יבש 3 – מערכות הפעלה

חלק א

1. נתאר את השתלשלות האירועים לקבלת פגיעה בתכונות:

נניח יש לנו 2 חוטים. הראשון נועל עם lock את הקטע הקריטי ולפתע הוא נופל באמצע הביצוע. לאחר מכן מגיע חוט אחר ל()lock ומנסה לנעול את המנעול – כאשר הוא בעצם עושה בדיקה עם AtomicSwap(%lockval) אבל lockval עדיין 0 כי החוט הראשון לא עדכן אותו כי הוא קרס ולכן החוט השני יתקע בלולאה ולא יתקדם לעולם.

לכן נפגעת ההתקדמות. מאותה סיבה גם נפגעת ההוגנות כי חוט 2 אף פעם לא יתבצע למרות שמגיע לו.

2. הבעיה שתפגע משמעותית ביעילות זמן המעבד הוא כאשר חוט כלשהו ינעל את המנעול, יגיע החוט ה-2 ואז מכיוון ש lockVal==0 הוא יסתובב בלולאה על הwhile כל הזמן ויבזבז את כל הקוואנטום המוגדר עבורו, למרות שהיה עדיף שיצא לתור המתנה וייתן לתהליכים אחרים לרוץ על המעבד.

2. התכונות שיפגעו הפעם הינן:

MUTEX נפגע – נניח שיש כרגע חוט אחד בקטע הקריטי. מגיעים MAXITER ניסיונות לנעול את המנעול ואז מה שיקרה הוא שהמנעול יפתח כאשר עדיין יש חוט אחד בפנים והחוט הראשון בתור ייכנס לקטע הקוד הקריטי גם הוא ולכן mutual exclusion יפגע.

3. מקסימום = 55 – כאשר נוצר חוט חדש באמצעות pthread_Create אשר מקבל את הפונקציה workload, הוא מקבל מחסנית משהו וקונטקסט משלו ולכן לכל חוט יהיה עותק אחר של workload. כאשר כל החוטים מסיימים את הביצוע של הפונציה שלהם ומסיימים, בכל פעם מוסיפים את הID של החוט מ1 ועד 10. לכן הסכום שיתקבל הינו 55 ואז תקרה ההדפסה של האב כי הוא מחכה רק לאחד מהם שיסיים אבל בגלל החלפות הקשר ייתכן מצב שכל החוטים יסיימו, הוא יחכה לאחרון, כולם הסתיימו ולכן האב יכול להמשיך ואז ידפיס את הסכום 55.

מינימום = 10 – נשים לב שהאב מחכה בjoin לאחרון שנוצר – מוצבע ע"י המשתנה t. לכן בהנחה שאחרי יצירת כל החוטים הראשון אשר יסיימו את ריצתו הוא העשירי אזי בסכום יהיה 10 (מובטח מהאטומיות של ההוספה), האב יתחיל לעבוד ישר אחריו, יעבור את הjoin (כי החוט הסתיים) וידפיס 10. אם כל אחד מהאחרים יסתיים לפני העשירי, האב עדיין יחכה לעשירי להסתיים ואז הסכום רק יילך ויגדל ולכן זה לא יהיה המינימום.

4.מקסימלי = 200 – זה המקרה שהאב יוצר את שני החוטים, שניהם רצים ללא הפרעה וסוכמים כל אחד 100 ואז סך הכל בתוצאה נקבל 200 והאב ימשיך לרוץ אחרי סיומם וידפיס 200. נעיר כי האב מבצע join עבור שניהם בנפרד ולכן הוא יחכה ששניהם יסתיימו ולא יוכל להמשיך אחרת.

מינימאלי = 100 – מקרה זה יתקבל אם החוט הראשון יתחיל לעבוד ראשון ויטען את הערך 0 לresult, כלומר בביצוע $result + 1$ הערך שיש לו בresult (מצד ימין) הוא 0. לאחר מכן החלפת הקשר לחוט 2 שהוא רץ במלואו ומשנה את הערך הגלובלי result ל-100 . לאחר מכן חוזרים לחוט הראשון והוא מכניס 0+1 לresult ובעצם מתחיל את הספירה מחדש שבסופה יהיה 100 בresult. האב יראה ששניהם הסתיימו וימשיך גם וידפיס 100.

5. אין צורך להגן על sum שכן בביצוע fork מבצעים קריאת מערכת sys_clone אשר מעתיקה את כל התהליך כולו לתהליך הבן כולל את הזכרון. לכן sum יגדל ב-5 ע"י הבן וגם ב-1 ע"י האב אבל כל אחד הוא בעצם עותק של sum והבן והאב יראו שני עותקים ולכן אין צורך להגן כי אין להם מרחב זכרון משותף.

חלק ב

```
typedef struct mutex {
    singlephore h;
} mutex;
void mutex_init (mutex * m) {
    singlephore_init(&m->h);
}
void mutex_lock (mutex * m) {
    H(&m->h, 0, -1);
}
void mutex_unlock (mutex * m) {
    H(&m->h, -1, 1);
}
```

.1

.2

```
typedef
struct
condvar
{
            mutex m;
            singlephore h;
            unsigned int count;
            unsigned int value;
        } condvar;
        // Initilize the condition variable
        void cond_init(condvar* c) {
            pthread_mutex_init(&(c->m));
            singlephore_init(&(c->h));
            count = 0;
            value = 0;
        }
        // Signal the condition variable
        void cond_signal(condvar* c) {
            c->m->lock();
            if (count - value > 0) {
                H(&(c->h), MIN_INT, 1)
                value++;
            }
            c->m->unlock();
        }
        // Block until the condition variable is signaled. The mutex m must be locked by
        the
        // current thread. It is unlocked before the wait begins and re-locked after the
        wait
        // ends. There are no sleep-wakeup race conditions: if thread 1 has m locked and
        // executes cond_wait(c,m), no other thread is waiting on c, and thread 2
        executes
        // mutex_lock(m); cond_signal(c); mutex_unlock(m), then thread 1 will always
        recieve the
        // signal (i.e., wake up).
        void cond_wait(condvar* c, mutex* m) {
            c->m->lock();
            count++;
            c->m->unlock();
            m->unlock();
            H(&c->h, 1, -1);
            m->lock();
            c->m->lock();
```

3. נדגים את אי נכונות המימוש בעזרת 3 חוטים והמימוש של תור שראינו בתרגול (ידוע לנו שהשימוש במנעולים ומשתני תנאי שם חוקי ולכן אם המימוש של המשתנה תנאי נכון אז המימוש של התור המאובטח גם הינו נכון).

T_1 מתחיל בכך שהוא עושה *dequee* כלומר הוא מוריד את ה*value* של *singlephore(SP)* ל-1. לאחר מכן הוא חוזר מהקוד של ה*cond_wait* ובודק את הלולאה ורואה כי ה*size==0* עדיין ולכן הוא מבצע שוב *con_wait*. הפעם הוא עושה *yield* כי $-1 < 0$.

מגיע T_2, מכניס איבר ושולח סיגנאל = מגדיל את *value* להיות 0 , מעלה את הגודל *size=1* ומסתיים.

T_3 עושה *deque*, רואה כי ה*size* אינו 0 ולכן הוא מצליח להוציא איבר ולהקטין את הגודל *size* להיות 0 ואז הוא מסתיים.

T_1 חוזר לעבוד ואז בודק את הלולאה שוב ושוב רואה *size=0* כלומר עושה *wait* שוב.

מסקנה: נשלח סיגנאל ע"י T_2 להעיר חוט אשר ישן אך בפועל אף אחד לא קולט את הסיגנאל הזה כאשר בפועל יש את T_1 אשר מחכה לאות אשר ישחרר אותו. כלומר קיבלנו מצב של *lost signal*.

הערה: כאשר אנו אומרים שחוט מתחיל זה כי קרתה החלפת הקשר אליו.

חלק ג

1) פתרון זה לא יעבוד כיוון שנניח והגיעו n-1 חוטים לbarrier.wait. כיוון שהsemaphore התחיל מ-0 אז כל wait מקטין ב-1 את המונה בsemaphore ונגיע שיש בו $-(n-1)$. כלומר n-1 אשר מחכים לקבל סיגנאל להמשיך. לאחר מכן מגיע החוט ה-n, רואה כי count==n ושולח סיגנאל. כלומר מגדיל ב-1 את המונה בsemaphore ומשחרר אחד מהם לריצה. הבעיה היא שבהמשך אין שליחת סיגנאל נוספת לשחרור שאר הממתינים, ה-n-2, ולכן הם יתקעו שם לנצח.

2) הבעיה תתרחש כבר מהחוט הראשון שיבצע את הbarrier.

החוט הראשון מגיע, נועל את קטע הקוד הקריטי עם מנעול, מגדיל את count, לא שולח סיגנאל ונכנס להמתנה כי הוא הוריד ל1- את הsemaphore כי הוא התחיל ב-0. הבעיה היא שהוא לא שחרר את המנעול של mutex שהוא סגר ולכן אף חוט אחר לא יוכל להיכנס לקטע הקוד כי המנעול סגור.

3) נדגים ע"י 2 חוטים כיצד הbarrier אינו חוזר למצבו ההתחלתי אחרי סדרת פעולות הבאה:

1. חוט 1 – מגדיל את count ל1 ונתקע על הwait כלומר יש 1- בvalue של הsemaphore.
2. חוט 2 – מגדיל ל-2 את count, שולח סיגנאל וערך הsemaphore הינו 0 והגיע לשורה 9 כאשר מתבצעת החלפת הקשר.
3. חוט 1 עובר את המחסום ושולח סיגנאל ומגדיל את הערך של הsemaphore ל1. עובר את הקטע הקריטי, מקטין את count ל-1, עובר את הנעילה ואז החלפת הקשר בשורה אחרונה.
4. חוט 2 יכול לעבור את המחסום ומשאיר (אחרי הקטנה והגדלה) את ערך הsemaphore ב-1. גם הוא עובר את הקטע הקריטי ומקטין את count להיות 0. הוא רואה שיש 0 בcount ומבצע wait כלומר ערך הsemaphore הינו 0 והוא מסתיים כלומר הוא יכל לעבור את המחסום.
5. חוט 1 ממשיך משורה 18, רואה שיש 0 בcount, מבצע wait וערך הsemaphore הינו 1- כעת. כלומר חוט זה נתקע ואין מי שישחרר אותו. כלומר המערכת לא אותחלה למצבה ההתחלתי כמו שצריך להיות.

4) גם פה תהיה בעיה שחוטים יהיו בקטע הקריטי. למשל עם שני חוטים.

1. חוט 1 וחוט 2 מגיעים שניהם ועוברים בהצלחה את הכל עד הקטע והקריטי ונכנסים אליו (הוסבר בסעיף קודם שאין בעיה עם שורות 11-1). Count=2 וערך הsemaphore הוא 0.
2. חוט 1 מסיים את הקטע הקריטי, מקטין את count ל-1 ויוצא מהמנעול. חוט 2 עדיין נמצא בקטע הקריטי כי הוא קיבל החלפת הקשר כאשר היה בתוכו.
3. נניח שיש barrier בהמשך של הקוד של חוט 1 (ו2). הוא מגיע למנעול, מעלה את count ל-2ו שולח סיגנאל כלומר העלה את ערך הsemaphore ל-1. הוא לא ממתין בwait אלא מקטין את ערך הsemaphore ל-0, שולח סיגנאל (לא רלוונטי כרגע) ונכנס לקטע הקריטי.

קיבלנו ששני החוטים נמצאים באותו קטע קוד הקריטי שזה פוגע בmutual exclusion שרצינו.

5)

```
#include "Barrier.h"

Barrier::Barrier(unsigned int num_of_threads) : num_of_threads(num_of_threads), counter(0) {
    // if semaphore is initialized with 0, it will hold everyone from going in
    sem_init(&sem, 0, 0);
    sem_init(&sem2, 0, 0);

    // acts as a mutex
    sem_init(&mutex, 0, 1);
}

void Barrier::wait() {
    // locking before critical section
    sem_wait(&mutex);

    // count how many threads are waiting to pass
    counter++;

    // if we reached the maximum number of threads, start releasing signals
    if (counter == num_of_threads) {
        for (unsigned int i = 0; i < num_of_threads; i++) {
            sem_post(&sem);
        }
    }
```

```
    // unlock
    sem_post(&mutex);

    // this semaphore now have 'num_of_threads' threads passing
    // so they lock this section and start decreasing the counter
    sem_wait(&sem);
    sem_wait(&mutex);
    counter--;

    // if all of the threads got through, let them pass the second semaphore
    if (counter == 0) {
        for (unsigned int i = 0; i < num_of_threads; i++) {
            sem_post(&sem2);
        }
    }

    // unlock and salamat
    sem_post(&mutex);
    sem_wait(&sem2);
}
```