מערכות הפעלה - הרצאה 6

שרון מלטר, אתגר 17

9 ביולי 2024

1 Chapter 3 Cont.

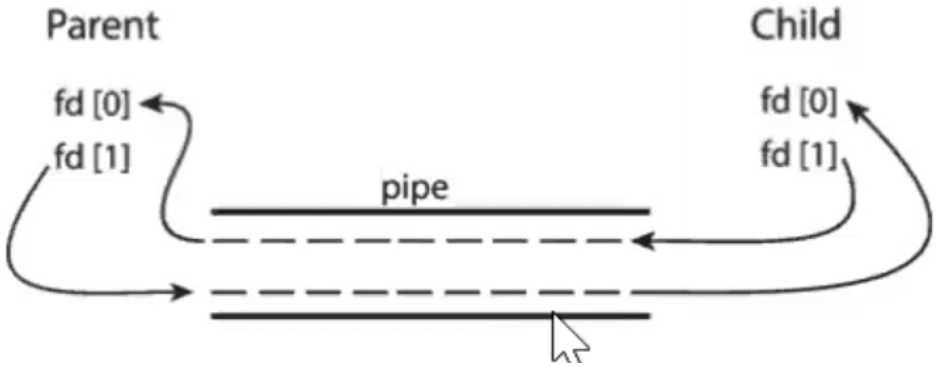
נלמד על דרך נוספת להעביר מסרים בין שתי תהליכים, צינורות ($pipes$). בעיות שיש לפתור:

- האם התקשורת היא חד־כיוונית היא דו־כיוונית?
- במקרה של תקשורת דו־כיוונית, האם היא לחלוטין או חצי $duplex$? (כלומר האם ניתן להעביר נתונים באותו הזמן באותו הצינור)
- האם חייבת להיות מערכת יחסית (כלומר הורה־ילד) בין התהליכים המתקשרים?
- האם הצינורות יכולים להישמש ברשת?

ישנם שני סוגים של צינורות. $Ordinary\ pipes$ - לא ניתן להיכנס אליהם מחוץ לתהליך שיצר אותם. לרוב, תהליך הורה יוצר צינור ומשתמש בו בשביל לתקשר עם תהליך ילד שהוא יצר.
$Named\ pipes$ - ניתן לגשת אליהם גם ללא יחסי הורה־ילד.

1.0.1 Ordinary Pipes

צינורות רגילים מאפשרים תקשורת בצורה סטנדרטית של יצרן־צרכן. היצרן כותב לצד אחד של הצינור (ה־ $write-end$ של הצינור), הצרכן קורא מהצד השני (ה־ $read-end$ של הצינור) לכן צינורות רגילים הם חדי־כיוון וזקוקים למערכת הורה־ילד בין התהליכים (על מנת לקבוע מי היצרן ומי הצרכן) במערכת ההפעלה ווינדוס מכנים צינורות אלה $anonymous\ pipes$.



1.0.2 Named Pipes

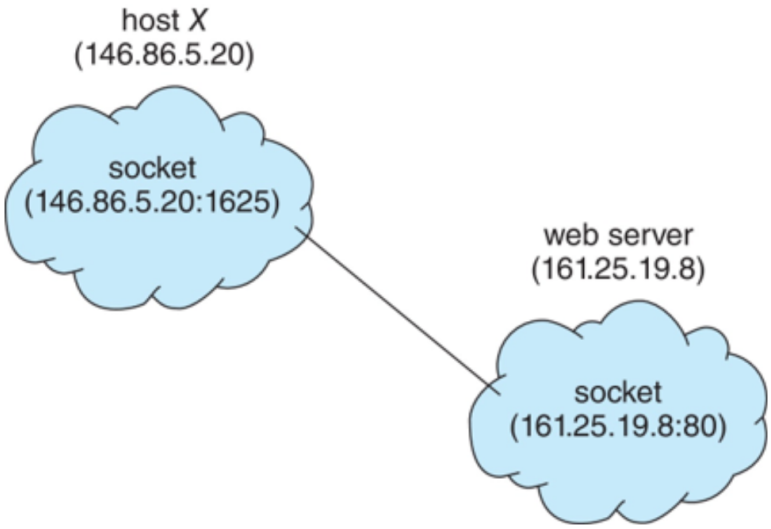
לצינורות אלה יותר יכולות, מכיוון שהתקשורת הינה דו־כיוונית ואין צורך ביחסי הורה־ילד. כמו כן מספר תהליכים יכולים להשתמש בצינור לתקשורת. סוג זה של צינור מסופק במערכות $Unix$ ו־ווינדוס.
כעת נעבור לתקשורת במערכות $client-server$.

1.1 תקשורת במערכות שרת־לקוח

ישנן שתי שיטות תקשורת במערכות אלה, *Sockets* ו־ *Remote Procedure Calls*.

1.1.1 *Sockets*

Socket מוגדר כנקודת קצה לתקשורת. זהו שרשור של כתובת *IP* ונמל־ מספר הכלול בתחילת המסר כדי להפריד בין שירותי הרשת למארח. לדוגמה, השקע 161.25.19.8 : 1625 מתייחס לנמל 1625 במארח 161.25.19.8. עם זאת, כל הנמלים שמספרם מתחת ל־ 1024 הם *well known*, כלומר שייכים ל־ *services* סטנדרטים. כמו כן ישנה כתובת *IP* מיוחדת (*loopback*) 127.0.01 המרפררת למערכת שבה התהליך רץ. ניתן לתקשר בין כל זוג שקעים, להלן דוגמה;



- $Connection - oriented\ (TCP)$
- $Connectionless\ (UDP)$
- $Multicastsockets$ ־ דאטה יכולה להישלח למספר נמענים.

להלן דוגמה של מימוש שקע של שרת השולח תאריך ב־ $JAVA$;

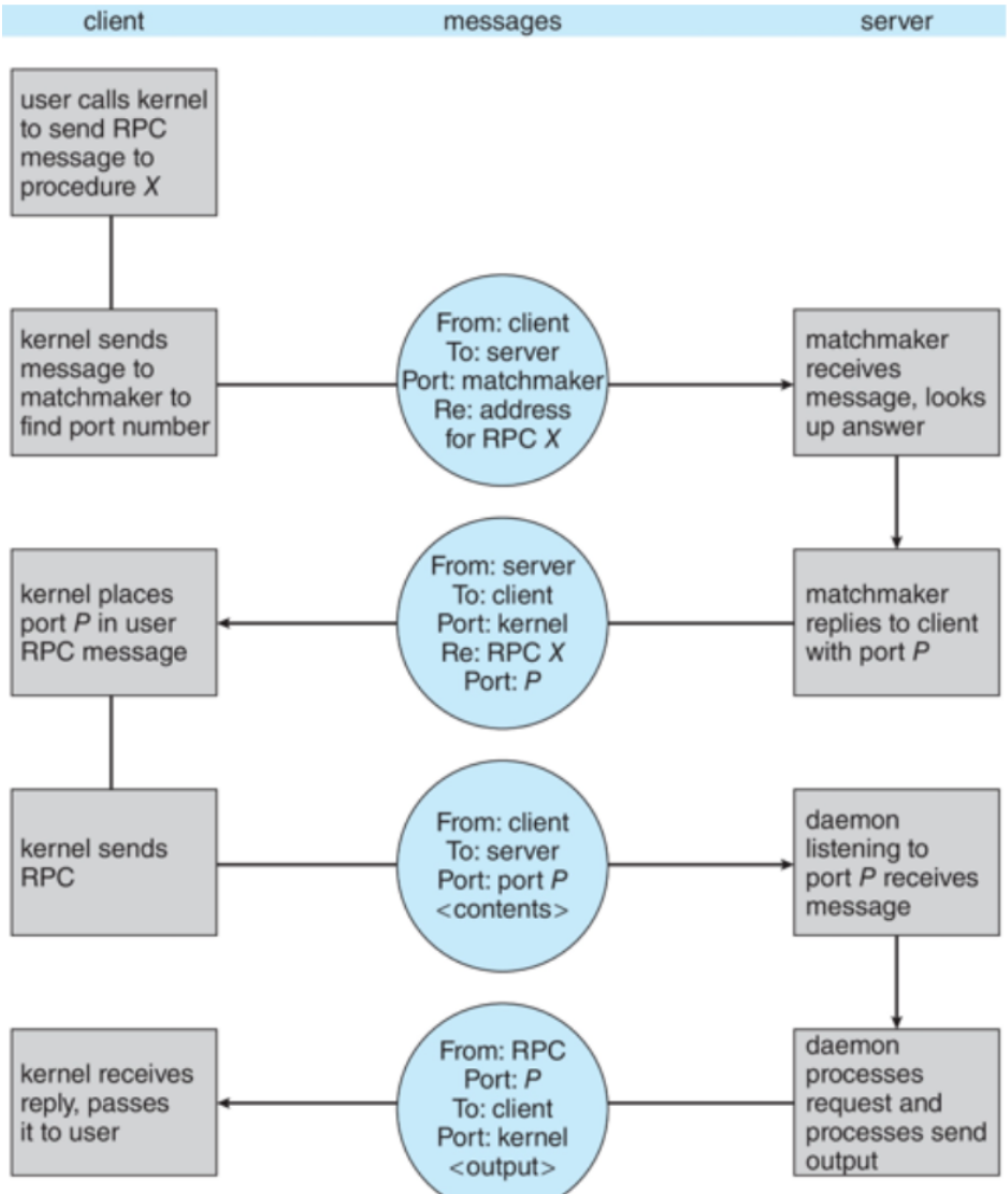
```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         /* now listen for connections */
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            /* write the Date to the socket */
            pout.println(new java.util.Date().toString());

            /* close the socket and resume */
            /* listening for connections */
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

ומימוש השקע הנמען;

```
import java.net.*;
import java.io.*;

public class DateClient
{
   public static void main(String[] args) {
      try {
         /* make connection to server socket */
         Socket sock = new Socket("127.0.0.1",6013);

         InputStream in = sock.getInputStream();
         BufferedReader bin = new
            BufferedReader(new InputStreamReader(in));

         /* read the date from the socket */
         String line;
         while ( (line = bin.readLine()) != null)
            System.out.println(line);

         /* close the socket connection*/
         sock.close();
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

1.1.2 *Remote Procedure Calls*

דוגמה ל־ RPC; בערוצים בהם ישנו פרק חדש מסדרה פעם בשבוע, יוזרים המבקשים יוכלו לקבל התרעה לאחר כל פרק חדש.
להלן דוגמה לביצוע של RPC; הסבר: תחילה היוזר מבקש מהקרנל לשלוח RPC לפרוצדורה X. הקרנל שולח



מסר ל־ $matchmaker$ (לנמל שלו) כדי שיימצא את כתובת ה־ $RPC\ X$. $server$ מחזיר ל־ $client$ כתשובה את הנמל P הקרנל שולח RPC כך שה־ $client$ יכול לשלוח נתונים ל־ $server$. לבסוף, $daemon\ processes$ מוציאים פלט, ה־ RPC מעביר את הפלט לנמל P
הסבר: לאחר שהיוזר יוצר בקשה, הקרנל פונה ל־ $matchmaker$, הוא תהליך שיודע למצוא את ה־ RPC הנכון אשר יוכל לענות על הבקשה. הוא יישלח לנמל של הקרנל את הנמל שאליו צריכים לגשת לשם כך (נמל זה יכול להיות אחד שהוחלט עליו באותו הרגע)
סיימנו את פרק 3!

Outline **2.1**

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

Objectives **2.2**

- Identify the basic components of a thread, and contrast threads and processes
- Describe the benefits and challenges of designng multithreaded applications
- Illustrate different approaches to implicit threading including thread pools, fork-join, and Grand Central Dispatch
- Describe how the Windows and Linux operating systems represent threads
- Designing multithreaded applications using the Pthreads, Java, and Windows threading APIs

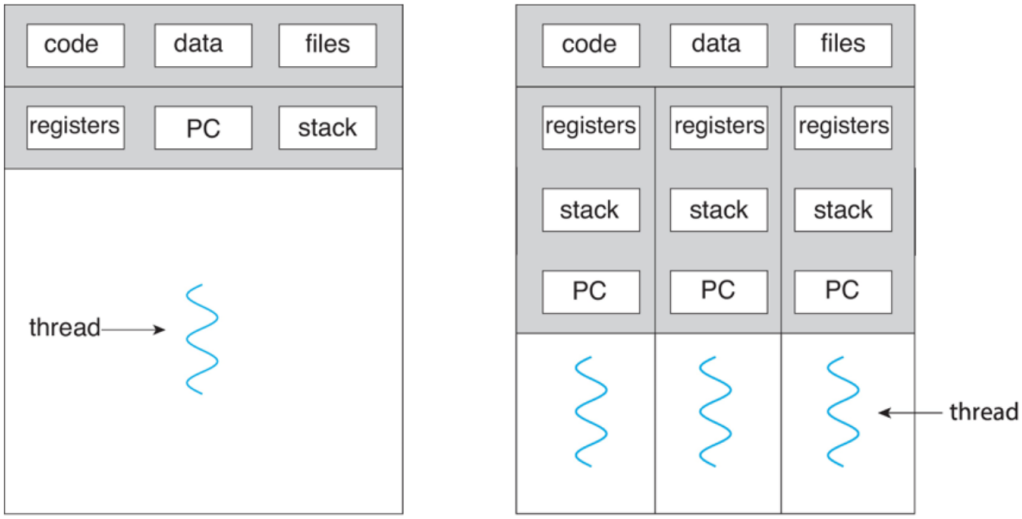
2.3 מוטיבציה לנושא

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

נעבור לחוטים עצמם.

2.4 מהו חוט

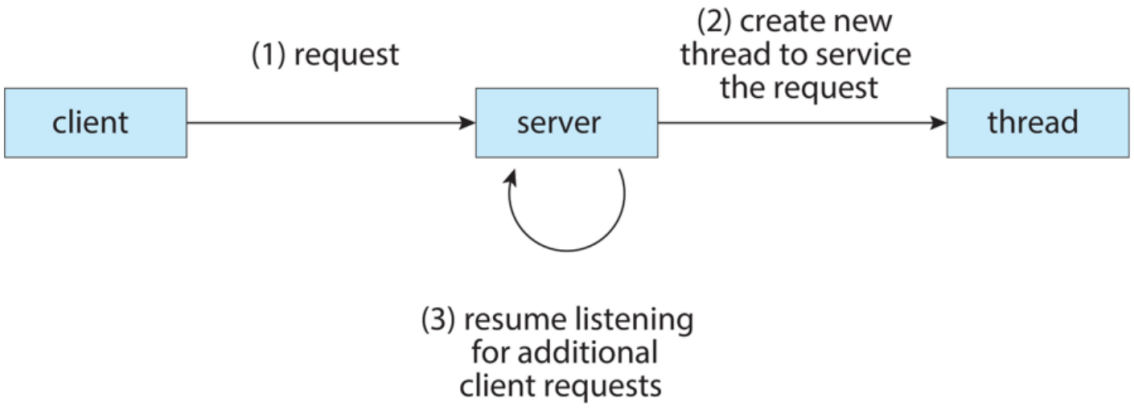
כל תהליך מורכב לפחות מחוט אחד. החוט הראשון של תהליך הוא החוט הראשי, ותהליך שזהו החוט היחיד שלו נקרא $single-threaded\ process$. להלן תרשימים המייצגים תהליך בעל חוט יחיד ותהליך בעל מספר חוטים;



single-threaded process multithreaded process

שימי לב שחוטים חולקים את הקוד של התהליך, את הקבצים שלו ואת ה־ $data$ (היכן ששמורים משתנים גלובאליים) אך יש להם רגיסטרים, $stack$ (שבהם שמורים משתנים לוקאליים) ו־ $program\ counter$ משלהם.

להלן ארכיטקטורה של שרת סעל מספר חוקים; בכל פעם שישנה בקשה, נוצר חוט חדש מהשרת אשר מטפל



בה.
יתרונות של גישה זו:

- *Responsiveness* - ניתן לאפשר הרצה מתמשכת אם חלק מהתהליך חסום, חשוב במיוחד עבור ממשק משתמש.
- *Resource Sharing* - חוטים חולקים משאבים של תהליך, מה שיותר נוח לנהל מאשר זיכרון משותף או העברת מסרים.
- *Economy* - יצירת חוט יותר זולה מיצירת תהליך וכך גם החלפת חוט זולה מ- *context switch*
- *Scalability* - תהליך יכול לנצל ארכיטקטורה מרובת ליבות.

נעמיק ביתרון האחרון שצוין, ונלמד כיצד ניתן לנצל באופן יעיל מספר ליבות.

2.5 *Multicore Programming*

מערכות *Multicore* או *Multiprocessor* מסבכות מעט יותר את המתכנתים, בין היתר האתגרים שהן מציבות הם:

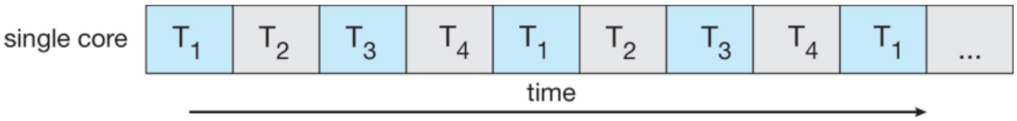
- פירוק *activities*
- איזון
- פירוק דאטה
- תלות בין נתונים
- בדיקה ודיבוג

כדי להבין איך להתעמת עם אתגרים אלה, יש עוד שני מושגים שנצטרך להכיר;

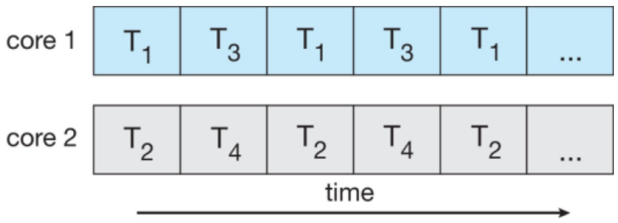
- *Parallelism* - מרמז שמערכת יכולה לבצע יותר ממשימה אחת באותו הזמן.
  ישנם שני סוגים של מקביליות:
  - *Data parrallelism* - חלוקת חלקי הדאטה בין מספר ליבות, עם אותה הפעולה על כולם.
  - *Task parrallelism* - חלוקת החוטים בין הליבות השונות, כך שכל חוט מבצע פעולה מיוחדת.
- *Concurrency* - מרמז שמערכת תומכת בכך שיותר ממשימה אחת תתקדם. ישנה ליבה אחת או מעבד יחיד, וה- *schduler* מאפשר *concurrency*.
  כמו בדוגמה בהרצאה הראשונה, מדובר במצב בו עורך דין עובד חצי יום עם לקוח ראשון ובחצי השני עם לקוח אחר, על מנת לתת לשני הלקוחות את התחושה שמטרתם מתקדמת.

להלן תרשים פשוט המתאר את שני המושגים;

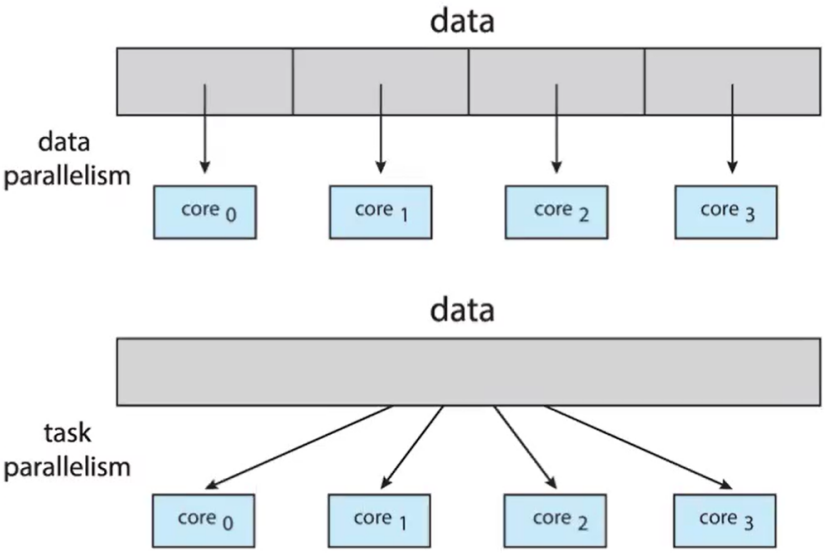
- **Concurrent execution on single-core system:**



- **Parallelism on a multi-core system:**



כמו כן, תרשים המתאר מקביליות דאטה ומקביליות משימות;



2.6 חוק אמדל

זהו חוק המסייע לנו לזהות מתי הוספת ליבה אכן מסייעת לייעל את המערכת ומתי לא.
נניח שיש לנו מערכת עם N מעבדי ליבה ו־ S הוא אחוז המשימות הסדרתיות (שחייבות להתבצע אחת אחרי השנייה), אזי הגדלת המהירות תהיה;

$$speedup \leqslant \frac{1}{S + \frac{(1-S)}{N}}$$

כלומר, אם $S = 25\%$ ואנחנו רוצים לעבור מליבה אחת לשתי ליבות, מתקיים ההאצה שנקבל היא $spedup \leqslant \frac{1}{0.25+\frac{(1-0.25)}{2}} = 1.6$

מדוע $\leqslant$? ־ מכיוון שחוק אמדל מניח שכל גישה מקבילית לזיכרון (שהיא $1-S$ מסך הגישות) נעשתה באופן אופטימלי, כך שהזמן הנדרש בשביל כולן, מאחר ש־ N מהן יכולות לקרות באותו הרגע, הוא בדיוק $\frac{1-S}{N}$. כמובן שזהו זמן אופטימלי, שלא לוקח בחשבון בדיקה ב־ $cache$, לכן ההאצה יכולה להיות קטנה יותר. נשים לב שכאשר $N \to \inf$, מתקיים $speedup \leqslant \frac{1}{S}$ כך שלא תמיד נשפר ביצועים כאשר נוסיף ליבה. כמו כן, יש השפעה גדולה

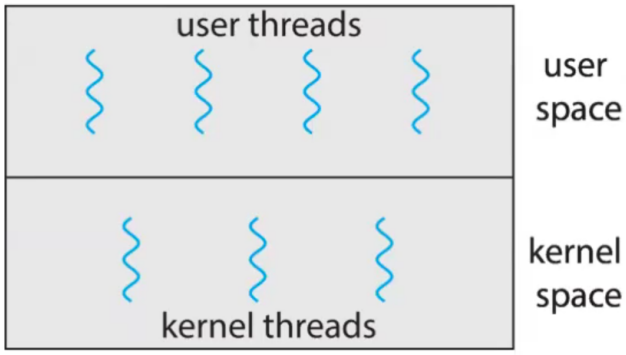
לאחוז הפעולות הסדרתיות שיש לבצע לכמות הליבות האופטימלית.
כעת נעבור לדבר על החוטים המשמשים יוזרים וחוטים המשמשים את הקרנל.

2.7 User Threads and Kernel Threads

ניהול חוטי יוזר נעשה באמצעות ספריית חוטים שברמת היוזר. ישנן שלוש ספריות עיקריות;

- POSIX Pthreads
- Windows threads
- Java threads

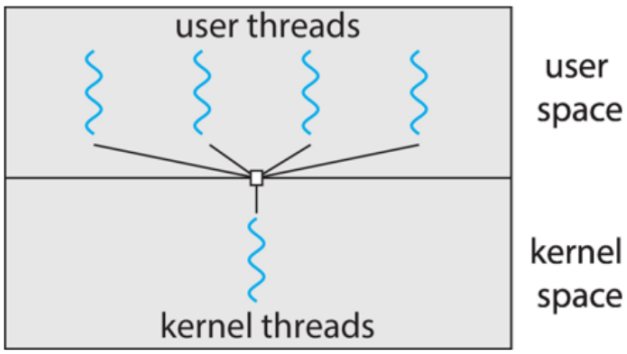
חוטי הקרנל נתמכים על ידי הקרנל :)
להלן שרטוט מאוד מסובך המתאר את מרחב חוטי היוזר ומרחב חוטי הקרנל;



ישנן מספר גישות ליחסים בין חוטי היוזר והקרנל. זכרי שהקרנל מבצע פעולות לאחר שהיוזר מבצע קריאת מערכת.

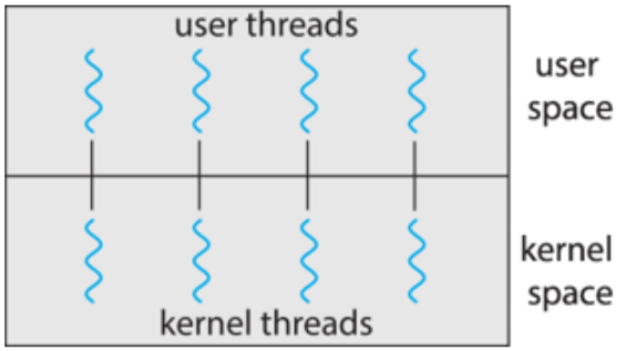
2.7.1 Many − to − One

במודל זה, חוטי יוזר רבים ממופים לחוט קרנל יחיד. כמובן שאם חוט קרנל יחיד זה חסום גורמת לעצירה של כל משימות היוזר שהיו אמורות להתבצע באותם חוטים. כמובן שמשימות הקרנל לא יכולות להתבצע במקביליות, כמו כן גם לא ניתן לקיים concurrency עבור חוטי הקרנל. . מערכות הפעלה שמשתמשות במודל־ Threads Solaris Green Threads, GNU Portable



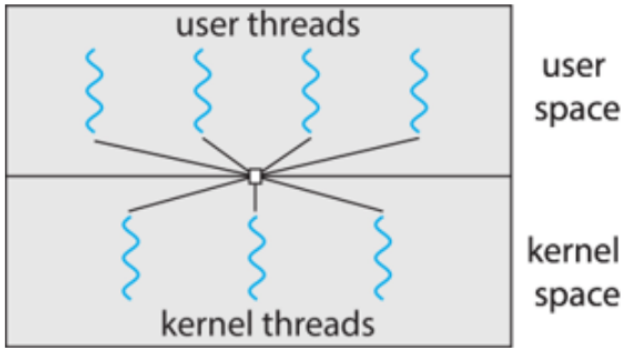
2.7.2 One − to − One

כל חוט של יוזר ממופה לחוט של קרנל. כמובן שכעת ניתן לקיים concurrency עם חוטי הקרנל. עם זאת, בכל פעם שניצור חוט ליוזר נצטרך גם ליצור חוט במרחב הקרנל, כך שעקב overhead ייתכן שמספר החוטים הניתנים לתהליך חסום. מערכות המשתמשות במודל זה הן ווינדוס ו־ Unix.



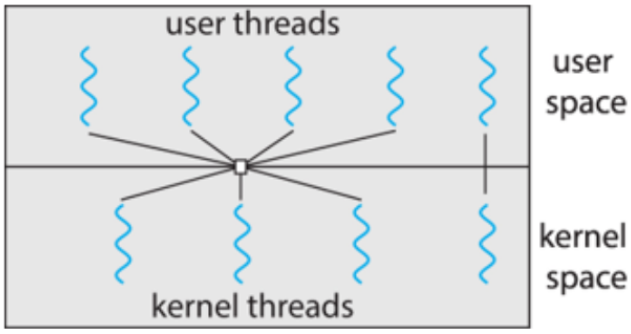
2.7.3 $Many - to - Many$

מודל זה מאפשר למספר חוטי יוזר להימפות למספר חוטי קרנל. גישה זו מאפשרת למערכת ליצור מספר מספק ויעיל של חוטי קרנל. זהו מודל לא נפוץ, מערכת אחת שמשתמשת בו היא ווינדוס עם חבילת $ThreadFiber$.



2.7.4 $Two - Level$

מודל זה דומה ל־ $M : M$, אך הוא מאפשר לחוט יוזר להיות כבול לחוט קרנל. כעת נעבור לספריות חוטים,



אשר קובעות כיצד הם מתנהלים.

2.8 $Thread\ Libraries$

ספריות חוטים מספקות למתכנת API ליצירת וניהול חוטים. ישנן שתי דרכים עיקריות למימוש ספרייה: כל הספרייה נמצאת במרחב היוזר או ספריית הקרנל נתמכת על ידי מערכת ההפעלה. נסתכל על כמה דוגמאות של ספריות.

2.8.1 *Pthreads*

ספרייה זו מספקת גם ברמת היוזר וגם ברמת הקרנל. היא הסטנדרט של *POSIX* ליצירת וסינכרון חוטים. עם זאת, מדובר בפירוט (*specification*) ולא מימוש, מתכנת הספרייה יכול להחליט בעצמו כיצד לבצע את המימוש. מודל זה נפוץ במערכות הפעלה של *Unix*.
דוגמה למימוש *Pthreads*:

```
#include <pthread.h>
#include <stdio.h>

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   /* set the default attributes of the thread */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}
```

```
/* The thread will execute in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

ולהלן לקוד *Pthreads* בו מחברים 10 חוטים:

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

להלן תכנית ב־ C של מערכת ווינדוס שתומכת ב־ *Multithreading*;

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 1; i <= Upper; i++)
      Sum += i;
   return 0;
}
```

בשיעור הבא נדבר על *implicit threads*, כלומר חוטים אשר קוראים להם באופן לא ישיר.