**חלק יבש**

**שאלה 1.**

1. ערך מינמלי 100 – במקרה ששני החוטים ירוצו ותיהיה תמיד החלפת הקשר אחרי שחוט מסויים קרא את הערך ב-result, ולפני שהציב את התוצאה (של result+1) חזרה ב-result. כך ריצות החוטים לא ישפיעו אחת על השנייה, ונקבל תוצאה במתקבלת מהרצה של חוט יחיד.

ערך מקסימלי 200 – במקרה ששני החוטים ירוצו ותיהיה תמיד החלפת הקשר לאחר ההצבה ב-result, ולפני הקריאה של result באיטרציה הבאה. כך כל חוט יקרא את הערך שהחוט השני עידכן בדיוק לפני החלפת ההקשר, ונקבל סכום מצטבר לסירוגין משתי ההרצות, כלומר 200.

# הערה: הכל בהנחה שהשמת ערך לתוך משתנה בזכרון היא פעולה אטומית. אחרת תיתכן כמובן החלפת הקשר במהלך עדכון המשתנה ואז התהליך השני ייקרא זבל (ואז אין לדעת מה התוצאה).

1. נשנה את התוכנית כך:

Int result;

Int main(){

Pthread\_mutex\_t res\_mutex;

Pthread\_t\_threads[2];

Int i;

Result = 0;

Pthread\_mutex\_init(&res\_mutex,NULL);

For(i=0; i<2;i++)

Pthread\_create(&threads[i],NULL,do\_calc,NULL);

For(i=0;i<2;i++)

Pthread\_join(threads[i],NULL);

Pthread\_mutex\_destroy(&res\_mutex);

Printf("%d\n",result);

Return 0;

}

Void\* do\_calc(){

Int I;

For ( i=0 ; i < 100: i++){

Pthread\_mutex\_lock(&res\_mutex);

Result = result +1;

Pthread\_mutex\_unlock(&res\_mutex);

}

}

וכך נקבל תוצאה 200 תמיד. כיוון שבקטע הקריטי של קריאת הערך והשמה אליו מובטח שהתהליך האחר לא יוכל לשנות או לקרוא ערך ביניים.

**שאלה 2**

**1)**חוטים משתפים את הזיכרון, המשתנים הגלובליים והסטטים. מה שלא משותף הוא המחסנית ,הרגיסטרים ומבנה הנתונים שמתארים אותם, על מנת לאפשר להם לרוץ בנפרד.

**2)**ההבדל ביו חוטים ותהליכים: דבר רישון חוט הוא סוג של תהליך אבל זמן ה"יצירה" שלו יותר מהר,

חותים משתפים הרבה דברים ביניהם, חוטים משתמשים באזור זיכרון משותף לתקשורת ביניהם(pipes).

**3)**multi-threading useless :

למשל אם רוצים לבצעה חישוב פשוט של ביטוי אלגברי כך שאנחנו חייבים להמתין ל תום של חישוב בשביל

להשתמש בתוצאה להמשך החישוב.

multi-threading can be effective:

לדוגמה אם נרצה לחשב מספר פיבונצי, חוט אחד מחשב F(n-1) ו שני F(n-2) ואז מחזירים סכום שלהם.

הפונקציה תהיה בארך פי2 יותר מהירה מי פונקציה ללא חוטים.

**4)**

**1:1**

Creates a kernel thread or task for each application thread, every user thread is controlled by kernel scheduler.

**+**

1. Simplest to implement.

2. Threads can run on number of processors

**-**

Each thread created by user becomes more expensive of kernel resources.

**M:1**

All threads in user space and appears to the kernel as single-threaded process.

**+**

This style is the most portable, in that it does not require any special features from the underlying kernel. Can to create many treads.

**-**

That it requires that all blocking system calls, this style also does not allow the application to take advantage of any multiprocessor scheduling. Since to the kernel scheduler it is still a single-threaded process.

**M:N**

M kernel threads for N user threads.

**+**

Thread are not blocked by sys calls, scheduler can switch between threads more faster, cheap

Creation and execution.

**-**

Very difficult to implementation, maintaining.

**(5**

1. We don’t use other processors for our needs (current application), and we waste our resources

(processors).We can create more threads for do it better.

1. Its good example for good usage of threads model, because we have a potential to run all threads on the all processors.
2. All allocated threads can’t run concurrency and our performance is worse than example 2,we must wait for free processor and time of all application increase . Maybe if we don’t create the

threads its will be take less time .

**Question 3**

1. Process with pid 16742 it’s a process that run the main. Process with pid 16743 it’s a group leader of threads, this process create in first time when we call the pthreads create. And this process become the group leader with gpid=pid of the thread. After that we create two thread with functions clone and in the reason of that they have ppid for group leader witch 16743.And they get the pid 16744 and 16745 in the order.
2. • if the manager thread is killed before the other threads, the rest of the process has to be manually cleaned up.

• The signal system is severely broken. It does not conform with the behavior

POSIX specifies. Sending a signal to the process as a whole is not implemented.

• since it is 1:1 implementation it uses signals to implement synchronization, thing that cause problems: high latency , more complicated signal handling system and spurious wakeups.

•incorrect implementation of SIGSTOP and SIGCONT signals, which cause the user to not be able to stop multithreaded process, but to stop only one thread.

•different PID for each thread cause compatibility issues with other POSIX implementations, which assume one ID for all threads of a process.

• On IA-32 the limit on the number of threads (8192, minus one for the manager) is

proving to be a problem for some people. Although threads are often misused in

such situations, the offending applications are known to work on other platforms.

• Processes with hundreds or thousands of threads make the /proc file system

barely usable. Each thread shows up as a separate process.

• The problems with the signal implementation are mainly due to missing kernel

support. Special signals like SIGSTOP would have to be handled by the kernel

and for all threads.

• The misuse of signals to implement synchronization primitives adds even more

to the problems. Delivering signals is a very heavy-handed approach to ensure

synchronization.

1. \* \*\*\*

* NPTL, as opposed to LinuxThreads, doesn’t not use manager thread. Some of the jobs of the manager thread are now done by the kernel: sending fatal signal to all threads of the process,deallocate the memory used by each thread stack, manages termination of all threads by waiting on them before clearing the parent thread.
* Since NPTL doesn’t use manager thread it has better synchronization and scalability mechanism.
* NPTL introduce a new mechanism called futex. A futex work on shared memory region and hence can be shared between processes. That way it avoids synchronization of threads by mean of signals.
* NPTL is more POSIX compliant – it handles signals per-process basis. Getpid() will return same ID for all threads of a process, and if for example SIGSTOP is sent – the whole process will stop (and not only the thread as in LinuxThreads).
* Since all threads have one parent process in NPTL, the resource usages reported to the parent are reported for the entire process (not for just one thread).
* An important feature that was introduced by NPTL was support for ABI, which helps backward compatibility with LinuxThreads. This can be done with the help of LD\_ASSUME\_KERNEL.