Pthreads

Computer Science Department, University of Crete

Παράλληλος Προγραμματισμός

Διεργασίες και Νήματα

- Threads, Νήματα
- Χρησιμοποιούνται για την υλοποίηση παραλληλισμού σε παράλληλους επεξεργαστές με κοινή μνήμη (shared memory)
- Ιστορικά κάθε κατασκευαστής επεξεργαστών υλοποιούσε διαφορετικά τα threads
  - Δυσκολίες σε portability
- Συχνά προβλήματα συμβατότητας APIs, ιδιωτικά APIs, κλπ
- Λύση: POSIX standard

POSIX

- Portable Operating System Interface
  - X
  - Τυποποίηση του interface μεταξύ λειτουργικού συστήματος και εφαρμογών
- Λειτουργικά *nix
  - Linux, MacOs, Android, κλπ
  - Υποστήριξη με runtime environment σε Windows*
- Το πρότυπο interface περιέχει
  - System calls, signals, process management
  - C standard library
  - Κέλυφος και utility προγράμματα
  - Διαχείριση Threads

POSIX Threads

- Πρότυπο που ορίζει το interface για threads
- Διάφορες υλοποιήσεις
  - Μπορεί να χρησιμοποιεί user ή kernel threads
  - Τα προγράμματα δεν χρειάζονται αλλαγή
  - Η συμπεριφορά και η ταχύτητα μπορεί να διαφέρει
- API
  - Διαχείριση threads
  - Μεταβλητές mutex, συγχρονισμός με locks
  - Condition variables, συγχρονισμός με wait
  - Σημαφόροι

Τι είναι Threads

- Τυπικά: *μια ανεξάρτητη σειρά εντολών που μπορεί να εκτελεστεί από το λειτουργικό σύστημα.*
- Μια διαδικασία που εκτελείται ανεξάρτητα από το κυρίως πρόγραμμα που την περιέχει.
- Ένα πρόγραμμα θα μπορούσε να περιέχει πολλές «διαδικασίες».
- Το λειτουργικό σύστημα μπορεί να τις εκτελέσει ανεξάρτητα από το κυρίως πρόγραμμα, ταυτόχρονα με αυτό.

Διεργασία (Process)

- Μια διεργασία (process) είναι όλα τα δεδομένα που διατηρεί το λειτουργικό σύστημα για να εκτελέσει ένα πρόγραμμα, όση ώρα εκτελείται.
- Περιέχει πληροφορίες για το πρόγραμμα που εκτελείται και την κατάστασή του, όπως:
  - Process ID, process group ID, user ID, and group ID
  - Environment
  - Working directory
  - Program instructions
  - Registers
  - Stack
  - Heap
  - File descriptors
  - Signal actions
  - Shared libraries
  - Αντικείμενα για επικοινωνία μεταξύ διεργασιών: ουρές μηνυμάτων, pipes, semaphores, περιοχές κοινής μνήμης, κλπ.

Process state

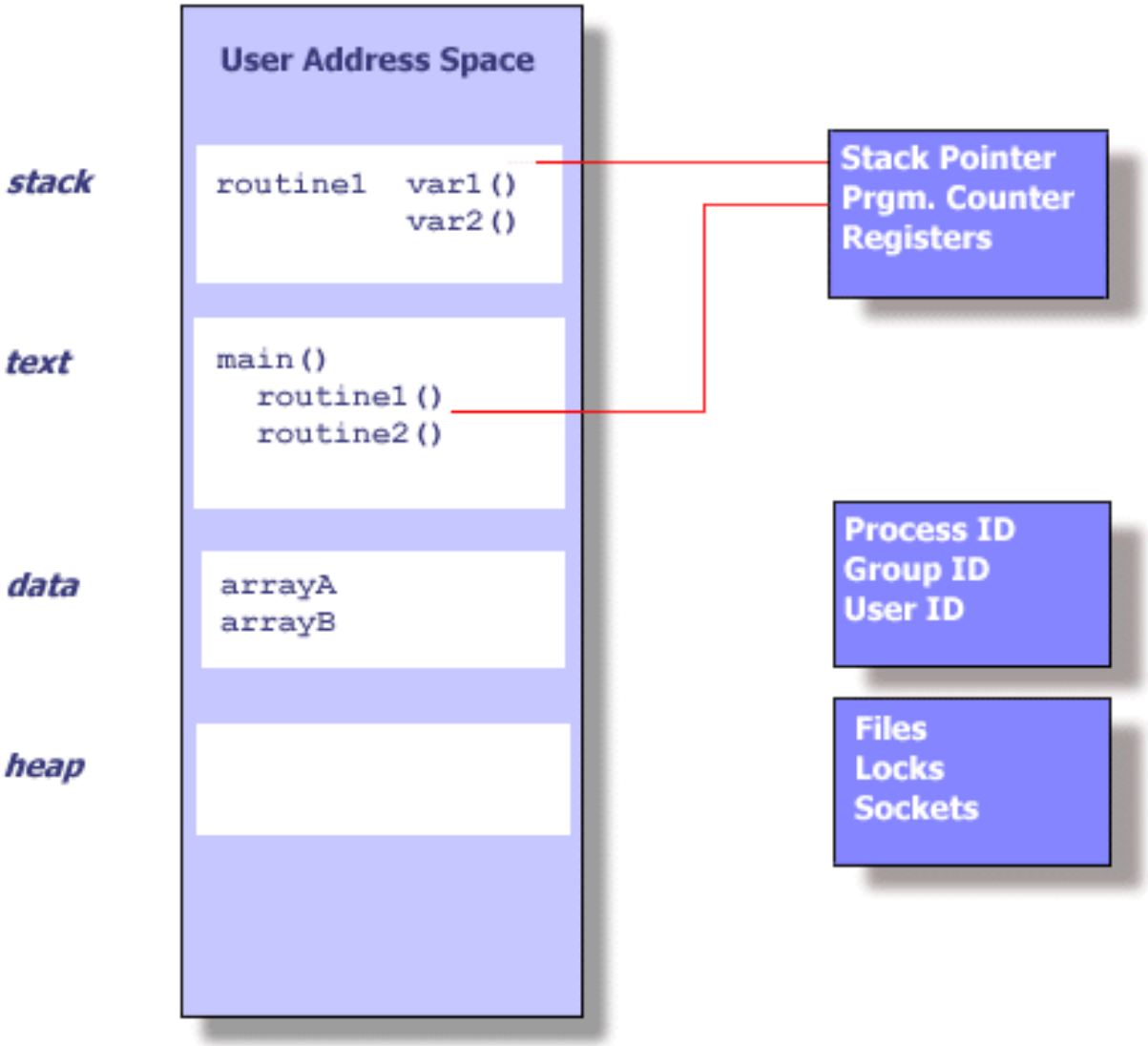


Figure source: https://computing.llnl.gov

Threads

- Κάθε thread υπάρχει εντός μιας διεργασίας
- Χρησιμοποιεί τα δεδομένα που ορίζουν τη διεργασία
- Ορίζει ξανά (δικά του) τα απολύτως απαραίτητα για την εκτέλεση από το λειτουργικό σύστημα:
  - Stack pointer
  - Registers
  - Priority, scheduling policy
  - Signals
  - (Συν δεδομένα χρήστη που ορίζονται per-thread)
- Μοιράζεται τα υπόλοιπα με το process
  - Αλλαγές στην κατάσταση της διεργασίας είναι κοινές (π.χ., σε αρχεία)
  - Η ίδια διεύθυνση μνήμης είναι η ίδια θέση μνήμης για όλους
  - Μπορεί να προκύψει ταυτόχρονη πρόσβαση στη μνήμη
  - Όταν δεν πρέπει να γίνει ταυτόχρονη πρόσβαση, χρειάζεται συγχρονισμός

Thread state

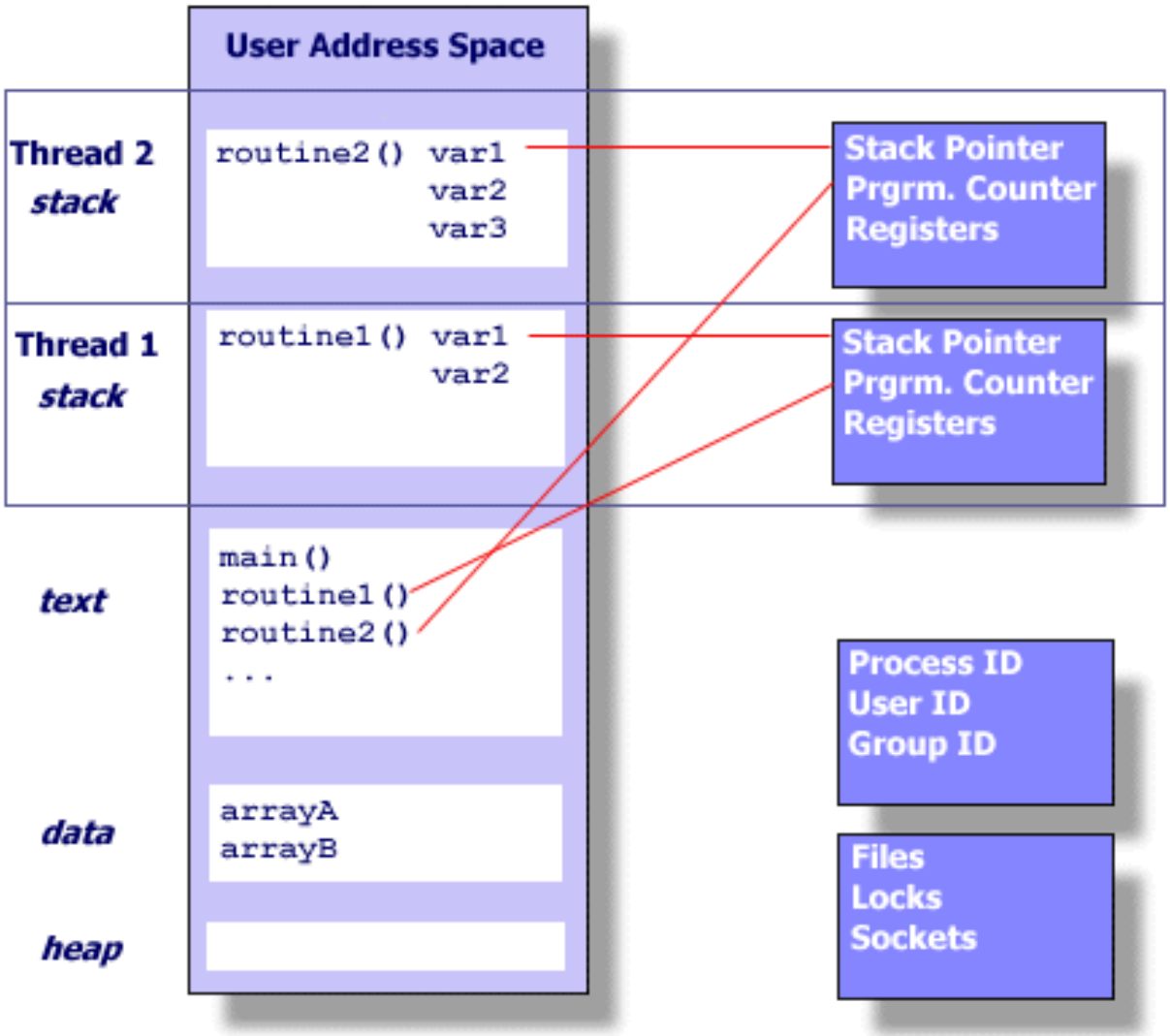


Figure source: https://computing.llnl.gov

Pthreads

- Χρήση ως βιβλιοθήκη
  - `#include <pthread.h>`
  - Μεταγλώττιση με `-pthread`
  - Σε κάποια βιβλία/tutorials χρησιμοποιείται το `-lpthread` στη μεταγλώττιση
  - ΜΗΝ χρησιμοποιείτε το `-lpthread` στη μεταγλώττιση
  - Μόνο το `-pthread`

Δημιουργία – pthread_create

```
int pthread_create(pthread_t *tid,
  const pthread_attr_t * attr,
  void * (*start_routine) (void*),
  void * arg);
```

- Δημιουργεί ένα thread που μόλις αρχίσει να εκτελείται, θα καλέσει τη συνάρτηση `start_routine(arg)`
- Γράφει στο `tid` το ID του νέου thread

Τερματισμός – pthread_exit

```
int pthread_exit(void * value_ptr);
```

- Τερματίζει το thread που την καλεί
- Το όρισμα αποθηκεύεται και αντιστοιχεί στο “return value”

Αναμονή Τερματισμού – pthread_join

```
int pthread_join(pthread_t tid, void ** value_ptr);
```

- Σταματά μέχρι το thread `tid` να τερματίσει
- Το δεύτερο όρισμα επιστρέφει (by reference) το αποτέλεσμα του thread που τελείωσε, αν υπάρχει

Τερματισμός από έξω – pthread_cancel

```
int pthread_cancel(pthread_t tid);
```

- Τερματίζει το thread `tid`
- Την καλεί ένα άλλο thread για να τερματίσει το `tid`

Χρονοδρομολόγηση – pthread_yield

```
int pthread_yield(void);
```

- Ζητά από την υλοποίηση των pthreads να παραχωρηθεί ο επεξεργαστής σε άλλο thread
- Οδηγία, όχι εντολή: Μπορεί να μη γίνει

Παράδειγμα

example1.c

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

void *do_something(void *p) {
  printf("Hello from %s thread\n", (char*)p);
  return NULL;
}

int main() {
  pthread_t tid;
  char *msg1 = "parent", *msg2 = "child";

  pthread_create(&tid, NULL, do_something, msg2);
  do_something(msg1);
  return 0;
}
```

Παράδειγμα – Εκτέλεση

```
$ gcc example1.c -pthread
$ ./a.out
Hello from parent thread
Hello from child thread
Hello from child thread
$ ./a.out
Hello from parent thread
Hello from Hello from child thread
$
```

- Πρόβλημα
- Race condition
  - Στο τέλος της `main()` το “parent” thread γράφει το buffer της printf στην έξοδο
  - Το “child” thread γράφει ακόμη στο buffer της printf, θα το γράψει στην έξοδο

Παράδειγμα

example2.c

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

void *do_something(void *p) {
  printf("Hello from %s thread\n", (char*)p);
  return NULL;
}

int main() {
  pthread_t tid;
  char *msg1 = "parent", *msg2 = "child";

  pthread_create(&tid, NULL, do_something, msg2);
  do_something(msg1);
  pthread_join(tid, NULL);
  return 0;
}
```

Παράδειγμα

- Σωστότερο
- Η `pthread_join` εισάγει συγχρονισμό
- Το πρόγραμμα παραμένει μη ντετερμινιστικό
  - Η σειρά των εκτυπώσεων
- Η `printf` σιγουρεύει ότι δεν θα αναμειχθούν
  - Όταν η μεταγλώττιση γίνει με `-pthread`
  - Όχι, όταν γίνει μόνο με `-lpthread`
- «Δεν έχει data race» *δεν σημαίνει* «είναι ντετερμινιστικό»

Κοινή Μνήμη

- Όλα τα threads έχουν πρόσβαση σε όλη τη μνήμη της διεργασίας
  - Μπορούν να έχουν τα δικά τους δεδομένα ανά thread
    - `__thread`
    - Παραμένουν στο global address space (address-of)
- Ο προγραμματιστής είναι υπεύθυνος για το συγχρονισμό
  - Ποιά δεδομένα είναι κοινά
    - Προστασία από ταυτόχρονη πρόσβαση, data races
  - Ποιά δεδομένα δεν χρειάζονται συγχρονισμό (thread-local)
  - Παραμένουν στην κοινή μνήμη (memory corruption bugs!)

Κοινή μνήμη

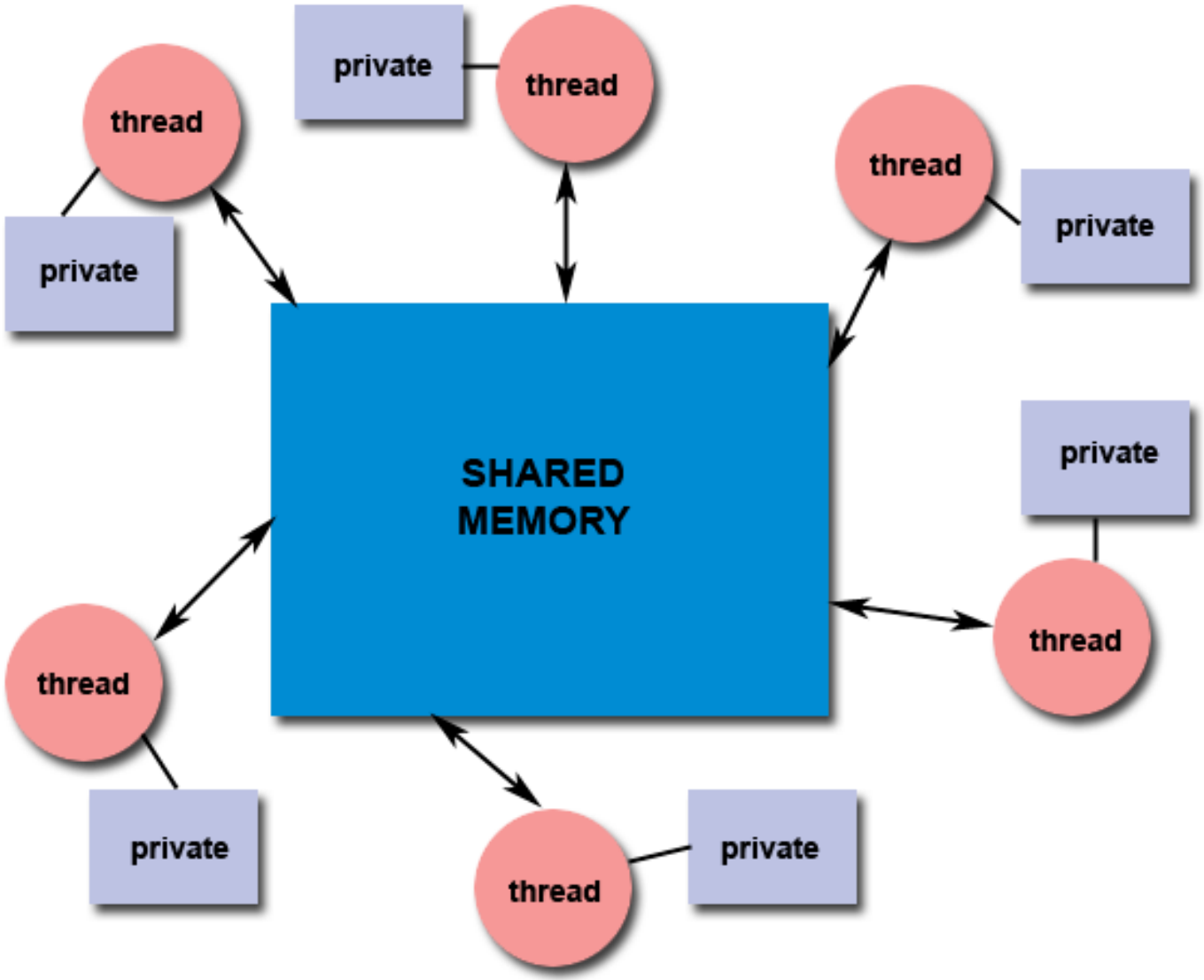


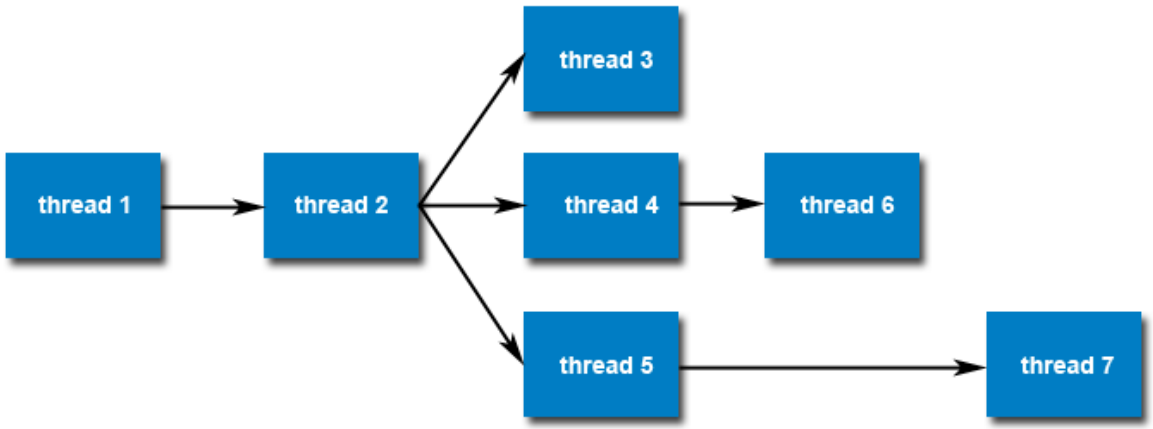
Figure source: https://computing.llnl.gov

“Thread-safe”

- “Thread-safe”: Όρος που περιγράφει την ιδιότητα ενός προγράμματος να μπορεί να εκτελεστεί ταυτόχρονα, σε threads
  - Βιβλιοθήκες που διατηρούν state (printf buffers, file I/O)
  - Δομές δεδομένων: insert, delete, sort, ταυτόχρονα;
  - Συχνά υλοποιείται με locking
- Ο προγραμματιστής είναι υπεύθυνος για το συγχρονισμό
  - Ποιά δεδομένα είναι κοινά
    - Προστασία από ταυτόχρονη πρόσβαση, data races
  - Ποιά δεδομένα δεν χρειάζονται συγχρονισμό (thread-local)
  - Παραμένουν στην κοινή μνήμη (memory corruption bugs!)
- Αν μια βιβλιοθήκη δεν είναι σίγουρα thread-safe, το ασφαλές είναι να μην καλεστεί ποτέ ταυτόχρονα
  - I/O thread, communication thread, κλπ

Εκτέλεση

- Μόλις δημιουργηθεί ξεκινά το thread καλώντας τη συνάρτηση που δίνεται
- Δεν υπάρχει κατασκευαστική διαφορά μεταξύ των threads της διεργασίας, εκτός:
  - "main thread": το context που ξεκινάει το πρόγραμμα
    - Τελειώνοντας τερματίζουν όλα
    - Αποφυγή, με `pthread_exit()`
  - "child thread": threads που ξεκίνησαν με `pthread_create`
  - Δεν υπάρχει δεδομένη «ιεραρχία» για το τι μπορεί να κάνει κάθε thread
  - Μπορεί να υπάρξει «κληρονομικότητα» χαρακτηριστικών και παραμέτρων



Ιδιότητες και χαρακτηριστικά

- Detached / Joinable
  - Ανάγκη ή όχι για `pthread_join`
- Scheduling policy
  - FIFO, RR, OTHER
  - Προτεραιότητα
- Στοίβα
  - Θέση, μέγεθος, guard

Ιδιότητες – pthread_attr_t

```
int pthread_attr_init(pthread_attr_t *);
int pthread_attr_destroy(pthread_attr_t *);

int pthread_attr_getdetachstate(const pthread_attr_t *, int *);
int pthread_attr_setdetachstate(pthread_attr_t *, int);

int pthread_attr_getstack(const pthread_attr_t *, void **, size_t *);
int pthread_attr_setstack(pthread_attr_t *, void *, size_t);

int pthread_attr_getstacksize(const pthread_attr_t *, size_t *);
int pthread_attr_setstacksize(pthread_attr_t *, size_t);

int pthread_attr_getstackaddr(const pthread_attr_t *, void **);
int pthread_attr_setstackaddr(pthread_attr_t *, void *);

int pthread_attr_getguardsize(const pthread_attr_t *, size_t *);
int pthread_attr_setguardsize(pthread_attr_t *, size_t);

int pthread_attr_getinheritsched(const pthread_attr_t *, int *);
int pthread_attr_setinheritsched(pthread_attr_t *, int);

int pthread_attr_getschedparam(const pthread_attr_t *, struct sched_param *);
int pthread_attr_setschedparam(pthread_attr_t *, const struct sched_param *);

int pthread_attr_getschedpolicy(const pthread_attr_t *, int *);
int pthread_attr_setschedpolicy(pthread_attr_t *, int);
```

Data Races

- Ταυτόχρονη πρόσβαση στην ίδια μνήμη, τουλάχιστον ένα write
  - Memory corruption
- Αποφυγή ταυτόχρονης πρόσβασης
  - Χωρίς συγκεκριμένη σειρά, απλά όχι ταυτόχρονα
  - Μη ντετερμινιστικό, συχνά αρκεί
  - Low-level: mutex
  - High-level: atomic
- Παράδειγμα: Ταυτόχρονη κατάθεση

mutex

- Mutual Exclusion — αμοιβαίος αποκλεισμός
  - Lock
  - Μηχανισμός, αντικείμενο 2 καταστάσεων
  - Κλειδωμένο, Ξεκλείδωτο
  - Όποιος το κλειδώσει πρέπει να το ξεκλειδώσει (όχι semaphore)
  - Μόνο ένα thread μπορεί να έχει το κάθε mutex ανά πάσα στιγμή
- Το μόνο data race που επιτρέπεται είναι το race for the lock
  - Κατασκευασμένο έτσι ώστε μόνο ένα thread να επιτύχει να κλειδώσει, ακόμη κι αν πολλά προσπαθήσουν ταυτόχρονα
  - Μπορεί να είναι διαφορετικός μηχανισμός σε κάθε επεξεργαστή!

mutex

- Τυπική χρήση
  - Πολλά thread προσπαθούν να κλειδώσουν ένα mutex
  - Μόνο ένα επιτυγχάνει, τα άλλα σταματούν και περιμένουν (blocking)
  - Το thread που κλείδωσε το mutex συνεχίζει με την κρίσιμη πρόσβαση ή προσβάσεις και ξεκλειδώνει το mutex όταν τελειώσει
  - Ένα από τα blocked threads κλειδώνει το mutex, κ.ο.κ.
- Υποστήριξη ασύγχρονου κλειδώματος
  - trylock

Σωστή Χρήση – Δυσκολίες

- Τα mutexes είναι low-level μηχανισμός
  - Ευθύνη του προγραμματιστή η σωστή χρήση
- Πρέπει κάθε πρόσβαση να προστατεύεται από mutex
  - Αν ένα thread δεν κλειδώσει πριν γράψει υπάρχει data race
- Καθολική ιδιότητα
  - Ένα mutex «προστατεύει» μια διεύθυνση αν την προστατεύει παντού
- Δυσκολίες
  - Οργάνωση του κώδικα
  - Κώδικας σε βιβλιοθήκες
  - Έμμεσες αναφορές σε μνήμη
  - Σύνθεση μεγάλων προγραμμάτων από μικρά
  - κλπ

Δημιουργία - pthread_mutex_init

- Στατικά

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

- Δυναμικά

```
int pthread_mutex_init(pthread_mutex_t *mutex,
  const pthread_mutexattr_t *attr);
```

  - Επιτρέπει attributes: priority inheritance, re-entrant

Χρήση

- Κλείδωμα

```
int pthread_mutex_lock(pthread_mutex_t *);
```

- Non-blocking

```
int pthread_mutex_trylock(pthread_mutex_t *);
```

- Ξελείδωμα

```
int pthread_mutex_unlock(pthread_mutex_t *);
```

  - Πρέπει να κληθεί από το thread που έχει κλειδώσει το lock
  - Αλλιώς επιστρέφεται λάθος

Παράδειγμα: Producer-consumer με mutex

prodcons-mutex.c

```
pthread_mutex_t task_lock;
int task_available;

int main() {
  task_available = 0;
  pthread_mutex_init(&task_lock, NULL);
  /* create and join producer and consumer threads */
}
```

```
void *producer(void *producer_thread_data) {
  int inserted;
  struct task my_task;

  while (!done()) {
    inserted = 0;
    create_task(&my_task);
    while (inserted == 0) {
      pthread_mutex_lock(&task_lock);
      if (task_available == 0) {
        insert_into_queue(my_task);
        task_available = 1;
        inserted = 1;
      }
      pthread_mutex_unlock(&task_lock);
    }
  }
}
```

```
void *consumer(void *consumer_thread_data) {
  int extracted;
  struct task my_task;

  while (!done()) {
    extracted = 0;
    while (extracted == 0) {
      pthread_mutex_lock(&task_lock);
      if (task_available == 1) {
        extract_from_queue(&my_task);
        task_available = 0;
        extracted = 1;
      }
      pthread_mutex_unlock(&task_lock);
    }
    process_task(my_task);
  }
}
```

Συγχρονισμός με condition variables

- Άλλος ένας μηχανισμός συγχρονισμού
  - Συγχρονισμός με συνθήκη και mutex
- Ένα thread μπορεί να περιμένει μέχρι μια συνθήκη να γίνει αληθής
  - Κλειδώνει το mutex
  - Ελέγχει αν μια μεταβλητή ικανοποιεί τη συνθήκη
  - Αν η συνθήκη δεν ικανοποιείται, περιμένει, *αφήνοντας το lock*
  - Μόλις η συνθήκη ικανοποιηθεί από κάποιο άλλο thread, αυτό ξυπνά όσα threads περιμένουν στη συνθήκη
  - Ένα από αυτά θα κλειδώσει και θα συνεχίσει έχοντας το mutex

Condition API

```
int pthread_cond_init(pthread_cond_t *cond,
  const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cond_wait(pthread_cond_t *cond,
  pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
  pthread_mutex_t *mutex, const struct timespec *wtime);

int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Παράδειγμα: Producer-Consumer με Condition

```
pthread_cond_t cond_q_empty, cond_q_full;
pthread_mutex_t task_lock;
int task_available;

int main() {
  task_available = 0;
  pthread_cond_init(&cond_q_empty, NULL);
  pthread_cond_init(&cond_q_full, NULL);
  pthread_mutex_init(&task_lock, NULL);
  /* create and join producer and consumer threads */
}
```

```
void *producer(void *producer_thread_data) {
  while (!done()) {
    create_task();
    pthread_mutex_lock(&task_lock);
    while (task_available == 1)
      pthread_cond_wait(&cond_q_empty, &task_lock);
    insert_into_queue();
    task_available = 1;
    pthread_cond_signal(&cond_q_full);
    pthread_mutex_unlock(&task_lock);
  }
}
```

```
void *consumer(void *consumer_thread_data) {
  while (!done()) {
    pthread_mutex_lock(&task_lock);
    while (task_available == 0)
      pthread_cond_wait(&cond_q_full, &task_lock);
    my_task = extract_from_queue();
    task_available = 0;
    pthread_cond_signal(&cond_q_empty);
    pthread_mutex_unlock(&task_lock);
    process_task(my_task);
  }
}
```

Συγχρονισμός με reader-writer locks

- Διαφοροποιούν την ανάγνωση από την εγγραφή
  - Δεν ορίζεται η προτεραιότητα
- Το ίδιο lock μπορεί να κλειδωθεί ταυτόχρονα από πολλούς readers
  - Οι writers περιμένουν
- Αλλά μόνο ένας writer
  - Οι readers και οι υπόλοιποι writers περιμένουν

RWlock API

```
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
int pthread_rwlock_init(pthread_rwlock_t *,
  const pthread_rwlockattr_t *);

int pthread_rwlock_rdlock(pthread_rwlock_t *);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *);
int pthread_rwlock_timedrdlock(pthread_rwlock_t *rwlock,
  const struct timespec *abstime);

int pthread_rwlock_wrlock(pthread_rwlock_t *);
int pthread_rwlock_trywrlock(pthread_rwlock_t *);
int pthread_rwlock_timedwrlock(pthread_rwlock_t *rwlock,
  const struct timespec *abstime);

int pthread_rwlock_unlock(pthread_rwlock_t *);

int pthread_rwlock_destroy(pthread_rwlock_t *);
```

Παράδειγμα: Reader-Writer Access

```
pthread_rwlock_t data_rw_lock;
int data[1000];
pthread_t readers[NUM_READERS];
pthread_t writer;

int main() {
  /* create and join threads */
  for(int i;  i  < NUM_READERS; i++) {
    pthread_create(&reader[i], NULL, reader_func, NULL);
  }
  pthread_create(&writer, NULL, rare_writer_func, NULL);
  /*  ...  */
}
```

```
void *reader_func(void *reader_thread_data) {
  while (!done()) {
    pthread_rwlock_rdlock(&data_rw_lock);
    compute(data);
    pthread_rwlock_unlock(&data_rw_lock);
  }
}
```

```
void *rare_writer_func(void *writer_thread_data) {
  while (!done()) {
    pthread_rwlock_wrlock(&data_rw_lock);
    update(data);
    pthread_rwlock_unlock(&data_rw_lock);
  }
}
```

Barriers: Συγχρονισμός μεταξύ πολλών threads

- Λειτουργούν σαν “rendez-vous” συγχρονισμός
  - Αναμονή μέχρι να φτάσουν όλοι
  - Ο τελευταίος τους ξυπνά όλους
- Όχι απαραίτητα *όλα* τα threads
  - Συγκεκριμένος αριθμός
  - Ισοδύναμο με μετρητή + lock ή conditional

Barrier API

```
int pthread_barrier_init(pthread_barrier_t *barrier,
  const pthread_barrierattr_t *attr, unsigned count)
int pthread_barrier_destroy(pthread_barrier_t *barrier)
int pthread_barrier_wait(pthread_barrier_t *barrier)
```

Παράδειγμα: Barrier

```
pthread_rwlock_t data_rw_lock;
int data[1000];
pthread_t workers[NUM_THREADS];
pthread_barrier_t barrier;

void *worker_func(void *arg) {
  pthread_barrier_wait(&barrier);
  phase_1(); /* everybody starts together */
  pthread_barrier_wait(&barrier);
  /* everybody waits for all to finish phase 1*/
  phase_2(); /* everybody starts together */
  pthread_barrier_wait(&barrier);
  /* wait for everyone to finish together */
}

int main() {
  /* create and join threads */
  pthread_barrier_init(&barrier, NULL, NUM_THREADS);
  for(int i;  i < NUM_THREADS; i++) {
    pthread_create(&workers[i], NULL, worker_func, NULL);
  }
  /* ...  */
}
```

Πιθανά προβλήματα με threads

- `fork()` και `exec()`
  - Στο καινούριο process μόνο ένα thread
  - Άγνωστη κατάσταση μνήμης, τα locks διατηρούν την κατάσταση
  - `pthread_atfork()`
- Thread-safety
  - Συναρτήσεις που δεν βασίζονται σε global state, δεν έχουν global resources, κλπ
  - Συναρτήσεις βιβλιοθήκης re-entrant (τελειώνουν σε _r)
  - Ασφαλέστερο (slow): “Wrap in a lock”
- Λάθη
  - Η `errno` αλλάζει σε macro (διαφορετική για κάθε thread)
  - Η `strerror()` δεν είναι re-entrant, χρειάζεται συγχρονισμό
- Signals
  - Κάθε thread έχει δική του μάσκα σημάτων
  - Χειρισμός: ένα thread για τα σήματα, αποκλειστικά

Deadlock

```
/* ... */
pthread_mutex_lock(&lockA);
pthread_mutex_lock(&lockB);
work(a, b);
pthread_mutex_unlock(&lockA);
pthread_mutex_unlock(&lockB);
/* ... */
```

```
/* ... */
pthread_mutex_lock(&lockB);
pthread_mutex_lock(&lockA);
work(b, a);
pthread_mutex_unlock(&lockA);
pthread_mutex_unlock(&lockB);
/* ... */
```

- Deadlock: Κάθε thread περιμένει το άλλο
- Οσοδήποτε μεγάλος κύκλος
- Συνήθως διορθώνεται με σταθερή κοινή σειρά
- Όχι πάντα δυνατόν: τότε χρήση *trylock*

Livelock

```
/* ... */
while(retrying) {
  if(pthread_mutex_trylock(&lockA)) {
    if(pthread_mutex_trylock(&lockB) {
      work(a, b);
      pthread_mutex_unlock(&lockB);
      retrying = 0;
    }
    pthread_mutex_unlock(&lockA);
  }
}
/* ... */
```

```
/* ... */
while(retrying) {
  if(pthread_mutex_trylock(&lockB)) {
    if(pthread_mutex_trylock(&lockA) {
      work(a, b);
      pthread_mutex_unlock(&lockA);
      retrying = 0;
    }
    pthread_mutex_unlock(&lockB);
  }
}
/* ... */
```

- Κάθε thread προσπαθεί συνεχώς αλλά κανείς δεν προχωρά
- Dining philosophers
- Δύσκολο debugging

Priority Inversion

- Ένα thread H με υψηλή προτεραιότητα προσπαθεί να κλειδώσει
- Ένα thread L με χαμηλή προτεραιότητα έχει το lock
- Ένα thread M με μεσαία προτεραιότητα μπορεί να σταματήσει το L
  - Καθυστερεί και το H
  - Mars Bug

Πρότυπα Παραλληλισμού

- Συχνές περιπτώσεις παραλληλισμού
- `doall` ή `forall`
  - Embarassingly parallel
- Pipeline
  - Υπολογισμός σε στάδια
- Task (recursive)
  - Αναδρομικά μικρότερο πρόβλημα
  - Divide-and-Conquer
- MapReduce
  - Πρότυπο υπολογισμού με (key, value) ζευγάρια
  - Υπολογισμός ορίζεται με 2 συναρτήσεις:
    - map(key, value): παίρνει ένα ζευγάρι εισόδου και παράγει (emit) πολλά ενδιάμεσα
    - reduce(key, values): παίρνει ένα ενδιάμεσο key και όλα τα values του και επιστρέφει ένα ζευγάρι εξόδου
  - Το πρότυπο κάνει τα υπόλοιπα

doall ή forall

```
for(i = 0; i < 1000000; i++) {
  C[i] = A[i] + B[i];
}
```

forall με pthreads

```
struct thread_arg {
  int from;
  int to;
  pthread_t thread;
};

void *do_work(void *voidarg) {
  struct thread_arg *arg = (struct thread_arg *)voidarg;
  for(i = arg->from; i < arg->to; i++) {
    C[i] = A[i] + B[i];
  }
}

void parallelfor() {
  struct thread_arg[NUM_THREADS];
  int from = 0, to = 1000000;
  int step = to / NUM_THREADS;
  int i;

  for(i = 0; i < NUM_THREADS; i++) {
    thread_arg[i].from = from;
    thread_arg[i].to = (i < NUM_THREADS-1) ? (from + step) : to;
    from = to;
    pthread_create(&thread_arg[i].thread, NULL, &do_work, &thread_arg[i]);
  }
  for(i = 0; i < NUM_THREADS; i++) {
    pthread_join(thread_arg[i].thread, NULL);
}
```

forall με pthreads

- Καινούριο thread για κάθε step
- Κόστος δημιουργίας και join
- Κώδικας βιβλιοθήκης – παράλληλη χρήση παράλληλου κώδικα
- Διαχείριση πόρων

Thread Pool

```
pthread_t threads[NUM_THREADS];
queue_t available_work;

void* worker_thread(void* arg) {
  /* ... */
  while(!done()) {
    task_t work = dequeue(available_work);
    run(task);
  }
}

int main() {
  /* ... */
  for(i = 0; i < NUM_THREADS; i++) {
    pthread_create(&threads[i], NULL, &worker_thread, arg);
  }

  for(i = 0; i < WORK_PIECES; i++) {
    /* create work */
    enqueue(available_work, task);
  }

  for(i = 0; i < NUM_THREADS; i++) {
    pthread_join(&threads[i], NULL);
  }
}
```

Load Balancing

- Καλύτερος καταμερισμός εργασίας
- Περισσότερα κομμάτια
- Overhead: χρόνος dequeue()
- Fine-grain, coarse-grain: ισορροπία
  - Διαφορετικά για κάθε επεξεργαστή!

False sharing

- Διαδοχικές θέσεις μνήμης των πινάκων
- Ίδιο cache line, διαφορετικά threads
- Κάθε εγγραφή στο cache line το κάνει invalidate από τις caches των άλλων επεξεργαστών
- False sharing: Δεν υπάρχει μνήμη που να ανήκει ταυτόχρονα σε 2 επεξεργαστές, αλλά και πάλι ο ένας καθυστερεί τον άλλο
- `memalign`, padding, κλπ

memalign

```
int posix_memalign(void **memptr, size_t alignment, size_t size);
```

- Δεσμεύει `size` bytes
- Αποτέλεσμα στον δείκτη `*memptr` (call by reference)
- Διεύθυνση πολλαπλάσιο του `alignment` (δύναμη του 2)

```
void * memalign(size_t alignment, size_t size);
```

- Παρόμοια χρήση
- Obsolete — Υπάρχει σε linux
- Επιστρέφει το δείκτη (όπως η `malloc`)
- Σε περίπτωση λάθους θέτει `errno` και επιστρέφει `NULL`

Pipeline

- Γενίκευση του “producer-consumer”
- “Γραμμή παραγωγής”
- Κάθε thread εκτελεί μια λειτουργία
- Τα δεδομένα μετακινούνται από thread σε thread κατά την επεξεργασία
- Επικοινωνία με buffers, point-to-point
  - Χαμηλό contention, λιγότερο false sharing
  - Ευκολότερο: barrier (less parallel)
- Load balancing?

Παράδειγμα

pipeline.c

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

queue_t ph1_ph2_buffer;
queue_t ph2_ph3_buffer;

void *phase1(void *a) {
  while(!done()) {
    read_from_file(data);
    process1(data);
    enqueue(ph1_ph2_buffer, data);
  }
}

void *phase2(void *a) {
  while(!done()) {
    data = dequeue(ph1_ph2_buffer);
    process2(data);
    enqueue(ph2_ph3_buffer, data);
  }
}

void *phase3(void *a) {
  while(!done()) {
    data = dequeue(ph2_ph3_buffer);
    save_to_file(data);
  }
}
```

DataFlow: Γενίκευση

- Κάθε φάση μπορεί να έχει thread pool για εσωτερικό παραλληλισμό
- Πάνω από μία φάσεις υπολογισμού ταυτόχρονα (branching)
- Φάσεις με m εισόδους και n εξόδους
  - Χρήση “copying phase”
- Προσοχή: load balancing

Recursively Parallel

- Divide-and-conquer
  - Quicksort https://en.wikipedia.org/wiki/Quicksort
  - Mergesort https://en.wikipedia.org/wiki/Merge_sort
- Δυναμικός προγραμματισμός
  - Caching – Memoizing
  - Longest common subsequence https://en.wikipedia.org/wiki/Longest_common_subsequence_problem
  - Matrix-chain multiplication https://en.wikipedia.org/wiki/Matrix_chain_multiplication
  - Edit distance https://en.wikipedia.org/wiki/Edit_distance
  - Shortest path https://en.wikipedia.org/wiki/Dijkstra's_algorithm

Divide and Conquer

- Μικρότερα παράλληλα κομμάτια υπολογισμών
  - Αναδρομικά μικρότερα
  - Συνένωση στο τέλος
- Εύκολος παραλληλισμός
  - Μικρότερο συμφέρον σειριακό κομμάτι: granularity
  - Overhead
  - Δημιουργία αναδρομικά μικρότερων υπολογισμών
  - Συνένωση αποτελεσμάτων σε μεγαλύτερα

Παράδειγμα: Mergesort

```
void merge(int A[], int length, int middle) {
  int i, j, k;
  int *x = malloc(length * sizeof (int));
  for (i = 0, j = middle, k = 0; k < length; k++) {
    x[k] = j == length ? A[i++]
         : i == middle ? A[j++]
         : A[j] < A[i] ? A[j++]
         :               A[i++];
  }
  for (i = 0; i < length; i++) {
    A[i] = x[i];
  }
  free(x);
}

void merge_sort(int A[], int length) {
  int middle;

  if (length < 2) return;
  middle = length / 2;
  merge_sort(A, middle);
  merge_sort(A+middle, length -middle);
  merge(A, length, middle);
}
```

Παράδειγμα: Parallel Mergesort

```
struct thread_arg {
  int *a;
  int length;
};

void * par_merge_sort(void *voidarg) {
  thread_t child;
  int middle;
  struct thread_arg arg = (struct thread_arg*) voidarg;
  struct thread_arg childarg;

  if (arg->length < 2) return;
  middle = arg->length / 2;

  childarg.a = arg->a;
  childarg.length = middle;

  pthread_create(&child, NULL, &par_merge_sort, &childarg)
  par_merge_sort(arg->a + middle, arg->length -middle);
  pthread_join(&child, NULL);
  merge(arg->a, length, middle);
}
```

Παράδειγμα: Parallel Mergesort

```
#define THRESHOLD 1000

struct thread_arg {
  int *a;
  int length;
};

void * par_merge_sort(void *voidarg) {
  thread_t child;
  struct thread_arg arg = (struct thread_arg*) voidarg;
  struct thread_arg childarg;
  int middle;

  if (arg->length < THRESHOLD) {
    merge_sort(arg->a, arg->length);
    return;
  }
  middle = arg->length / 2;

  childarg.a = arg->a
  childarg.length = middle;

  pthread_create(&child, NULL, &par_merge_sort, &childarg)
  par_merge_sort(arg->a + middle, arg->length -middle);
  pthread_join(&child, NULL);
  merge(A, length, middle);
}
```

Δυναμικός Προγραμματισμός

- Memoization προσωρινών αποτελεσμάτων: “ως τώρα καλύτερο”
  - Εξαρτήσεις επόμενων λύσεων από προηγούμενες
  - Διαφορά από divide-and-conquer:
    - Χρήση “κοινών” προηγούμενων λύσεων από πολλές επόμενες

Παράδειγμα: Levenshtein Distance

```
int edit_distance(char *str1, char *str2) {
  int i, j;
  int result;
  const int length1 = strlen(str1);
  const int length2 = strlen(str2);

  int **dist = alloc_2d_array(length1+1, length2+1, sizeof(int));
  for(i = 0; i < length1; i++) dist[i][0] = i;
  for(j = 0; j < length2; j++) dist[0][j] = i;
  for(i = 1; i <= length1; i++) {
    for(j = 1; j <= length2; j++) {
      dist[i][j] = (s1[i - 1] == s2[j -1]) ? dist[i-1][j-1]
                 : 1 + min3(dist[i-1][j], dist[i][j-1], dist[i-1][j-1]);
    }
  }
  result = dist[length1][length2];
  free_2d_array(dist);
  return result;
}
```

MapReduce

- Βασίζεται σε συναρτησιακό προγραμματισμό
  - Lisp 1960
  - Κάθε δεδομένο γράφεται μια φορά όταν δημιουργείται
  - Immutable data structures
  - Κάθε υπολογισμός επιστρέφει νέα δεδομένα
  - Δε χρειάζεται συγχρονισμός σε writes επειδή όλα τα writes είναι thread-local
- Revival: Google Big Data
- Αν κάτι μπορεί να εκφραστεί, πολύ εύκολος παραλληλισμός
- Pattern: Αναλαμβάνει load balancing, sorting, reduction