Εργασία 1: Νήματα με υποστήριξη autoscheduling

Προθεσμία παράδοσης 1ης φάσης: TBA

Υλοποιήστε υποστήριξη για την ταυτόχρονη εκτέλεση κώδικα με νήματα όπου η εναλλαγή γίνεται ρητά από το κάθε νήμα (non-preemptive scheduling), μετά από μη επιτυχή προσπάθεια απόκτησης σηματοφόρου, εθελοντική παραχώρηση του επεξεργαστή, ή μετά τον τερματισμό του νήματος που τρέχει. Η υλοποίηση σας πρέπει να παρέχει κατάλληλη διεπαφή προγραμματισμού:

int thread_lib_init(int native_threads);	Αρχικοποίηση περιβάλλοντος εκτέλεσης
thread_t *thread_self();	Επιστρέφει το αναγνωριστικό του νήματος
int thread_getid();	Επιστρέφει το ID (0..N) του νήματος

int thread_create(
  thread_t *thread_descriptor,
  void (body)(void *),
  void *arg,
  int deps,
  thread_t *successors
);
	Αρχικοποίηση νέου νήματος (thread) με εξαρτήσεις, για την εκτέλεση της συνάρτησης body με παράμετρο arg. Στην παράμετρο thread_descriptor επιστρέφεται το handle του νέου thread.
	Τιμή επιστροφής συνάρτησης: 0 σε περίπτωση επιτυχούς δημιουργίας, -1 διαφορετικά.
	
	successors: NULL terminated array

int thread_yield();	Εθελοντική εναλλαγή / παραχώρηση του επεξεργαστή. Εάν το νήμα έχει >0 εξαρτήσεις θα μπλοκάρει, διαφορετικά θα ξαναμπεί στη ready queue.
int thread_inc_dependency(int num_deps);	Αύξηση του αριθμού εξαρτήσεων του νήματος που κάνει την κλήση
void thread_exit();	Τερματισμός νήματος που κάνει την κλήση
int thread_lib_exit();	Καταστροφή περιβάλλοντος εκτέλεσης

Η συνάρτηση δημιουργίας ενός νήματος (thread_create) ορίζει τον αριθμό εξαρτήσεων του νέου νήματος από προηγούμενα (deps), καθώς και τα νήματα τα οποία εξαρτώνται από αυτό (successors).

Επίσης, υλοποιήστε γενικούς σηματοφόρους, ελέγχοντας κατάλληλα την εναλλαγή έτσι ώστε να αποφεύγονται συνθήκες ανταγωνισμού κατά την εκτέλεση των λειτουργιών down/up.

int sem_init(sem_t *s, int val);	Αρχικοποίηση σηματοφόρου.
int sem_down(sem_t *s);	Μείωση σηματοφόρου.
int sem_up(sem_t *s);	Αύξηση σηματοφόρου.
int sem_destroy(sem_t *s);	Καταστροφή σηματοφόρου.

Σημειώσεις

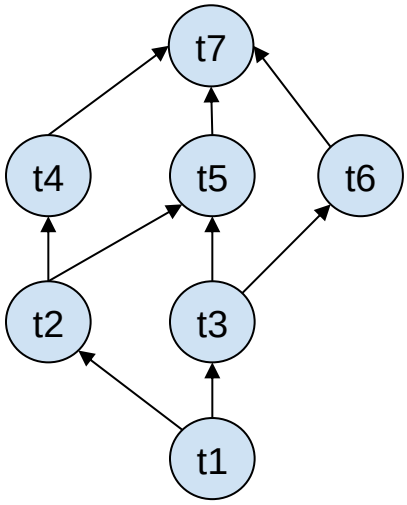
Θα σας είναι χρήσιμες οι λειτουργίες getcontext(), makecontext(), setcontext(), swapcontext() που παρέχει το λειτουργικό για την δημιουργία και την εναλλαγή ανάμεσα σε ξεχωριστά νήματα εκτέλεσης. Στο man page της makecontext υπάρχει και ένα καλό παράδειγμα χρήσης των συναρτήσεων.
Κάθε νήμα αναμένουμε ότι θα αντιπροσωπεύεται από έναν περιγραφέα (descriptor) με τα εξής πεδία:

```
struct thread_descriptor {
        struct thr_descriptor *next, *prev;
        int id;
        char *stack;
        ucontext_t context;        /* see man getcontext */
        int deps;
        int num_successors;
        struct thr_descriptor *successors[];
}
```

Μπορείτε να προσθέσετε ό,τι έξτρα πεδία πιθανώς χρειάζεστε, ή να αλλάξετε αυτά τα πεδία. Συνίσταται τα 2 πρώτα πεδία του struct να είναι δείκτες προς το επόμενο / προηγούμενο.

Τα νήματα που είναι έτοιμα προς εκτέλεση (επιλυμένες εξαρτήσεις), τοποθετούνται σε μια ουρά (ready_queue). Κατά την εναλλαγή νημάτων ο αλγόριθμος εναλλαγής νημάτων επιλέγει ένα νήμα από τα έτοιμα της ουράς. Ένα νήμα είναι έτοιμο προς εκτέλεση όταν όλα τα νήματα από τα οποία εξαρτάται (depends) έχουν τερματίσει. Με τη σειρά του, κάθε νήμα που τερματίζει, μειώνει κατά ένα τον αριθμό εξαρτήσεων (deps) καθενός από τους successors του. Όταν ο αριθμός αυτός φτάσει στο μηδέν τότε το νήμα εισέρχεται στην ουρά (ready_queue).

Κατά την αρχικοποίηση του νέου νήματος, η υλοποίηση δεσμεύει κατάλληλα μια περιοχή μνήμης για να χρησιμοποιηθεί ως στοίβα (stack) από το νέο νήμα. Η μνήμη αυτή πρέπει να είναι ευθυγραμμισμένη (aligned) σε πολλαπλάσια του μεγέθους της stack, η οποία πρέπει να είναι πολλαπλάσιο σελίδας και δύναμη του 2. Το μέγεθος της σελίδας μνήμης του συστήματος μπορεί να βρεθεί είτε στατικά είτε δυναμικά για οποιοδήποτε σύστημα. Θα προτείναμε τα πρώτα 16 bytes της στοίβας περιέχουν ένα δείκτη (pointer) στη δομή struct thread_descriptor που περιγράφει το νήμα μαζί με όσο padding χρειάζεται. Επομένως, η χρήσιμη περιοχή της στοίβας ξεκινάει από τη διεύθυνση stack + 16. Σημειώστε ότι σε συστήματα x86 η stack μεγαλώνει ανάποδα, δηλαδή από μεγαλύτερες προς μικρότερες διευθύνσεις. Συνεπώς σκεφτείτε αν ο δείκτης προς τον thread descriptor θα πρέπει να τοποθετηθεί στην αρχή ή στο τέλος της stack. Η δομή thread_descriptor δεσμεύεται ξεχωριστά και το μέγεθός της εξαρτάται από τον αριθμό των εξαρτήσεων.



Επειδή κάθε νήμα για να δημιουργηθεί πρέπει να έχει δείκτη (pointer) προς τα successor νήματα, αυτό συνεπάγεται πως πρέπει να δημιουργηθεί μετά από αυτά. Οπότε, στη γενική περίπτωση ενός γράφου εξαρτήσεων μεταξύ νημάτων, πρώτα δημιουργούνται τα νήματα που θα τρέξουν τελευταία και δεν έχουν κανέναν successor. Επαναληπτικά και κατά συνέπεια τα νήματα που θα τρέξουν πρώτα και δεν έχουν καμία εξάρτηση (dependency) θα δημιουργηθούν τελευταία. Στο διπλανό σχήμα το βέλος δηλώνει εξάρτηση (t3 depends on t1).

Δεδομένου του συγκεκριμένου γράφου εξαρτήσεων η σειρά δημιουργίας των νημάτων είναι: t7, {t4, t5, t6}, {t2, t3}, t1.

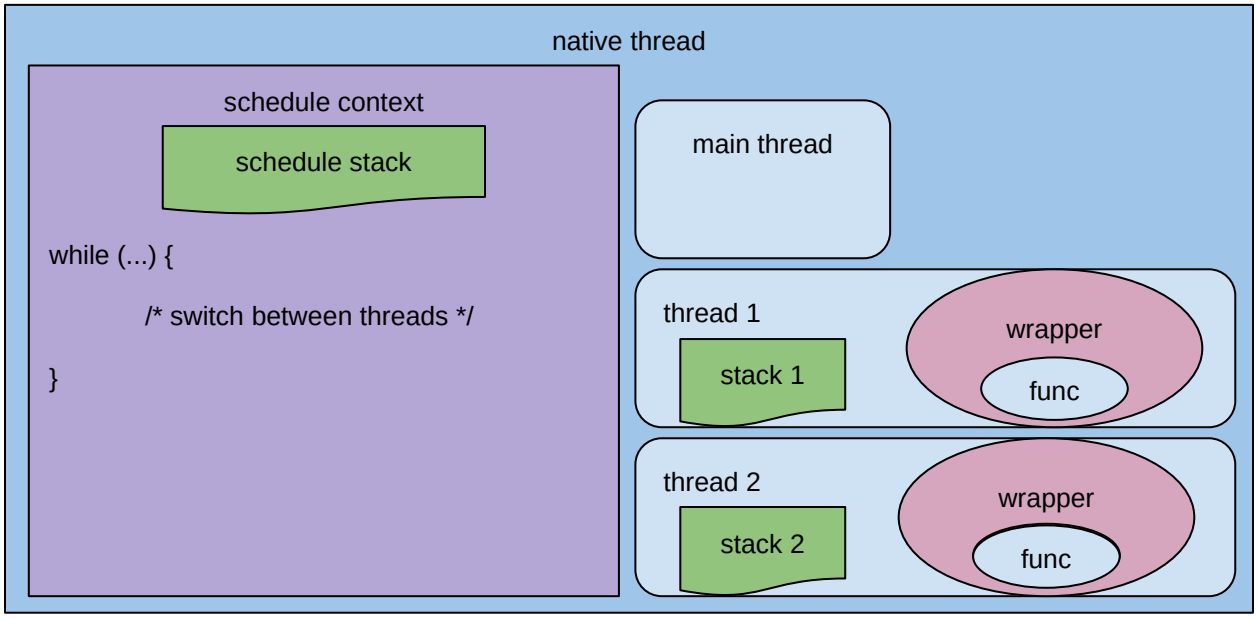
Μετά τη δημιουργία των νημάτων η εκτέλεση ξεκινάει με την πρώτη κλήση thread_yield() από το κυρίως νήμα (main thread).

Η περίπτωση του κυρίως νήματος (main thread) χρειάζεται ειδική μεταχείριση από τα επιπλέον νήματα που φτιάχνονται. Ο λόγος είναι η διεύθυνση της στοίβας του κυρίως νήματος η οποία δεσμεύεται από το λειτουργικό σύστημα, ενώ τα επιπλέον νήματα έχουν δυναμικά δεσμευμένη στοίβα. Ο κώδικας της thread_self() πρέπει να μπορεί να εντοπίζει εάν καλείται από το κυρίως νήμα ή από διαφορετικό και να επιστρέφει την κατάλληλη διεύθυνση του περιγραφέα νημάτων (thread_descriptor).

Εισάγετε ένα νέο context το οποίο διαχειρίζεται την εναλλαγή και αποδέσμευση μνήμης των νημάτων (scheduling context). Το κυρίως νήμα έχει το δικό του context όπως όλα τα νήματα και αντιμετωπίζεται ως ισότιμο από τον μηχανισμό εναλλαγής νημάτων που θα υλοποιήσετε (scheduler).

Όταν τερματίζει ένα νήμα, η υλοποίηση αποδεσμεύει τη μνήμη που έχει δεσμεύσει για το συγκεκριμένο νήμα. Η αποδέσμευση γίνεται από διαφορετικό context (δεν μπορούμε να αποδεσμεύσουμε τη στοίβα ενός thread από το ίδιο το thread, καθώς αυτό θα οδηγήσει σε error όταν το thread θα επιχειρήσει να ξαναχρησιμοποιήσει τη στοίβα του). Η διαχείριση της στοίβας του κυρίως νήματος είναι και σε αυτή την περίπτωση διαφορετική.

Η εκτέλεση της main συνάρτησης κάθε νήματος περιβάλλεται από μία βοηθητική συνάρτηση (wrapper) έτσι ώστε να μπορεί η υλοποίηση να μεταβάλλει το περιβάλλον του κάθε νήματος πριν αυτό ξεκινήσει και αφού τερματίσει.



Recycling δομών

Σε μια απλή υλοποίηση θα περίμενα ότι οι δομές δεδομένων (thread descriptor, stack, ενδεχομένως και άλλες, ανάλογα με την υλοποίησή σας) χρησιμοποιούνται από ένα μόνο thread και μετά καταστρέφονται.

Μία βελτιστοποίηση είναι να διατηρούνται σε ουρές και να επαναχρησιμοποιούνται. Αν η αντίστοιχη ουρά βρεθεί άδεια τότε θα δεσμεύεται καινούρια δομή. Αν επίσης το μέγεος της ουράς υπερβολικά θα απελευθερώνετε κάποιες από τις δομές. Θα πρέπει, ως ένα βήμα βελτιστοποίησης, να υλοποιήσετε την ανακύκλωση δομών (με τρόπο που με ένα preprocessor #define να μπορεί να ενεργοποιηθεί / απενεργοποιηθεί) και να μετρήσετε την αποδοτικότητά της (είτε με τα παραδείγματα που σας δίνουμε είτε με άλλα). Ένα χαρακτηριστικό test θα ήταν ο μέσος χρόνος του κύκλου ζωής ενός thread που εκτελεί ένα κενό function.

Το recycling δομών είναι εύκολο για τη stack αλλά ενδεχομένως να είναι πιο δύσκολο για τους thread descriptors (συνεπώς δοκιμάστε πρώτα με τις stacks).

Ενδεικτικά Tests:

Έχουν ανέβει 4 ενδεικτικά test για να δοκιμάσετε τη βιβλιοθήκη σας:

α) main.c: Κατασκευάζει 4 threads (t1, t2, t3, t4). Το t1 εξαρτάται από το t2 και το t2 από τα t3 και t4.
β) matmul.c: Απλός πολλαπλασιασμός πινάκων. Κάθε γραμμή του πίνακα αποτέλεσμα υπολογίζεται από ένα thread.
γ) matmul_yield.c: Ομοίως με το (β), με τη διαφορά ότι κάθε thread κάνει yield μετά τον υπολογισμό ενός στοιχείου.
δ) matmul_multilevel.c: Ομοίως με το (β), όμως κάθε thread χωρίζει τη γραμμή που του έχει ανατεθεί να υπολογίσει στον πίνακα αποτελέσματος σε κομμάτια (chunks) και φτιάχνει ένα thread για τον υπολογισμό κάθε chunk.

Τα παραπάνω tests δοκιμάζουν πολλά (αλλά όχι απαραίτητα όλα) τα σημεία της βιβλιοθήκης. Μπορείτε να φτιάξετε τα δικά σας, να αλλάξετε τα ήδη υπάρχοντα κλπ. Σημειώστε ότι αυτά τα tests δεν ελέγχουν τη λειτουργικότητα των σημαφόρων.

Βήματα:
1ο στάδιο
Υλοποιήστε το interface (API) της βιβλιοθήκης (header file που βλέπει ο χρήστης, ενδεχομένως "εσωτερικό/ά" header file/s, ένα ή περισσότερα c αρχεία). Στα c αρχεία υλοποιείτε τις συναρτήσεις του API της βιβλιοθήκης ως κενές συναρτήσεις. Στόχος σας είναι να κάνουν compile τα ενδεικτικά προγράμματα (αλλά βεβαίως όχι να κάνουν κάτι ως προς την εκτέλεσή τους).

Φτιάξτε συναρτήσεις (για εσωτερική χρήση στη βιβλιοθήκη σας) για χειρισμό 2πλα διασυνδεμένων ουρών με 2 άκρα. Χρειάζεται να μπορείτε να αρχικοποιήσετε μια ουρά, να καταστρέψετε μια ουρά, να απαντήσετε αν η ουρά είναι άδεια, να βάλετε κάτι στην αρχή της ουράς, να βάλετε κάτι στο τέλος της ουράς, να βγάλετε κάτι από την αρχή της ουράς). Οι συναρτήσεις θα πρέπει να χειρίζονται κόβους που τα 2 πρώτα πεδία τους είναι δείκτες προς το προηγούμενο και το επόμενο στοιχείο, με ονόματα prev και next, αλλά κατά τα άλλα δε θα πρέπει να σας ενδιαφέρει τι ακριβώς τύπου είναι ο κόμβος.

Διαβάστε τα man pages των συναρτήσεων makecontext, getcontext, swapcontext. Θα τις χρησιμοποιήσουμε για user-level context switching. Στο man page της makecontext υπάρχει ένα πολύ καλό παράδειγμα.

2ο στάδιο
Ξεκινάτε να πειραματίζεστε με την κατασκευή των native threads (όχι ακόμα των user-level threads). Για την ώρα δουλεύετε με ένα μόνο native thread, το βασικό που ξεκίνησε το πρόγραμμα. Θα χρειαστεί όμως

για αυτό να φτιάξουμε ένα ακόμα context, αυτό του scheduler. Το context αυτό θα πρέπει να τρέχει ένα scheduling function (για την ώρα υπότυπώδες) και να έχει τη δική του stack. Για τα context χρήσιμες θα σας είναι οι συναρτήσεις getcontext, makecontext, swapcontext. Δεσμεύστε ό,τι διαχειριστικές δομές θεωρείτε ότι θα σας φανούν χρήσιμες. Π.χ. Μπορεί να χρειαστείτε μία δομή που να περιγράφει κάθε native thread (παρόλο που για την ώρα θα είναι 1).

Για τα χαρακτηριστικά των stacks των native threads στο σύστημά σας θα φανούν χρήσιμες οι συναρτήσεις getpagesize, getrlimit. Για τη δέσμευση μνήμης με συγκεκριμένο alignment υπάρχει η συνάρτηση posix_memalign. Θυμηθείτε, όταν δώσετε στο context τη stack, ότι σε επεξεργαστές x86 η stack κινείται από μεγαλύτερες προς μικρότερες διευθύνσεις.

Ιδανικά θα θέλαμε να έχετε καταφέρει να εναλλαγείτε μεταξύ του context με το οποίο ξεκινήσατε την εφαρμογή και του scheduler context (και πάλι πίσω).

Τελευταία ανανέωση: 3/3/2022 11:00