

Λειτουργικά Συστήματα, 6^ο Εξάμηνο ΗΜΜΥ

Ακαδημαϊκή Περίοδος 2019-2020

Άσκηση 3: Συγχρονισμός

Ομάδα: oslabb33

Γιαννούλης Παναγιώτης (03117812)

Κανελλόπουλος Σωτήρης (03117101)

1.1 Μεταγλωττίζοντας και τρέχοντας το έτοιμο πρόγραμμα παρατηρήσαμε ότι τα δύο νήματα πράγματι δεν συγχρονίζουν σωστά την εκτέλεσή τους, αφού η μεταβλητή τελικά είναι διάφορη του 0.

Επεκτείνουμε τον κώδικα που μας δόθηκε ώστε τα δύο νήματα να συγχρονίζονται με mutexes και με χρήση ατομικών λειτουργιών.

Το makefile που μας δίνεται παράγει δύο εκτελέσιμα που κάνουν τις δύο αυτές λειτουργίες. Παρακάτω φαίνονται οι έξοδοι των δύο αυτών εκτελέσιμων:

```
pangiann@WhiteRose13:~/6o Εξάμηνο/opsys/3herg/sync\∞ ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
```

```
pangiann@WhiteRose13:~/6o Εξάμηνο/opsys/3herg/sync\∞ ./simplesync-atomic
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.
```

Απαντήσεις στις ερωτήσεις:

1. Παρακάτω φαίνονται οι έξοδοι του διορθωμένου προγράμματος με χρήση της εντολής time(1) για τη μέτρηση του χρόνου εκτέλεσης:

```

pangiann@WhiteRose13:~/6o Εξάμηνο/opsys/3herg/sync\∞ time ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.

real    0m0,119s
user    0m0,229s
sys     0m0,000s
pangiann@WhiteRose13:~/6o Εξάμηνο/opsys/3herg/sync\∞ time ./simplesync-mutex
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.

real    0m1,360s
user    0m1,740s
sys     0m0,940s

```

Συγκρίνουμε και με το χρόνο του αρχικού μη συγχρονισμένου προγράμματος:

```

oslabb33@os-nodel:~/sync$ time ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = 3249614.

real    0m0.038s
user    0m0.072s
sys     0m0.000s
oslabb33@os-nodel:~/sync$ time ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = -155632.

real    0m0.038s
user    0m0.072s
sys     0m0.000s

```

Παρατηρούμε ότι το εκτελέσιμο με τις ατομικές λειτουργίες είναι γρηγορότερο από αυτό με τα mutexes, ενώ το αρχικό (μη συγχρονισμένο) είναι γρηγορότερο κι απ τα δύο. Αυτό είναι λογικό, αφού για να συγχρονιστούν οι δύο λειτουργίες μεταξύ τους προφανώς χρειάζεται να «περιμένουν» η μία την άλλη, πράγμα το οποίο αναγκαστικά καθιστά το πρόγραμμα πιο αργό.

2. Από το screenshot που παραθέσαμε στο προηγούμενο ερώτημα βλέπουμε ότι η εκτέλεση με ατομικές λειτουργίες είναι γρηγορότερη από την εκτέλεση με mutexes. Αυτό οφείλεται στο γεγονός ότι τα mutexes πραγματοποιούν κλήσεις συναρτήσεων και εν συνεχεία system calls για το κλείδωμα σε αντίθεση με τις ατομικές λειτουργίες που εκτελείται μία επιπλέον εντολή στον επεξεργαστή (όπως θα δούμε και παρακάτω).

3. Από τα παρακάτω screenshots παρατηρούμε ότι η χρήση ατομικών λειτουργιών αντιστοιχεί στην εντολή lock addl \$1, (%rbx).

```
.L2:
    lock addl    $1, (%rbx)
    subl    $1, %eax
    jne     .L2
```

```
public increase_fn
increase_fn proc near
endbr64
push    rbx
mov     rbx, rdi
mov     rdi, cs:stderr@GLIBC_2_2_5
mov     ecx, 989680h
lea     rdx, aAboutToIncreas ; "About to increase variable %d times\n"
mov     esi, 1
xor     eax, eax
call    sub_1160
mov     eax, 989680h
nop     dword ptr [rax+00h]
```

```
loc_1400:
lock add dword ptr [rbx], 1
sub     eax, 1
jnz     short loc_1400
```

```
mov     rcx, cs:stderr@GLIBC_2_2_5
mov     edx, 1Ah
mov     esi, 1
lea     rdi, aDoneIncreasing ; "Done increasing variable.\n"
call    sub_1150
xor     eax, eax
pop     rbx
retn
increase_fn endp
```

4. Από τα παρακάτω screenshots βλέπουμε σε ποιες εντολές αντιστοιχεί η χρήση mutexes.

```
.p2align 3
.L17:
    movq    %r12, %rdi
    call    pthread_mutex_lock@PLT
    movl    0(%rbp), %eax
    movq    %r12, %rdi
    subl    $1, %eax
    movl    %eax, 0(%rbp)
    call    pthread_mutex_unlock@PLT
    subl    $1, %ebx
    jne     .L17
```

```
call    0x55555555551b0 <pthread_mutex_lock@plt>
mov     rax,QWORD PTR [rbp-0x8]
mov     eax,DWORD PTR [rax]
lea     edx,[rax-0x1]
mov     rax,QWORD PTR [rbp-0x8]
mov     DWORD PTR [rax],edx
lea     rdi,[rip+0x2c5a]      # 0x5555555555040 <mutex>
call    0x5555555555150 <pthread_mutex_unlock@plt>
```

Παρατηρούμε ότι στην πρώτη περίπτωση έχουμε μόνο μία εντολή της αρχιτεκτονικής, ενώ στη δεύτερη έχουμε κλήση συνάρτησης από τη βιβλιοθήκη pthread.

1.2 Παραθέτουμε μερικά κομβικά κομμάτια του κώδικά μας. Ο κώδικας υπάρχει ολόκληρος στο συμπίεσμένο αρχείο που κατατέθηκε.

Στην παρακάτω φωτογραφία φαίνεται το πώς αρχικοποιούμε τα threads, χρησιμοποιώντας `safe_malloc` και ορίζοντας τα πεδία τους. Για τον πρώτο σημαφόρο ($i = 0$), κάνουμε `sem_post` ώστε να μπορεί να ξεκινήσει το πρώτο thread την εκτύπωση της πρώτης γραμμής.

```

sem = safe_malloc(thrcnt * sizeof(sem_t));
for (int i = 0; i < thrcnt; i++) {
    sem_init(&sem[i], 0, 0);
}

thr = safe_malloc(thrcnt * sizeof(*thr));
for (int i = 0; i < thrcnt; i++) {

    if (i == 0)
        sem_post(&sem[i]);
    /* Initialize per-thread structure */
    thr[i].fd = 1;
    thr[i].thrcnt = thrcnt;
    thr[i].line = i;

    /* Spawn new thread */
    ret = pthread_create(&thr[i].tid, NULL, thread_compute_and_output_mandel_line, &thr[i]);
    if (ret) {
        perror("pthread_create");
        exit(1);
    }
}

```

Παρακάτω βλέπουμε την κλήση της `compute_and_output_mandel_line` από το κάθε νήμα. Το `reset_xterm_color` μετά την εκτύπωση της κάθε γραμμής έχει προστεθεί για να μην αλλάζει το χρώμα των γραμμάτων (βλ. ερώτηση 4).

```

void *thread_compute_and_output_mandel_line(void *arg) {

    struct thread_info_struct *thr = arg;
    for (int line = thr->line; line < y_chars; line += thr->thrcnt) {
        compute_and_output_mandel_line(1, line, thr->thrcnt);
        reset_xterm_color(1);
    }
    return NULL;
}

```

Τέλος, στην επόμενη εικόνα βλέπουμε τη χρήση των σημαφόρων στο συγχρονισμό του προγράμματος (βλ. ερώτηση 1), κάνοντας ένα `wait` για τον `current` σημαφόρο πριν την εκτύπωση της γραμμής και ένα `post` για τον `next` μετά την εκτύπωση, επιτρέποντας στο επόμενο `thread` να μπει στο κρίσιμο κομμάτι του κώδικα. Ουσιαστικά με αυτό τον τρόπο τα `threads` τρέχουν κυκλικά.

```

void compute_and_output_mandel_line(int fd, int line, int thrcnt)
{
    /*
     * A temporary array, used to hold color values for the line being drawn
     */
    int color_val[x_chars];
    int current = (line) % thrcnt;
    int next = (current + 1) % thrcnt;
    compute_mandel_line(line, color_val);

    sem_wait(&sem[current]);

    output_mandel_line(fd, color_val);

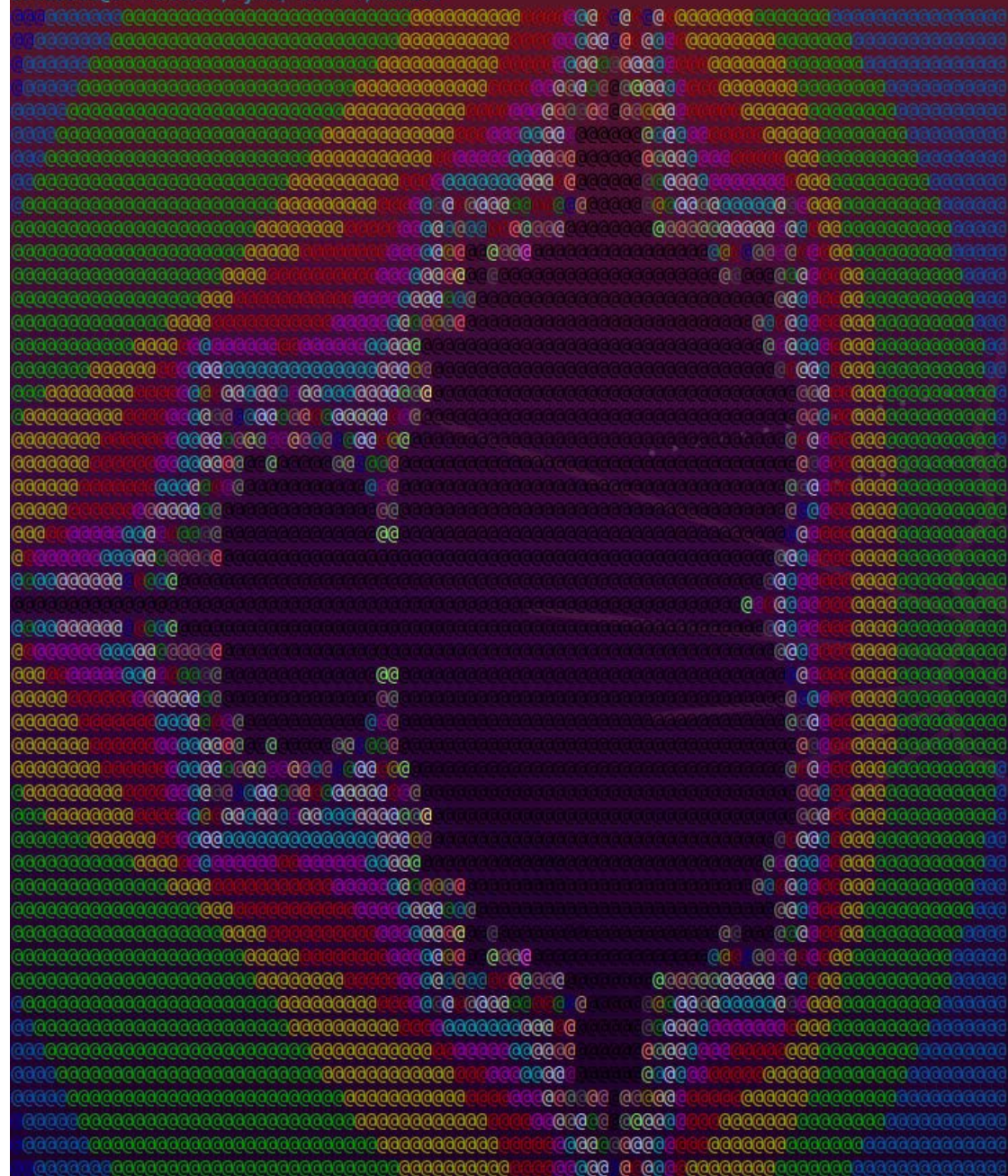
    sem_post(&sem[next]);
}

```

Απαντήσεις στις ερωτήσεις:

1. Χρειαζόμαστε τόσους σημαφόρους όσα και τα threads, ώστε να εξασφαλίσουμε ότι τρέχει μόνο ένα νήμα κάθε φορά και τα υπόλοιπα περιμένουν. Στην ουσία έχουμε η πόρτες και τις ανοίγουμε κυκλικά. Αν σκεφτούμε τη λειτουργία της εκτύπωσης ως την είσοδο σε ένα λεωφορείο, αν είχαμε μία μόνο πόρτα τότε θα βρισκόμασταν σε race condition μιας και μόλις άνοιγε δε θα μπορούσε να αποφασιστεί το ποιος θα εισέλθει. Με η πόρτες ελέγχουμε πλήρως και τη σειρά με την οποία θα μπουν οι διεργασίες στο λεωφορείο.
2. Στα παρακάτω screenshots φαίνονται οι χρόνοι εκτέλεσης του σειριακού και του παράλληλου προγράμματος με δύο νήματα:


```
oslab33@os-node1:~/sync$ time ./mandel
```



```
real    0m1.023s
user    0m0.968s
sys     0m0.028s
```



```
^C
real    0m0.456s
user    0m0.440s
sys     0m0.004s
oslab33@os-node1:~/sync$ ^C
oslab33@os-node1:~/sync$
```

Για να το διορθώσουμε αυτό βάζουμε την εντολή `reset_xterm.color(1)` μετά την εκτύπωση γραμμής. Έτσι έχουμε την εξής έξοδο διακόπτοντας το πρόγραμμα:

```
pangiannc@WhiteRose13:~/60 Εξάμηνο/opsys/3herg/sync$ time ./mandel 25
^C
real    0m0,186s
user    0m0,421s
sys     0m0,015s
```

Πράγματι, τα γράμματα έχουν επιστρέψει στο default χρώμα.

2.1 Παρακάτω φαίνονται η έξοδος του αρχικού και ένα κομμάτι της εξόδου του διορθωμένου προγράμματος. Παρατηρούμε ότι στην έξοδο του διορθωμένου προγράμματος ότι η αναλογία παιδιών καθηγητών παραμένει σωστή.

```
oslab33@os-nodel:~/sync$ ./kgarten 10 7 3
Thread 1 of 10. START.
Thread 2 of 10. START.
Thread 3 of 10. START.
Thread 0 of 10. START.
Thread 0 [Child]: Entering.
Thread 4 of 10. START.
Thread 4 [Child]: Entering.
THREAD 4: CHILD ENTER
Thread 1 [Child]: Entering.
THREAD 1: CHILD ENTER
Thread 7 of 10. START.
Thread 7 [Teacher]: Entering.
THREAD 0: CHILD ENTER
Thread 0 [Child]: Entered.
    Thread 0: Teachers: 0, Children: 3
Thread 8 of 10. START.
Thread 8 [Teacher]: Entering.
THREAD 8: TEACHER ENTER
Thread 9 of 10. START.
Thread 9 [Teacher]: Entering.
THREAD 9: TEACHER ENTER
*** Thread 0: Oh no! Little Jim drank a bottle of acid with his lunch!
*** Why were there only 0 teachers for 3 children?!
```

```
Thread 3 [Child]: Exited.
    Thread 3: Teachers: 3, Children: 4
Thread 3 [Child]: Entering.
THREAD 3: CHILD ENTER
Thread 3 [Child]: Entered.
    Thread 3: Teachers: 3, Children: 5
Thread 8 [Teacher]: Exiting.
THREAD 8: TEACHER EXIT
Thread 8 [Teacher]: Exited.
    Thread 4: Teachers: 2, Children: 5
Thread 4 [Child]: Entering.
THREAD 4: CHILD ENTER
Thread 4 [Child]: Entered.
    Thread 4: Teachers: 2, Children: 6
    Thread 8: Teachers: 2, Children: 6
Thread 8 [Teacher]: Entering.
THREAD 8: TEACHER ENTER
Thread 8 [Teacher]: Entered.
    Thread 8: Teachers: 3, Children: 6
    Thread 0: Teachers: 3, Children: 6
```

Απαντήσεις στις ερωτήσεις:

1. Στον κώδικά μας όταν βγαίνει κάποιο παιδί πρώτα γίνεται έλεγχος για το αν μπορεί τώρα να φύγει κάποιος καθηγητής (και αν μπορεί, ενεργοποιείται η κατά συνθήκη μεταβλητή teacher_out και γίνεται το αντίστοιχο broadcast) και μετά για το αν μπορεί να μπει νέο παιδί. Παραθέτουμε το αντίστοιχο κομμάτι του κώδικα:

```
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);
    ++(thr->kg->space);

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

    if ( (t - 1) * r >= c)
        pthread_cond_broadcast(&thr->kg->teacher_out);

    if (thr->kg->space > 0)
        pthread_cond_broadcast(&thr->kg->child_in);

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

Το ίδιο συμβαίνει και όταν μπαίνει κάποιος καθηγητής. Επομένως, **όταν κάποιος καθηγητής περιμένει να βγει έχει προτεραιότητα έναντι των παιδιών που περιμένουν να βγουν.**

2. Θα δείξουμε ότι μπορεί να υπάρξει race condition με το παρακάτω παράδειγμα:

Έστω ότι είναι άδαιο το νηπιαγωγείο και θέλουν ένα παιδί και ένας καθηγητής να μπουν, όμως ο κώδικας είναι λανθασμένος και έτσι μπαίνει πρώτο το παιδί. Ταυτόχρονα, περιμένει στο lock ο καθηγητής για να μπει με την πρώτη ευκαιρία. Με το που βγαίνει το νήμα του παιδιού από το κρίσιμο τμήμα, συνεχίζει και προσπαθεί να μπει στο κρίσιμο τμήμα του verify. Αφού έγινε unlock, ο καθηγητής ενδεχομένως προλαβαίνει να μπει στο δικό του κρίσιμο τμήμα και έτσι αλλάζει την αναλογία παιδιών-καθηγητών πριν προλάβει το νήμα του παιδιού να κάνει verify. Έτσι, το verify του παιδιού βλέπει την αναλογία ως σωστή και το πρόγραμμα συνεχίζεται κανονικά, παρόλο που θα έπρεπε να βγάλει σφάλμα.

```
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);
    }
}
```