Ε-85: Ειδικά Θέματα Λογισμικού
Προγραμματισμός Συστημάτων Υψηλών Επιδόσεων

Χειμερινό Εξάμηνο 2009-10

«Ειδικά Θέματα Προγραμματισμού Νημάτων»

Παναγιώτης Χατζηδούκας
(Π.Δ. 407/80)

Ειδικά Θέματα

- Αντικείμενα ιδιοτήτων (attributes objects)
  - Ιδιότητες νημάτων
  - Ιδιότητες των mutexes
  - Ιδιότητες μεταβλητών συνθήκης
- Αρχικοποίηση μιας φοράς (one-time initialization)
- Ιδιωτικά δεδομένα νήματος
- Ακύρωση νημάτων (cancellation)
- Χειριστές καθαρισμού
- Σήματα
- Σημαφόροι

Αντικείμενα Ιδιοτήτων

- Δημιουργία νημάτων και δυναμική αρχικοποίηση mutexes και μεταβλητών συνθήκης με το δεύτερο όρισμα στην αντίστοιχη ρουτίνα ίσο με NULL
- Το όρισμα είναι δείκτης σε ένα αντικείμενο ιδιοτήτων
- Τιμή NULL σημαίνει την εξ' ορισμού τιμή των ιδιοτήτων (όπως στην στατική αρχικοποίηση)
- Ένα αντικείμενο ιδιοτήτων μπορεί να θεωρηθεί ως μια ιδιωτική δομή δεδομένων.
- Η προσπέλαση των πεδίων της (για ανάγνωση και εγγραφή) δεν πραγματοποιείται άμεσα αλλά πάντα μέσω κατάλληλων ρουτινών
  - **`pthread_attr_getstacksize, pthread_attr_setstacksize`**

Ιδιότητες Νημάτων

```
pthread_attr_t attr;
int pthread_attr_init(&attr);
int pthread_attr_destroy(&attr);
```

- Δήλωση, αρχικοποίηση και καταστροφή ενός αντικειμένου ιδιοτήτων νημάτων

```
int pthread_attr_getdetachstate(&attr, (int *) &ds);
int pthread_attr_setdetachstate(&attr, (int) ds);
```

- Καθορισμός αν ένα νήμα είναι αποδεσμευμένο ή όχι
  - **PTHREAD_CREATE_JOINABLE**
  - **PTHREAD_CREATE_DETACHED**

Ιδιότητες Νημάτων

```
int pthread_attr_getstacksize(&attr, (size_t *) &ss);
int pthread_attr_setstacksize(&attr, (size_t) ss);
```

- Καθορισμός του μεγέθους της στοίβας ενός νήματος
- Ελάχιστο μέγεθος στοίβας: **PTHREAD_STACK_MIN**

```
int pthread_attr_getstackaddr(&attr, (void **) &sa);
int pthread_attr_setstacksize(&attr, (void *) sa);
```

- Καθορισμός της στοίβας ενός νήματος
- Η μνήμη για τη στοίβα δεσμεύεται ρητά (π.χ. malloc)
- Πρέπει να είναι τουλάχιστον **PTHREAD_STACK_MIN**
- Δεν παρέχει πλήρη μεταφερσιμότητα

Ιδιότητες Νημάτων

```
int pthread_attr_getscope(&attr, (int *) &cs);
int pthread_attr_setscope(&attr, (int_t) cs);
```

- Καθορισμός του χώρου όπου δρομολογείται ένα νήμα (διεργασία ή λειτουργικό σύστημα)
- Δυνατές επιλογές: **PTHREAD_SCOPE_PROCESS, PTHREAD_SCOPE_SYSTEM**

```
int pthread_attr_getschedpolicy(&attr, (int *) &sp);
int pthread_attr_setschedpolicy(&attr, (int) sp);
```

- Καθορισμός της πολιτικής δρομολόγησης με την οποία θα εκτελεστούν τα νήματα
- Δυνατές επιλογές: **SCHED_FIFO,SCHED_RR,SCHED_OTHER**
- Η προκαθορισμένη πολιτική εξαρτάται από την υλοποίηση

Ιδιότητες Mutexes

```
pthread_mutexattr_t attr;
int pthread_mutexattr_init(&attr);
int pthread_mutexattr_destroy(&attr);
```

- Δήλωση, αρχικοποίηση και καταστροφή ενός αντικειμένου ιδιοτήτων mutexes

```
#ifdef _POSIX_THREAD_PROCESS_SHARED
   int pthread_mutexattr_getpshared(&attr, (int *) &ps);
   int pthread_mutexattr_setpshared(&attr, (int) ps);
#endif
```

- Διαμοίραση ή μη ενός mutex μεταξύ διαφορετικών διεργασίων
- **ps: PTHREAD_PROCESS_{PRIVATE, SHARED}**
- Το mutex πρέπει να έχει δεσμευτεί σε περιοχή μνήμης που είναι κοινή μεταξύ των διεργασιών

Ιδιότητες Mutexes

```
int pthread_mutexattr_gettype(&attr, (int *) &type);
int pthread_mutexattr_settype(&attr, (int ) type);
```

- Καθορισμός – ορισμός του τύπου ενός mutex που δημιουργείται με το αντικείμενο ιδιοτήτων attr
- Δυνατές επιλογές
  - **PTHREAD_MUTEX_NORMAL**: ταχύτερη υλοποίηση χωρίς έλεγχο σφαλμάτων
  - **PTHREAD_MUTEX_RECURSIVE**: ένα mutex μπορεί να κλειδωθεί πολλές φορές από το ίδιο νήμα
  - **PTHREAD_MUTEX_ERRORCHECK**: για debugging κατά την ανάπτυξη εφαρμογών
  - **PTHREAD_MUTEX_DEFAULT**: αντιστοιχίζεται από την εκάστοτε πλατφόρμα σε κάποια από τις παραπάνω επιλογές (συνήθως στην πρώτη)

Ιδιότητες Μεταβλητών Συνθήκης

```
pthread_condattr_t attr;
int pthread_condattr_init(&attr);
int pthread_condattr_destroy(&attr);
```

- Δήλωση, αρχικοποίηση και καταστροφή ενός αντικειμένου ιδιοτήτων mutexes

```
#ifdef _POSIX_THREAD_PROCESS_SHARED
   int pthread_condattr_getpshared(&attr, (int *) &ps);
   int pthread_condattr_setpshared(&attr, (int) ps);
#endif
```

- Διαμοίραση ή μη μιας μεταβλητής συνθήκης μεταξύ διαφορετικών διεργασίων
- **ps: PTHREAD_PROCESS_{PRIVATE, SHARED}**
- Η μεταβλητή συνθήκης πρέπει να έχει δεσμευτεί σε περιοχή μνήμης που είναι κοινή μεταξύ των διεργασιών

Αρχικοποίηση μιας Φοράς

- Αρχικοποίηση δεδομένων (π.χ. mutexes) από την main
- Στην ανάπτυξη βιβλιοθηκών δεν υπάρχει αυτή η πολυτέλεια
- Σε εφαρμογές με ένα νήμα μια κατάλληλη boolean μεταβλητή αρκεί για να δηλώσει ότι έγινε η αρχικοποίηση
- Ένα στατικά αρχικοποιημένο mutex αρκεί μα αν απαιτείται δυναμική αρχικοποίηση τότε χρησιμοποιείται η pthread_once

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;
int pthread_once(pthread_once_t *once_control,
                 void (*init_routine)(void));
```

- Οποιοδήποτε νήμα μπορεί να καλέσει την συνάρτηση
- Ελέγχεται αυτόματα η αντίστοιχη μεταβλητή και καλείται η ρουτίνα αρχικοποίησης
- Αν ένα άλλο νήμα καλέσει την συνάρτηση, θα μπλοκάρει μέχρι να ολοκληρωθεί η αρχικοποίηση

Αρχικοποίηση μιας Φοράς

```
pthread_once_t once_block = PTHREAD_ONCE_INIT;
pthread_mutex_t mutex;

/* Initialization routine */
void once_init_routine(void) {
    pthread_mutex_init(&mutex, NULL);
}

/* Thread start routine that calls pthread_once */
void *thread_routine(void *arg) {
    pthread_once(&once_block, once_init_routine);
    ...
}

int main() {
    pthread_create(&t, NULL, thread_routine, NULL);
    pthread_once(&once_block, once_init_routine);
    ...
}
```

Ιδιωτικά Δεδομένα Νημάτων

- Διατήρηση της τιμής μιας μεταβλητής μεταξύ κλήσεων διαφορετικών ρουτινών σε ένα συγκεκριμένο νήμα
- Μεταβλητές που ορίζονται ως static σε μια ρουτίνα / αρχείο μπορούν να προσπελαστούν από όλα τα νήματα
- Μόνο το περιεχόμενο των καταχωρητών είναι πραγματικό ιδιωτικό σε ένα νήμα. Ακόμα και η στοίβα που θα μπορούσε να προσπελαστεί από ένα άλλο νήμα

```
pthread_key_t key;
int pthread_key_create(pthread_key_t *key, void
(*destructor)(void *));
int pthread_key_delete(pthread_key_t key);
```

- Δημιουργία και διαγραφή κλειδιού για αποθήκευση ιδιωτικών δεδομένων

```
int pthread_setspecific(pthread_key_t key, const void
*value);
void *pthread_getspecific(pthread_key_t key);
```

- Ανάθεση και ανάκτηση τιμής κλειδιού

Ιδιωτικά Δεδομένα Νημάτων

```
pthread_key_t key;
pthread_once_t key_once = PTHREAD_ONCE_INIT;

void once_routine(void) {
   pthread_key_create(&key, NULL);
}

void routine() {
   long *value;
   value = pthread_setspecific(key);
}

void *thread_routine(void *) {
   long *value;
   pthread_once(&key_once, once_routine);

   value = malloc(sizeof(long));
   *value = (long)pthread_self();
   pthread_setspecific(key, value);
   routine();
}
```

Χειρισμός σφαλμάτων

- Στις παραδοσιακές διεργασίες η errno είναι μια καθολική μεταβλητή ακέραιου (int) τύπου
- Μια τέτοια μεταβλητή θα ήταν άχρηστη σε ένα πολυνηματικό πρόγραμμα
- Στην πραγματικότητα, η errno είναι ένα macro που ορίζεται στο αρχείο <errno.h> και επιστρέφει πληροφορία ανά νήμα
- Κάθε νήμα έχει ένα ιδιωτικό αντίγραφο της errno

```
#if (defined(_REENTRANT)
extern int *___errno();
#define errno (*(___errno()))
#else
extern int errno;
#endif
```

Ακύρωση Νημάτων

```
int pthread_cancel(pthread_t thread);
```

- Ακυρώνει την εκτέλεση ενός νήματος

```
int pthread_setcancelstate(int state, int *ostate);
```

- Ένα νήμα επιτρέπει την ακύρωση του ή όχι
  - **PTHREAD_CANCEL_ENABLE, PTHREAD_CANCEL_DISABLE**

```
int pthread_setcanceltype(int type, int *otype);
```

- Καθορισμός του τύπου ακύρωσης
  - Με αναβολή (deferred): **PTHREAD_CANCEL_DEFERRED**
  - Ασύγχρονος (asynchronous): **PTHREAD_CANCEL_ASYNCHRONOUS**

```
int pthread_testcanceltype();
```

- Ένα νήμα ελέγχει αν έχει πάρει εντολή για να ακυρωθεί

Περιπτώσεις Ακύρωσης Νήματος

- Πρώτη περίπτωση
  - Κατάσταση: απενεργοποιημένη
  - Είδος ακύρωσης: Και τα δύο (με αναβολή ή ασύγχρονη)
  - Η ακύρωση παραμένει σε εκκρεμότητα μέχρι να ενεργοποιηθεί
- Δεύτερη περίπτωση
  - Κατάσταση: ενεργοποιημένη
  - Είδος ακύρωσης: Με αναβολή
  - Η ακύρωση θα πραγματοποιηθεί όταν το νήμα φτάσει σε ένα σημείο ακύρωσης (π.χ. pthread_testcancel)
- Τρίτη περίπτωση
  - Κατάσταση: ενεργοποιημένη
  - Είδος ακύρωσης: Ασύγχρονη
  - Η ακύρωση θα πραγματοποιηθεί οποιαδήποτε χρονική στιγμή

Ακύρωση Νημάτων

```
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;

void *test_wait(void *arg){
   pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
   pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);

   pthread_mutex_lock(&mut);  /* already locked */
   return NULL;
}

int main() {
   ...
   pthread_mutex_lock(&mut);
   pthread_create(&pth, NULL, test_wait, NULL);
   sleep(5);
   pthread_cancel(pth);       /* cancel blocked thread */
   pthread_mutex_unlock(&mut);
   pthread_join(pth, NULL);   /* join cancelled thread */
}
```

Χειριστές καθαρισμού

```
#include <pthread.h>
void pthread_cleanup_push(void (*routine) (void *),
                          void *arg);
void pthread_cleanup_pop(int execute);
```

- Οι χειριστές καθαρισμού είναι ρουτίνες που καλούνται όταν ένα νήμα τερματίζει, είτε καλώντας `pthread_exit` είτε λόγω ακύρωσης.
- Εγκαθίστανται και αφαιρούνται ακολουθώντας μια μορφή στοίβας
- Σκοπός τους είναι η απελευθέρωση πόρων που ένα νήμα μπορεί να κρατά ενώ τερματίζει
- Παράδειγμα: Ένα νήμα τερματίζει ή ακυρώνεται ενώ κρατά κλειδωμένο ένα mutex. Η καλύτερη λύση είναι η χρήση ενός χειριστή καθαρισμού που θα ξεκλειδώνει το mutex
- Παρόμοιες χρήσεις: απελευθέρωση δυναμικής μνήμης (malloc/free), κλείσιμο περιγραφέων αρχείων

Χειριστές καθαρισμού

```
#include <pthread.h>
void pthread_cleanup_push(void (*routine) (void *),
                          void *arg);
void pthread_cleanup_pop(int execute);
```

- Η pthread_cleanup_push εγκαθιστά έναν χειριστή καθαρισμού. Από το σημείο αυτό μέχρι την αντίστοιχη pthread_cleanup_pop, η ρουτίνα καθαρισμού routine θα κληθεί με όρισμα arg όταν το νήμα τερματίσει ή ακυρωθεί
- Αν αρκετοί χειριστές καθαρισμού είναι ενεργοί, θα κληθούν όλοι με σειρά LIFO: ο πιο πρόσφατα εγκαταστημένος χειριστής θα κληθεί πρώτος
- Η pthread_cleanup_pop αφαιρεί τον πιο πρόσφατα εγκαταστημένο χειριστή
  - Αν το όρισμα execute δεν είναι 0, εκτελεί επίσης τον χειριστή
  - Αν το όρισμα execute είναι 0, ο χειριστής αφαιρείται μα δεν εκτελείται
- Τα ζευγάρια pthread_cleanup_push και pthread_cleanup_pop πρέπει να εμφανίζονται μέσα στην ίδια ρουτίνα, στο ίδιο επίπεδο κώδικα.
- Στην πραγματικότητα είναι macros που εισάγουν ένα ζευγάρι αγκυλών στο κώδικα του χρήστη, δηλαδή { και } αντίστοιχα

Παράδειγμα

```
pthread_cleanup_push(pthread_mutex_unlock,
                     (void *)&mut);
pthread_mutex_lock(&mut); /* do some work */
pthread_mutex_unlock(&mut);
pthread_cleanup_pop(0);
```

- Οι τελευταίες δύο γραμμές μπορούν να αντικατασταθούν με:
  `pthread_cleanup_pop(1);`

- Ο παραπάνω κώδικας δεν είναι ασφαλής στην περίπτωση ασύγχρονης ακύρωσης: το νήμα μπορεί να προσπαθήσει να ξεκλειδώσει ένα mutex που δεν έχει κλειδώσει

Παράδειγμα

- Ο παρακάτω κώδικας επιλύει το προηγούμενο πρόβλημα της ασύγχρονης ακύρωσης

```
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED,
&oldtype);
pthread_cleanup_push(pthread_mutex_unlock, (void *)
&mut);
pthread_mutex_lock(&mut); /* do some work */
pthread_cleanup_pop(1);
pthread_setcanceltype(oldtype, NULL);
```

Παράδειγμα

```
void cleaner (void *data) {
   printf ("Clean %d\n", *(int *)data);
}

void *thread (void *param) {
   int data1 = 1, data2 = 2;

   pthread_cleanup_push (cleaner, &data1);
   pthread_cleanup_push (cleaner, &data2);
   pthread_cleanup_pop (1);

   pthread_exit (param);

   pthread_cleanup_pop (0);/* not reached */

   return NULL;
}
```

Σήματα

```
#include <pthread.h>
#include <signal.h>
int pthread kill(pthread t thread, int signo);
```

- Επιστρέφει 0 για επιτυχία, μη μηδενική τιμή σε περίπτωση σφάλματος
- Τιμές σφαλμάτων
  - **ESRCH**: Δεν αντιστοιχεί νήμα στο συγκεκριμένο αναγνωριστικό
  - **EINVAL**: μη αποδεκτός αριθμός σήματος

Αποστολή σήματος σε νήμα

- Τα νήματα δεν χρησιμοποιούνται για τερματισμό νημάτων
- Η **`pthread_kill`** χρησιμοποιείται όπως η **`kill`**
  - Το σήμα αποστέλλεται σε ένα συγκεκριμένο νήμα μα ο χειρισμός του μπορεί να επηρεάσει όλη τη διεργασία
  - Το **`SIGKILL`** θα τερματίσει όλη τη διεργασία
  - Το **`SIGSTOP`** θα μπλοκάρει όλη τη διεργασία (νήματα)
- Ένα νήμα μπορεί να στείλει ένα σήμα στον εαυτό του (παρόμοια με την κλήση **`raise`**):

```
pthread_kill(pthread_self(), signo);
```

Χειρισμός σημάτων και νήματα

- Όλα τα νήματα μοιράζονται τους χειριστές σημάτων της διεργασίας
- Κάθε νήμα έχει την δική του μάσκα σημάτων που αυτό μπλοκάρει
- Υπάρχουν τρεις τύποι σημάτων με διαφορετικές μεθόδους παράδοσης
  - **Ασύγχρονα:** Σήματα που δεν παραδίδονται κάποια καθορισμένη χρονική στιγμή ούτε σχετίζονται με κάποιο συγκεκριμένο νήμα (π.χ. SIGINT). Παραδίδονται σε κάποιο νήμα που δεν έχει μπλοκάρει το σήμα
  - **Σύγχρονα**: Σήματα που παράγονται στο ίδιο σημείο κατά την εκτέλεση ενός νήματος (π.χ. SIGFPE, για αριθμητικό σφάλμα όπως διαίρεση με το μηδέν). Παραδίδονται στο νήμα που προκάλεσε το σήμα.
  - **Κατευθυνόμενα**: Σήματα που αποστέλλονται σε ένα συγκεκριμένο νήμα με την κλήση pthread_kill. Παραδίδονται στο νήμα που καθορίζεται.

Χειρισμός σημάτων και νήματα

- Όλα τα νήματα υιοθετούν τη μάσκα σημάτων (ποια σήματα είναι μπλοκαρισμένα) από το νήμα που τα δημιουργεί
- Η μάσκα σημάτων μπορεί να τεθεί με την κλήση pthread_sigmask
- Οι ενέργειες σημάτων έχουν εμβέλεια διεργασίας. Δεν μπορεί να υπάρχει για το ίδιο σήμα ένα χειριστής σε κάποιο νήμα και ένας διαφορετικός χειριστής σε κάποιο άλλο νήμα
- Ένα σήμα παραδίδεται σε ένα νήμα. Αν ένα δεύτερο σήμα φτάσει ενώ το πρώτο σήμα βρίσκεται υπό επεξεργασία, το σήμα παραδίδεται σε κάποιο άλλο νήμα, εφόσον υπάρχει κάποιο που δεν το έχει μπλοκαρισμένο (ισχύει για ασύγχρονα σήματα μόνο)
- Πρακτικά, η μάσκα σημάτων και οι χειριστές σημάτων τίθενται στο κύριο νήμα, μια φορά

Χειρισμός σημάτων και νήματα

- Τα σήματα πάντα παραδίδονται αρκεί να υπάρχουν νήματα που δεν τα μπλοκάρουν, ακόμα κι αν ένα άλλο νήμα εκτελεί έναν χειριστή σήματος
- Συνηθισμένη τακτική είναι ένα συγκεκριμένο νήμα να χειρίζεται όλα τα ασύγχρονα σήματα
  - Το κύριο νήμα μπλοκάρει όλα τα εισερχόμενα σήμα. Αφού τα νήμα που δημιουργούνται υιοθετούν τη μάσκα σημάτων, όλα τα νήματα θα έχουν τα σήματα μπλοκαρισμένα
  - Δημιουργείται ένα ξεχωριστό νήμα για χειρισμό σημάτων, το οποίο καλεί την sigwait. Τα σήματα για τα οποία έχουν οριστεί χειριστές ξεμπλοκάρονται για το συγκεκριμένο νήμα.
  - Το νήμα καλεί την sigwait μέσα σε ένα βρόχο, αφού η συνάρτηση επιστρέφει όταν ο χειριστής σήματος καλείται

Χειρισμός σημάτων και νήματα

- Μια συνάρτηση των νημάτων POSIX δεν πρέπει να καλείται μέσα από έναν χειριστή σήματος
- Μια συνάρτηση είναι *async-signal safe* εφόσον μπορεί να διακοπεί με ασφάλεια από ένα σήμα και να ξεκινήσει αφού ο χειριστής σήματος επιστρέψει, ακόμα κι αν ο χειριστής σήματος καλέσει την ίδια την ρουτίνα
- Συναρτήσεις που χρησιμοποιούν εσωτερικά στατικές δομές δεν είναι async-safe εκτός κι αν μπλοκάρουν τα σήματα ώστε να μη διακοπεί η εκτέλεσή τους
- Καμία συνάρτηση των νημάτων POSIX δεν είναι ασφαλής. Το μπλοκάρισμα των σημάτων έχει υψηλό κόστος αφού πρόκειται για κλήση συστήματος
- Οι συναρτήσεις των νημάτων POSIX έχουν σχεδιαστεί για ταχύτητα παρά για ασφάλεια

Σημαφόροι

- Πρέπει να αρχικοποιούνται δυναμικά

```
sem_t sem;
sem_init( &sem, 0, 5 );
sem_destroy( &sem );
```

- Βασικές λειτουργίες

```
sem_post( &sem ); // +
sem_wait( &sem ); // - (block if <= 0)
sem_trywait( &sem );
sem_getvalue( &sem );
```

- Μπορούν να χρησιμοποιηθούν με ασφάλεια μέσα από χειριστές σημάτων
- Η κλήση sem_wait() μπορεί να διακοπεί:

```
do { err = sem_wait(&sem); } while ( err=EINTR );
```

Σημαφόροι

```
sem_t mutex;
sem_init(&mutex, 0, 1);
```

```
THREAD 1
sem_wait (&mutex);
a = data;
a++;
data = a;
sem_post (&mutex);
```

```
THREAD 2
sem_wait (&mutex);
/* blocked */
/* blocked */
/* blocked */
/* blocked */
b = data;
b--;
data = b;
sem_post (&mutex);
```