Λειτουργικά Συστήματα

7ο εξάμηνο, Ακαδημαϊκή περίοδος 2017-2018

Άσκηση 3:
Συγχρονισμός

oslabc19

Χαράλαμπος Κάρδαρης
A.M. 03114074
bkardaris@hotmail.com

Μιχάλης Παπαδόπουλλος
A.M. 03114702
mechatr0nic.cy@gmail.com

Άσκηση 1:

Απαντήσεις:

$ time ./simplesync – (without synchronization)
./simplesync 0.33s user 0.00s system 184% cpu 0.178 total

$ time ./simplesync – (with GCC builtins)
./simplesync 1.50s user 0.00s system 179% cpu 0.833 total

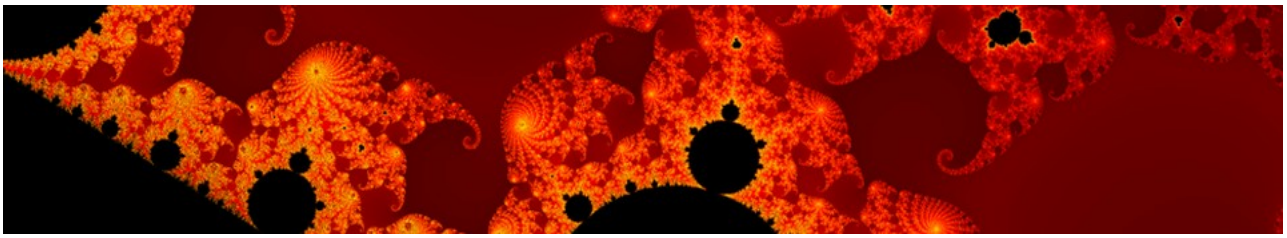
$ time ./simplesync – (with mutex)
./simplesync 2.74s user 1.64s system 183% cpu 2.393 total

- Παρατηρούμε ότι ο χρόνος εκτέλεσης αυξάνεται όπως και είναι λογικό εφ’ όσον υλοποιούμε σχήμα συγχρονισμού στην ενημέρωση των τιμών της κοινής μεταβλητής
- Όταν χρησιμοποιείται mutex αντί για τα atomic operations, παρατηρούμε μια αύξηση στο χρόνο εκτέλεσης. Αυτή η αύξηση οφείλεται στο overhead που εισάγει η δομή mutex.
- Οι ατομικές προσθεσεις `__sync_fetch_and_add (ip, 1);` εξασφαλίζουν οτι θα εκτελεστουν μαζι ως συνολο, και δεν πρόκειται η σειρά τους να αλλάξει απο κάποιο επεξεργαστη που υποστηριζει out of order execution

```
__sync_fetch_and_add (ip, 1);
        movq   -16(%rbp), %rax
        lock addl       $1, (%rax)
```

```
pthread_mutex_lock ( &mutex );
++(*ip);
pthread_mutex_unlock (&mutex);
        movl   $mutex, %edi
        call    pthread_mutex_lock
        testl   %eax, %eax
        je      .L3
.L3:
        movq   -16(%rbp), %rax
        movl   (%rax), %eax
        leal    1(%rax), %edx
        movq   -16(%rbp), %rax
        movl   %edx, (%rax)

        movl   $mutex, %edi
        call    pthread_mutex_unlock
```



Άσκηση 2: Mandelbrot Set

Περιγραφή:
Παρατηρούμε ότι ο υπολογισμός για το χρώμα κάθε γραμμής μπορεί να εκτελεστεί ανεξάρτητα. Συνεπώς δεν έχουμε κάποιας μορφής data dependency για τον υπολογισμό και έτσι μπρούμε να παραλληλοποιήσουμε την διαδικασία αυτή.
Ωστόσο κάθε γραμμή πρέπει να τυπωθεί κατα αύξουσα σειρά ώστε το παραγόμενο output να είναι συνεπές με το σκοπό του προγράμματος.

Version 1: Σκεφτήκαμε να δημιουργήσουμε το διδιάστατο πίνακα ο οποίος θα είναι shared ανάμεσα στα threads που θα δημιουργηθούν, ώστε κάθε thread να υπολογίζει τις γραμμές που του αντιστοιχούν. Δηλαδή: Για κάθε νήμα t με σχετικό αναγνωριστικό i, ο t υπολογίζει τις γραμμές i+k*total_threads για κάθε k τ.ω η πιο πάνω ποσότητα lineid <= y_chars.

Ωστόσο τρέχοντας το πρόγραμμα, παρατηρήσαμε ότι το tradeoff μεταξύ απλότητας υλοποιήσης και απόδοσης ήταν μεγάλο και γι' αυτό αποφασίσαμε να υλοποιήσουμε σχήμα συγχρονισμού με τη βοήθεια σημαφόρων (semaphores). Στη συγκεκριμένη υλοποίηση για κάθε παραγόμενο thread φτιαχνουμε τον αντιστοιχο σημαφόρο. Όλοι οι σημαφόρο αρχικοποιούνται με 0 (όταν κάποιο νήμα καλέσει `sem_wait ( &semaphore_id )` μπλοκάρει (blocks) μέχρι η τιμή του σημαφόρου να είναι θετική) εκτός απο τον σημαφόρο που αντιστοιχεί στο thread που αναλαμβάνει τον υπολογισμό για την 1η γραμμή – όπου αρχικοποιείται με 1.

Στο πιο κάτω κομμάτι κώδικα, βλέπουμε τη συνάρτηση έναρξης των νημάτων:

```
void *start_fn(void *arg)
{
    pthread_info_t thread = *(pthread_info_t *)arg;
        int color_val[x_chars];

    for (int i = thread.rid; i < y_chars; i += total_threads) {
        compute_mandel_line(i, color_val);
        sem_wait(&semaphores[thread.rid % total_threads]);    /* block on current line */
        output_mandel_line(1, color_val);
        sem_post(&semaphores[(thread.rid + 1) % total_threads]);
    }
    return NULL;
}
```

- Ο υπολογισμός της κάθε γραμμής γίνεται ανεξάρτητα, με δομή που είναι private μεταξύ των νημάτων.
- Το κρίσιμο τμήμα, ορίζεται μεταξύ των `sem_wait ()` και `sem_post ()` και είναι υπεύθυνο ώστε το output κάθε γραμμής να εμφανίζεται με την λογικά ορθή σειρά. Αυτό το καταφέρνουμε με την κυκλική "ενεργοποίηση" των σημαφόρων σε εκτέλεση.

Δηλαδή, το εκαστοτε νήμα `t` με relative id `(rid + 1) % total_threads`, θα μπεί στο κρίσιμο τμήμα μόνο όταν το νήμα με relative id `rid % total_threads` εξέλθει απο την κρίσιμη περιοχή.

Η υλοποιήση αυτή είναι πιο γρήγορη, απο την υλοποίηση με κοινή μνημη και μπορεί να γίνει ακόμη πιο αποδοτική, εαν αποφύγουμε το overhead χρόνου που εισάγεται απο τη χρήση σημαφόρων. Όπως βλέπουμε, στη συγκεκριμένη υλοποίηση οι σημαφόροι χρησιμοποιούνται ως mutexes, εφ' όσον μεταβάλλονται μεταξύ δύο καταστάσεων 0,1. Έτσι μπορούμε να υλοποιήσουμε το κρίσιμο τμήμα με χρήση ατομικών built-in λειτουργιών του compiler GCC για την υλοποίηση του κρίσιμου τμήματος.

Απαντήσεις:

1. Στη συγκεκριμένη υλοποίηση χρειαζόμαστε τόσους σημαφόρους, όσος και ο αριθμός των νημάτων που δημιουργούνται
2. Εκτελώντας `cat *proc*/*cpuinfo*` βλέπουμε πληροφορίες σχετικά με το σύστημα μας.
   Το σύστημά μας, είναι 4-πύρηνο, όπως φαίνεται πιο κάτω:

   $ cat /*proc*/*cpuinfo* | *grep cores* | *uniq*
   cpu cores : 4

   $ time ./mandel – (Το σειριακό πρόγραμμα)
   ./mandel 0.74s user 0.02s system 97% cpu 0.783 total

   $ time ./mandel 2 – (Η υλοποίηση με σημαφόρους, με 2 νηματα)
   ./mandel 2 0.76s user 0.01s system 131% cpu 0.520 total

   $ time ./mandel 2 – (Η υλοποίηση με σημαφόρους, με 4 νηματα)
   ./mandel 4 0.98s user 0.03s system 380% cpu 0.264 total

Όπως βλέπουμε το παράλληλο πρόγραμμα όντως παρουσιάζει επιτάχυνση, ωστόσο σύμφωνα και με το νόμο του Amdahl, η μέγιστη θεωρητική επιτάχυνση που ενδέχεται να παρουσιάσει το πρόγραμμα, ειναι bounded απο το ποσοστό της "σειριακής" εκτέλεσης καποιου υποσυνόλου εντολών του προγράμματος. Στην δική μας περίπτωση, η ανάγκη για σειριοποίηση εμφανίζεται στο τύπωμα των γραμμών.

4. Κατα την εκτέλεση του προγράμματος, όταν πατηθεί `Ctrl-C` δηλαδή σταλεί στη διεργασία ένα σήμα ***SIGINT*** με αποτέλεσμα να σταματήσει την εκτέλεση και να εμφανίσει οποιοδήποτε output με το χρώμα που έχει υπολογίσει. Για την αποφυγή αυτής της κατάστασης, πρέπει να υλοποιήσουμε ενα signal handler – που θα εκτελεί κάθε φόρα που κάνει catch το εν λόγω σήμα – μια ρουτίνα που θα καλεί τη `reset_xterm_color ()` και μετά να τερματίζει τη λειτουργία του προγράμματος με κλήση της `exit ()`.

Άσκηση 3: Kindergarten

Code

Οι αλλαγές που κάναμε στον κώδικα φαίνονται στις παρακάτω συναρτήσεις. Οπως φαίνεται προτιμήσαμε να αλλάξουμε τη θέση της κλήσης της συνάρτησης verify και της εμφάνισης του μηνύματος “Entered/Exited”, και την τοποθετήσαμε μέσα στην κρίσιμη περιοχή, εντός του lock, στις επιμέρους συναρτήσεις, ώστε να έχουμε πιο σωστά δεδομένα. Παρατηρήσαμε ότι όπως ήταν δοσμένη, μέχρι να εκτελεστεί, παρεμβάλονταν η εκτέλεση άλλων threads με συνέπεια να έχουμε μεν πάντα τα πιο καινούρια δεδομένα, αλλά να μην υπάρχει σχέση μεταξύ της ενέργειας που εκτελούσε ένα thread και του αποτελέσματος της verify που καλούσε το ίδιο thread.

```
void child_enter(struct thread_info_struct *thr)
{
if (!thr->is_child) {
    fprintf(stderr, "Internal error: %s called for a Teacher thread.\n", __func__);
     exit(1);
}

fprintf(stderr, "THREAD %d: CHILD ENTER\n", thr->thrid);

pthread_mutex_lock(&thr->kg->mutex);
int c = thr->kg->vc;
   int t = thr->kg->vt;
   int r = thr->kg->ratio;
while(c >= t * r){
    pthread_cond_wait(&thr->kg->cond1,&thr->kg->mutex);
    c = thr->kg->vc;
    t = thr->kg->vt;
    r = thr->kg->ratio;
   }
c = ++(thr->kg->vc);
fprintf(stderr, "Thread %d [%s]: Entered.\n", thr->thrid, thr->is_child ? "Child" : "Teacher");
verify(thr);

if (c < t * r) pthread_cond_signal(&thr->kg->cond1);
//if (c <= (t-1)*r) pthread_cond_signal(&thr->kg->cond2);

pthread_mutex_unlock(&thr->kg->mutex);

}
```

```
void child_exit(struct thread_info_struct *thr)
{

if (!thr->is_child) {
    fprintf(stderr, "Internal error: %s called for a Teacher thread.\n",
                                                        __func__);
     exit(1);
   }

fprintf(stderr, "THREAD %d: CHILD EXIT\n", thr->thrid);

pthread_mutex_lock(&thr->kg->mutex);
--(thr->kg->vc);
fprintf(stderr, "Thread %d [%s]: Exited.\n", thr->thrid, thr->is_child ? "Child" : "Teacher");
verify(thr);

int c = thr->kg->vc;
   int t = thr->kg->vt;
   int r = thr->kg->ratio;
if (c <= (t-1)*r) pthread_cond_broadcast(&thr->kg->cond2);
if (c < t * r) pthread_cond_signal(&thr->kg->cond1);

pthread_mutex_unlock(&thr->kg->mutex);
}

void teacher_enter(struct thread_info_struct *thr)
{
if (thr->is_child) {
    fprintf(stderr, "Internal error: %s called for a Child thread.\n",
                                                        __func__);
     exit(1);
   }

fprintf(stderr, "THREAD %d: TEACHER ENTER\n", thr->thrid);

pthread_mutex_lock(&thr->kg->mutex);
++(thr->kg->vt);
fprintf(stderr, "Thread %d [%s]: Entered.\n", thr->thrid, thr->is_child ? "Child" : "Teacher");
verify(thr);

int c = thr->kg->vc;
   int t = thr->kg->vt;
   int r = thr->kg->ratio;

pthread_cond_broadcast(&thr->kg->cond1);
if (c <= (t-1) * r) pthread_cond_signal(&thr->kg->cond2);

pthread_mutex_unlock(&thr->kg->mutex);
}
```

```
void teacher_exit(struct thread_info_struct *thr)
{
if (thr->is_child) {
    fprintf(stderr, "Internal error: %s called for a Child thread.\n",
                                                        __func__);
     exit(1);
   }

fprintf(stderr, "THREAD %d: TEACHER EXIT\n", thr->thrid);


pthread_mutex_lock(&thr->kg->mutex);
int c = thr->kg->vc;
 int t = thr->kg->vt;
   int r = thr->kg->ratio;
while(c > (t-1) * r){
    pthread_cond_wait(&thr->kg->cond2, &thr->kg->mutex);
    c = thr->kg->vc;
    t = thr->kg->vt;
    r = thr->kg->ratio;

   }
t = --(thr->kg->vt);
fprintf(stderr, "Thread %d [%s]: Exited.\n", thr->thrid, thr->is_child ? "Child" : "Teacher");
verify(thr);

//if (c < t * r) pthread_cond_signal(&thr->kg->cond1);
if (c <= (t-1) * r) pthread_cond_signal(&thr->kg->cond2);

pthread_mutex_unlock(&thr->kg->mutex);

}

/*
 * A single thread.
 * It simulates either a teacher, or a child.
 */
void *thread_start_fn(void *arg)
{
/* We know arg points to an instance of thread_info_struct */
struct thread_info_struct *thr = arg;
char *nstr;

fprintf(stderr, "Thread %d of %d. START.\n", thr->thrid, thr->thrcnt);


nstr = thr->is_child ? "Child" : "Teacher";
for (;;) {
    fprintf(stderr, "Thread %d [%s]: Entering.\n", thr->thrid, nstr);
```

```
    if (thr->is_child)
                                                child_enter(thr);
        else
                                               teacher_enter(thr);


    //fprintf(stderr, "Thread %d [%s]: Entered.\n", thr->thrid, nstr);

        /*
     * We're inside the critical section,
     * just sleep for a while.
        */
    /* usleep(rand_r(&thr->rseed) % 1000000 / (thr->is_child ? 10000 : 1)); */

        /*
       pthread_mutex_lock(&thr->kg->mutex);
    verify(thr);
       pthread_mutex_unlock(&thr->kg->mutex);

      */

    usleep(rand_r(&thr->rseed) % 1000000);

    fprintf(stderr, "Thread %d [%s]: Exiting.\n", thr->thrid, nstr);
    /* CRITICAL SECTION END */

    if (thr->is_child)
                                                 child_exit(thr);
        else
                                               teacher_exit(thr);

    //fprintf(stderr, "Thread %d [%s]: Exited.\n", thr->thrid, nstr);

    /* Sleep for a while before re-entering */
    /* usleep(rand_r(&thr->rseed) % 100000 * (thr->is_child ? 100 : 1)); */
    usleep(rand_r(&thr->rseed) % 100000);

        /*
       pthread_mutex_lock(&thr->kg->mutex);
    verify(thr);
       pthread_mutex_unlock(&thr->kg->mutex);
         */
     //break;
   }

fprintf(stderr, "\t\t\t\t\t\t\tThread %d of %d. END.\n", thr->thrid, thr->thrcnt);

return NULL;
}
```

Output

$ ./kgarten-2 10 6 2

```
Thread 0 of 10. START.
Thread 0 [Child]: Entering.
THREAD 0: CHILD ENTER
Thread 1 of 10. START.
Thread 1 [Child]: Entering.
THREAD 1: CHILD ENTER
Thread 2 of 10. START.
Thread 2 [Child]: Entering.
THREAD 2: CHILD ENTER
Thread 3 of 10. START.
Thread 3 [Child]: Entering.
THREAD 3: CHILD ENTER
Thread 5 of 10. START.
Thread 5 [Child]: Entering.
THREAD 5: CHILD ENTER
Thread 4 of 10. START.
Thread 4 [Child]: Entering.
THREAD 4: CHILD ENTER
Thread 6 of 10. START.
Thread 6 [Teacher]: Entering.
THREAD 6: TEACHER ENTER
Thread 6 [Teacher]: Entered.
        Thread 6: Teachers: 1, Children: 0
Thread 7 of 10. START.
Thread 8 of 10. START.
Thread 7 [Teacher]: Entering.
Thread 9 of 10. START.
Thread 9 [Teacher]: Entering.
THREAD 9: TEACHER ENTER
THREAD 7: TEACHER ENTER
Thread 8 [Teacher]: Entering.
THREAD 8: TEACHER ENTER
Thread 2 [Child]: Entered.
        Thread 2: Teachers: 1, Children: 1
Thread 0 [Child]: Entered.
        Thread 0: Teachers: 1, Children: 2
Thread 9 [Teacher]: Entered.
        Thread 9: Teachers: 2, Children: 2
Thread 5 [Child]: Entered.
        Thread 5: Teachers: 2, Children: 3
Thread 4 [Child]: Entered.
        Thread 4: Teachers: 2, Children: 4
Thread 7 [Teacher]: Entered.
        Thread 7: Teachers: 3, Children: 4
Thread 1 [Child]: Entered.
        Thread 1: Teachers: 3, Children: 5
```

Thread 3 [Child]: Entered.
        Thread 3: Teachers: 3, Children: 6
Thread 8 [Teacher]: Entered.
        Thread 8: Teachers: 4, Children: 6
Thread 7 [Teacher]: Exiting.
THREAD 7: TEACHER EXIT
Thread 7 [Teacher]: Exited.
        Thread 7: Teachers: 3, Children: 6
Thread 8 [Teacher]: Exiting.
THREAD 8: TEACHER EXIT
Thread 7 [Teacher]: Entering.
THREAD 7: TEACHER ENTER
Thread 7 [Teacher]: Entered.
        Thread 7: Teachers: 4, Children: 6
Thread 8 [Teacher]: Exited.
        Thread 8: Teachers: 3, Children: 6
Thread 8 [Teacher]: Entering.
THREAD 8: TEACHER ENTER
Thread 8 [Teacher]: Entered.
        Thread 8: Teachers: 4, Children: 6
Thread 6 [Teacher]: Exiting.
THREAD 6: TEACHER EXIT
Thread 6 [Teacher]: Exited.
        Thread 6: Teachers: 3, Children: 6
Thread 8 [Teacher]: Exiting.
THREAD 8: TEACHER EXIT
Thread 6 [Teacher]: Entering.
THREAD 6: TEACHER ENTER
Thread 6 [Teacher]: Entered.
        Thread 6: Teachers: 4, Children: 6
Thread 8 [Teacher]: Exited.
        Thread 8: Teachers: 3, Children: 6
Thread 8 [Teacher]: Entering.
THREAD 8: TEACHER ENTER
Thread 8 [Teacher]: Entered.
        Thread 8: Teachers: 4, Children: 6
Thread 3 [Child]: Exiting.
…
…
…

$ ./kgarten-2 6 4 2
Thread 1 of 6. START.
Thread 1 [Child]: Entering.
THREAD 1: CHILD ENTER
Thread 0 of 6. START.
Thread 0 [Child]: Entering.
THREAD 0: CHILD ENTER
Thread 2 of 6. START.

Thread 2 [Child]: Entering.
Thread 4 of 6. START.
Thread 4 [Teacher]: Entering.
THREAD 4: TEACHER ENTER
Thread 4 [Teacher]: Entered.
	Thread 4: Teachers: 1, Children: 0
Thread 5 of 6. START.
Thread 1 [Child]: Entered.
	Thread 1: Teachers: 1, Children: 1
Thread 5 [Teacher]: Entering.
THREAD 5: TEACHER ENTER
Thread 3 of 6. START.
Thread 3 [Child]: Entering.
THREAD 3: CHILD ENTER
Thread 0 [Child]: Entered.
	Thread 0: Teachers: 1, Children: 2
THREAD 2: CHILD ENTER
Thread 5 [Teacher]: Entered.
	Thread 5: Teachers: 2, Children: 2
Thread 2 [Child]: Entered.
	Thread 2: Teachers: 2, Children: 3
Thread 3 [Child]: Entered.
	Thread 3: Teachers: 2, Children: 4
Thread 4 [Teacher]: Exiting.
THREAD 4: TEACHER EXIT
Thread 3 [Child]: Exiting.
THREAD 3: CHILD EXIT
Thread 3 [Child]: Exited.
	Thread 3: Teachers: 2, Children: 3
Thread 3 [Child]: Entering.
THREAD 3: CHILD ENTER
Thread 3 [Child]: Entered.
	Thread 3: Teachers: 2, Children: 4
Thread 5 [Teacher]: Exiting.
THREAD 5: TEACHER EXIT
Thread 0 [Child]: Exiting.
THREAD 0: CHILD EXIT
Thread 0 [Child]: Exited.
	Thread 0: Teachers: 2, Children: 3
Thread 0 [Child]: Entering.
THREAD 0: CHILD ENTER
Thread 0 [Child]: Entered.
	Thread 0: Teachers: 2, Children: 4
Thread 0 [Child]: Exiting.
THREAD 0: CHILD EXIT
Thread 0 [Child]: Exited.
	Thread 0: Teachers: 2, Children: 3
Thread 1 [Child]: Exiting.
THREAD 1: CHILD EXIT

Thread 1 [Child]: Exited.
	Thread 1: Teachers: 2, Children: 2
Thread 4 [Teacher]: Exited.
	Thread 4: Teachers: 1, Children: 2
Thread 2 [Child]: Exiting.
THREAD 2: CHILD EXIT
Thread 2 [Child]: Exited.
	Thread 2: Teachers: 1, Children: 1
Thread 1 [Child]: Entering.
THREAD 1: CHILD ENTER
Thread 1 [Child]: Entered.
	Thread 1: Teachers: 1, Children: 2
Thread 4 [Teacher]: Entering.
THREAD 4: TEACHER ENTER
Thread 4 [Teacher]: Entered.
	Thread 4: Teachers: 2, Children: 2
Thread 5 [Teacher]: Exited.
	Thread 5: Teachers: 1, Children: 2
Thread 0 [Child]: Entering.
THREAD 0: CHILD ENTER
Thread 2 [Child]: Entering.
THREAD 2: CHILD ENTER
Thread 5 [Teacher]: Entering.
THREAD 5: TEACHER ENTER
Thread 5 [Teacher]: Entered.
	Thread 5: Teachers: 2, Children: 2
Thread 0 [Child]: Entered.
	Thread 0: Teachers: 2, Children: 3
Thread 2 [Child]: Entered.
	Thread 2: Teachers: 2, Children: 4
Thread 3 [Child]: Exiting.
THREAD 3: CHILD EXIT
Thread 3 [Child]: Exited.
	Thread 3: Teachers: 2, Children: 3
Thread 3 [Child]: Entering.
THREAD 3: CHILD ENTER
Thread 3 [Child]: Entered.
	Thread 3: Teachers: 2, Children: 4
…
…
…

Απαντήσεις

1. Ο κώδικας που έχουμε υλοποιήσει δεν δίνει προτεραιότητα σε παιδιά ή δασκάλους. Αυτό σημαίνει ότι όταν πολλά παιδια θέλουν να μπουν και δάσκαλοι να βγουν έχουν ίδια διαδικασία που πρέπει να ακολουθήσουν. Αυτή είναι να πάρουν το lock, και ακολούθως να ελέγξουν αν ισχύει η συνθήκη διατήρησης του ratio (κάνοντας cond_wait ή συνεχίζοντας με την ενέργεια τους). Οπότε τελικά, όποιος προλάβει έχει τη δυνατότητα να κάνει την ενέργεια που θέλει και μόνο αν επιτρέπεται.
Από εκεί και πέρα υπάρχει μια διαφοροποίηση όσον αφορά το signal ή broadcast που κάνει κάθε thread. Αν ένας δάσκαλος κάνει enter, κάνουμε broadcast πρώτα στα παιδιά, οπότε έχουν καλύτερη πιθανότητα να πάρουν το lock. Αν ένα παιδί κάνει exit, κάνουμε signal στους δασκάλους, για να έχουν ίσως την ευκαρία να φύγουν.
Σε κάθε περίπτωση για τις συνθήκες που περιγράφει η ερώτηση, ένας δάσκαλος που περιμένει να φύγει έχει περισσότερες πιθανότητες να το κάνει αν φύγει κάποιος μαθητής.
Γενικά, η σειρά που εκτελούμε τα signal ή broadcast μπορεί να ευνοήσει τη μία ομάδα ή την άλλη.

Όσον αφορά κάθε thread, αυτό προσπαθεί να πάρει το lock με την εντολή pthread_mutex_lock. Απο τα manpages έχουμε:

> “The mutex object referenced by mutex shall be locked by a call to pthread_mutex_lock() that returns zero or [EOWNERDEAD]. If the mutex is already locked by another thread, the calling thread shall block until the mutex becomes available.”

Τα παιδιά που θέλουν να μπουν και οι δάσκαλοι που θέλουν να βγουν χρησιμοποιούν condition variables. Όταν η συνθήκη για block ισχύει η κλήση της συνάρτησης pthread_cond_wait κάνει unlock το mutex, που έχει γίνει πριν lock από το ίδιο thread. Στη συνέχεια, η εκτέλεση looράρει(κάθε φορά το thread παίρνει το lock για να προχωράει αυτό το loop) και κάνει pthread_cond_wait μέχρι να μην ικανοποιείται η συνθήκη όταν και το thread έχει ξαναπάρει το lock και τότε προχωράει η εκτέλεση στο κρίσιμο τμήμα. Από τα manpages:

> “The pthread_cond_wait() function shall block on a condition variable.
> This function atomically releases mutex and causes the calling thread to block on the condition variable cond; atomically here means “atomically with respect to access by another thread to the mutex and then the condition variable”.”

Κάθε thread με το πέρας της εκτέλεσης του κώδικα στην κρίσιμη περιοχή στέλνει σήματα ptrhread_cond_broadcast ή pthread_cond_signal στα threads που είναι blocked λόγω του condition. Το broadcast τα ξυπνάει όλα, ενώ το signal ξυπνάει ένα τυχαία επιλεγμένο. Από τα manpages:

> “The pthread_cond_broadcast() function shall unblock all threads currently blocked on the specified condition variable cond.
> The pthread_cond_signal() function shall unblock at least one of the threads that are blocked on the specified condition variable cond (if any threads are blocked on cond).
> If more than one thread is blocked on a condition variable, the scheduling policy shall determine the order in which threads are unblocked.”

Στο τέλος του κρίσιμου τμήματος κάθε thread φυσικά απελευθερώνει το mutex με την κλήση της pthread_mutex_unlock.

2. Ο αρχικός κώδικας, όπως αναφέραμε και στις αλλαγές που κάναμε, περιείχε ένα τέτοιο race condition κατά την κλήση της verify. Αυτό που παρατηρήσαμε ήταν ότι, ενώ για παράδειγμα εκτελούνταν η verify για τον πρώτο δάσκαλο που έκανε enter, το output της πιθανόν να έδειχνε περισσότερους δασκάλους και παιδιά, επειδή τα threads τους είχαν προλάβει να τρέξουν στο ενδιάμεσο. Πάντως, αν και συνέβαινε αυτό, αυτό το race condition δεν ήταν critical, και αυτό γιατί η εξασφάλιση της συνθήκης γινόταν πάντα εντός της κρίσιμης περιοχής, και δεν υπήρχε περίπτωση να παραβιάσουμε τον κανόνα μας. Όμως, δεν έπαυε να παρουσιάζει μια απόκρυψη πληροφορίας, ως προς την σειρά που γίνονταν οι διάφορες κινήσεις. Για αυτό το λόγο, αποφασίσαμε να κάνουμε verify εντός της κρίσιμης περιοχής, ώστε να διευκολυνθούμε και στην εξακρίβωση της ορθότητας του κώδικα μας.