νήματα

• Υπάρχει πιθανότητα όπου το κύριο νήμα δεν «ενδιαφέρεται» για την ολοκλήρωση ή μη των υπόλοιπων νημάτων και επομένως δεν χρειάζεται να εκτελέσει τη συνάρτηση pthread_join.
• Τα νήματα για τα οποία δεν εκτελείται η join από το κύριο νήμα λέγονται απομονωμένα νήματα.
• Όταν τα απομονωμένα νήματα τερματίζουν, καταστρέφονται από το λειτουργικό σύστημα και οι πόροι του απελευθερώνονται
• Ένα νήμα μπορεί να ορισθεί ως απομονωμένο, θέτοντας την κατάλληλη τιμή στο δεύτερο όρισμα της pthread_create()
• Τα απομονωμένα νήματα έχουν λιγότερη επιβάρυνση και επομένως θα πρέπει να χρησιμοποιούνται όποτε είναι δυνατόν.



Παράδειγμα δημιουργίας και τερματισμού νημάτων (υπολογισμός του π)

```
#include <pthread.h>
#include <stdlib.h>
#define MAX_THREADS 512
void *compute_pi (void *);
....
main() {
    ...
    pthread_t
    p_threads[MAX_THREADS];
    pthread_attr_t attr;
    pthread_attr_init (&attr);
    for (i=0; i< num_threads;
    i++) {
        hits[i] = i;
        pthread_create(&p_thread
        s[i], &attr, compute_pi,
            (void *) &hits[i]);
    }
    for (i=0; i< num_threads;
    i++) {
        pthread_join(p_threads[i
        ], NULL);
        total_hits += hits[i];
    }
    ...
}
```

```
void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double rand_no_x, rand_no_y;
    int local_hits;
    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i <
    sample_points_per_thread; i++) {
        rand_no_x
        =(double)(rand_r(&seed))/(double
        )((2<<14)-1);
        rand_no_y
        =(double)(rand_r(&seed))/(double
        )((2<<14)-1);
        if (((rand_no_x - 0.5) *
        (rand_no_x - 0.5) +
            (rand_no_y - 0.5) *
            (rand_no_y - 0.5)) < 0.25)
            local_hits ++;
        seed *= i;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
```

Αν αντί για την εντολή local_hits ++ είχαμε την εντολή (*hit_pointer)++ θα είχαμε πιο αργή εκτέλεση λόγω του προβλήματος του false sharing. Το πρόβλημα τώρα αποφεύγεται γιατί η μεταβλητή local_hits είναι τοπική μεταβλητή.

Concurrency - Ταυτοχρονισμός

• Η εκτέλεση των εντολών μίας διεργασίας ή νήματος εξαρτάται από το λειτουργικό σύστημα.
• Σε ένα σύστημα μονού επεξεργαστή, τα νήματα/διεργασίες εκτελούνται συνήθως μέχρι να ανασταλεί η εκτέλεση τους από κάποια λειτουργία, π.χ. I/O λειτουργία.
• Σε ένα πολυεπεξεργαστικό σύστημα, οι εντολές των νημάτων/διεργασιών επικαλύπτονται χρονικά.
• Για παράδειγμα, αν υπάρχουν δύο διεργασίες με τις ακόλουθες εντολές

Process 1	Process 2
Instruction 1.1	Instruction 2.1
Instruction 1.2	Instruction 2.2
Instruction 1.3	Instruction 2.3

υπάρχουν πολλές χρονικές επικαλύψεις των εντολών των δύο διεργασιών, π.χ.:

Instruction 1.1
Instruction 1.2
Instruction 2.1
Instruction 1.3
Instruction 2.2
Instruction 2.3

• Αν δύο διεργασίες τυπώνουν μηνύματα, τα μηνύματα θα μπορούσαν να εμφανίζονται με διαφορετική σειρά ανάλογα με τη χρονοδρομολόγηση των διεργασιών
• Θα μπορούσε να είναι επίσης πιθανό το σενάριο όπου, οι χαρακτήρες από διαφορετικά μηνύματα παρεμβάλλονται μεταξύ τους

Thread-Safe Routines (Ρουτίνες ασφαλείς σε περιβάλλον νημάτων)

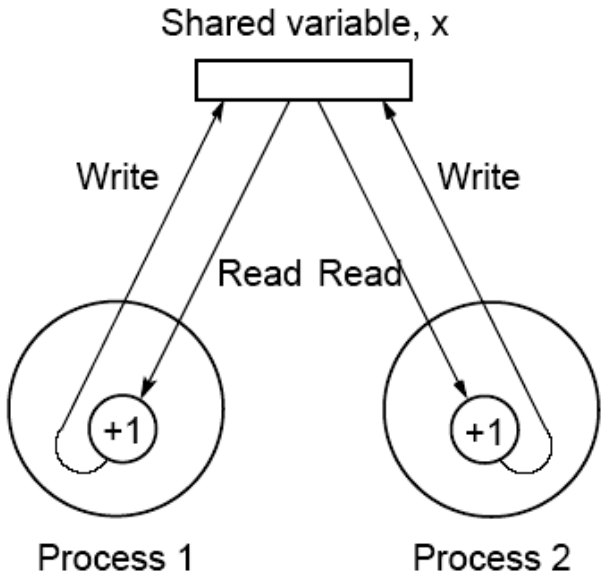
• Συναρτήσεις συστήματος ή συναρτήσεις βιβλιοθηκών καλούνται thread safe αν όταν καλούνται από πολλά νήματα ταυτόχρονα πάντα παράγουν σωστά αποτελέσματα, π.χ. εκτύπωση μηνυμάτων χωρίς τη παρεμβολή χαρακτήρων από άλλα μηνύματα.
• Όλες οι I/O ρουτίνες είναι συνήθως thread safe και έτσι δεν προκύπτουν προβλήματα με τη παρεμβολή χαρακτήρων.
• Πάντως, ρουτίνες που προσπελαύνουν διαμοιραζόμενα δεδομένα ή στατικές (static) μεταβλητές μπορεί να μην είναι thread safe. Π.χ. ρουτίνες που επιστρέφουν χρόνο μπορεί να μην είναι thread safe.
• Μπορούμε να μετατρέψουμε οποιαδήποτε διαδικασία σε thread safe αν επιβάλουμε μόνο μία διεργασία να εκτελεί τη διαδικασία. Αυτό μπορεί να επιτευχθεί με τη τοποθέτηση της διαδικασίας εντός ενός κρίσιμου τμήματος. Αυτός όμως ο τρόπος επίλυσης του προβλήματος έχει επιπτώσεις στο χρόνο εκτέλεσης

Προσπέλαση Διαμοιραζόμενων Δεδομένων

- Η προσπέλαση διαμοιραζόμενων δεδομένων απαιτεί προσεκτικό χειρισμό.
- Ας θεωρήσουμε δύο διεργασίες, κάθε μία από τις οποίες προσθέτει τη μονάδα σε μία κοινή μεταβλητή, x.
- Οι ενέργειες που κάνει κάθε διεργασία είναι η ανάγνωση της μεταβλητής x, η πρόσθεση της μονάδας στα περιεχόμενα της x και στη συνέχεια αποθήκευση της νέας τιμής στην μεταβλητή x.
- Στο σχήμα που ακολουθεί βλέπουμε τη χρονική στιγμή που εκτελούνται οι διάφορες ενέργειες από τις δύο διεργασίες.
- Αν η αρχική τιμή της x είναι 0, η τελική τιμή της x θα είναι 1 αντί 2 όπως θα ήταν επιθυμητό μετά την εκτέλεση των δύο διεργασιών

Instruction	Process 1	Process 2
`x = x + 1;`	`read x`	`read x`
	`compute x + 1`	`compute x + 1`
	`write to x`	`write to x`

Time



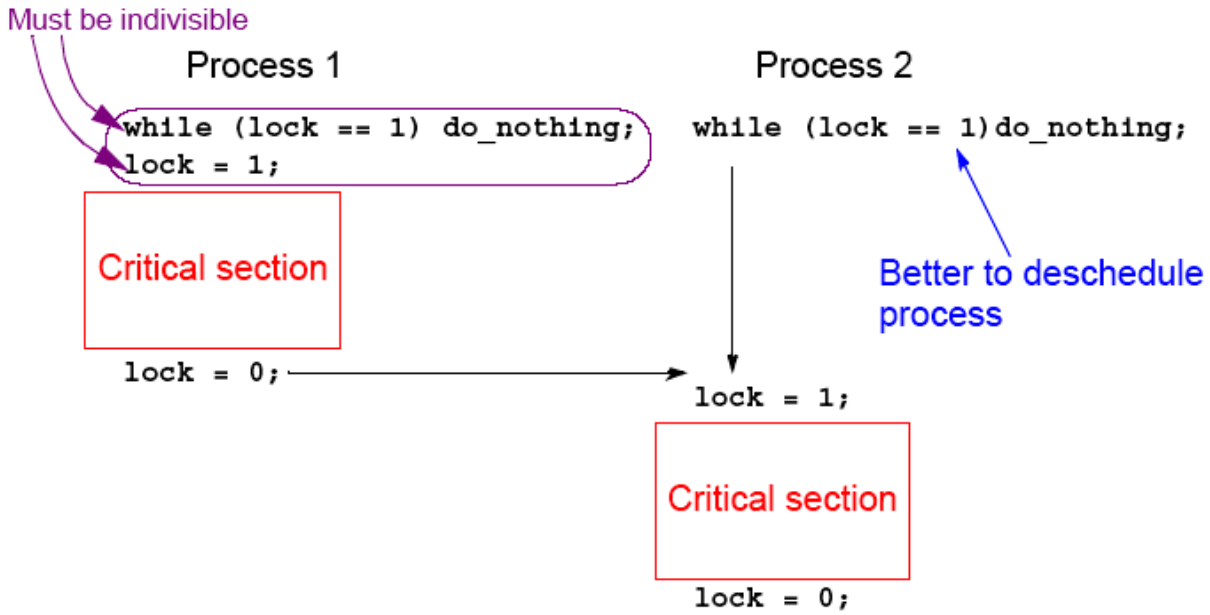
Κρίσιμο Τμήμα (Critical Section)

• Στο προηγούμενο παράδειγμα, το πρόβλημα προήλθε από την ταυτόχρονη εκτέλεση του ίδιου τμήματος κώδικα (εντολή x=x+1) από τις δύο διεργασίες.
• Ένα τμήμα κώδικα το οποίο πρέπει να εκτελείται από μία το πολύ διεργασία σε οποιαδήποτε χρονική στιγμή ονομάζεται κρίσιμο τμήμα (critical section).
• Ο μηχανισμός που εξασφαλίζει ότι μόνο μία διεργασία θα εισέρχεται πάντα στο κρίσιμο τμήμα ονομάζεται αμοιβαίος αποκλεισμός.
• Ο πιο απλός μηχανισμός που εξασφαλίζει αμοιβαίο αποκλεισμό των κρίσιμων τμημάτων είναι τα locks (κλειδαριές)
• Οι κλειδαριές είναι μεταβλητές boolean με την τιμή 1 να δηλώνει ότι μία διεργασία έχει εισέλθει στο κρίσιμο τμήμα και με 0 να δηλώνει ότι καμία διεργασία δεν έχει εισέλθει στο κρίσιμο τμήμα.

Οι κλειδαριές δουλεύουν όπως και η κλειδαριά μίας πόρτας:

Μία διεργασία που βρίσκει την «πόρτα» ενός κρίσιμου τμήματος ανοιχτή, εισέρχεται στο κρίσιμο τμήμα και κλειδώνει την πόρτα ώστε να αποτρέψει άλλες διεργασίες να εισέλθουν στο κρίσιμο τμήμα. Από τη στιγμή που η διεργασία ολοκληρώσει την εκτέλεση του κρίσιμου τμήματος, ξεκλειδώνει την πόρτα και φεύγει.

Έλεγχος του κρίσιμου τμήματος με busy waiting



• Ο πιο απλός τρόπος υλοποίησης των κλειδαριών είναι με τη χρήση ενός βρόχου όπου οι διεργασίες περιμένουν ενόσω η μεταβλητή lock είναι 1 πράγμα που σημαίνει ότι μία άλλη διεργασία είναι ήδη στο κρίσιμο τμήμα.
• Η διεργασία που εξέρχεται από το κρίσιμο τμήμα θέτει τη μεταβλητή lock ίση με 0 και επομένως κάποια άλλη διεργασία μπορεί να εισέλθει στο κρίσιμο τμήμα.
• Η διεργασία που περιμένει ελέγχει συνεχώς τη συνθήκη του βρόχου με αποτέλεσμα να δεσμεύει πολύτιμο χρόνο επεξεργασίας (busy waiting)
• Προτιμότερο θα ήταν η διεργασία να περιμένει σε μία ουρά αναμονής μέχρι να γίνει διαθέσιμο το κρίσιμο τμήμα.
• Άλλο ένα πρόβλημα που μπορεί να προκύψει είναι να εισέλθουν τελικά δύο διεργασίες στο κρίσιμο τμήμα.
• Για να αποφευχθεί αυτό θα πρέπει η εντολή while και η επόμενη της να εκτελούνται αδιαίρετα από μία διεργασία χωρίς να παρεμβάλλεται δηλ. η εκτέλεση κάποιας άλλης διεργασίας.

Λειτουργίες Κλειδώματος του Προτύπου Pthread

Οι κλειδαριές υλοποιούνται στο πρότυπο Pthreads με αμοιβαίως αποκλειόμενες μεταβλητές, γνωστές με όνομα “mutex”:

.
pthread_mutex_lock(&mutex1);
critical section
pthread_mutex_unlock(&mutex1);
.

Αν ένα νήμα φτάσει στην εντολή pthread_mutex_lock και βρεί την κλειδαριά mutex1 κλειδωμένη, περιμένει μέχρι η κλειδαριά να ξεκλειδώσει. Αν περισσότερα από ένα νήματα περιμένουν για να ανοίξει η κλειδαριά, το σύστημα θα επιλέξει ένα νήμα και θα του επιτρέψει να προχωρήσει. Μόνο το νήμα το οποίο έχει κλειδώσει τη μεταβλητή mutex1 μπορεί να τη ξεκλειδώσει με την εντολή pthread_mutex_unlock.
Ακολουθούν οι βασικές συναρτήσεις χειρισμού των μεταβλητών mutex

```
int pthread_mutex_lock (
    pthread_mutex_t *mutex_lock);
int pthread_mutex_unlock (
    pthread_mutex_t *mutex_lock);
int pthread_mutex_init (
    pthread_mutex_t        *mutex_lock,
    const pthread_mutexattr_t *lock_attr);
```

Η εντολή pthread-mutex_init αρχικοποιεί τη μεταβλητή mutex και θα πρέπει να εκτελείται νωρίτερα από τις υπόλοιπες.

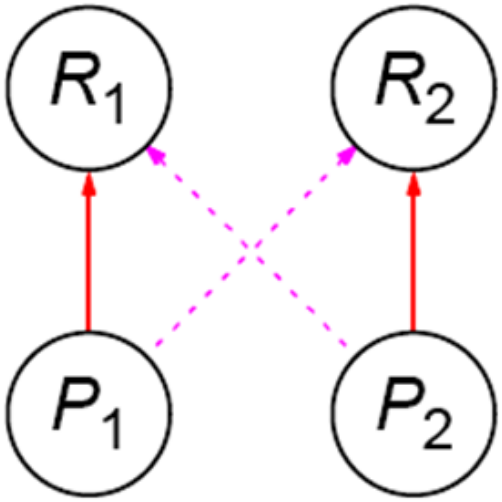
```
#include <pthread.h>
void *find_min(void *list_ptr);
pthread_mutex_t
minimum_value_lock;
int minimum_value,
partial_list_size;

main() {
/* declare and initialize data
structures and list */
minimum_value = MAX_INT;
pthread_init();
pthread_mutex_init(&minimum_value_
lock, NULL);

/* initialize lists, list_ptr, and
partial_list_size */
/* create and join threads here */
}
```

```
void *find_min(void *list_ptr) {
int *partial_list_pointer, my_min, i;
my_min = MIN_INT;
partial_list_pointer = (int *)
list_ptr;

for (i = 0; i < partial_list_size;
i++)
 if (partial_list_pointer[i] < my_min)
 my_min = partial_list_pointer[i];
/* lock the mutex associated with
minimum_value and update the variable
as required */

pthread_mutex_lock(&minimum_value_lock
);
 if (my_min < minimum_value)
 minimum_value = my_min;
 /* and unlock the mutex */


pthread_mutex_unlock(&minimum_value_lo
ck);
 pthread_exit(0);
}
```

• Στο προηγούμενο πρόγραμμα βρίσκεται το ελάχιστο μίας λίστας. Κάθε νήμα αναλαμβάνει να βρει το ελάχιστο σε ένα τμήμα της αρχικής λίστας και το συνολικό ελάχιστο αποθηκεύεται στη μεταβλητή minimum_value της οποίας ο έλεγχος και τροποποίηση προστατεύονται από τη μεταβλητή mutex `minimum_value_lock`.
• Το πρότυπο Pthreads προσφέρει επίσης τη ρουτίνα pthread_mutex_trylock() η οποία μπορεί να ελέγξει αν μία μεταβλητή mutex είναι κλειδωμένη χωρίς να μπλοκάρει το νήμα στο σημείο αυτό αν η μεταβλητή είναι κλειδωμένη.
• Αν η μεταβλητή δεν είναι κλειδωμένη, η trylock κλειδώνει τη μεταβλητή και επιστρέφει 0. Αν η μεταβλητή είναι κλειδωμένη επιστρέφει τη τιμή EBUSY χωρίς να μπλοκάρει τη διαδικασία.

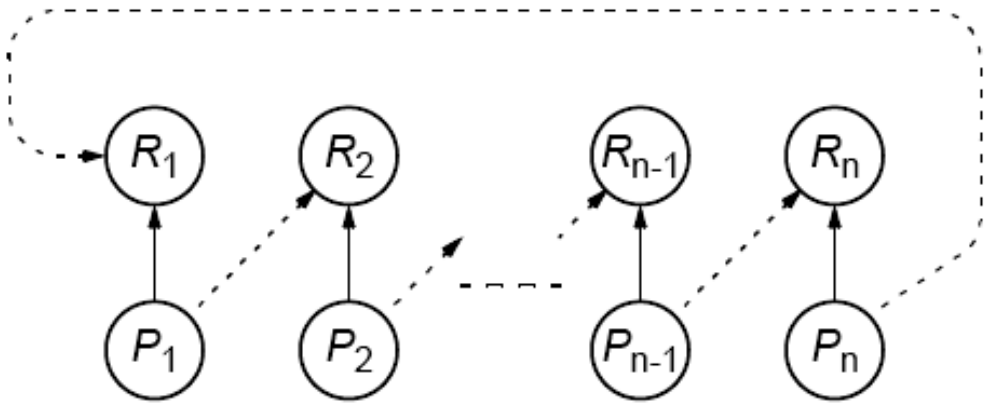
Αδιέξοδο (Deadlock)

• Είναι σημαντικό κατά το χειρισμό των κλειδαριών να αποφύγουμε την περίπτωση αδιεξόδου.
• Αδιέξοδο μπορεί να προκύψει όταν μία διεργασία ζητάει ένα πόρο ο οποίος είναι δεσμευμένος από μία άλλη διεργασία και αυτή η διεργασία ζητάει ένα πόρο ο οποίος κρατείται από τη πρώτη διεργασία.
• Στο σχήμα αριστερά, έχουμε ένα αδιέξοδο που εμπλέκει δύο διεργασίες.



Πόρος

Διεργασία



• Το αδιέξοδο μπορεί να συμβεί και σε κυκλική διάταξη όπου κάθε διεργασία κατέχει ένα πόρο ο οποίος ζητείται από μία άλλη διεργασία.

Σημαφόροι - Semaphores

Αμοιβαίο αποκλεισμό μπορούμε να πετύχουμε με το μηχανισμό των σημαφόρων. Ο σημαφόρος s είναι ένας θετικός ακέραιος (συμπεριλαμβανομένου του μηδενός) στον οποίο ορίζονται οι ακόλουθες δύο λειτουργίες:

Λειτουργία P: Η λειτουργία αυτή περιμένει μέχρι ο s να γίνει μεγαλύτερο του μηδενός και στη συνέχεια μειώνει τον s κατά 1 και επιτρέπει τη διαδικασία να συνεχίσει. Οι λειτουργίες που περιμένουν τοποθετούνται σε μία ουρά αναμονής και μία από αυτές επιλέγεται όταν εκτελεσθεί η λειτουργία V.

Λειτουργία V: Η λειτουργία αυτή αυξάνει τον s κατά 1 και ελευθερώνει μία από τις διεργασίες που περιμένουν (αν υπάρχουν)

Αμοιβαίος αποκλεισμός μπορεί να επιτευχθεί με ένα σημαφόρο ο οποίος έχει μόνο δύο τιμές, 0 ή 1 (δυαδικός σημαφόρος). Στο παράδειγμα που ακολουθεί, ο σημαφόρος s αρχικοποιείται στη τιμή 1 και στη συνέχεια μόνο μία διεργασία επιτρέπεται να εισέλθει στη κρίσιμη περιοχή

Process 1	Process 2	Process 3
Noncritical section	Noncritical section	Noncritical section
.....	.....	.....
P(s)	P(s)	P(s)
Critical section	Critical section	Critical section
V(s)	V(s)	V(s)
.....	.....	.....
Noncritical section	Noncritical section	Noncritical section

• Στη γενική περίπτωση, ένας σημαφόρος μπορεί να πάρει οποιαδήποτε τιμή πέρα από τις δυαδικές τιμές. Παρέχει ένα μέσο για να καταγράψουμε το πλήθος των πόρων που είναι διαθέσιμα προς χρήση και βρίσκουν εφαρμογή στα προβλήματα παραγωγών/καταναλωτών (producer/ consumer problems).

• Αν και το πρότυπο Pthreads δεν παρέχει σημαφόρους, μπορούμε να χρησιμοποιήσουμε τους σημαφόρους που περιέχει το POSIX Unix. Οι βασικές λειτουργίες χειρισμού των σημαφόρων είναι οι ακόλουθες:

```
#include <semaphore.h>
int sem init( sem t*  semaphore p /* out */, int shared /* in */, unsigned
initial val /* in */);
int sem destroy(sem t* semaphore p /* in/out  */);
int sem post(sem t* semaphore p / in/out  */);
int sem wait(sem t* semaphore p /* in/out */);
```

• Η λειτουργία sem_init αρχικοποιεί το σημαφόρο σε μία αρχική τιμή, η sem_destroy καταστρέφει το σημαφόρο, η sem_post είναι η λειτουργία V και η sem_wait είναι η λειτουργία P.

Στο παράδειγμα αυτό, τα νήματα είναι σε διάταξη δακτυλίου και κάθε νήμα στέλνει στο επόμενο του στη διάταξη. Κάθε νήμα περιμένει στο δικό του σημαφόρο μέχρι το προηγούμενο από αυτό νήμα να τον ξεκλειδώσει

```
/ messages is allocated and initialized to NULL in main /
/ semaphores is allocated and initialized to 0 (locked) in
main /
 void Send msg(void rank) {
 long my rank = (long) rank;
 long dest = (my rank + 1) % thread count;
 char my msg = malloc(MSG MAXsizeof(char));

 sprintf(my msg, "Hello to %ld from %ld", dest, my rank);
 messages[dest] = my msg;
 sem post(&semaphores[dest])
/ ‘‘Unlock’’ the semaphore of dest /

 / Wait for our semaphore to be unlocked /
 sem wait(&semaphores[my rank]);
 printf("Thread %ld > %snn", my rank, messages[my rank]);

return NULL;
} / Send msg /
```

Μεταβλητές Συνθήκης (Condition Variables)

• Συχνά, ένα κρίσιμο τμήμα εκτελείται μόνο όταν μία συγκεκριμένη συνθήκη ικανοποιηθεί π.χ. όταν μία μεταβλητή πάρει μία συγκεκριμένη τιμή.
• Με τη χρήση κλειδαριών, η μεταβλητή θα πρέπει να ελέγχεται ανά τακτά χρονικά διαστήματα με αποτέλεσμα τη σπατάλη πολύτιμων υπολογιστικών πόρων.
•Το πρόβλημα αυτό μπορεί να αποφευχθεί με τη χρήση των μεταβλητών συνθήκης (condition variables). Ουσιαστικά υλοποιείται η ακόλουθη λειτουργικότητα:

```
lock mutex;
if condition has occurred
    signal thread(s);
else { unlock the mutex and block;
 }
unlock mutex; /* when thread is unblocked, mutex is relocked */
```

Οι βασικές ρουτίνες χειρισμού των μεταβλητών συνθήκης είναι οι ακόλουθες:

```
int pthread_cond_wait(pthread_cond_t *cond,
    pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *cond,
    const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

• Η pthread_cond_init και η pthread_cond_destroy δημιουργεί και καταστρέφει τη μεταβλητή συνθήκης αντίστοιχα.
• Η pthread_cond_signal θα ξεμπλοκάρει ένα νήμα που περιμένει ενώ pthread_cond_broadcast ξεμπλοκάρει όλες τα νήματα που περιμένουν.
• Η pthread_cond_wait θα ξεκλειδώσει τη μεταβλητή mutex και στη συνέχεια το νήμα θα μπλοκάρει μέχρι να ξεμπλοκάρει από τη κλήση της συνάρτησης pthread_cond_signal ή της συνάρτησης pthread_cond_broadcast. Ουσιαστικά κατά τη κλήση της pthread_cond_wait εκτελούνται αδιαίρετα τα εξής:

```
pthread mutex unlock(&mutex p);
wait on signal(&cond var p);
pthread mutex lock(&mutex p);
```

• Πάντα μία μεταβλητή συνθήκης θα πρέπει να συνδυάζεται με μία μεταβλητή κλειδώματος

• Ένα παράδειγμα χρήσης των μεταβλητών συνθήκης ακολουθεί:

```
pthread_cond_t cond1;
pthread_mutex_t mutex1;
.....
pthread_cond_init(&cond1, NULL);
pthread_mutex_init (&mutex1, NULL);
```

```
action()
{
.....
pthread_mutex_lock(&mutex1);
while (c !=0)

pthread_cond_wait(cond1,mutex
1);
pthread_mutex_unlock(&mutex1)
;
take_action();
.....
}
```

```
counter()
{
.....
pthread_mutex_lock(&mutex1);
c--;
if (c==0)
pthread_cond_signal(cond1);
pthread_mutex_unlock(&mutex1);
....
}
```

• Στο προηγούμενο παράδειγμα, ένα ή περισσότερα νήματα αναλαμβάνουν δράση όταν ο μετρητής c γίνει μηδέν. Ένα άλλο νήμα είναι υπεύθυνο για τη μείωση αυτού του μετρητή.
• Επίσης γίνεται η υπόθεση ότι η ρουτίνα action φτάνει στο κρίσιμο τμήμα πρώτα, αφού τα σήματα (signals) δεν αποθηκεύονται κάπου και μπορούν να χαθούν αν η ρουτίνα counter φτάσει πρώτο στο κρίσιμο τμήμα.

Ο κώδικας που ακολουθεί υλοποιεί ένα φράγμα με τη βοήθεια των μεταβλητών συνθήκης. Συγκεκριμένα, όλα τα νήματα πρέπει να περιμένουν μέχρι “thread count” νήματα φθάσουν στο φράγμα.

```
/* Shared */
int counter = 0;
pthread mutex t mutex;
pthread cond t cond var;
...
void Thread work(. . .) { . . .
/* Barrier */
pthread mutex lock(&mutex);
counter++;
if (counter == thread count) {
counter = 0;
pthread cond broadcast(&cond var);
} else {
 while (pthread cond wait(&cond var, &mutex)
!= 0);
}
thread mutex unlock(&mutex);
...
}
```

OpenMP

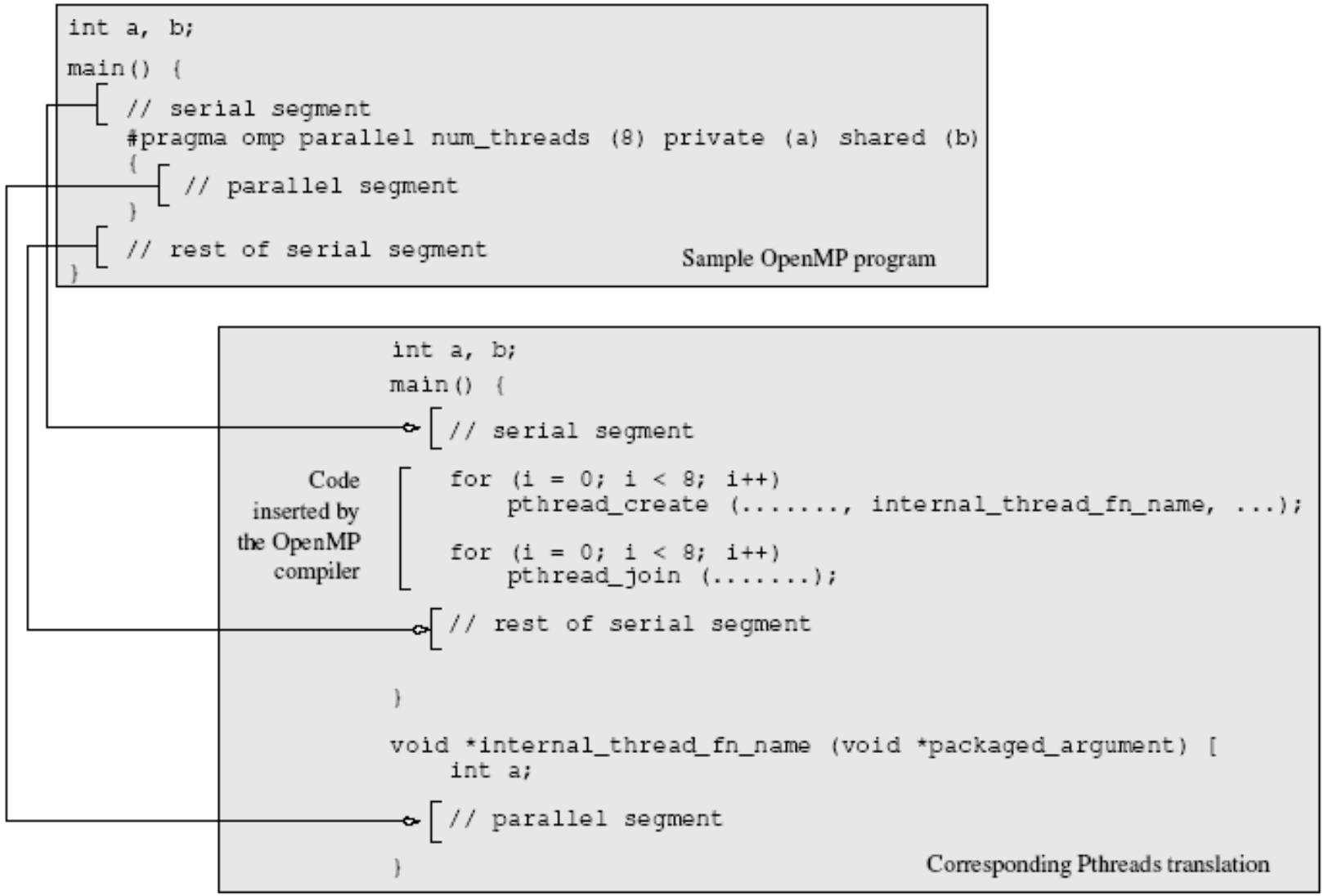
• Η OpenMP είναι ένα πρότυπο για το Παράλληλο Προγραμματισμό.
• Είναι μία εναλλακτική πρόταση σε σχέση με τις βιβλιοθήκες παράλληλου προγραμματισμού που έχουμε μελετήσει (MPI και Pthreads).
• Στην OpenMP, ξεκινούμε από μια γλώσσα ακολουθιακού προγραμματισμού και μετατρέπουμε τμήματα του κώδικα σε παράλληλο κώδικα εισάγοντας στο πρόγραμμα συγκεκριμένες οδηγίες (directives) μεταγλωττιστή.
• Οι οδηγίες αυτές προσδιορίζουν το τρόπο παραλληλοποίησης του σχετικού κώδικα, παρέχουν λεπτομέρειες για το συγχρονισμό μεταξύ των νημάτων, τον χειρισμό των δεδομένων χωρίς να χρειάζεται ο προγραμματιστής να χρησιμοποιήσει μεταβλητές κλειδωματος, μεταβλητές συνθήκης κτλ.
• Οι οδηγίες OpenMP στη C και C++ βασίζονται στην οδηγία `#pragma` του μεταγλωττιστή.
• Μία οδηγία αποτελείται από το όνομα της οδηγίας και ακολουθείται από μία λίστα προτάσεων.

```
#pragma omp directive [clause list]
```

• Τα προγράμματα σε OpenMP εκτελούνται ακολουθιακά μέχρι να συναντήσουν την οδηγία `parallel`, η οποία δημιουργεί μία ομάδα νημάτων:

```
#pragma omp parallel [clause list]
/* structured block */
```

• Όταν το κύριο νήμα συναντήσει την οδηγία `parallel` γίνεται ο master αυτής της ομάδας των νημάτων και το αναγνωριστικό αυτού του νήματος είναι 0 μέσα σε αυτή την ομάδα.



Το παράδειγμα δείχνει την αντιστοιχεία των εντολών της OpenMP και των εντολών του Ptheads

Η λίστα των προτάσεων προσδιορίζει τον υπό συνθήκη παραλληλισμό, το πλήθος των νημάτων ή τον χειρισμό των δεδομένων.

Υπό συνθήκη παραλληλισμός: Η συνθήκη `if (scalar expression)` προσδιορίζει αν η οδηγία `parallel` θα έχει ως αποτέλεσμα τη δημιουργία νημάτων.

Βαθμός παραλληλισμού: Η πρόταση `num_threads(integer expression)` προσδιορίζει το πλήθος των νημάτων που θα δημιουργηθούν.

Χειρισμός δεδομένων: Η πρόταση `private (variable list)` προσδιορίζει ποιες μεταβλητές θα είναι τοπικές σε κάθε νήμα. Η πρόταση `shared (variable list)` προσδιορίζει ποιες μεταβλητές θα είναι κοινές σε όλες τις διεργασίες.

Για παράδειγμα:

```
#pragma omp parallel if (is_parallel== 1) num_threads(8)
    private (a) shared (b) {
 /* structured block */
}
```

Αν η τιμή της μεταβλητής `is_parallel` είναι ίση με 1, οκτώ νήματα δημιουργούνται.
Η μεταβλητή a είναι ιδιωτική σε κάθε νήμα ενώ η μεταβλητή b είναι κοινή σε όλα τα νήματα.
Η πρόταση `reduction` προσδιορίζει πως πολλαπλά τοπικά αντίγραφα μίας μεταβλητής σε διαφορετικά νήματα συνδυάζονται σε ένα μοναδικό αντίγραφο στο νήμα master όταν όλα τα νήματα τερματίζουν.
Η πρόταση `reduction` συντάσσεται ως εξής: `reduction (operator: variable list)`.
Οι μεταβλητές στη λίστα ορίζονται αυτόματα ως ιδιωτικές σε κάθε νήμα. Ο τελεστής `operator` μπορεί να είναι ένας από τους ακόλουθους τελεστές: `+`, `*`, `-`, `&`, `|`, `^`, `&&`, and `||`.

```
#pragma omp parallel reduction(+: sum) num_threads(8) {
/* compute local sums here */
}
/*sum here contains sum of all local instances of sums */
```

Ο παραπάνω κώδικας υπολογίζει την τιμή του π με τη χρήση της οδηγίας `reduction`

```
/*
******************************************************
*
An OpenMP version of a threaded program to compute
PI.
******************************************************
* */
#pragma omp parallel default(private) shared
(npoints)
    reduction(+: sum) num_threads(8)
{
    num_threads = omp_get_num_threads();
    sample_points_per_thread = npoints / num_threads;
    sum = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x
        =(double)(rand_r(&seed))/(double)((2<<14)-1);
        rand_no_y
        =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) <
            0.25)
            sum ++;
    }
}
```

Η οδηγία `parallel` μπορεί να χρησιμοποιηθεί μαζί με τις οδηγίες `for` και `sections` για να προσδιορίζουμε πως θα παραλληλοποιηθεί ένας επαναληπτικός βρόχος ή πως θα εκτελεστούν παράλληλα διαφορετικά τμήματα κώδικα.

Η οδηγία `for` χρησιμοποιείται για να μοιράσουμε τις επαναλήψεις ενός βρόχου στα διαθέσιμα νήματα. Η γενική μορφή για την οδηγία for έχει ως εξής:

```
#pragma omp for [clause list]
    /* for loop */
```

Με τη χρήση της οδηγίας for, ο υπολογισμός του π μπορεί να απλοποιηθεί τώρα ως εξής:

```
#pragma omp parallel default(private) shared (npoints)
reduction(+: sum) num_threads(8)
{
    sum = 0;
    #pragma omp for
    for (i = 0; i < npoints; i++) {
        rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
        rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            sum ++;
    }
}
```

• Η πρόταση `schedule` της οδηγίας `for` καθορίζει τον τρόπο που θα μοιραστούν οι επαναλήψεις στα νήματα.
• Η γενική μορφή της οδηγίας `schedule` είναι `schedule(scheduling_class[, chunksize])`.
• Η OpenMP υποστηρίζει τέσσερις κλάσεις χρονοδρομολόγησης: `static, dynamic, guided` και `runtime`

```
sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
 reduction(+:sum) schedule(static,1)
for (i = 0; i <= n; i++)
sum += f(i);
```

• Στο παράδειγμα αν υπάρχουν τρία νήματα (thread_count=3) και (n=9) επαναλήψεις , τα νήματα θα αναλάβουν τις εξής επαναλήψεις:

Νήμα 1: 1,4,7
Νήμα 2: 2, 5, 8
Νήμα 3: 3, 6, 9

• Αν είχαμε χρησιμοποιήσει την οδηγία `schedule(static,2),` τα νήματα θα αναλάβουν τις εξής επαναλήψεις:

Νήμα 1: 1,2, 7, 8
Νήμα 2: 3, 4, 9
Νήμα 3: 5, 6

• Γενικά, στη δρομολόγηση static, οι επαναλήψεις κατανέμονται στις διεργασίες εκ των προτέρων πριν ο βρόχος εκτελεστεί. Στη δρομολογήσεις dynamic και guided, οι επαναλήψεις κατανέμονται στα νήματα δυναμικά καθώς ο βρόχος εκτελείται. Όταν ένα νήμα ολοκληρώνει ένα σύνολο επαναλήψεων, μπορεί να ζητήσει νέες επαναλήψεις από το σύστημα. Η δρομολόγηση runtime καθορίζεται την ώρα της εκτέλεσης ενώ στην auto, η δρομολόγηση καθορίζεται είτε στη φάση της

Πέρα από τη παράλληλη εκτέλεση των επαναλήψεων ενός βρόχου, η OpenMP υποστηρίζει και την παράλληλη υλοποίηση οποιοδήποτε τμημάτων κώδικα με τη χρήση της οδηγίας `sections` . Η γενική μορφή της οδηγίας `sections` έχει ως εξής:

```
#pragma omp sections [clause list]
{
    [#pragma omp section
        /* structured block */
    ]
    [#pragma omp section
        /* structured block */
    ]
    ...
}
```

Στο παράδειγμα που ακολουθεί, δημιουργούνται τρία νήματα τα οποία εκτελούν παράλληλα τα έργα A, B και C.

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
        }
        #pragma omp section
        {
            taskC();
        }
    }
}
```

Οδηγίες συγχρονισμού

• Η OpenMP προσφέρει διάφορες οδηγίες για το συγχρονισμό των νημάτων
• Η οδηγία critical επιτρέπει σε ένα μόνο νήμα να εισέλθει στο σχετικό block κώδικα. Συγκεκριμένα, όταν ένα ή περισσότερα νήματα φτάσουν στην οδηγία critical:

```
#pragma omp critical name
    structured_block
```

περιμένουν μέχρι κανένα άλλο νήμα να μην εκτελεί το ίδιο κρίσιμο τμήμα και τότε μόνο ένα από αυτά τα νήματα προχωρεί για να εκτελέσει το κρίσιμο τμήμα. Στο κώδικα που ακολουθεί η ουρά θεωρείται κρίσιμο τμήμα και η πρόσβαση σε αυτή προστατεύεται με τις οδηγίες critical.

```
#pragma omp parallel sections
{
 #pragma parallel section
{
 /* producer thread */
 task = produce_task();
 #pragma omp critical ( task_queue)
{
 insert_into_queue(task);
}}
 #pragma parallel section
{
 /* consumer thread */
 #pragma omp critical ( task_queue)
{
task = extract_from_queue(task);
}
consume_task(task);
}}
```

Επίσης, η OpenMP προσφέρει και το μηχανισμό του φράγματος που υλοποιείται με την οδηγία barrier

```
#pragma omp barrier
```

Με την εισαγωγή αυτής της οδηγίας στο κώδικα, τα νήματα περιμένουν να φτάσουν όλα μαζί στο φράγμα πριν προχωρήσουν.

Η οδηγία atomic

```
#pragma omp atomic
    expression_statement
```

υλοποιεί ένα κρίσιμο τμήμα αποδοτικά όταν το κρίσιμο τμήμα είναι μία απλή εντολή (πρόσθεση, αφαίρεση ή άλλη αριθμητική εντολή σε μία μεταβλητή η οποία ορίζεται από την expression_statement).

Παραδείγματα προγραμμάτων διαμοιραζόμενης μνήμης

Παραδείγματα

Άθροιση των στοιχείων ενός πίνακα 1000 στοιχείων, **a[1000]**:

```
int sum, a[1000];
        sum = 0;
        for (i = 0; i < 1000; i++)
                sum = sum + a[i];
```

Διεργασίες UNIX

Ο Υπολογισμός διαιρείται σε δύο τμήματα. Το ένα τμήμα θα αθροίζει τις ζυγές θέσεις και το άλλο τις μονές, δηλ.,

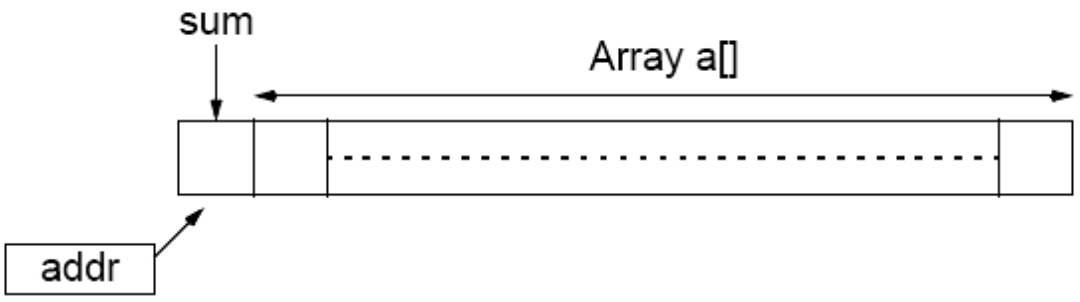
Process 1
sum1 = 0;
for (i = 0; i < 1000; i = i + 2)
sum1 = sum1 + a[i];

Process 2
sum2 = 0;
for (i = 1; i < 1000; i = i + 2)
sum2 = sum2 + a[i];

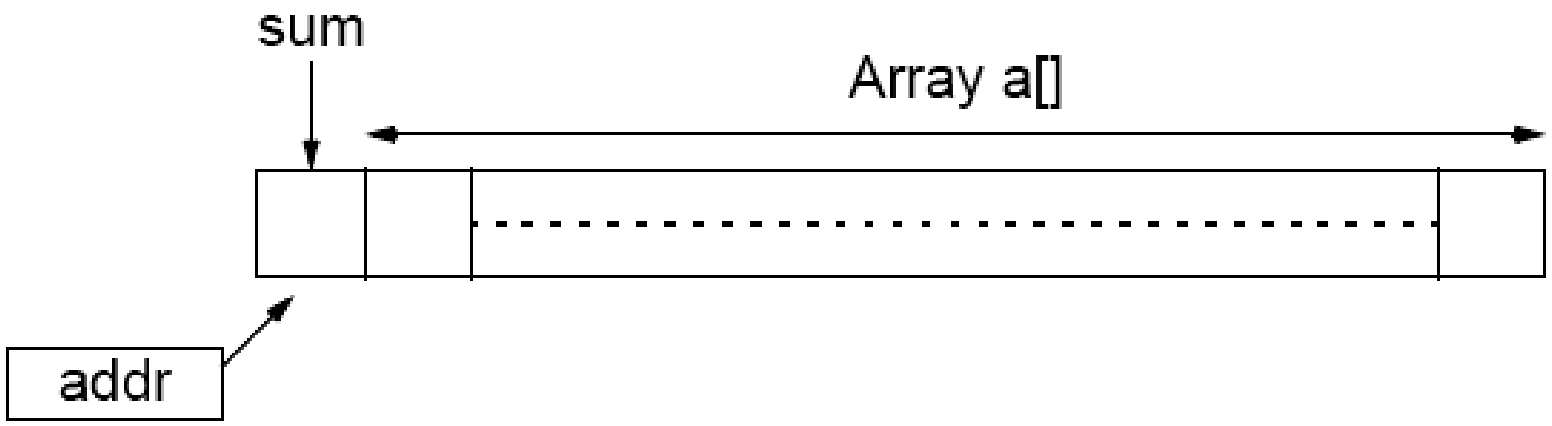
Κάθε διεργασία θα προσθέσει το αποτέλεσμά της στη μεταβλητή sum:

sum = sum + sum1; **sum = sum + sum2;**

Η μεταβλητή Sum επειδή είναι κοινή μεταβλητή θα πρέπει να προστατευθεί με μηχανισμό κλειδώματος. Επίσης και ο πίνακας Α είναι κοινός στις δύο διεργασίες:



Θέσεις μνήμης με πρόσβαση και από τις δύο διεργασίες UNIX



```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <errno.h>
#define array_size 1000        /* no of elements in shared memory */
extern char *shmat();
void P(int *s);
void V(int *s);
int main()
{
int shmid, s, pid;             /* shared memory, semaphore, proc id */
char *shm;                     /*shared mem. addr returned by shmat()*/
int *a, *addr, *sum;           /* shared data variables*/
int partial_sum;               /* partial sum of each process */
int i;
                               /* initialize semaphore set */
int init_sem_value = 1;
s = semget(IPC_PRIVATE, 1, (0600 | IPC_CREAT))
if (s == -1) {                 /* if unsuccessful*/
   perror("semget");
   exit(1);
}
if (semctl(s, 0, SETVAL, init_sem_value) < 0) {
   perror("semctl");
   exit(1);
}
```

```
                                        /* create segment*/
shmid = shmget(IPC_PRIVATE,(array_size*sizeof(int)+1),
   (IPC_CREAT|0600));
if (shmid == -1) {
   perror("shmget");
   exit(1);
}
                                      /* map segment to process data space */
shm = shmat(shmid, NULL, 0);
                                      /* returns address as a character*/
if (shm == (char*)-1) {
   perror("shmat");
   exit(1);
}
```

```
addr = (int*)shm;                    /* starting address */
sum = addr;                          /* accumulating sum */
addr++;
a = addr;                            /* array of numbers, a[] */

*sum = 0;
for (i = 0; i < array_size; i++) /* load array with numbers */
   *(a + i) = i+1;

pid = fork();                        /* create child process */
if (pid == 0)    {                   /* child does this */
   partial_sum = 0;
   for (i = 0; i < array_size; i = i + 2)
     partial_sum += *(a + i);
else {                               /* parent does this */
   partial_sum = 0;
   for (i = 1; i < array_size; i = i + 2)
     partial_sum += *(a + i);
}
P(&s);                               /* for each process, add partial sum */
   *sum += partial_sum;
V(&s);
```

```
printf("\nprocess pid = %d, partial sum = %d\n", pid, partial_sum);
if (pid == 0) exit(0); else wait(0);       /* terminate child proc */
printf("\nThe sum of 1 to %i is %d\n", array_size, *sum);

                                           /* remove semaphore */
if (semctl(s, 0, IPC_RMID, 1) == -1) {
   perror("semctl");
   exit(1);
}
                                           /* remove shared memory */
if (shmctl(shmid, IPC_RMID, NULL) == -1) {
   perror("shmctl");
   exit(1);
}
}                                          /* end of main */
```

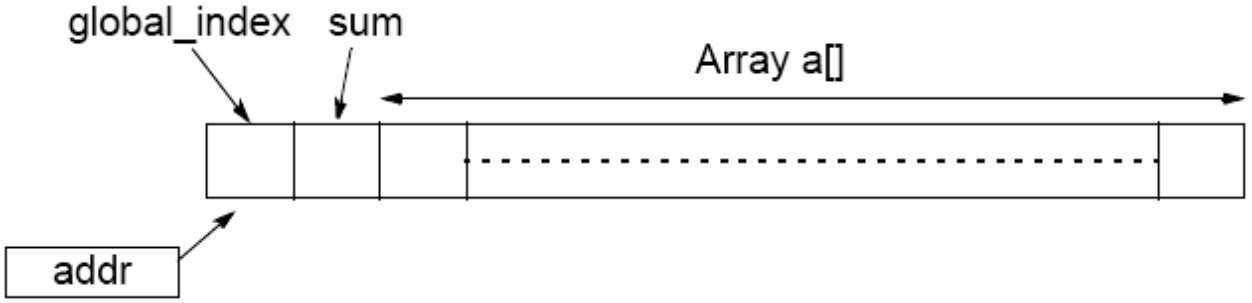
```
void P(int *s)                                  /* P(s) routine*/
{
   struct sembuf sembuffer, *sops;
   sops = &sembuffer;
   sops->sem_num = 0;
   sops->sem_op = -1;
   sops->sem_flg = 0;
   if (semop(*s, sops, 1) < 0) {
      perror("semop");
      exit(1);
   }
return;
}

void V(int *s)                                  /* V(s) routine */
{
   struct sembuf sembuffer, *sops;
   sops = &sembuffer;
   sops->sem_num = 0;
   sops->sem_op = 1;
   sops->sem_flg = 0;
   if (semop(*s, sops, 1) <0) {
      perror("semop");
      exit(1);
   }
return;
}
```

SAMPLE OUTPUT

```
process pid = 0, partial sum = 250000
process pid = 26127, partial sum = 250500
The sum of 1 to 1000 is 500500
```

Παράδειγμα με Pthreads

Δημιουργούνται n νήματα και κάθε νήμα παίρνει αριθμούς από τη λίστα και τους αθροίζει. Όταν ολοκληρωθούν οι τοπικοί υπολογισμοί, τα νήματα προσθέτουν τα μερικά τους αποτελέσματα σε μία κοινά διαμοιραζόμενη μεταβλητή sum.
Επίσης χρησιμοποιείται η κοινά διαμοιραζόμενη μεταβλητή global_index
Μετά από κάθε ανάγνωση της global_index, η μεταβλητή αυτή αυξάνεται για να δείχνει το επόμενο προς επεξεργασία στοιχείο του πίνακα a[].
Το τελικό αποτέλεσμα θα αποθηκευθεί στη μεταβλητή sum η οποία όπως και προηγουμένως είναι κοινή σε όλα τα νήματα και επομένως πρέπει προστατεύεται από μηχανισμό κλειδώματος.



```
#include <stdio.h>
#include <pthread.h>
#define array_size 1000
#define no_threads 10
                                /* shared data */
int a[array_size];              /* array of numbers to sum */
int global_index = 0;           /* global index */
int sum = 0;                    /* final result, also used by slaves */
pthread_mutex_t mutex1;         /* mutually exclusive lock variable */
void *slave(void *ignored)      /* Slave threads */
{
int local_index, partial_sum = 0;
do {
   pthread_mutex_lock(&mutex1);/* get next index into the array */
      local_index = global_index;/* read current index & save locally*/
      global_index++;           /* increment global index */
   pthread_mutex_unlock(&mutex1);

   if (local_index < array_size) partial_sum += *(a + local_index);
} while (local_index < array_size);

pthread_mutex_lock(&mutex1);  /* add partial sum to global sum */
   sum += partial_sum;
pthread_mutex_unlock(&mutex1);

return ();                      /* Thread exits */
}
```

```
   main () {
int i;
pthread_t thread[10];                    /* threads */
pthread_mutex_init(&mutex1,NULL);        /* initialize mutex */

for (i = 0; i < array_size; i++)         /* initialize a[] */
   a[i] = i+1;

for (i = 0; i < no_threads; i++)         /* create threads */
   if (pthread_create(&thread[i], NULL, slave, NULL) != 0)
      perror("Pthread_create fails");

for (i = 0; i < no_threads; i++)         /* join threads */
   if (pthread_join(thread[i], NULL) != 0)
      perror("Pthread_join fails");
printf("The sum of 1 to %i is %d\n", array_size, sum);
}                                        /* end of main */
```

SAMPLE OUTPUT

```
The sum of 1 to 1000 is 500500
```

Παράδειγμα σε Java

```
public class Adder
{
   public int[] array;
   private int sum = 0;
   private int index = 0;
   private int number_of_threads = 10;
   private int threads_quit;

   public Adder()
   {
      threads_quit = 0;
      array = new int[1000];
      initializeArray();
      startThreads();
   }

   public synchronized int getNextIndex()
   {
      if(index < 1000) return(index++); else return(-1);
   }
```

```
    public synchronized void addPartialSum(int partial_sum)
    {
       sum = sum + partial_sum;
       if(++threads_quit == number_of_threads)
          System.out.println("The sum of the numbers is " + sum);
    }

    private void initializeArray()
    {
       int i;
       for(i = 0;i < 1000;i++) array[i] = i;
    }

    public void startThreads()
    {
       int i = 0;
       for(i = 0;i < 10;i++)
       {
          AdderThread at = new AdderThread(this,i);
          at.start();
       }
}
    {
       Adder a = new Adder();
    }

}
```

```
   public static void main(String args[])

class AdderThread extends Thread
{
   int partial_sum = 0;
   Adder parent;
   int number;
   public AdderThread(Adder parent,int number)
   {
      this.parent = parent;
      this.number = number;
   }

   public void run()
   {
      int index = 0;
      while(index != -1) {
         partial_sum = partial_sum + parent.array[index];
         index = parent.getNextIndex();
      }
      System.out.println("Partial sum from thread " + number + " is "
           + partial_sum);
      parent.addPartialSum(partial_sum);
   }
}
```