Operating Systems

מערכות הפעלה בוקר טוב

יום ראשון
04 דצמבר
2016

ניהול המעבד
תהליכים, חוטים,
קיפאון – Dead-Lock

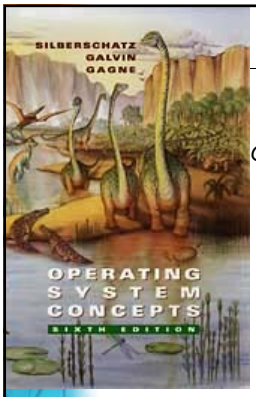
Saul Coval
Computer Systems
ד' כסלו תשע"ז

Operational Systems – Scheduler ver03

מערכות הפעלה
סיכום נושא תהליכים

COVAL Systems
saul@coval.net - http://www.coval.net

מקורות מידע



- ספר הקורס (ראשי) למצגת זו:
  Silberschatz and Galvin,
  Operating Systems Concepts (5th ed.)
  Wiley

המצגת מורכבת מעבודות של סטודנטים.
ייתכן שחלקם נלקחו מתוך אתרים פרטים.
אני רק ריכזתי את עבודות הסטודנטים ללא דרישות לזכויות כל שהם.

מקור מידע מאינטרנט

http://webcourse.technion.ac.il/236364

Popular Operating Systems

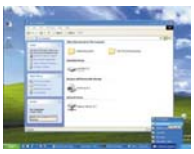
MS-DOS

UNIX

LINUX

MAC OS

WINDOWS



MS-DOS

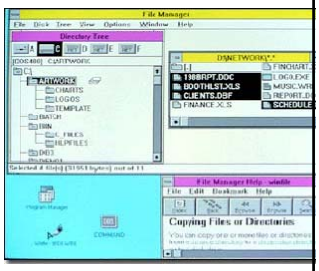
- Developed for IBM PCs in 1981
- Uses command-line interface
- Use is diminishing

MAC OS

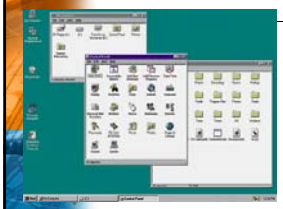
- First to use graphical user interface in 1984
- Easiest operating system for beginners

Windows 3.X

- Includes Windows 3.0, 3.1, 3.11, and Windows for Workgroups 3.1
- Not a true operating system
- Uses cooperative multitasking



Windows 95 and 98



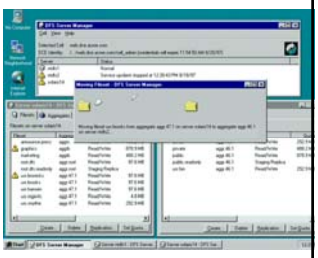
- Windows 95
  - True operating system
  - Uses preemptive multitasking
  - Downward compatible with DOS
  - Considered a transitional system
- Windows 98
  - Improved version of Windows 95
  - More stable than Windows 95

Windows CE

- System used in PDAs or palmtops
- Runs simplified versions of Windows programs
- Data can be transferred to PCs
- Includes handwriting and speech recognition

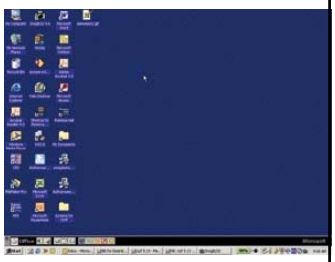
Windows NT

- Designed for client/server systems
- Two components:
  - Windows NT Workstation
  - Windows NT Server
- Oriented to business needs
- Offers security, remote administration, directory services, and server



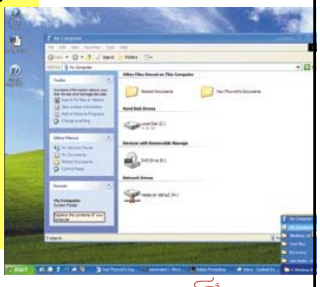
Windows 2000

- Two versions:
  - Microsoft Windows 2000 Professional
  - Microsoft Windows 2000 Server
- Better stability and more features than Windows NT

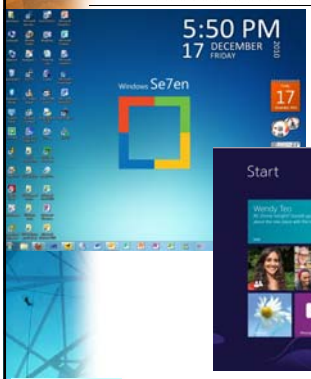


Windows XP

- Replaces all previous versions of Windows
- Three versions:
  - Windows XP Home Edition
  - Windows XP Professional
  - Windows XP Server



Windows 7 / Windows 8



מערכות הפעלה
מתקדמות של חברת
מיקרוסופט למחשבים
רגילים ומסכי מגע

Linux

- Developed in 1991 by UNIX
- Competes with Windows and MAC
- Powerful and free
- Growing fast in acceptance
- Uses Apache web server

Solaris Op System

ORACLE
(Sun- Oracle)

How Windows 9x Differs from Windows 3.x and DOS

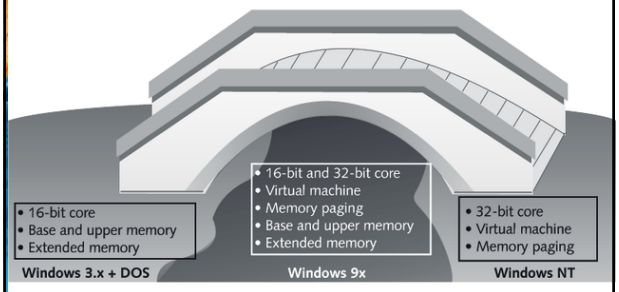
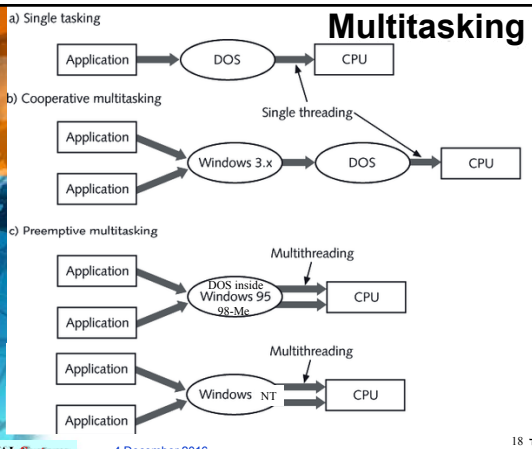


Figure 1-12 Windows 9x is the bridge from DOS to Windows NT

Comparing OSs: Terminology

- Thread
- Single-tasking
- Multitasking
  - Cooperative
  - Preemptive
- Environment
- 16-bit (real) mode and 32-bit (protected) mode
- FAT (file allocation table)
  - Tracks
  - Sectors
  - Clusters
- RAM (random access memory)

Multitasking



A Multitasking Environment

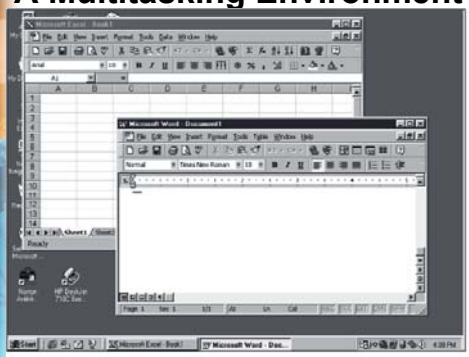
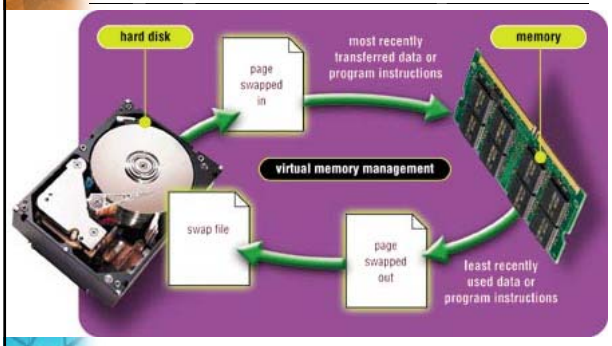


Figure 1-8 A multitasking environment allows two or more applications to run simultaneously

Managing Virtual Memory



מבוא לניהול תהליכים

מבוא לניהול תהליכים

- **תהליכים מאפשרים הפרדה נקייה בין משימות שמתבצעות "בו-זמנית"**
  - ביצוע סדרתי של פקודות
  - מצב תהליך משתנה ומשפיע על ביצוע התכנית
    - מצב כולל: קוד, רגיסטרים, זיכרון, רגיסטר בקרה, מצב קבצים פתוחים (מיקום), וכו'.
- **תהליך לעומת תוכנית**
  - תהליך היא חלק ממצב התוכנית.
  - תוכנית יכולה ליצר מספר תהליכים.

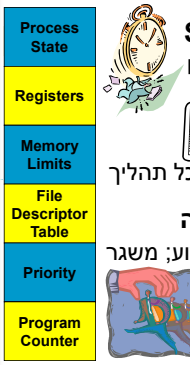
ביצוע תהליכים – דוגמא "vi myfile.txt"

- **Keyboard Device Driver**
  - אוגר ב-buffer את התווים שמתקבלים, עד להקשת "Enter"
- **Command Line Interpreter**
  - מיצר תהליך חדש (קריאה ל-fork)
- **File Manager**
  - מזהה גישה לקובץ "vi"
  - פונה דרך ה-Disk Device Driver לדיסק כדי להביא את הקובץ לזיכרון
- **Loader**
  - מאתחל את מצב התהליך, ומעביר לו ארגומנטים (מחרוזת "myfile.txt")

ביצוע תהליכים – דוגמא "vi myfile.txt" (המשך)

- **Memory Manager**
  - מקצה מקום לקובץ ומעדכן את טבלאות המיפוי של הזיכרון
- **Process Manager**
  - התהליך מסומן כמוכן לביצוע (Ready)
- **Scheduler**
  - עם הזמן, התהליך נבחר לביצוע ומקבל את המעבד
- **File Manager**
  - פותח וטוען לזיכרון את תוכן הקובץ "myfile.txt" בשיתוף עם ה-Memory Manager

מעקב אחר תהליכים



- **תזמון תהליכים Scheduller**
  - מתבצע ע"י מערכת ההפעלה במתזמן
  - מערכת ההפעלה מבטיחה הוגנות
- **טבלת תהליכים PCB**
  - מכילה Process Control Block עבור כל תהליך
- **משגר (Dispatcher) הוא לולאה**
  - מקבל שליטה; בוחר תהליך לביצוע; משגר את תהליך; מקבל שליטה; ...
  - בהמשך נראה
    - איך נבחר תהליך
    - איך תהליכים מתקשרים זה לזה

אופן פעולה כפול (UNIX)
Dual Mode Operation

- **תהליך יכול להימצא באחד משני מצבים**
  - גרעין (Kernel Mode)
    - עם זכויות יתר (Privileged Mode), למשל גישה ישירה לקלט/פלט
    - הקוד והנתונים של הגרעין נמצאים כל הזמן בזיכרון (non-pageable)
    - מתבצע בהקשר של תהליך המשתמש
  - משתמש (User Mode)
    - עם זכויות מוגבלות, למשל ללא גישה לכל הזיכרון או התקני קלט/פלט
- **מעבר ממשתמש לגרעין**
  - נעשה ע"י קריאת מערכת (System Call)

POSIX הגדרה:

- **POSIX or "Portable Operating System Interface" is the collective name of a family of related standards specified by the IEEE to define the Application Programming Interface (API) for software compatible with variants of the Unix operating system. Originally, the name stood for IEEE Std 1003.1-1988, which as the name suggests, was released in 1988. The family of POSIX standards is formally designated as IEEE 1003 and the international standard name is ISO/IEC 9945. The standards emerged from a project that began near 1985. The term *POSIX* was suggested by Richard Stallman in response to an IEEE request for a memorable name; before that the standards effort was called IEEE-IX.**

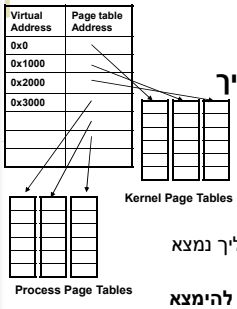
המשך: POSIX cont.

- **POSIX Threads is a POSIX standard for threads. The standard defines an API for creating and manipulating threads.**
- **Libraries implementing the POSIX Threads standard are often named Pthreads. Pthreads are most commonly used on Unix-like POSIX systems such as Linux and Solaris, but Microsoft Windows implementations also exist. For example, the pthreads-w32 is available and supports a subset of the Pthread API [1]. (Note: in text, Pthreads is written with an upper-case P.)**

POSIX

- **POSIX, ממשק מערכת הפעלה נייד מבוסס Unix, אוסף של תקנים למערכות הפעלה מבוססות יוניקס.**
- **POSIX, קיצור של POrtable System Interface. ה־ X בסוף מסמנת UNIX.**
- **סטנדרטים המתוארים של ידי ארגון IEEE להגדרת ממשק לתכנות יישומים לתוכנות תואמות למשתנים של מערכת הפעלה Unix. Application Programming Interface (API)**
- **הוא תקן של ה־ IEEE שראשיתו בסוף שנות השמונים שנועד לאפשר כתיבת תוכניות פורטביליות (ניידות) עבור מערכות דמויות יוניקס.**
- **כיום מערכת יכולה להיקרא UNIX אם היא עומדת בתקן הזה.**

אופן פעולה כפול (UNIX)
איזורי זיכרון

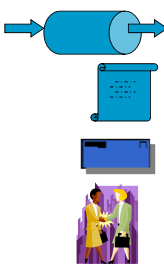


- **מרחב הכתובות של תהליך מחולק ל-**
  - User Space –
    - ניתן לגישה כל הזמן
  - Kernel Space –
    - ניתן לגישה רק כאשר התהליך נמצא ב-Kernel Mode
- **טבלאות ניהול הזיכרון לא חייבות להימצא בזיכרון כל הזמן (Memory Resident)**

תקשורת בין תהליכים

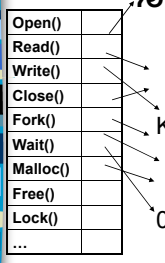
- **מטרה**
  - העברת מידע
  - תיאום

- **כיצד**
  - באמצעות pipes ו-signals (תיסקר בתרגול)
  - באמצעות זיכרון משותף (shared memory)
  - באמצעות העברת הודעות (message passing)
  - תקשורת ישירה ע"י זיהוי מפורש של התהליכים
    - Send / Receive



אופן פעולה כפול (UNIX)
קריאות מערכת (System Calls)

- **קריאות מערכת מוגדרות כספריה של פונקציות "C"**
- **כאשר יש קריאת מערכת מתבצע Operating System Trap**
  - העתקת פרמטרים ל- Kernel Space
  - שמירת מצב התהליך
  - מעבר ל- Kernel Mode
  - קריאה לפונקציה המתאימה בספריה 0



תיבות דואר - Mailboxes

- **מאפשרות תקשורת עקיפה**
  - זוג תהליכים יכול להתקשר דרך כמה תיבות-דואר

  - כמה תהליכים יכולים לגשת לאותה תיבה

  - אם כמה תהליכים מתחרים, **מי זוכה?**

בעית היצרן / צרכן
Producer/Consumer

- **תהליך היצרן מכין מידע הנצרך ע"י הצרכן**

```
producer:   repeat
                nextp = new item;
                send(consumer, nextp);
            until false;
consumer:   repeat
                receive( producer, nextp);
                consume nextp;
            until false;
```

```
P1: send(Q, id, msg);   P2: send(Q, id, msg);
                Q: receive(id, msg);
```

  - תקשורת סימטרית
    - שני התהליכים מזהים זה את זה
  - תקשורת אסימטרית
    - שליחת הודעה לתהליך ספציפי
    - קבלת הודעה מתהליך כלשהו

תיבות דואר - Mailboxes

- **תיבת-דואר בבעלות תהליך**
  - רק התהליך יכול לגשת לתיבה
  - כאשר התהליך מסתיים, התיבה נעלמת
- **תיבת-דואר בבעלות המערכת**
  - יש פקודות ליצור/להשמיד אותה, ולשלוח/לקבל הודעות
  - התהליך היוצר מקבל בעלות והרשאת קריאה
  - ניתן להעביר בעלות, להעניק זכויות קריאה/כתיבה
    - למשל בעזרת קריאות מערכת ו/או יצירת תהליך בן
  - במימושים מסוימים, כאשר מסתיימים כל התהליכים המשתמשים, מערכת ההפעלה משמידה אותה (למשל כאשר use-count=0).

חוצצים - Buffers

- **הגבלה על מספר ההודעות**
  - קיבולת אפס
    - תקשורת **סינכרונית** חוסמת עם Rendez-vous (מפגש/פגישה)
    - למשל Remote Procedure Call
  - קיבולת חסומה
    - השולח נחסם כאשר החוצץ (buffer) מלא.

תקשורת **אסינכרונית**

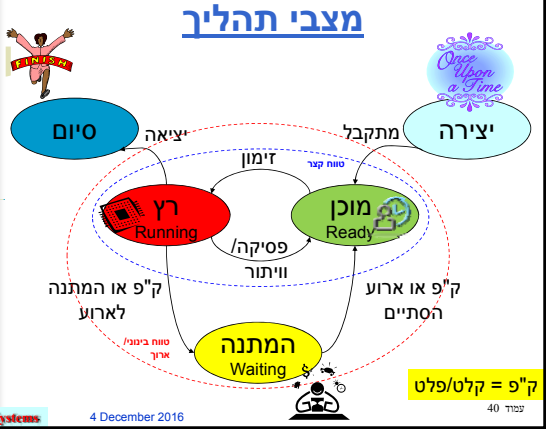
  - קיבולת אינסופית
    - השולח לא נחסם.
  - בתקשורת אסינכרונית, מתי ההודעה מגיעה מקבל?
    - אפשרויות: שליחת אישור (ack) – לפעמים כולל זיהוי המקבל .

זימון תהליכים
Process Scheduling

טיפול בחריגים

- **כאשר תהליך מסתיים, יתכן ...**
  - שתהליך אחר מחכה (או יחכה) להודעה ממנו
  - שישנן הודעות שנשלחו אליו שלא נקראו
- **הודעות אבודות**
  - למשל בגלל שחוצץ התמלא
  - המערכת או השולח יכולים לשלוח אותן מחדש
    - למשל, זיהוי ע"י time-out
- **הודעות משובשות**
  - למשל בגלל רעש בקו
  - זיהוי ע"י קודים (checksum, message digest).

מצבי תהליך

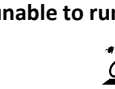


נושאים

- **מבוא לזימון תהליכים**
  1. מצבי תהליך
  2. בעיית זימון התהליכים
  3. מדדים לאיכות זימון תהליכים
- **מדיניויות זימון**
  - א- אלגוריתם First-Come First-Serve
  - ב- אלגוריתם Round Robin
  - ג- מזעור זמן שהייה ממוצע (Shortest Processing Time First)
  - ד- זמני הגעה לא אחידים (Shortest Remaining Time to Completion First)
  - ה- שימוש בכמה תורים (דוגמא Windows NT)

Process States

- Running - actually using the CPU
- Ready – runnable, temporarily stopped to let another process run
- Blocked - unable to run until some external event happens



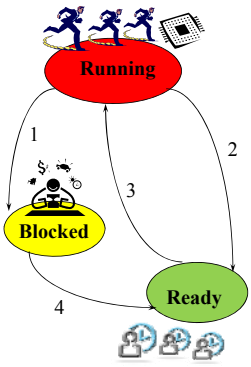
A process can *block itself*, but not "run" itself

41

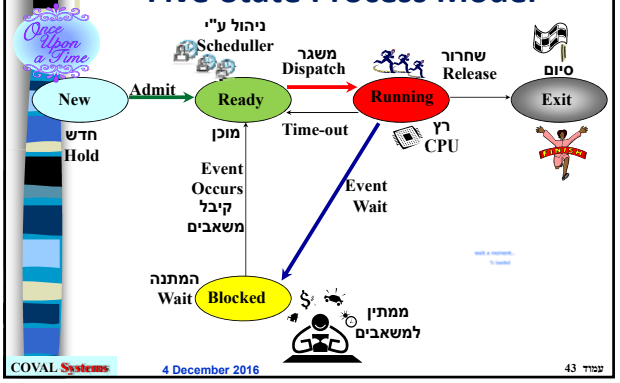
Process State Transitions

When do these transitions occur?

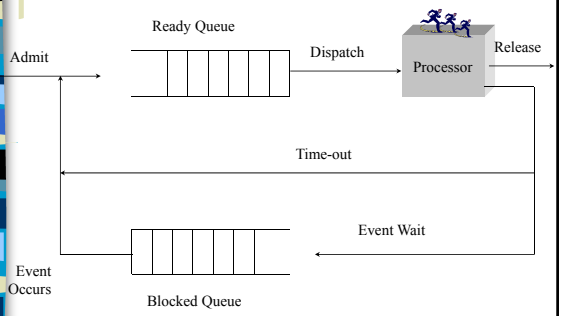
1. Process *blocks* for input or waits for an event
2. End of time-slice, or preemption
3. Scheduler switches back to this process
4. Input becomes available, event arrives



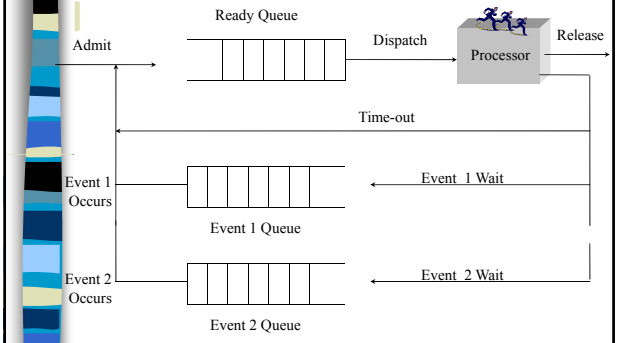
5 מצבי תהליך
Five-State Process Model



Scheduling: Single Blocked Queue

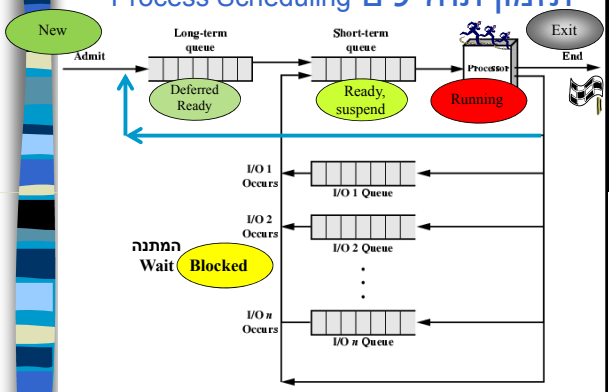


Scheduling: Multiple Blocked Queues



45

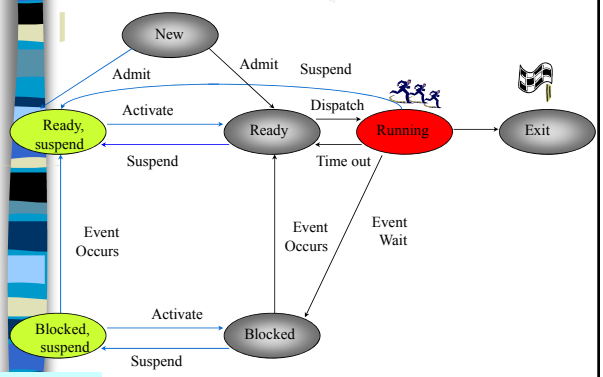
Process Scheduling תזמון תהליכים



Suspended Processes

- Processor is much faster than I/O so many processes could be waiting for I/O
- Swap some of these processes to disk to free up more memory
- Blocked state becomes blocked-suspended state when swapped to disk, ready becomes ready-suspended
- Two new states
  - Blocked-suspended
  - Ready-suspended

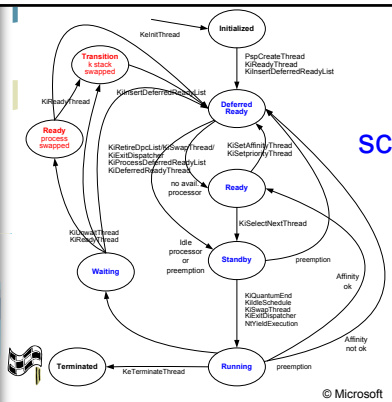
Process State Transition Diagram with Two Suspend States



מתי "זימון תהליכים" נכנס לפעולה?

- **ניתן לבצע החלטות זימון כאשר תהליך עובר ממצב למצב**
  - למשל, תהליך שמבצע פקודת ק"פ (I/O), עובר ממצב **"רץ"** למצב **"המתנה"**
- **האם ניתן לאלץ תהליך לעבור ממצב "רץ" למצב "מוכן"?**
  - אם כן, הזימון נקרא **בר הפקעה** (Preemptive)
  - אם לא, הזימון נקרא **ללא הפקעה** (Non-preemptive)
    - תהליך חייב לשתף פעולה (Cooperative)
    - למשל ב-Windows 3.1
  - התכנות בשתי הסביבות יכול להיות מאד שונה. **מדוע?**

Thread scheduling states



בעיית זימון התהליכים

- **הבעיה**
  - הקצאת המעבד (המשאב) לתהליך מסוים לפרק זמן מסוים

- **דרגות החופש**
  - מבין התהליכים שמוכנים לרוץ, איזה תהליך לבחור
  - לכמה זמן נקצה את המעבד לתהליך שנבחר

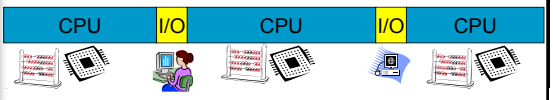
תקורה (overhead) בזימון תהליכים

- **זימון תהליכים אינו חינם**
  - מעבר ל-Kernel mode
  - החלפת הקשר (Context switch)
    - שמירת המצב\הקשר של התהליך היוצא
    - טעינת המצב\הקשר של התהליך הנכנס
  - מעבר ל-User mode וקפיצה לפקודה הבאה

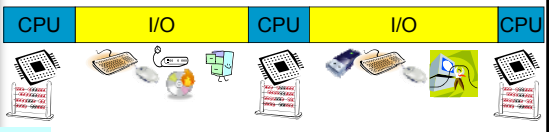
- **ישנו גם מחיר מוסתר...**
  - Page faults
  - Cache misses

אפיון תהליכים

- **תהליך עתיר חישובים (CPU-bound)**



- **תהליך עתיר קלט/פלט (I/O-bound)**



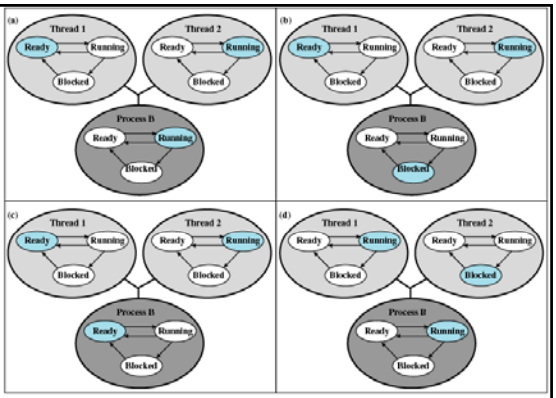


Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States

מהו המדד הנכון?

- **תלוי...**
  - –בסביבה אינטראקטיבית
    - חשוב זמן תגובה קצר
    - ... אך שונות נמוכה חשובה לא פחות.
  - –בסביבת אצווה (batch)
    - רצוי למקסם תפוקה (תפוקה מקסימאלית)

מדדים הקובעים את "טיב" מדיניות הזימון

- **ניצולת המעבד** (CPU utilization)
  - – אחוז הזמן שהמעבד "פעיל"
- **תפוקה** (Throughput)
  - – מספר יחידות העבודה שהושלמו בפרק זמן נתון
- **זמן ביצוע תהליך** (Turn-around-time)
  - – מרגע שיגור התהליך עד לסיומו
- **זמן המתנה למעבד** (Waiting time)
  - – סכום הזמנים בהם התהליך במצב "מוכן"
- **זמן תגובה** (Response time)
  - – בתהליך אינטראקטיבי, פרק הזמן בין הגשת בקשה ע"י המשתמש עד לתגובת המערכת (תצוגת פלט)

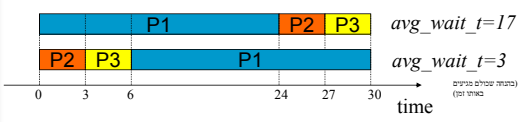
מדינויות זימון

הגדרת מדדים

- **יחס הענישה של תהליך P_i הוא t_i/T_i כאשר**
  - – T_i - הוא **זמן השהייה** - סה"כ זמן שתהליך בטווח הקצר - במצב "רץ" או "מוכן"
  - – t_i - הוא סה"כ זמן שתהליך במצב "רץ"
- **זמן שהייה ממוצע של תהליך, תחת מדיניות זימון A**
  - – כאשר N הוא מספר התהליכים $H_A = \frac{1}{N} \cdot \sum_{k=1}^{N} T_k$

First Come First Serve (FCFS)

- **הגדרה**
  - – המעבד מוקצה לתהליך הראשון שדורש אותו
- **פרטי מימוש**
  - – Non-preemptive (Cooperative)
  - – ממומש ע"י תור FIFO של התהליכים במצב מוכן
  - – זמן שהייה של תהליך תלוי בזמן ובסדר הגעתו
  - – יחס ענישה גבוה לתהליכים קצרים ונמוך לארוכים

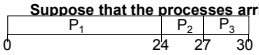


First Come First Served Scheduling

- **Example:**

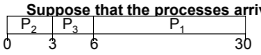
Process	Burst Time
P_1	24
P_2	3
P_3	3

- **Suppose that the processes arrive in the order: P_1 , P_2 , P_3**



Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
Average waiting time: (0 + 24 + 27)/3 = 17

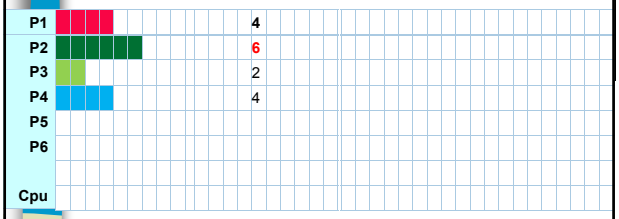
- **Suppose that the processes arrive in the order: P_2 , P_3 , P_1 .**



Waiting time for P_1 = 6; P_2 = 0; P_3 = 3
Average waiting time: (6 + 0 + 3)/3 = 3

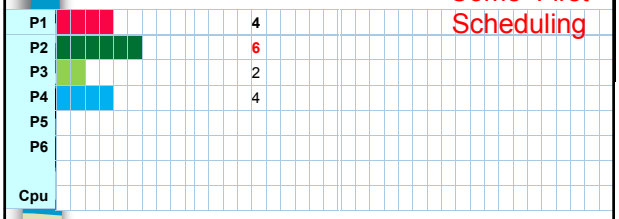
- **Non-preemptive scheduling**
- **Troublesome for time-sharing systems**
  - *Convoy effect* short process behind long process –
- יג-8 23 נובמבר 2015

First Come First Served Scheduling



Shortest Time First Scheduling

עדיפות שנייה First Come First Scheduling



First Come First Serve (FCFS) (Convoy effect) אפקט השיירה

- **אפיון**
  - תהליך אחד עתיר חישובים: C
  - מספר תהליכים עתירי קלט/פלט: $I_1,\ldots,I_n$
  - (בהנחת non-preemption)
- **אפקט**
  - ברגע שתהליך C תופס את המעבד, תהליכי I_j מצטברים בתור המוכנים
  - התקני הקלט/פלט מובטלים!

Round Robin (RR)

q = Quantum

- **הגדרה**
  - המעבד מוקצה לתהליך הראשון בתור לזמן ≥ q
  - אם זמן החישוב של התהליך גדול מ-q, התהליך מופסק ומועבר לסוף תור ה-"מוכנים"
- **פרטי מימוש**
  - Preemptive
  - Quantum (פרוסת זמן / time-slice) אופייני 10-100msec
  - ממומש בעזרת timer שמייצר פסיקה כל פרק זמן q

Round Robin (RR) אפיונים

- **אם ה-quantum קטן, הזימון "הוגן"**
  - תחושה שכל תהליך רץ במעבד משלו עם בקצב 1/N, כאשר N הוא מספר התהליכים – יחס ענישה אחיד.
  - זמן תגובה (כמעט) ליניארי בזמן החישוב וב-N.
  - אך התקורה עלולה להיות גבוהה!

Round-Robin

First Come First Served

תהליך	FCFS	RR
1	100	991
2	200	992
...		
10	1000	1000

- **דוגמא**
  - מספר תהליכים N=10
  - זמן חישוב של תהליך 100
  - q=1

הכללה: Selfish Round-Robin

- **הגדרה**
  - תהליכים חדשים מוחזקים בתור המתנה FIFO
  - תהליכים וותיקים מוחזקים בתור Round-Robin לביצוע
  - כאשר אין תהליך בתור הוותיקים, בוחרים את הראשון בתור החדשים ומצרפים אותו לוותיקים
  - "הזדקנות" – תהליך הופך מחדש לוותיק, למשל על ידי כך שמגדילים את עדיפות כל תהליך בכל יחידת זמן עד מעבר לסף שהופך אותו לוותיק
- **איפיון**
  - אם ההזדקנות קצרה (או לא קיימת) מתקבל RR
  - אם ההזדקנות ארוכה (או אינסופית) מתקבל FCFS

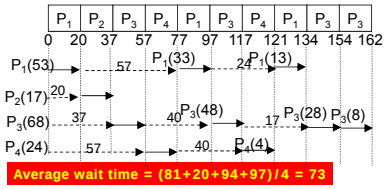
Round-Robin

First Come First Served

Round Robin Scheduling

- Each process is given CPU time in turn, (i.e. time quantum: usually 10-100 milliseconds), and thus waits no longer than (n – 1) * time quantum
- time quantum = 20

Wait Time	Burst Time	Process
57 +24 = 81	53	P_1
20	17	P_2
37 + 40 + 17= 94	68	P_3
57 + 40 = 97	24	P_4



Average wait time = (81+20+94+97)/4 = 73

Burst time is an assumption of how long a process requires the cpu between I/O waits. It can not be predicted exactly, before a process starts.

Round Robin Scheduling

Process Number	Arrival Time	זמן CPU
P1	0	25
P2	12	5
P3	5	12
P4	21	25
P5	35	30
P6	42	5

Quantum = 10 msec

שיטת RR ללא סדרי עדיפויות

יש לחשב:

1- AVG של המתנה

2- AVG של סיום

3- AVG של ריצה (Ready) + המתנה קצרה (Ready)

Round Robin Scheduling

P1
P2
P3
P4
P5
P6
Cpu

0 10 20 30 40

Quantum = 10 msec

יש לחשב:
1- AVG של המתנה
2- AVG של סיום
3- AVG של ריצה (Ready) + המתנה קצרה (Ready)

Process Number	Arrival Time	זמן CPU
P1	0	20
P2	10	5
P3	5	10
P4	20	20
P5	30	30
P6	40	5

Round Robin Scheduling

P1
P2
P3
P4
P5
P6
Cpu

0 10 20 30 40

Quantum = 10 msec

יש לחשב:
1- AVG של המתנה
2- AVG של סיום
3- AVG של ריצה (Ready) + המתנה קצרה (Ready)

Process Number	Arrival Time	זמן CPU
P1	0	25
P2	12	5
P3	5	12
P4	21	25
P5	35	30
P6	42	5

Round Robin Scheduling

Process Number	Arrival Time	CPU זמן	I/O זמן	CPU זמן	I/O זמן	CPU זמן
P1	0	25	0	-	-	-
P2	12	5	10	20	-	-
P3	5	12	15	5	-	-
P4	21	25	5	18	5	10
P5	35	30	10	25	-	
P6	42	5	20	13	-	-

Quantum = 10 msec

שיטת RR ללא סדרי עדיפויות

יש לחשב:
1- AVG של המתנה
2- AVG של סיום
3- AVG של ריצה (Ready) + המתנה קצרה (Ready)

Round Robin Scheduling

P1
P2
P3
P4
P5
P6
Cpu

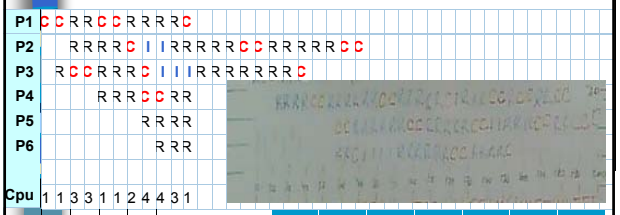
0 10 20 30 40

Quantum = 15 msec

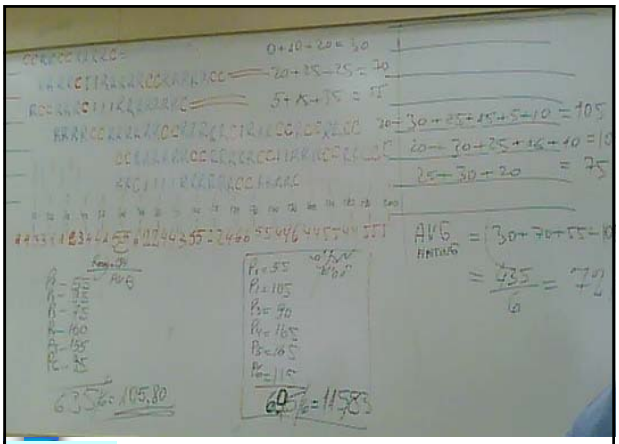
יש לחשב:
1- AVG של המתנה
2- AVG של סיום
3- AVG של ריצה (Ready) + המתנה קצרה (Ready)

Process Number	Arrival Time	CPU זמן	I/O זמן	CPU זמן	I/O זמן	CPU זמן
P1	0	25	0	-	-	-
P2	12	5	10	20	-	-
P3	5	12	15	5	-	-
P4	21	25	5	18	5	10
P5	35	30	10	25	-	
P6	42	5	20	13	-	-

Round Robin Scheduling

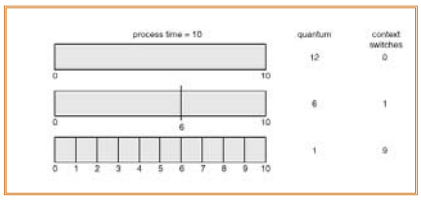


Process Number	Arrival Time	CPU זמן	I/O זמן	CPU זמן	I/O זמן	CPU זמן
P1	0	25	0	-	-	-
P2	12	5	10	20	-	-
P3	5	12	15	5	-	-
P4	21	25	5	18	5	10
P5	35	30	10	25	-	
P6	42	5	20	13	-	-



Round Robin Scheduling

- **Typically, higher average turnaround than SJF, but better *response*.**
- **Performance**
  - *q* large ⇒ FCFS (First Come First Served)
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high.



Shortest Processing Time First (SPTF)

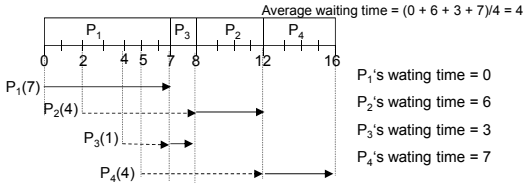
- **הנחה**
  - כל N התהליכים מגיעים יחד לביצוע
  - זמן הריצה של תהליך P_i ידוע מראש (t_i)
- **הגדרה**
  - בוחרים את התהליך עם t_i מינימלי שירוץ עד סיומו
- **פרטי מימוש**
  - Non-preemptive
  - אבל... איך מנחשים את t_i?

Shortest Job First Scheduling

- **Example:**

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- **Non preemptive SJF**

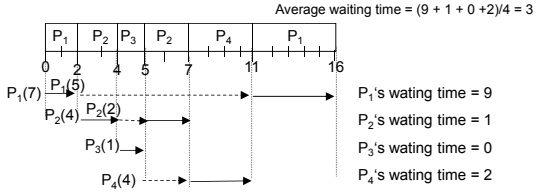


Shortest Job First Scheduling Cont'd

- **Example:**

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- **Preemptive SJF**



- המשימה העיקרית של ליבה היא לאפשר הרצה של תוכניות מחשב ולספק להן שירותים כגון **אבסטרקציה** של החומרה. תהליך מגדיר אילו חלקים מהזיכרון יהיו נגישים לתוכנית, וניהול התהליכים שמתבצע בליבה צריך לקחת בחשבון את החומרה המובנית לצורך הגנה על הזיכרון.
- כדי להריץ תוכנית, בדרך כלל הליבה מגדירה עבורה מרחב כתובות, טוענת לזיכרון את הקובץ שמכיל את הקוד של התוכנית, יוצרת **מחסנית קריאות**, מצביעה למיקום מסוים בתוך התוכנית ובכך מתחילה את הביצוע שלה.
- ליבות התומכות **בריבוי משימות** מסוגלות ליצור אשליה עבור המשתמש, כאילו מספר התהליכים שרצים בו זמנית על גבי המחשב הוא גדול יותר מהמספר המקסימלי של תהליכים שהמחשב פיזית יכול להריץ בו זמנית. בדרך כלל מספר התהליכים שמערכת יכולה להריץ בו זמנית שווה למספר המעבדים שמותקנים במחשב (עם זאת, המצב יכול להיות שונה במידה והמעבדים תומכים ב- simultaneous multithreading).

במערכת המבוססת על ריבוי משימות מקדים (pre-emptive multitasking), הליבה תיתן לכל תוכנית פרוסת זמן ותבצע מיתוג (switch) מתהליך לתהליך במהירות כזאת עד שהדבר יראה למשתמש כאילו תהליכים אלה מורצים בו זמנית. הליבה משתמשת באלגוריתמי תזמון על מנת לקבוע מה יהיה התהליך הבא שירוץ וכמה זמן יוקצה לו. האלגוריתם הנבחר יכול לאפשר לחלק מהתהליכים להיות בעלי קדימות גבוהה יותר מזו של אחרים. הליבה בדרך כלל גם מספקת לתהליכים אלה אמצעי כלשהו להתקשר; דבר זה ידוע בתור תקשורת בין תהליכים (IPC), והשיטות העיקריות לצורך כך הן זיכרון משותף, העברת מסרים והפעלת פרוצדורות מרחוק.

מערכות אחרות (בייחוד על גבי מחשבים קטנים ופחות עוצמתיים) יכולות להיות מבוססות על ריבוי משימות שיתופי (co-operative multitasking), שבו כל תהליך מורשה לרוץ ללא פסיקות ("ללא הפרעה"), עד שהוא מגיש בקשה שאומרת לליבה שהיא יכולה לבצע מיתוג לתהליך אחר. בקשות כאלה נקראות (yielding יתור), בדרך כלל הן מתרחשות בתגובה לבקשות לתקשורת בין תהליכים, או כאשר מחכים לאירוע שיתרחש. גרסאות ישנות יותר של Windows ו- Mac OS השתמשו ב- co-operative multitasking, אבל שתיהן עברו לשיטת ה- pre-emptive ככל שגדלה עוצמת המחשבים שעבורם תוכננו.

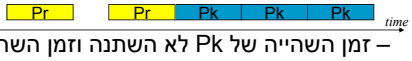
מערכת ההפעלה עשויה לתמוך גם ב- multiprocessing (בארכיטקטורת Symmetric multiprocessing או Non-Uniform Memory Access); במקרה זה, תוכניות ותהליכונים שונים יכולים לרוץ על מעבדים שונים. הליבה עבור מערכת כזאת צריכה להיות מתוכננת כך שתוכל בו זמנית להריץ בבטחה שני חלקים שונים של הקוד שלה. בדרך כלל המשמעות של כך היא שיש לספק מנגנוני סנכרון (כדוגמת spinlock על מנת להבטיח שלא יקרה מצב בו שני מעבדים מנסים לשנות את אותם הנתונים באותו זמן.

הפקעה = העברה לרשות הכלל

עם הפקעה או ללא הפקעה? *(המשך)*

- **הוכחה**
  - בהינתן זימון של N תהליכים לפי זימון A
  - יהי Pk התהליך שמסתים אחרון

  - נצופף את ריצת Pk לסוף הזימון



  - זמן השהייה של Pk לא השתנה וזמן השהייה של התהליכים האחרים לא גדל

הפקעה = העברה לרשות הכלל

עם הפקעה או ללא הפקעה?

- **נזכיר:** זמן שהיה ממוצע עבור מדיניות זימון A:

$$H_A = \frac{1}{N} \cdot \sum_{k=1}^{N} T_k$$

- **למה**
  - לכל מדיניות זימון עם הפקעה A (עבור N תהליכים שמגיעים יחד), קיימת מדיניות A' ללא הפקעה כך ש- $H_{A'} \le H_A$

אופטימליות של H_A לפי מדד SPTF

Shortest Processing Time First

- **משפט**
  - אם כל התהליכים מגיעים יחד, לכל מדיניות זימון A $H_{SPTF} \le H_A$

- **הוכחה**
  - לפי הלמה, ניתן להניח ש-A היא מדיניות ללא הפקעה (הפקעה = העברה לרשות הכלל)
  - אם A לא SPTF, חייבים להיות שני תהליכים P_k ו-P_r כך ש- $t_k < t_r$ ו-P_r מתוזמן מיד אחרי P_k

Shortest Processing Time First = (SPTF)

אופטימליות של H_A לפי מדד SPTF

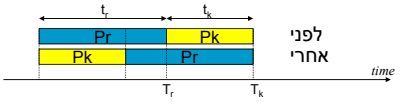
(המשך)

- **הוכחה (המשך)**
  - זמן שהייה ממוצע אחרי ההחלפה:

$$\frac{1}{N} \cdot [T_1 + \ldots + (T_r + t_k) + \ldots + (T_k - t_r) + \ldots + T_N] =$$

$$= \frac{1}{N} \cdot \left[\sum T_i + (t_k - t_r)\right]$$

  - זמן השהייה הממוצע לא גדל!
  - חוזרים על ההחלפות עד שהתזמון נהיה SPTF



Shortest Processing Time First = (SPTF)

זמן שהייה ממוצע תחת RR

- **משפט**
  - אם כל התהליכים מגיעים יחד, אזי $H_{RR} \le 2H_{SPTF}$
- **הוכחה**
  - נסמן ב-$delay_A(i, j)$ את העיכוב שנגרם לתהליך P_j עקב זימונים של תהליך P_i תחת מדיניות A
  - עבור מדיניות A כלשהי, מתקיים:

$$N \cdot H_A = \sum_{i=1}^{N} t_i + \sum_{1 \le i < j \le N} [delay_A(i,j) + delay_A(j,i)]$$

Shortest Processing Time First = (SPTF) – Round Robin = (RR)

זמן שהייה ממוצע תחת RR *(המשך)*

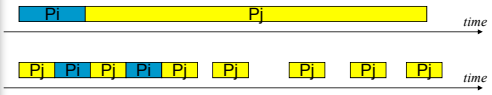
עבור SPTF

$$N \cdot H_{SPTF} = \sum_{i=1}^{N} t_i + \sum_{1 \le i < j \le N} \min(t_i, t_j)$$

עבור Round Robin

$$N \cdot H_{RR} = \sum_{i=1}^{N} t_i + \sum_{1 \le i < j \le N} 2 \cdot \min(t_i, t_j)$$

i<j



Shortest Remaining Time to Completion First (SRTF)

- **הגדרה**
  - כאשר מגיע תהליך Pi עם זמן חישוב נותר קצר יותר משארית זמן החישוב של התהליך שרץ כרגע Pk, אזי מפקיעים את המעבד מ-Pk ומכניסים את Pi
- **משפט**
  - SRTF ממזער את זמן השהייה הממוצע במערכת (בדומה ל-SPTF)
- **הוכחה**
  - דומה להוכחת האופטימליות של SPTF
- **מה ההבדל בין SRTF ו- SPTF?**
  - SRTF עובד טוב גם כאשר תהליכים לא מגיעים יחד

אומדן זמן החישוב של תהליך

- **הבעיה**
  - קשה לנבא את זמן החישוב שתהליך יצטרך
- **פיתרון**
  - אומדן סטטיסטי (מריצים התהליך כמה פעמים ומנחשים לעתיד)
    - τ_i – הערכת זמן החישוב לסיבוב ה-i
    - t_i – זמן החישוב בפועל בסיבוב ה-i

$$\tau_{i+1} = \alpha \cdot \tau_i + (1-\alpha) \cdot t_i \qquad 0 \le \alpha \le 1$$

- **שימו לב!**
  - $\alpha = 0$ – זמן הריצה האחרון בפועל קובע (היסטוריה קרובה)
  - $\alpha = 1$ – זמן ריצה בפועל לא משפיע!

שימוש במספר תורים

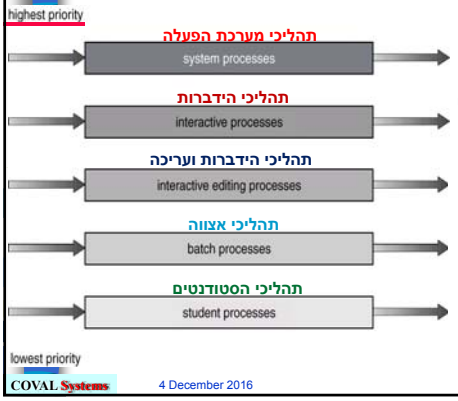
- **ניתן להפריד תהליכים לקבוצות שונות לפי איפיונם**
  - foreground, background, …
- **מדיניות תזמון מתפרקת ל-**
  - מדיניות תזמון בין התהליכים השייכים לאותה קבוצה
  - מדיניות תזמון בין קבוצות שונות

מודול ניהול תהליכים

ריבוי תורים
Multilevel Feedback Queues

- **הגדרה**
  - קיימים מספר תורים לפי סדר עדיפות
    - תור גבוה לתהליכים בעלי עדיפות גבוהה יותר
  - תהליך מתחיל בתור הגבוה ביותר
  - אם הוא צורך את כל ה-quantum, הוא יורד לתור נמוך יותר
  - אם תהליך משחרר המעבד לפני סוף ה-quantum (עקב פעולת ק"פ [I/O]), הוא חוזר לתור גבוה יותר
- **פרטי מימוש**
  - לתורים הנמוכים נקצה אחוז קטן יותר של זמן מעבד
  - ה-quantum עולה עבור תורים נמוכים יותר

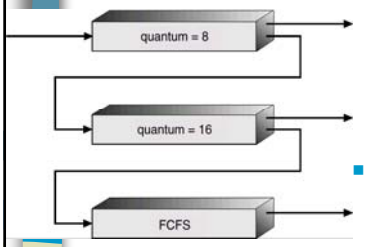
Multilevel Queue Scheduling



לכל טור אלגוריתם לתזמון משלו

- **Each queue has its own scheduling algorithm,**
  - foreground (interactive) – RR, 80% CPU time
  - background (batch) – FCFS, 20% CPU time

Multilevel Feedback-Queue Scheduling



- **A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.**
- **At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.**

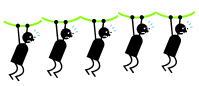
דוגמא: זימון ב-Windows NT

- **שימוש בתורים לפי עדיפות**
  - 32 תורים (dispatcher ready queues) –
    - Real time priority (עדיפויות 16-31)
    - Variable priority (עדיפויות 1-15)
    - (עדיפות 0 שמורה למערכת)
- **מדיניות Round Robin**
  - על התור העדיף ביותר שאינו ריק –
- **העדיפות יורדת אם נצרך כל ה-quantum**
- **העדיפות עולה אם תהליך עובר מ-wait ל-ready**
  - העדיפות נקבעת ע"פ סוג הק"פ [I/O] .
    - ק"פ [I/O] מהמקלדת מקנה עדיפות גבוהה

חוטים
Threads

נושאים

- **הגדרות**
  - חוטים –
  - חוטים לעומת תהליכים
- **תמיכת מערכת ההפעלה בחוטים**
- **דוגמאות**
  - Mach –
  - Windows NT –

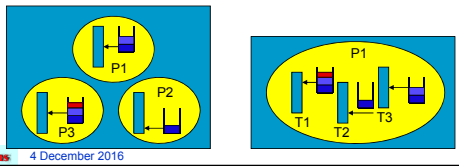


מוטיבציה

- **תהליכים דורשים משאבי מערכת רבים**
  - מרחב כתובות, גישה לקלט/פלט (File Table), וכו'
- **זימון תהליכים הינו פעולה כבדה**
  - החלפת מצב/הקשר במעבד
- **...אך הרבה משימות הן בעלות אופי "מקבילי"**
  - תהליך שרת, פעולות מתימטיות, וכו'

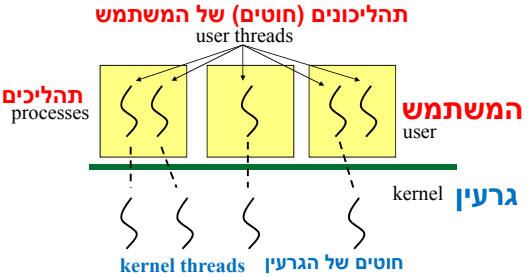
הגדרה

- **Thread:** זרם ביצוע סדרתי - A sequential execution stream
- **Address space:** נתחי זיכרון כל מה שצריך כדי להפעיל תוכנית
  Chunks of memory and everything needed to run a program
- **Process:** חוט + מרחב הכתובות - An address space + thread(s)
- **Two types of threads**
  - Kernel threads
  - User-level threads



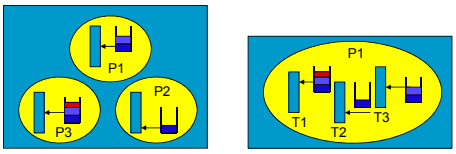
המתזמן מקיר רק את חוטי הגרעין

OS scheduler only knows about kernel threads



הגדרה

- **חוט הינו יחידת ביצוע (בקרה) נפרדת בתוך תהליך**
- **המשאבים הדרושים הם:**
  - Program Counter –
  - מחסנית –
  - רגיסטרים –



חוטים לעומת תהליכים

	תהליך	חוט
Program Counter	☐	☐
Registers	☐	☐
Execution Stack	☐	☐
Address Space	☐	☐
Open Files	☐	☐
File position	☐	☐
ערך כתובת וירטואלית	☐	☐
ערך כתובת פיזית	☐	☐
מנעול	☐	☐

חוטים לעומת תהליכים

	תהליך	חוט
Program Counter	☑	☑
Registers	☑	☑
Execution Stack	☑	☑
Address Space	☑	☐
Open Files	☑	☐
File position	☑	☐
ערך כתובת וירטואלית	☑	☐
ערך כתובת פיזית	☑	☐
מנעול	☑	☐

דוגמא: שרת קבצים

תהליך עם חוט בקרה יחיד

- **המימוש**

```
do forever
  get request;
  execute request;
  return results;
end;
```

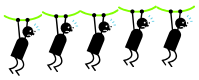
- **יתרונות**
  - פשטות
- **חסרונות**
  - ניצול משאבים, למשל בהמתנה ל-קלט/פלט
- **הערה**
  - ניתן ליעל ע"י ביצוע ק"פ (קלט/פלט) אסינכרוני וטיפול במספר בקשות בו זמנית

דוגמא: שרת קבצים

- **תהליך מקבל בקשות**
  - קריאה (כתיבה)
  - זיהוי קובץ
  - מיקום בקובץ
  - אורך
  - חוצץ (Buffer)
- **התהליך מחזיר**
  - סטטוס (הצלחה/כישלון)
  - אורך (קריאה/כתיבה בפועל)
  - מחרוזת תווים (במקרה של קריאה)

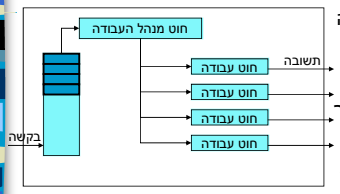
יתרונות

- **יצירת חוט יעילה (קלה) יותר**
  - דרושה רק יצירת Thread Control Block והקצאת מחסנית
- **ניצול טוב יותר של משאבים**
  - למשל, במערכות מרובות מעבדים
- **תקשורת נוחה יותר בין חוטים השייכים לאותו תהליך**
  - זיכרון משותף
- **תכנות מובנה יותר**



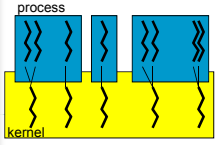
דוגמא: שרת קבצים - ריבוי חוטים

- **מימוש**
  - חוט מנהל אחד
    - מקבל בקשה
    - מעביר את הבקשה לביצוע ע"י אחד מחוטי העבודה
  - חוטי עבודה
    - מבצע את הבקשה
    - מחזיר תשובה
- **יתרונות**
  - תפוקה גבוהה יותר
- **חסרונות**
  - תכנות מורכב יותר



חוטי משתמש לעומת חוטי מערכת

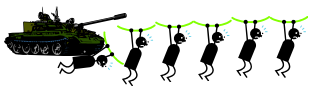
- **חוט משתמש (user threads)**
  - מוגדרים ע"י סביבת התכנות (לא מוכרים ע"י מערכת ההפעלה)
  - יעילים יותר
    - לא דורשים קריאות מערכת
    - לא מתבצעת החלפת הקשר
  - מה קורה כאשר חוט נחסם?



- **חוט גרעין (kernel threads)**
  - מוגדרים ע"י מערכת ההפעלה
- **מבט היסטורי**
  - מערכות עברו מתמיכה בחוט יחיד (single threaded) לתמיכה בחוטי משתמש (ללא שינויים במערכת ההפעלה), לתמיכה בחוטי גרעין

חסרונות

- **חוסר הגנה בין חוטים באותו תהליך**
  - חוט עלול לדרוס את המחסנית של חוט אחר



שירותי תמיכה בחוטים

- **יצירה והריסת חוטים, לדוגמא**
  - thread_create(char *stack)
  - thread_exit(…), thread_join(…), thread_kill(…)
- **מנגנוני סנכרון**
  - למשל עבור תיאום בגישה לזיכרון משותף
- **מבני נתונים פנימיים**
  - Thread Control Block –
    - Program Counter, stack, registers •

איפיון מערכות הפעלה

	מרחב כתובות יחיד	הרבה מרחבי כתובות
חוט יחיד בכל מרחב כתובות	DOS, MacOS (<8)	Unix (traditional)
מספר חוטים בכל מרחב כתובות	Embedded Systems (מערכות קבועות)	Linux, Mach, Solaris, Windows NT

דוגמא: Remote Procedure Call
פרטי מימוש

- **תהליך קורא**
  - אורז את ה-input parameters, שולח אותם לתהליך השרת, בתוספת זיהוי הפונקציה המבוקשת ונחסם
- **תהליך נקרא**
  - פותח את ה-input parameters
  - קורא לפרוצדורה (הלוקלית) לביצוע הבקשה
  - אורז את ה-output parameters, ושולח אותם חזרה
  - מחכה לבקשה הבאה
- **תהליך קורא**
  - פותח את ה-output parameters
  - חוזר

דוגמא: Remote Procedure Call

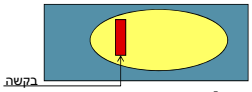
- **Remote Procedure Call (RPC) מדמה את מנגנון קריאה לפרוצדורה**
  - בהבדל שהפרוצדורה הקוראת והנקראת רצות (אולי) במכונות שונות
  - המנגנון מסתיר את העברת הפרמטרים והתוצאות
- **RPC היא גם הפשטה נוחה לתקשורת בין תהליכים (או חוטים) על אותה מכונה**

Mach
מערכת הפעלה מבוססת הודעות

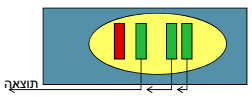
- **היחידה הבסיסית: משימה (task)**
  - יכולה להכיל מספר חוטים
- **מנגנון בסיסי לתקשורת: תיבות דואר (ports)**
  - תקשורת בין חוטים
  - תקשורת מ-/למערכת ההפעלה
- **משימה נוצרת עם שתי תיבות דואר**
  - Kernel mailbox מהתהליך למערכת ההפעלה
  - Notify mailbox להעברת signals ממערכת ההפעלה

דוגמא: Remote Procedure Call
מימוש באמצעות pop-up threads

- **בהתחלה, בתהליך השרת יש חוט יחיד**



- **כאשר מתקבלת בקשה, החוט מקבל את הפרמטרים ומייצר חוט שמטפל בבקשה**



Windows NT

- **היחידה הבסיסית הינה <u>תהליך</u>**
  - תהליך יכול להכיל כמה **<u>חוטים</u>** – kernel threads
  - חוט יכול להכיל כמה **<u>סיבים</u>** (fibers) – user threads
  - זימון נעשה ברמת חוטים
  - תהליך מגדיר בעיקר את מרחב הזיכרון
- **תקשורת בין חוטים**
  - באמצעות pipes ותיבות דואר

Mach
הודעות

- **קריאות מערכת לתקשורת באמצעות הודעות**
  - msg_send, msg_receive, msg_rpc, port_allocate
- **הודעה מכילה**
  - Header – אורך ההודעה, תיבות מקור ויעד
  - תוכן ההודעה (שדות עם type) באורך משתנה
- **אם תיבת הדואר מלאה**
  - החוט ממתין עד שיתפנה מקום, או
  - החוט ממתין עד חסם (time-out) מסוים
  - Caching של ההודעה
- **Mach ממפה (לא מעתיקה) את ההודעה ממרחב התהליך השולח למרחב התהליך המקבל**
  - Copy-on-write – אם אחד התהליכים משנה את תוכן ההודעה, ההודעה מועתקת

תיאום בין תהליכים: יסודות

- **דוגמאות לבעיות תיאום**
- **הגדרות: קטע קריטי, מנעולים**
- **אלגוריתם קופת-חולים**

Windows NT
תקשורת

- **חוט נוצר עם message queue**
  - לקבלת הודעות מחוטים אחרים (כולל מעצמו)
- **מבנה הודעה**
  - שדה סוג (32 bits)
  - שדה נתונים (64 bits)
- **קריאות**
  - חוסמות / לא חוסמות לשליחת או קבלת הודעות

<u>דוגמא: בנק הפועלים</u>

- **מימשנו פונקציה למשיכת כסף מחשבון בנק.**

```
int withdraw( account, amount) {
  balance = get_balance( account );
  balance -= amount;
  put_balance( account, balance);
  return balance;
}
```

- **בחשבון יש $50000, ושני בעלי החשבון ניגשים לכספומטים שונים ומושכים $30000 ו-$20000 בו-זמנית.**

תיאום

- **תהליכים משתפים פעולה:**
  - גישה למשאבים משותפים, למשל זיכרון משותף (בעיקר חוטים).
  - העברת נתונים מתהליך אחד לשני דרך התקן משותף.
- **חייבים לתאם את השיתוף:**
  - מניחים שביצוע התהליכים משולב באופן שרירותי. למתכנת האפליקציה אין שליטה על זימון התהליכים.
  - שימוש במנגנוני תיאום (synchronization).

בנק הפועלים: יש כסף!

- תהליך נפרד מבצע כל פעולת משיכה (על אותו מעבד).

```
balance = get_balance(account);
balance -= amount;      // 50K-30K
```

```
balance = get_balance(account);
balance -= amount;      // 50K-20K
put_balance(account, balance);
return balance;         // = 30K
```

```
put_balance(account, balance);
return balance;         // = 20K
```

מי שמח עכשיו?

בנק הפועלים: אין כסף?

- תהליך נפרד מבצע כל פעולת משיכה (על אותו מעבד)

```
balance = get_balance(account);
balance -= amount;      // 50K-30K
put_balance(account, balance);
return balance;         // = 20K
```

```
balance = get_balance(account);
balance -= amount;      // 20K-20K
put_balance(account, balance);
return balance;         // = 0
```

בחשבון $0....

יותר מידי חלב!

שעה	סיון:	עמיר:
3:00	מסתכלת במקרר	
3:05	הולכת לסופר	מסתכל במקרר
3:10	קונה חלב	הולך לסופר
3:15	חוזרת הביתה	קונה חלב
3:20	מכניסה חלב למקרר	חוזר הביתה
3:25		מכניס חלב למקרר

עוד דוגמא

- שני חוטים מבצעים אותו קוד
- החלפת חוטים אפשרית בכל מקום

```
shared in, out

procedure echo();
  read( in, keyboard);
  out = in;
  write( out , screen);
end echo
```

race condition (מצב מירוץ)

תוצאת הריצה אינה צפויה.

בעיה עם פתרון 1

```
if (no milk) then
   if (no note) then
```

```
      leave note
      buy milk
      remove note
```

```
if (no milk) then
   if (no note) then
      leave note
      buy milk
      remove note
```

פעמיים חלב!

פיתרון 1

להשאיר פתק לפני שהולכים לסופר

```
if (no milk) then
   if (no note) then
      leave note
      buy milk
      remove note
```

פיתרון 2

משאירים פתקה לפני שבודקים את המקרר:

```
Thread A:
  leave note A
  if (no note B) then
    if (no milk) then
      buy milk
  remove note A
```

```
Thread B:
  leave note B
  if (no note A) then
    if (no milk) then
      buy milk
  remove note B
```


בעיה עם פתרון 2

```
leave note A
if (no note B) then
remove note A
```

```
leave note B
if (no note A) then
remove note B
```

אין חלב!

פיתרון 3 ☺

לא סימטרי

```
Thread A:
  leave note A
  while (note B) do nop
  if (no milk) then
    buy milk
  remove note A
```

```
Thread B:
  leave note B
  if (no note A) then
    if (no milk) then
      buy milk
  remove note B
```

אם שניהם משאירים פתק בו זמנית (race condition), A יקנה חלב!

– לא הוגן

רק לשני תהליכים ☹

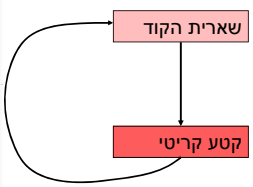
לב הבעיה

- **שני תהליכים נגשים בו-זמנית לאותו משאב, ללא תיאום.**
  - למשל, בגישה למשתנים גלובליים.
  - התוצאה אינה צפויה.
- **מנגנון לשליטה בגישות מקביליות למשאבים משותפים.**
  - כך שנוכל לחזות באופן דטרמיניסטי את התוצאות.
- **לכל מבנה נתונים של מערכת הפעלה (וגם להרבה תוכניות משתמש מרובות-חוטים).**
  - Buffers, queues, lists, hash tables

קטע קריטי

Youd dalet 8 28 nov 2016

הקוד שניגש למשאב המשותף מכונה קטע קריטי

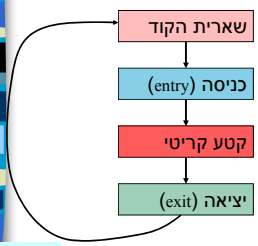


שימו לב: לא בהכרח אותו קוד לכל החוטים

– חוט אחד מגדיל מונה וחוט שני מקטין אותו.

קטע קריטי

הקוד שניגש למשאב המשותף מכונה קטע קריטי עוטפים אותו בקוד כניסה וקוד יציאה (של הקטע הקריטי).



רוצים להבטיח אטומיות – Atomicity

(ערכות גרעינית)

- בביצוע סדרת פקודות ע"י חוט אחד,חוטים אחרים אינם יכולים לראות תוצאות חלקיות
- ניתן להשיג בעזרת מנגנון של מניעה הדדית

באלגוריתם לבעית החלב

```
Thread A:
  leave note A
  while (note B) do nop
  if (no milk) then
    buy milk
  remove note A

Thread B:
  leave note B
  if (no note A) then
    if (no milk) then
      buy milk
  remove note B
```

קוד כניסה

קטע קריטי

קוד יציאה

תכונות רצויות

מניעה הדדית: חוטים לא מבצעים בו-זמנית את הקטע הקריטי. (mutual exclusion)

– חוטים מעוניינים מחכים בקטע הכניסה.

– כאשר החוט הנוכחי יוצא מהקטע הקריטי, יכולים להיכנס.

MUTual Exclusion (MUTEX)

שיטה של סנכרון גישות מרובות למקורות מידע משותפים ע"י מנגנון "lock–unlock" הנועל ופותח לסירוגין את הגישה לתכנית אחת בלבד.

תכונות רצויות

מניעה הדדית: חוטים לא מבצעים בו-זמנית את הקטע הקריטי. (mutual exclusion)

התקדמות: אם יש חוטים שרוצים לבצע את הקטע הקריטי, חוט כלשהו יצליח להיכנס. (no deadlock, אין קיפאון)

אלא אם חוט אחר נמצא בתוך הקטע הקריטי.

תכונות רצויות ...המשך

מניעה הדדית: חוטים לא מבצעים בו-זמנית את הקטע הקריטי. (mutual exclusion)

התקדמות: אם יש חוטים שרוצים לבצע את הקטע הקריטי, חוט כלשהו יצליח להיכנס. (no deadlock, אין קיפאון)

הוגנות: אם יש חוט שרוצה לבצע את הקטע הקריטי, לבסוף יצליח. 3 סוגים:

– no starvation (אין הרעבה) – החוט מתישהו יצליח

• אבל אפשר לעקוף אותו מספר פעמים לא חסום

– bounded waiting – ניתן לעקוף את החוט בכניסה לקטע הקריטי מספר פעמים חסום

– FIFO - החוטים ייכנסו לקטע הקריטי לפי סדר הבקשות

דוגמא לשימוש במנעולים

בחזרה לפונקציה למשיכת כסף מחשבון בנק.

```
int withdraw( account, amount) {
  acquire ( lock ) ;
  balance = get_balance( account );
  balance -= amount;
  put_balance( account, balance);
  release ( lock ) ;
  return balance;
}
```

קטע קריטי

מנעולים (locks)

- **אבסטרקציה אשר מבטיחה גישה בלעדית למידע באמצעות שתי פונקציות:**
  – acquire (lock) – נחסם בהמתנה עד שמתפנה המנעול.
  – release (lock) – משחרר את המנעול.

- **acquire ו release מופיעים בזוגות:**
  – אחרי acquire החוט מחזיק במנעול.
  – רק חוט אחד מחזיק את המנעול (בכל נקודת זמן).
  – יכול לבצע את הקטע הקריטי.

מימוש מנעולים

- **אם היו לנו מנעולים, היה נפלא...**
- **אבל מימוש מנעול מכיל קטע קריטי:**

– קרא מנעול
• אם מנעול פנוי, אז
– כתוב שהמנעול תפוס.

המשך הדוגמא

```
acquire(lock);
balance = get_balance(account);
balance -= amount;      // 50K-25K
acquire(lock);
put_balance(account, balance);
release(lock);
balance = get_balance(account);
balance -= amount;      // 25K-25K
put_balance(account, balance);
release(lock);
return balance;         // = 0
return balance;         // = 25K
```

- **שני תהליכים מבצעים את פעולת המשיכה**
- **מה קורה כשהאדום מבקש את המנעול?**
- **מדוע ה return מחוץ לקטע הקריטי?**
  – האם זה נכון?

אלגוריתם קופת-חולים

- **ידוע כאלגוריתם המאפיה (bakery).**
  [Lamport, 1978]

- **שימוש במספרים:**
  – חוט נכנס לוקח מספר.
  – חוט ממתין שמספרו הקטן ביותר נכנס לקטע הקריטי.

דרכים לממש מנעולים

- **פתרונות תוכנה:**
  – אלגוריתמים.
  – מבוססים על לולאות המתנה (busy wait).
- **שימוש במנגנוני חומרה:**
  – פקודות מיוחדות שמבטיחות מניעה הדדית.
  – לא תמיד מבטיחות התקדמות.
  – לא מובטחת הוגנות.
- **תמיכה ממערכת ההפעלה:**
  – מבני נתונים ופעולות שמהם ניתן לבנות מנגנונים מסובכים יותר.
  – בדרך-כלל, מסתמכים על מגנוני חומרה.

חלוקת מספרים: שבירת סימטריה: ניסיון

```
Thread i:
  initially number[i]=0
  number[i]=max{number[1],…,number[n]}+1;
  for all j≠i do
    wait until number[j]=0 or
      ((number[j],j)>(number[i],i)) // סדר לקסיקוגרפי
  critical section
  number[i]=0
```

משתמשים במספר החוט (thread id).

סדר לקסיקוגרפי = סדר אלפביתי

חלוקת מספרים: ניסיון 1

```
Thread i:
  initially number[i]=0
  number[i]=max{number[1],…,number[n]}+1;
  for all j≠i do
    wait until number[j]=0 or (number[j]>number[i])
  critical section
  number[i]=0 // Exit critical section
```

- **חוט i ו-j קוראים את המערך בו זמנית**
- **בוחרים את אותו מספר**

קיפאון!

חלוקת מספרים: ניסיון 3

מחכה שכל החוטים ב-entry יבחרו מספרים

```
Thread i:
  initially number[i]=0;
  choosing[i]=true;
  number[i]=max{number[1],…,number[n]}+1;
  choosing[i]=false;
  for all j≠i do
    wait until choosing[j]=false;
  for all j≠i do
    wait until number[j]=0 or
      ((number[j],j)>(number[i],i));
  critical section
  number[i]=0 // Exit critical section
```


בעיה עם האלגוריתם המתוקן

```
Thread 1:
number[1]=0
read{number[1],…,number[n]}
```

```
Thread 2:
number[2]=0
number[2]=max{number[1],…,number[n]}+1 // =1
wait until number[1]=0 or …
critical section
```

```
number[1]=1
wait until number[2]=0
       or (number[2],2)>(number[1],1))
critical section
```

אין מניעה הדדית!

תיאום בין תהליכים: שיטות מתקדמות

- **שימוש בחומרה למימוש מנעולים**
- **מנגנוני מערכת הפעלה לתיאום:**
  סמפורים, משתני תנאי

נכונות האלגוריתם

✓ **מבטיח מניעה הדדית.**

✓ **הוגן (אין הרעבה), כניסה לקטע הקריטי (פחות או יותר) לפי סדר הגעה.**

× **מסורבל**

- הרבה פעולות, הרבה משתנים
- פונקציה של מספר החוטים
- מבוסס על לולאות המתנה (busy wait)

דרכים אחרות לתיאום בין חוטים / תהליכים... עזרה מהחומרה.

חסימת פסיקות?

- **בעיות במערכת עם מעבד יחיד:**
  - תוכנית מתרסקת כאשר הפסיקות חסומות.
  - פסיקות חשובות הולכות לאיבוד.
  - עיכוב בטיפול בפסיקות I/O גורם להרעת ביצועים.
- **במערכות עם כמה מעבדים, לא די בחסימת פסיקות.**
  - לא ניתן לחסום פסיקות של מעבדים אחרים
  - חוטים יכולים לרוץ בו-זמנית (על מעבדים שונים)

מימוש מנעולים: חסימת פסיקות

```
lock_acquire(L):
  disableInterrupts()
  while L≠FREE do
    enableInterrupts()
    disableInterrupts()
  L = BUSY
  enableInterrupts()

lock_release(L):
  L = FREE
```

- **חסימת פסיקות מונעת החלפת חוטים ומבטיחה פעולה אטומית על המנעול**
- **למה מאפשרים פסיקות בתוך הלולאה?**

queue lock: מונע busy wait באמצעות ניהול תור של החוטים המחכים...

תמיכת חומרה במנעולים

test&set(boolvar)
- כתוב true ל- `boolvar` והחזר ערך קודם

- L = false – מנעול פנוי
- L = true – מנעול תפוס

```
lock_acquire(L):
  while test&set(L)
    do nop

lock_release(L):
  L = false
```


תמיכת חומרה מתקדמת

`compare&swap(mem, R1, R2)`
- אם בכתובת הזיכרון mem ערך זהה לרגיסטר R_1, כתוב את הערך אשר ברגיסטר R_2 והחזר הצלחה
- אחרת החזר כישלון

ממומש בארכיטקטורות IA32.

`load-linked / store conditional`
- **`LL(mem)`** – קרא את הערך בכתובת הזיכרון mem.
- **`SC(mem, val)`** – אם לא היתה כתיבה ל- **`mem`** מאז ה- **`LL(mem)`** האחרון שלך, כתוב ערך **`val`** ל- **`mem`** והחזר הצלחה (אחרת כשלון)

ממומש בהרבה ארכיטקטורות
- HP's Alpha, IBM's PowerPC, MIPS4000

Spinlocks

- **מימוש מנעול באמצעות busy waiting:**
  - בדוק האם המנעול תפוס (על-ידי גישה למשתנה).
  - אם המנעול תפוס, בדוק שנית.
  - גם בקופת-חולים...
- **מאוד בזבזני.**
  - חוט שמגלה כי המנעול תפוס מבזבז זמן cpu.
  - בזמן הזה החוט שמחזיק במנעול לא יכול להתקדם.

priority inversion: כאשר לחוט הממתין עדיפות גבוהה. ☹

מנגנוני תיאום גבוהים יותר

- **לנהל תור של החוטים הממתינים.** ☼
- **נמצא במנגנוני תיאום עיליים:**
  - סמפורים
  - משתני תנאי
  - ... מוניטורים

פעולות על סמפור

wait(semaphore)
- מקטין את ערך המונה ב-1
- ממתינים עד שערכו של הסמפור אינו שלילי
- נקרא גם P(), proben

signal(semaphore)
- מגדיל את ערך המונה ב-1
- משחרר את אחד הממתינים.
- נקרא גם V(), verhogen

סמפור

שני שדות:
- ערך שלם
- תור של חוטים / תהליכים ממתינים

[Dijkstra, 1968]

סמפור בינארי וסמפור מונה

- **סמפור בינארי**
  - ערך התחלתי =1; זה גם הערך המקסימאלי.
  - מאפשר גישה בלעדית למשאב (בדומה למנעול).
- **סמפור מונה**
  - ערך התחלתי N>0.
  - שולט על משאב עם N עותקים זהים.
  - חוט יכול לעבור ()wait בסמפור כל עוד יש עותק פנוי של המשאב.

יותר מידי חלב עם סמפורים

קוד זהה לשני החוטים.

סמפור `OKToBuyMilk`, **בהתחלה 1.**

```
wait( OKToBuyMilk);
if ( NoMilk ) {
   Buy Milk;      // critical section
      }
signal( OKToBuyMilk);
```

אם ערך הסמפור > 0 ⇐ ניתן להכנס לקטע הקריטי.

אם ערך הסמפור ≥ 0 ⇐ הסמפור תפוס, וערכו (בערך מוחלט) הוא מספר הממתינים

פתרון לבעיית יצרן / צרכן?

```
global variable
int c = 0;
```

```
Producer:
  repeat
    wait until (c < n);
    buff[pp] = new
  item;
    pp = (pp+1) mod n;
    c = c + 1;
  until false;
```

```
Consumer:
 repeat
    wait until (c ≥ 1);
    consume buff[cp];
    cp = (cp+1) mod n;
    c = c - 1;
  until false;
```

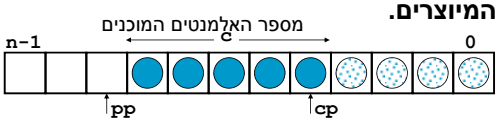
ואם ניגשים בו-זמנית ל-c???

דוגמה: בעיית יצרן / צרכן

שני חוטים רצים באותו מרחב זיכרון

- **היצרן** מיצר אלמנטים לטיפול (למשל, משימות)
- **הצרכן** מטפל באלמנטים (למשל, מבצע את המשימות)

מערך חסום (מעגלי) מכיל את העצמים המיוצרים.



יצרן / צרכן עם סמפורים

```
semaphore freeSpace,          מספר המקומות הפנויים
  initially n
Semaphore availItems,         מספר האיברים המוכנים
  intiailly 0
```

```
Producer:
  repeat
    wait( freeSpace);
    buff[pp] = new
  item;
    pp = (pp+1) mod n;
    signal(
  availItems);
  until false;
```

```
Consumer:
  repeat
    wait( availItems);
    consume buff[cp];
    cp = (cp+1) mod n;
    signal( freeSpace);
  until false;
```


דוגמה: **קוראים/כותבים**

חוטים קוראים וחוטים כותבים

- **מספר חוטים יכולים לקרוא בו-זמנית.**
- **כאשר חוט כותב, אסור שחוטים אחרים יכתבו ו/או יקראו.**

טבלת גישה

	Reader	Writer
Reader	✓	✘
Writer	✘	✘

מימוש (תיאורטי) של סמפורים?

```
struct semaphore_t {
   int value;
   queue waitQ;
}

void wait(semaphore_t *s){
   s->value--;
   if (s->value < 0){
        enQ(self, &s->waitQ);
        block }
}

void signal(semaphore_t *s){
   s->value++;
   if (s->value <= 0){
        P = deQ( &s->waitQ);
        wakeup(P) }
}
```


קוראים / כותבים עם סמפורים

```
int r = 0;                          מונה מספר הקוראים
semaphore sRead,                    מגן על מונה מספר הקוראים
   initially 1
semaphore sWrite,                   מניעה הדדית בין קוראים לבין כותבים
   initially 1                      (ובין כותבים לעצמם)

Writer:                             Reader:
   wait(sWrite)                        wait(sRead)
   [Write]                             r:=r+1
   signal(sWrite)                      if r=1 then
                                           wait(sWrite)
                                       signal( sRead)
                                       [Read]
                                       wait( sRead)
                                       r:=r-1
                                       if r=0 then
                                           signal(sWrite)
                                       signal( sRead)
```

היה busy-waiting ?

- **עדיין יש נעילה: על הגישה לתור השייך לסמפו**
  ול value
- **עדיין יש busy waiting...**

...אבל הקטע הקריטי קצר:

– רק לשים / להוריד אלמנט מתור.

– אפשר לתכנת כך שתהיה עבודה בו-זמנית בשני הקצוות של תור לא-ריק.

⇐ ההשלכות של busy-waiting מוקטנות.

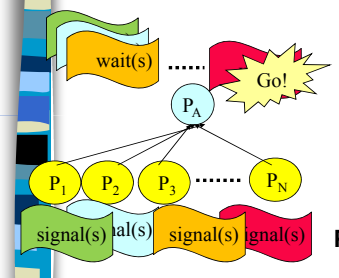
מימוש של סמפורים: תיקון

```
struct semaphore_t {
   int value;
   queue waitQ;
   lock_t l;
}
void wait(semaphore_t *s){
   lock(&s->l) ;
   s->value--;
   if (s->value < 0){
        enQ(self, &s->waitQ);
        unlock(&s->l) ;
        block }
   else unlock(&s->l) ;
}
```

```
void signal(semaphore_t *s){
   lock(&s->l) ;
   s->value++;
   if (s->value <= 0){
        P = deQ( &s->waitQ);
        wakeup(P) }
   unlock(&s->l) ;
}
```


ייצוג גרף תלויות עם סמפורים

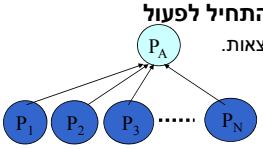
- **תהליך PA ממתין על-פי מספר התהליכים בהם הוא תלוי.**
- **תהליך שמסיים מבצע signal.**
- **האם האלגוריתם יעבוד גם אם P_i כלשהו יסיים ויבצע signal לפני ש PA קרא ל- wait?**



גרף תלויות

- **תהליכים P_1... P_N מבצעים חישוב**
- **תהליך P_A ממתין לתוצאות כל החישובים לפני שיוכל להתחיל לפעול**

– P_A מחשב את ממוצע התוצאות.



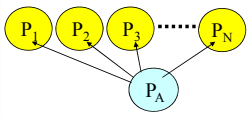
- **כיצד ניתן להבטיח ש P_A יתחיל לפעול רק לאחר שכל האחרים יסיימו?**

– סדר הסיום לא ידוע מראש.

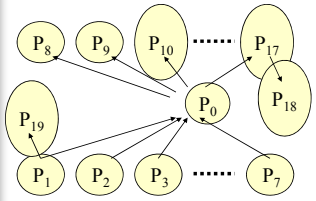
חסרונות של סמפורים

- **לא מפרידים:**
  - נעילה.
  - המתנה.
  - ניהול משאבים.
- **סמפורים אינם מספקים אבסטרקציה טובה**
  - קשר בין המנעול לבין המבנה המוגן על ידו אינו ברור מאליו
- **רעיון מודרני יותר: נקודות מפגש במשתני תנאי...**

גרפי תלויות אחרים



- **לפעמים המצב הוא הפוך.**
  - N תהליכים ממתינים לתהליך אחד לפני שיוכלו להתחיל לפעול.



- **ייתכנו גם מצבים מורכבים בהרבה.**
  - לא תמיד מספיק סמפור בודד.

משתני תנאי: הערות

- **כאשר תהליך מקבל signal הוא אינו מקבל את המנעול באופן אוטומטי, ועדיין צריך לחכות להשגתו**
  - mesa-style (ולכן החוט המוער חייב לבדוק את התנאי מחדש)
- **משתני תנאי הולכים נפלא עם מנעולים.**
  - wait(cond,&lock) קודם משחררת את המנעול lock.
- **בניגוד לסמפורים, signal לא זוכר היסטוריה.**
  - signal(cond) הולך לאיבוד אם אין ממתינים על cond.

פעולות על משתני תנאי

- wait(cond,&lock):
  - שחרר את המנעול (חייב להחזיק בו).
  - המתן לפעולת signal.
  - המתן למנעול (כשחוזר מחזיק במנעול).
- signal(cond):
  - הער את **אחד** הממתינים ל cond, אשר עובר להמתין למנעול.
  - הולך לאיבוד אם אין ממתינים.
- broadcast(cond):
  - הער את כל התהליכים הממתינים.
  - עוברים להמתין למנעול.
  - הולך לאיבוד אם אין ממתינים.

יצרן / צרכן עם משתני תנאי

```
condition not_full,
  not_empty;
lock bLock;

producer:
lock_acquire(bLock);
while (buffer is full)
  wait(not_full,&bLock);
add item to buffer ;
signal(not_empty);
lock_release(bLock);
```

```
consumer:
lock_acquire(bLock);
while (buffer is empty)
  wait(not_empty,&bLock
  );
get item from buffer
signal(not_full);
lock_release(bLock);
```


דוגמא – מימוש תור

```
lock QLock ;
condition notEmpty;

Enqueue (item):
lock_acquire( QLock)
put item on queue
signal(notEmpty)
lock_release( QLock)

Dequeue (item):
lock_acquire( QLock)
while queue empty
  wait(notEmpty
  ,&QLock)
remove item from queue
lock_release( QLock)
```

- **ממתינים כאשר התור ריק.**
- **נעילה להגן על הגישה לנתונים.**
- **קטע קריטי קצר.**
- **משתנה תנאי מאפשר לחכות עד שיתווסף איבר לתור, מבלי לבצע busy-wait.**
- **למה צריך while?**

משתני תנאי + מנעול ≅ מוניטור

- **ההקשר המקורי של משתני תנאי** [C.A.R. Hoare, 1974]
- **אובייקט (במובן של שפת-תכנות** (object-oriented)**, הכולל פרוצדורת אתחול וגישה.**
  - הגישה לאובייקט מקנה שליטה במנעול (באופן לא-מפורש)
  - שליחת signal משחררת את המנעול ומעבירה את השליטה בו למקבל ה signal.

נתמך בכמה שפות-תכנות מודרניות (ובראשן, Java וכל השפות בפלטפורמת .NET. כגון C#, Visual Basic).

מנגנוני תיאום ב POSIX

- **אובייקטים לתיאום**
  - נמצאים בזיכרון המשותף.
  - נגישים לחוטים של תהליכים שונים.
- **mutex locks (`pthread_mutex_xxx`)**
  - יצירה, השמדה: init, destroy
  - lock, unlock, trylock.
- **condition variables (`pthread_cond_xxx`)**
  - יצירה, השמדה: init, destroy
  - wait, signal, broadcast.

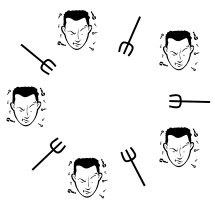
מנגנוני תיאום ב-Windows NT

- **כל רכיב במערכת ההפעלה הוא אובייקט תיאום**
  - ניתן להמתין ו/או לשחרר
- **אובייקטים מיוחדים**
  - mutex– מנעול הוגן
  - event – משתנה תנאי (עם אפשרות ל-broadcast כדי להעיר את כל הממתינים)
  - semaphore – סמפור מונה, לא הוגן
  - critical section – light-weight mutex המיועד לחוטים באותו תהליך

קיפאון
Deadlock

דוגמא – הפילוסופים הסועדים

- **פיתרון נאיבי בעזרת סמפורים**
  - נקצה סמפור fork[i] לכל מזלג
  - p(fork[i-1]); p(fork[i]); *eat*; v(fork[i-1]); v(fork[i]);
- **בעיה**
  - כל פילוסוף תופס את המזלג השמאלי בו-זמנית ותקוע בהמתנה לימני
- **קיפאון!**



נושאים

- **בעית הקיפאון**
  - דוגמא
  - תנאים לקיפאון
- **טיפול בקיפאון**
  - מניעה, זיהוי, התחמקות
  - אלגוריתמים למניעה והתחמקות

הגדרה

קפאון הוא מצב בו אף אחד מהתהליכים/חוטים במערכת אינו יכול לבצע את הפקודה הבאה בתכנית שלו.

אבחנה: כל אחד מארבעת התנאים הבאים חייב להתקיים כדי שייווצר קיפאון:

תנאים לקיפאון

1. **מניעה הדדית**
   - ישנם משאבים שרק תהליך אחד יכול לעשות שימוש בהם בו-זמנית
2. **החזק והמתן**
   - ישנו תהליך המחזיק משאב ומחכה למשאב אחר שבשימוש אצל תהליך אחר
3. **לא ניתן להפקיע משאבים**
   - לא ניתן להפקיע משאב מתהליך ללא הסכמתו
4. **המתנה מעגלית**
   - קיימים n תהליכים כך ש-P_i מחכה למשאב המוחזק ע"י $P_{(i+1) \bmod n}$ $(0<=i<=n-1)$

דרכי התמודדות עם קיפאון

- **מניעה (prevention)**
  - לקבוע מדיניות שתימנע קיפאון
- **התחמקות (avoidance)**
  - להשתמש במידע מוקדם על צרכיו של כל תהליך כדי להראות שלא קורה
- **גילוי והתאוששות (detection and recovery)**
  - מאפשרים למערכת להגיע לקיפאון
  - כאשר זה קורה, המערכת תגלה זאת ותיחלץ מהמצב
- **להתעלם מהבעיה**
  - להניח שהמצב כמעט ואינו קורה...

מניעת קיפאון – Prevention

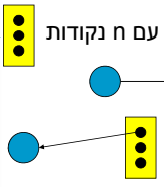
1. **מניעה הדדית**
   - למשל, בנית מערכת שבה כל המשאבים ניתנים לשימוש משותף בו-זמנית
2. **החזק והמתן**
   - למשל, לא נאפשר לתהליך להמתין למשאב כאשר הוא מחזיק משאב אחר
3. **לא ניתן להפקיע משאבים**
   - למשל, נאפשר להפקיע משאבים מתהליך שכרגע בהמתנה
4. **המתנה מעגלית**
   - נקבע מדיניות שתמנע המתנה מעגלית

מניעת קיפאון
מניעה המתנה מעגלית

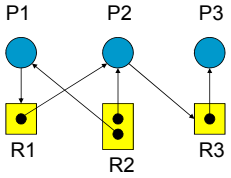
- **נקבע סדר מלא בין המשאבים**
  - F(tape)=1, F(printer)=2, F(scanner)=3, …
- **תהליכים יבקשו משאבים רק בסדר עולה**
  - תהליך שמחזיק את ה-printer לא יוכל לבקש את ה-tape
- **או... תהליך שמבקש משאב מסדר נמוך חייב לשחרר קודם משאבים מסדר גבוה**

גרף בקשות-הקצאות

- **נתאר מצב רגעי של מערכת ע"י הגרף המכוון הבא**
  - כל תהליך מתואר כעיגול
  - משאב בעל n עותקים מתואר כמלבן עם n נקודות
  - קשת מתהליך למשאב
    - התהליך ממתין למשאב
  - קשת מ(עותק של) משאב לתהליך
    - עותק של המשאב הוקצה לתהליך



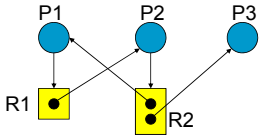
גרף בקשות-הקצאות
דוגמא



- **P1 מחזיק עותק של R2 ומחכה ל-R1**
- **P2 מחזיק עותקים של R1 ו-R2 ומחכה ל-R3**
- **P3 מחזיק עותק של R3**

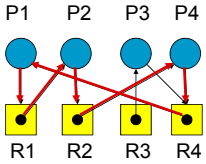
גילוי קיפאון

- **משפט: אם יש קיפאון אזי קיים מעגל מכוון בגרף בקשות-הקצאות**
  – נובע מתנאי "החזק והמתן" ו-"המתנה מעגלית"
- **אבל... קיום מעגל אינו תנאי מספיק לקיפאון!**



גילוי קיפאון
עותק יחיד של כל משאב

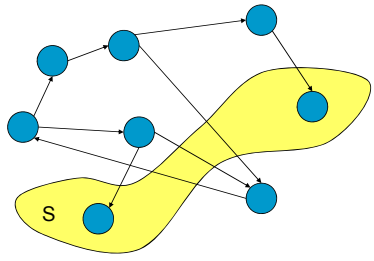
- **משפט:בהנחה שיש עותק יחיד של כל משאב, קיים מעגל מכוון בגרף בקשות-הקצאות אם ורק אם יש קיפאון**
  – המתנה מעגלית



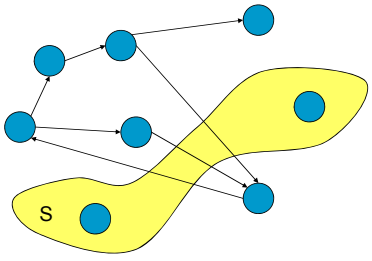
גילוי קיפאון - עותק יחיד של כל משאב

- **זיהוי מעגל**
  – S היא קבוצת כל הצמתים ללא קשתות יוצאות
- **אלגוריתם**
  – כל עוד יש קשת i♦j הנכנסת לצומת j, עבור j כלשהו ב-S
    - מחק את הקשת i♦j מהגרף
    - אם ל-i אין קשתות יוצאות, הוסף את i ל-S
  – אם בסוף S לא מכיל את כל הצמתים בגרף, אזי יש מעגל C
    - מהו היחס בין הצמתים ב-C ו-S?
- **סיבוכיות**
  – O(|E|) כאשר E היא קבוצת הקשתות

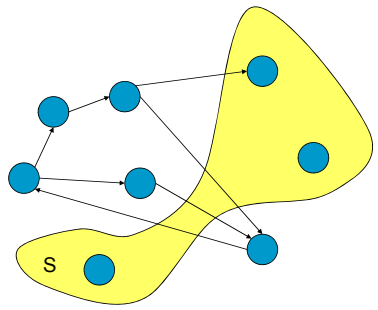
גילוי מעגל – דוגמא



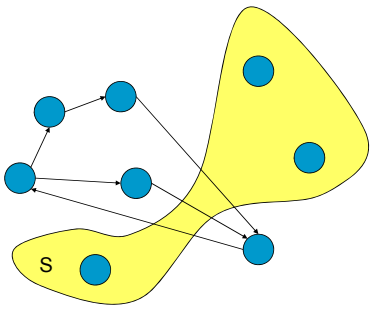
גילוי מעגל – דוגמא



גילוי מעגל – דוגמא



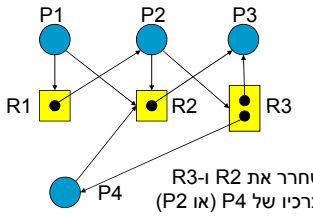
גילוי מעגל – דוגמא



זיהוי קיפאון
מספר עותקים מכל משאב

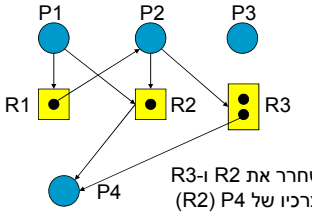
- **כפי שראינו, גילוי מעגל אינו תנאי מספיק**
  - דרוש כלי חזק יותר
- **פריקות של גרף בקשות-הקצאות**
  - ניתן לצמצם צומת/תהליך אם ניתן לספק את כל בקשותיו
    - בעת הצמצום משחררים את כל המשאבים שברשותו
  - גרף הוא פריק אם ניתן לצמצם תהליך כלשהו
  - גרף הוא פריק לחלוטין אם קיימת סדרת צמצומים שבסיומה כל צמתי הגרף מבודדים

צמצום גרף בקשות-הקצאות



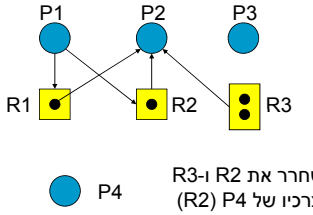
- ניתן לצמצם את P3 ולשחרר את R2 ו-R3
- כעת ניתן לספק את כל צרכיו של P4 (או P2)
- אך לא של P1
- אחר-כך ניתן לצמצם את P2 ו-P1

צמצום גרף בקשות-הקצאות



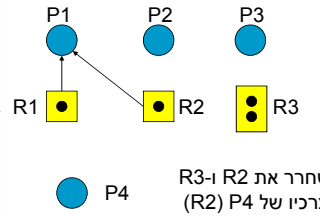
- ניתן לצמצם את P3 ולשחרר את R2 ו-R3
- כעת ניתן לספק את כל צרכיו של P4 (R2)
- אך לא של P1
- אחר-כך ניתן לצמצם את P2 ו-P1

צמצום גרף בקשות-הקצאות



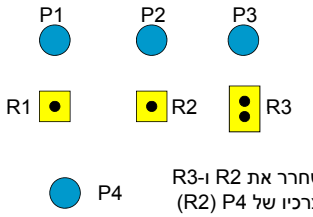
- ניתן לצמצם את P3 ולשחרר את R2 ו-R3
- כעת ניתן לספק את כל צרכיו של P4 (R2)
- אך לא של P1
- אחר-כך ניתן לצמצם את P2 ו-P1

צמצום גרף בקשות-הקצאות



- ניתן לצמצם את P3 ולשחרר את R2 ו-R3
- כעת ניתן לספק את כל צרכיו של P4 (R2)
- אך לא של P1
- אחר-כך ניתן לצמצם את P2 ו-P1

צמצום גרף בקשות-הקצאות



- ניתן לצמצם את P3 ולשחרר את R2 ו-R3
- כעת ניתן לספק את כל צרכיו של P4 (R2)
- אך לא של P1
- אחר-כך ניתן לצמצם את P2 ו-P1

הגרף פריק לחלוטין ⇦

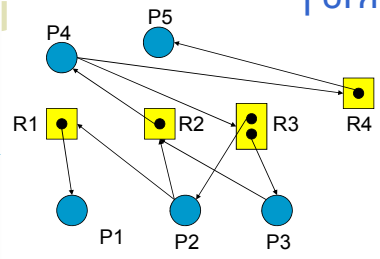
סדרת צמצומים מירבית בגרף בקשות-הקצאות

- **הגדרה**
  - סדרת צמצומים בגרף בקשות-הקצאות היא מירבית אם לא ניתן להמשיך לצמצם צמתים
- **טענה**
  - כל סדרות הצמצומים המרביות מביאות לאותו גרף בלתי פריק

סדרת צמצומים מירבית בגרף בקשות-הקצאות

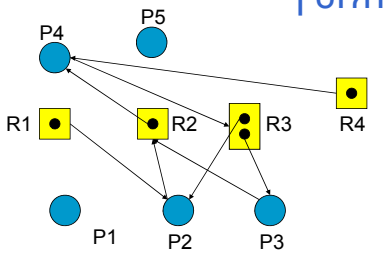
- **הוכחה**
  - נסתכל על שתי סדרות צמצומים מירביות
    - p1, …, pk ו- q1, …, qr
  - נניח (בדרך השלילה) שהן אינן מכילות את אותם התהליכים
  - נתבונן בתהליך הראשון pi שאינו מופיע ב- q1, …, qr
  - אחרי q1, …, qi-1 ניתן לצמצם את pi
  - ולכן אחרי q1, …, qr עדיין ניתן לצמצם את pi
  - סתירה!

פריקות לחלוטין



- ניתן לצמצם את P1, P5

פריקות לחלוטין



- לאחר מכן,לא ניתן לצמצם תהליכים אחרים
- הגרף אינו פריק לחלוטין!

פריקות לחלוטין וקפאון

- **משפט (עבור מקרה של מספר עותקים מכל משאב)**
  - גרף בקשות-הקצאות הוא פריק לחלוטין אם ורק אם אין קיפאון
- **הוכחה**
  - אם הגרף פריק לחלוטין סדרת צמצום מירבית משרה סדרת הקצאות שתביא לסיום כל התהליכים
  - אם הגרף לא פריק לחלוטין, מסדרת צמצום מירבית כלשהי יתקבל גרף לא פריק
    - קבוצת הצמתים הלא מבודדים משרה קבוצת תהליכים שאינם יכולים להתקדם

אלגוריתם לזיהוי קיפאון מספר עותקים מכל משאב

- **הרעיון: בדיקת פריקות גרף הקצאות-בקשות**
  - נאתחל קבוצה L עם כל הצמתים שאינם מחכים למשאבים
  - עבור כל תהליך Pi ב-L
    - שחרר את כל המשאבים התפוסים ע"י Pi
    - עבור כל תהליך Pk אשר ניתן לספק את כל צרכיו
      - הוסף את Pk ל-L
  - אם בסיום L אינו מכיל את כל התהליכים אזי יש קיפאון
- **סיבוכיות**
  - $O(m\ n^2)$
    - m – מספר המשאבים
    - n – מספר התהליכים

החלצות מקיפאון

- **אפשרויות החלצות מקיפאון**
  - ביטול כל התהליכים המצויים במצב קיפאון
  - ביטול תהליכים אחד אחד עד להחלצות
  - הפקעת משאבים
- **מציאת קבוצה אופטימלית (של תהליכים או משאבים) היא NP-Complete**
  - feedback vertex set
    - קבוצת צמתים מינימלית שתהפוך גרף לחסר מעגלים
  - feedback arc set
    - קבוצת קשתות מינימלית שתהפוך גרף לחסר מעגלים

התחמקות מקיפאון – Deadlock Avoidance

- **דואגים להשאיר את המערכת במצב בטוח**
  - תהליכים מצהירים על כוונתם לבקש משאבים
  - אסור לתהליך לבקש משאב שלא הצהיר עליו
  - מצב הוא בטוח אם קיימת סדרת הקצאות לכל המשאבים שהוצהרו, מבלי להיכנס לקיפאון
- **התחמקות מקיפאון**
  - בעת בקשה בפועל, אם היענות לבקשה תכניס את המערכת למצב לא בטוח, הבקשה נדחת
- **אלגוריתם שמרני**
  - דורש הנחות מחמירות לגבי דרישות התהליכים

התחמקות מקיפאון דוגמא

- **תהליכים P1 ו-P2 מצהירים על כוונה לבקש את המדפסת ואת כונן הסרטים**
  - תהליך P1 מבקש את המדפסת
    - הבקשה נענית
  - תהליך P2 מבקש את כונן הסרטים
    - הבקשה נדחית... אם P1 יבקש את המדפסת בהמשך, ייווצר קיפאון

התחמקות מקיפאון אלגוריתם הבנקאי

- **עקרונות האלגוריתם**
  - תהליכים המגיעים למערכת מצהירים על כוונתם המירביות לבקשות משאבים – לכל משאב
  - אם תהליך מבקש עותק ממשאב,הבקשה תיענה אם
    - אין חריגה מההצהרות המקוריות
    - יש מספיק עותקים פנויים
    - ההקצאה משאירה את המערכת במצב בטוח

אלגוריתם הבנקאי - פירוט

- **משתנים**
  - **available[1..m]** – מספר עותקים פנויים מכל משאב
  - **max[1..n, 1..m]** – מספר עותקים מקסימלי שתהליך יכול לבקש (הצהרה)
  - **allocation [1..n, 1..m]** – מספר העותקים של המשאב שכעת מוקצים לתהליך
  - **need[i,j]=max[i,j]-allocation[i,j]**
    - מספר העותקים של המשאב שהתהליך i עלול לבקש
- **פעולות וקטוריות**
  - X ☯ Y – אמ"ם X[i] ☯ Y[i] לכל 1 ☯ i ☯ m

אלגוריתם הבנקאי פירוט

```
L = ℵ
while there is a Pi∉L and
    need[i] ☯ available do
  available:=available+allocation[i]
  add Pi to L
end
if L contains all processes then
  this is a safe state
```


אלגוריתם הבנקאי – דוגמא

- **האם המצב הבא של המערכת הוא "בטוח"?**

	allocation			max			available			need		
	A	B	C	A	B	C	A	B	C	A	B	C
P1	0	1	0	3	1	1				3	0	1
P2	2	1	0	2	1	4				0	0	4
P3	1	0	3	3	2	3				2	2	0
Total	3	2	3				0	3	4			

- **ניתן לצמצם את P2**
  - לאחר שחרור משאביו, ניתן לצמצם את P3 ו-P1

סיכום

- **רוב מערכות ההפעלה אינם דואגות למנוע קיפאון**
  - פעולה כבדה
  - האחריות נשארת אצל המתכנת
  - הריגת תהליך מאפשרת להיחלץ מקיפאון
    - אך עלולות להשאיר את המערכת במצב לא תקין
- **ב-Windows NT**
  - פקודת WaitForMultipleObjects
  - מקצה את כל משאבים יחד, אם כולם פנויים

מהן פסיקות?

- **פסיקות (interrupts) הן אירועים הגורמים למעבד להשעות את פעילותו הרגילה ולבצע פעילות מיוחדת.**
- **שימושים:**
  - מימוש קריאות מערכת-הפעלה.
  - קבלת מידע וטיפול ברכיבי חומרה (שעון, חומרת קלט/פלט...).
  - טיפול בתקלות (חלוקה באפס, גישה לא חוקית לזיכרון...).
  - אכיפת חלוקת זמן מעבד בין תהליכים

פסיקות

- **סוגי פסיקות**
- **טיפול בפסיקות**
- **דוגמא: קלט בעזרת פסיקות**

סוגי פסיקות

- **פסיקות אסינכרוניות נוצרות על-ידי רכיבי חומרה שונים:**
  - ללא תלות בפעילותו הנוכחית של המעבד.
  - למשל: פעימה של שעון המערכת, לחיצה על מקש במקלדת...
- **פסיקות סינכרוניות נוצרות בשל פעילות של המעבד:**
  - למשל, כתוצאה מתקלות שונות.
  - נקראות גם חריגות (exceptions).
- **סוג אחר של פסיקות סינכרוניות הוא פסיקות יזומות, אשר נקראות גם פסיקות תוכנה (software interrupts):**
  - נוצרות על-ידי הוראות מעבד מיוחדות (למשל int).
  - שימושים: מימוש קריאות מערכת, debugging.

הזדמנות שנייה

- **ביצוע הוראות מסוימות יכול להכשל ולגרום לפסיקה סינכרונית**
- **מעבדים מודרניים מאפשרים למערכת ההפעלה, על-ידי טיפול בפסיקה, לתת להוראות "הזדמנות שנייה" להתבצע (נקראות restartable instructions)**
  - ההוראה נכשלה ⇦ פסיקה
  - מערכת-ההפעלה מטפלת בפסיקה
  - מריצים שוב אותה ההוראה (שהפעם אמורה להצליח).
- **שימושי למימוש זיכרון וירטואלי**
  - בהמשך הקורס

אם המעבד מגלה שהיו פסיקות לאחר ביצוע פעולה...

- **שומר על המחסנית את כתובת החזרה.**
  - בד"כ כתובת ההוראה הבאה.
  - במקרה של "הזדמנות שנייה", כתובת ההוראה שגרמה לפסיקה.
  - ב Linux, הערכים נשמרים על מחסנית הגרעין של התהליך, ולא מחסנית המשתמש.
- **אולי מידע נוסף.**
  - רגיסטרים מיוחדים – למשל רגיסטר הדגלים
  - מידע נוסף לגבי הפסיקה, למשל, סוג השגיאה המתמטית.
- **מפעיל את שגרת הטיפול של הפסיקה**
  - ב-Linux ניתן להפעיל מספר שגרות בעקבות פסיקה
  - השגרה מוגדרת עבור (התקן, פסיקה) ולא עבור (פסיקה) בלבד
  - מספר התקנים יכולים לחלוק את אותה פסיקה.

איך מפעילים את שגרת הטיפול בפסיקה?

- **לכל פסיקה שגרת טיפול משלה, הפועלת בהתאם לסוג הפסיקה**
  - למשל, קוראת את התו שנלחץ במקלדת ושומרת אותו במקום מתאים בזיכרון
- **לכל פסיקה יש מספר שונה (נקרא לפעמים interrupt vector).**
- **למעבד יש גישה לטבלה של מצביעים לשגרות טיפול בפסיקה.**
  - מספר הכניסות בטבלה כמספר וקטורי הפסיקות.
    - במעבדי IA16, הטבלה שמורה בכתובת 0000:0000 (ממש בתחילת הזיכרון) מכילה 256 כניסות של 4 בתים כ"א (4 בתים = גודל מצביע), סה"כ 1 KB.
  - טבלת הפסיקות מאותחלת על-ידי חומרת המחשב (ה- BIOS).
  - בטעינה, מערכת-ההפעלה מעדכנת את הטבלה, ומחליפה חלק מהשגרות.
  - מערכות-הפעלה מודרניות לא מסתמכות כלל על שגרות הטיפול של ה- BIOS, ומחליפות את כל הטבלה.

בעיות בטיפול בפסיקות

- **פסיקות א-סינכרוניות יכולות להגיע בכל זמן שהוא.**
  - המשתמש יכול ללחוץ על המקלדת בכל רגע...
  - מצד אחד, חשוב לטפל בהן במהירות האפשרית...
    - בעיקר כדי לא לעכב התקני קלט-פלט.
    - זמן הטיפול עשוי להיות רב (למשל, מידע שהגיע מהדיסק).
  - מצד שני, טיפול שדורש זמן רב ומתחיל מיד, מתבצע על חשבון התהליך הנוכחי
- **כמו כן, יש לצמצם את זמן חסימת הפסיקות למינימום**
- **בנוסף, יש לתמוך בקינון: פסיקה שמגיעה בזמן הטיפול בפסיקה אחרת**

מי מריץ את שגרת הטיפול בפסיקה?

- **הפסיקה קורה בזמן שתהליך כלשהו רץ.**
- **הפסיקה אינה בהכרח קשורה לתהליך זה.**
  - למשל, נגרמה על-ידי חומרה שתהליך אחר משתמש בה.
- **קוד הטיפול בפסיקה שייך ל-kernel ולא לתהליך.**
- **ועדיין, קוד הטיפול מורץ בהקשר של התהליך הנוכחי...**
  - לא מתבצעת החלפת תהליכים כדי לטפל בפסיקה!
  - בדרך-כלל, קוד הטיפול בפסיקה לא מתייחס כלל לתהליך הנוכחי, אלא למבני נתונים גלובליים של המערכת.

פתרון (חלקי): טיפול דו-שלבי בפסיקות אסינכרוניות

- **חלק עליון (top half):**
  - מורץ כשגרת הטיפול בפסיקה, באופן מהיר ככל האפשר.
  - רושם (בתור מיוחד) את העובדה שיש לתת לחומרה הרלוונטית טיפול הולם (למשל, לקרוא את כל המידע שהגיע מהדיסק).
- **חלק תחתון (bottom half):**
  - מורץ בזמן מאוחר יותר, ומבצע את "העבודה השחורה" של הטיפול בפסיקה.
  - פסיקות אחרות יכולות להתרחש תוך-כדי פעולתו.
- **המערכת שומרת תור של שגרות להרצה (חלקים תחתונים).**
  - שגרת הטיפול בפסיקה פשוט מוסיפה לתור זה את החלק התחתון המתאים.
  - בנקודות זמן מסוימות (נניח, בכל החלפת הקשר) המערכת בודקת אם התור אינו ריק, ומריצה את כל השגרות הממתינות.

שרשור פסיקות: שינוי שגרת הטיפול

- **ניתן לשנות את שגרת הטיפול בפסיקה בזמן פעולת המערכת.**
  - החלפת השגרה נעשית פשוט על-ידי שינוי הטבלה.
- **אבל מה אם רוצים להוסיף לשגרה הנוכחית, מבלי לבטל אותה?**
  - למשל, שינוי שגרת הטיפול במקלדת להשמיע קליק לאחר כל לחיצת מקש.
  - בהחלפת השגרה הקיימת, יושמעו קליקים, אבל הלחיצות עצמן לא תטופלנה!
- **לפני שמעדכנים את טבלת הפסיקות, שומרים את כתובת השגרה הנוכחית.**
- **שגרת הטיפול החדשה קוראת לשגרה הישנה (בכתובת שנשמרה):**
  1. קודם קוראים לשגרה המקורית, ורק אחר-כך מבצעים את השגרה החדשה.
  2. **או** קודם השגרה החדשה, ורק אחר-כך מבצעים את השגרה המקורית.
     - מאפשר לא לבצע את הפעולה המקורית במקרים מסוימים.

טיפול בפסיקות במערכת מרובת-מעבדים

- **פסיקות סינכרוניות: כל מעבד מטפל בפסיקות שיצר הקוד שהוא מריץ.**
- **פסיקות אסינכרוניות: איזה מעבד יתעכב כדי לטפל בקלט-פלט?**
  - חלוקה סטטית: לכל מעבד קבוצה של (סוגי) פסיקות שהוא אחראי עליה.
  - חלוקה דינאמית: המעבד שמריץ את התהליך עם העדיפות הגרועה יותר, יטפל בפסיקה
    - round-robin במקרה של שוויון
- **בנוסף, במערכות מרובות-מעבדים יש גם פסיקות המשמשות להעברת מידע בין המעבדים עצמם**
  - Inter-Processor Interrupts
  - מעבד רואה IPIs של מעבדים אחרים כפסיקות א-סינכרוניות רגילות.

דוגמא: קריאה מהדיסק 1

- **המשתמש מבצע fread כדי לקרוא מידע מהקובץ**

fread(...)

מציאת המקום על הדיסק בו שמור המידע (ע"י קריאת טבלאות, למשל)

מעבירים לדיסק בקשה: נא לקרוא את המידע מכתובת זו-וזו, ולשמור במיקום זה-וזה בזיכרון...

דוגמא: קריאה מהדיסק 2

- **התהליך שביצע את הקריאה ממתין**
- **בינתיים, רצים תהליכים אחרים...**

פסיקה אסינכרונית: הדיסק סיים את בקשת הקריאה (המידע המבוקש כבר נמצא בזיכרון)

דוגמא: קריאה מהדיסק 3

- **מופעלת השגרה לטיפול בפסיקות הדיסק.**
- **מוסיפה את החצי התחתון הרלוונטי לרשימה, ובזאת מסתיים הטיפול בפסיקה.**
- **בהמשך, מורץ החצי התחתון:**
  - בודק שהמידע שהדיסק העביר לא כולל דיווח על שגיאות.
  - מעביר את התהליך שביצע את הקריאה ממצב המתנה אל תור המוכנים.
  - אולי גם מעדכן את העדיפות של אותו תהליך.
- **התהליך חוזר מהקריאה fread ומקבל את המידע.**


סיכומים וחזרות לחומר הלימוד
תלמידים
צוות הוראה
הכל ברור
Any Questions ?
All Rights Reserved - שאול קובל מערכות מחשבים
229


R&D Advisers
Web & Net Design - System Analyst
Computer Engineering Teach Advice
Saul Coval Computers
כל הזכויות שמורות
http://www.coval.net
230