

מערכות הפעלה

סיכון הרצאות ותרגולים

ע"פ פרופ' דוד חי, מרץ 2019

מסכמים: עידן גבאי ומעין שטרית

5.....	הרצאה 1
5.....	המחשב ומערכות הפעלה
6.....	מקביליות
7.....	מערכות הפעלה כיום
9.....	תרגול 1
9.....	זיכרון המחשב
10.....	Debugging
11.....	הרצאה 2
11.....	User / Kernel Mode, System Calls
13.....	שכבות מערכות הפעלה
14.....	Interrupts
16.....	תרגול 2
16.....	שימוש נכון בזיכרון
17.....	Kernel Mode vs User Mode
17.....	OS - מערכת הפעלה
17.....	Interrupts - פסיקות
20.....	הרצאה 3
20.....	ניהול תהליכיים
21.....	Context
22.....	Processes – תהליכיים
23.....	Process Control Block (PCB)
24.....	Scheduling
26.....	תרגול 3
26.....	סיג널ים – Signals
27.....	פונקציות שימושיות לניהול סיגナルים
29.....	Threads
33.....	הרצאה 4
33.....	Threads - תהליכיונים
37.....	מקביליות אמיתית
38.....	Synchronization
41.....	תרגול 4
41.....	Thread Pools
43.....	ניהול תהליכיונים
44.....	Mutex
46.....	Monitors (Conditional Variables)
49.....	Semaphores

הרצאה 5	50
אלגוריתמים לפתרון	50
האלגוריתם של פיטרסון	50
Lamport's bakery algorithm	51
Read-Modify-Write Instructions	53
Burn's Algorithm	53
Synchronization Primitives	54
Coordination	55
תרגום 5	56
דוגמאות לביעות סינכרון	56
בעיות סינכרון קלאסיות	59
שאלות ממבחןים שהופיעו בתרגול	62
הרצאה 6	64
פתרון בעיות בעזרת סטטוסים	64
מוניטור	70
הרצאה 7	71
Monitor	71
Memory Hierarchy	72
Cache	74
תרגום 6	77
אסטרטגיות לניהולeadlock	77
Cache	79
הרצאה 8	82
Set Associativity	82
2-Way Set Associativity	83
index, tag, offset	84
סיבות ל Cache Miss	84
Block Replacement	85
תרגום 7	88
Memory Management	89
Segmentation	90
הרצאה 9	91
Pseudo-LRU	91
Memory Management	91
Virtual Memory	96
Thrashing	97
תרגום 8	98
Memory Management	98
Structure of the Page Table	101
שאלות חזרה על תרגום כתובות	103
הרצאה 10	107
Hierarchical Page Table	108
Memory Accesses and Page Fault	109

110.....	Hash Page Table
110.....	Inverted Page Table
112.....	שאלות ממבחןים
115.....	Paging "חוד'ם ל- Replacement Algorithm
117.....	תרגול 9
117.....	Inverted Page Tables
118.....	File Systems
120.....	File Descriptor (FD)
122.....	שאלות ממבחןים
126.....	הרצאה 11
126.....	File Systems
128.....	Hard Link & Symbolic Link
129.....	File Protection
130.....	File Access
133.....	תרגול 10
133.....	פרוטוקול תקשורת – Communication Protocol
135.....	Transport layer
136.....	Sockets
137.....	Streams
138.....	Socket's Address
141.....	Domain Name Service (DNS)
142.....	Server – שלבים להקמת נדש Socket Programing
144.....	The Client
148.....	הרצאה 12
150.....	Open files of a process
151.....	– מימוש ספריות
153.....	Disk Scheduling
154.....	Scheduling
157.....	מימושים של Schedulers
159.....	תרגול 11
159.....	Scheduling
167.....	Parallel Systems Scheduling
169.....	תרגול 12
169.....	System Calls
171.....	Process Control Calls
174.....	Memory Management calls
175.....	File Access Calls
178.....	File & Directory Management Calls
178.....	Hard Links
179.....	Device Management Calls
179.....	פונקציות נוספת
181.....	הרצאה 13
181.....	Schedulers

184	I/O (Input/Output)
188	תרגול 13
188	Interrupts
189	Signals
190	שאלות מבחנים מושנים קודמות
194	הרצאה 14
194	נקודות אחרונות לגבי התקני O/I
196	וירטואלייזציה
204	Containers
204	תרגול 14

המחשב ומערכת הפעלה

מערכת מחשב בנויה ממשתמשים (בני אדם או מכונות אחרות), אפליקציות (תוכנות), חומרה ומידע. מערכת הפעלה זו תוננה שספקת סביבה נוחה ובוטחה עבור תוכנות אחרות. מערכת הפעלה היא הגורם המתויר בין החומרה למשתמשים, רצה ברקע באופן תמיד ומשתמשת בחומרת המחשב באופן עיל. לעיתים יותר מערכות הפעלה אחת תרוץ על המחשב (Virtualization) ומשתמשת בחומרת המחשב באופן עיל. ולפעמים מערכות הפעלה تعمل על יותר ממעבד אחד (Multi – core architectures)

שינויים משמעותיים במערכות מחשב לאורך השנים:

1945 – 1955 : דור ראשון, מחשב שבוני מЛОחות וצינורות, ללא מערכת הפעלה.

1955 – 1965 : דור שני, ניקוב כרטיסיות והעברה ידנית שלahn למחשב אחר שייעוד את הפלט, למשל מדפסת. הcartisיות אוחדו לסלילים גדולים יותר, מה שהסב כמעט את העבודה (במוקם ללכנת למחשב אחר עבר כל כרטיס בנפרד). פעולה זו נקראת *Batching*.

1965 – 1980 : דור שלישי, מחשבים החלו לטפל ביותר מפעולה אחת בו זמנית.

1980 – present : דור רביעי, מחשבים אישיים, תקשורת, פלאפונים וכו'

present : דור חמישי, ריבוי מעבדים, וירטואלייזציה, ענן וכו'

Spooling (Simultaneous Peripheral Operation On – Line)

job pool, מבנה נתונים כלשהו שמערכות הפעלה משתמשות בו כדי לבחור (על פי שיקולים פנימיים) איזה עבודה תרוץ הבאה, כשהמטרה היא לקבל ניצול גבוה ככל הניתן של המעבד. *job queue*, מבנה נתונים ספציפי - תור, שמנהל את בחירת העבודות כך שמי שהגיע ראשון יוצא ראשון. (*FIFO*) בהקשר של מערכות הפעלה בד"כ מדובר ב-*Priority Queue*.

מנוחים חשובים:

זמן השהייה (Latency) – הזמן שלוקח לבצע עבודה או עיבוד מידע.

תפוקה (Throughput) – הקצב שבו מתבצעת העבודה, למשל כמות העבודה שבוצעה ביחידת זמן.

ニיטולות (Utilization) – שבר (בין 0 ל-1), הזמן שבו ה-*CPU* מבצע מטלות אמיתיות.

(מטלות לא אמיתיות הן מטלות שה-*CPU* חייב לבצע כדי לבצע מטלות אמיתיות. למשל אם להיות נכון בהערכתה זאת מטלת רצiosa, אז היליכה/נסיעה לאוניברסיטה זאת מטלת לא אמיתית, מטלת שאנוanno חייבים לעשות כדי שנוכל לבצע את המטלת האמיתית "לחיות נכון בהערכתה")

CPU Usage – הזמן שבו ה-*CPU* מבצע *Instructions*. ככל יותר הזמן שלוקח לו לבצע מטלות אמיתיות וגם מטלות לא אמיתיות. אך $CPU Usage \geq Utilization$. הרבה פעמים נתענין בניתולות אר מכיוון שאנוanno לא יכולים למדוד את זה, נבעוד עם ה-*CPU*.

תקורה (Overhead) – כמות המטלות הלא אמיתיות שהמחשב או מערכת הפעלה הינו חייבים לבצע כדי לאפשר את הפעלת המטלות האמיתיות. תקורה יכולה להימדד בזמן או בזכרון (כמויות המידע שצריך לשמר כדי לבצע את המטלת, למשל בזמן פתיחתקובץ).

מקבילות

ב-1965 מחשבים החלו להריץ מספר אפליקציות במקביל, למשל מוסיקה והקלדה בו זמןית. כדי לאפשר זאת, המעבד מטפל בהוראות של מספר אפליקציות בו זמןית ובכך ממקסם את הניתולת ומשפר את זמן התגובה. בפועל, ההוראות של מספר אפליקציות נשמרות בזיכרון, וה-*CPU* מחשב את ההוראות לפי אחת מהגישות הבאות:

- כאשר תוכנית מגיעה להוראה שמחכה לתשובה (למשל קלט מהמשתמש), היא "תועטר" על השליטה ב-*CPU* ותוכנית אחרת תרוץ ביניים.
- בנוסף לאפשרות שבתוכנית מסוימת יכולה לוווטר על ה-*CPU*, *Time Sharing* בגישה זו מערכת הפעלה תחליף "בכח" בין פעולות אם היא תזהה שההחלפה תשפר את הניתולת.

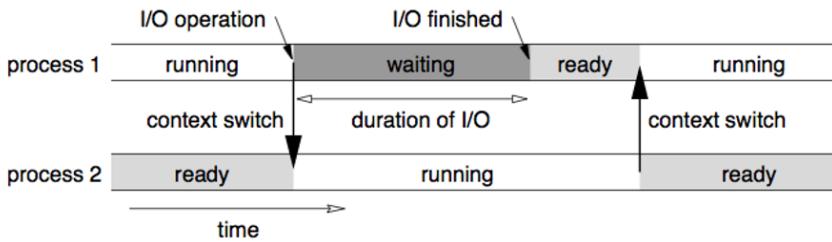
הערה 1: ב-*Time Sharing* התקורה גדולה יותר מכיוון שמערכת הפעלה מבצעת יותר מטלות לא אמיטיות (כדי לבדוק אם צריך לבצע תוכנית אחרת או לא). אז מה היתרון של *Time Sharing* לעומת *Multiprogramming*?
גישה זו מבטיחה לנו **זמן תגובה טוב יותר**. אנחנו לא צריכים לסמור על אפליקציות אחרות שיישחררו את המעבד מזומtan כדי שתתבצע פעולה כלשהי, למשל לחיצה על העכבר.
הערה 2: חשוב לציין שמעבד בודד יכול לבצע אחת בכל רגע נתון ולכן הפעולות מבוצעות לסירוגין ולא במאמה במקביל. גם הוראות של מערכת הפעלה הן הוראות רגילים, אך איך מערכת הפעלה יכולה לבדוק אם כדאי "להעיף" את האפליקציה הנוכחית? ככל כמה זמן מערכת הפעלה חוזרת לפעולה לא משנה מה מבוצע באותו רגע, וכך היא מחליט האם המשיך עם אותה אפליקציה או לעבור לאחת אחרת.

הנחות הכרחיות על חומרת המחשב כדי לאפשר את הגישות הנ"ל:

- *the-CPU* תמיד שם ולא להשתמש בו זה בזבוז
- *the-CPU* מהיר הרבה יותר מ-*I/O* (מכシリי קלט/פלט)
- הזיכרון גדול מספיק כדי לאחסן מספר תוכניות
- יש גישה ישירה לזכרון (*Direct Memory Access*) - תוכנה המאפשרת להתקנים לבצע קרייה או כתיבה מהזיכרון באופן עצמאי ובלתי תלוי במעבד
- יש יותר פעולה אחת לבצע בו זמןית

הנחות הכרחיות על מערכת הפעלה כדי לאפשר את הגישות הנ"ל:

- *I/O Primitives* - מערכת הפעלה תדע לתקשר עם התקנים חיצוניים כמו מקלדת ועכבר.
- ניהול זיכרון – מערכת הפעלה תדע להקצות מקטע זיכרון ובמקרה של גלישה ממקטע זיכרון אחד לשני היא תדע למנוע אותה ולזרוק שגיאה.
- *Scheduler* – ניהול העבודות שנכנסות למעבד והפלטים שיוצאים ממנה.
- *Spooler* – ניהול התקנים חיצוניים איטיים יותר כמו מדפסות.



דוגמה להרצת 2 תהליכים במקביל לפי גישת *Time Sharing*.
הערה: המעבר האחרון של Process 2 מ מצב *running* למצב *ready* מבטיח שמדובר ב- *Time Sharing* (כלומר שמערכת הפעלה הפסיקת את ריצת התהיליך באמצעות *context switch*).
אחרת, אם התוכנה הייתה מוגדרת על ה-*CPU* מרצונה, המצב היה הופך להיות *waiting*.

מערכות הפעלה כיווים

רכיבים עיקריים במערכות הפעלה כיווים:

- ביצוע פעולות לטובת הפעלת אפליקציות, ניהול תהליכיים
- ניהול זיכרון (מה זה זיכרון, זיכרון ירטואלי, Cache וכו')
- תקשורת מול התקני *I/O*, דרישים
- מערכות קבצים
- *UI(G)*
- תקשורת בין מחשבים שונים
- (בטחה, טיפול בשגיאות, הקצאת משאבים ועוד)

במחשבים כיום יש 2 שיטות ארכיטקטוניות:

- מספר המעבדים (*Multi – core*) – כיום, מערכות הפעלה יכולות להריץ מספר פעולות במקביל על מספר מעבדים שונים. מעבד בודד כבר לא מספיק לדרישות השוק במילוי אחד לאחר שחל האטה בעקבות שיפור המהירות של מעבד בודד לאורך השנים, ולכן דרישות השוק הובילו לפתרון של מקבילים – עבודה על מספר מעבדים במקביל.
- מכונות ירטואליות (*Virtualization*) – מאפשרות "הקפאת מצב" (*Snapshot*), העברת מכונה וירטואלית לשרת אחר (*Migration*), הוספה או מחיקה של מכונה וירטואלית לפי דרישת (*Isolation*), בידוד בין המכונה הוירטואלית לשרת המארח (*Scaling*).
הערה: יכולה להיות תקווה גבוהה ביצועים.

בנוסף כיום יש גם שיטות אפליקטיבים עיקריים:

- הענן – מאפשר חישוב על גבי האינטרנט. אין צורך במחשב פיזי אלא רק בדפסן. מספק סביבת אחסון נוחה, בטוחה וחזקת ווחסן עלויות, זמן ומשאבים. עלויות תחזקה לשרתים יהיו זולים יותר עבור>tagidsים גדולים כמו AMAZON, בנוספ' יש להם תשתיות טובות יותר ורבה ניסיון.
- מכשירים ניידים – מערכות הפעלה למכשירים ניידים (*Android, iOS*)

Time Sharing/Multi Programming הבדלים בין מחשבים נייחים למכשירים ניידים בהקשר של

- ה-*CPU* לא תמיד זמין, כדי לשמר על הסוללה
- הזיכרון לא בהכרח גדול מספיק לאחסון מספר אפליקציות
- לא בהכרח יש יותר ממשימה אחת לבצע בו同一zeitzeit

רכיבים עיקריים במערכות הפעלה של מכשירים ניידים כיום:

- קלים יותר לפעול עבור מפתחי אפליקציות
- זמן תגובה מהיר (אלגוריתמי זמן אחרים)
- ניצול שונה של ה-*CPU* לטובות סוללה
- אבטחה (מכשירים ניידים גניבים יותר מכשירים נייחים למשל)
- זיכרון *RAM* מוגבל, אחריות גדולה יותר לניצול זיכרון

Debugging

:Valgrind

סבירת עבודה המאפשרת דיבוג קוד על מערכות לינוקס. הכלי המוכר ביותר של *Valgrind* הוא *Memcheck* שתפקידו לזהות דליות / שגיאות זיכרון. *Memcheck* רלוונטי רק עבור זיכרון דינמי שהוקצה במהלך התוכנית (למשל בעזרת *malloc*)

בתרגול ראיינו מספר דוגמאות שבהן *Valgrind* עוזר לדבג בקשות בעיות זיכרון, למשל:

- דליות זיכרון, חריגה מגבלות מערכת וכו'
- שימוש לא נכון בפונקציות הקשורות לזיכרון כמו *malloc*, *free* (למשל מספר פעמים)
- שימוש במשתנים שלא אוחחלו.

(יש דוגמאות מפורטות יותר בתרגול הראשון בשקופיות 44 – 15)

:GDB (GNU Debugger)

דיבאגר שרצ על מערכות *Unix* עבור שפות תכנות רבות. כדי להריץ את *GDB* ידנית, נקمل את הקובץ *gdb myProg* – ולאחר מכן נריץ את התוכנית כך: *gdb myProg*. כדי להריץ את *GDB* ידנית, נקمل את הקובץ *shlomo* עם הדגל *g* – ולאחר מכן נריץ את התוכנית כך: *gdb myProg*. יש פירוט פקודות ידניות של ה-*GDB*, אך אין צורך לעבוד איתן. הדיבאגר של *Clion* הוא *GUI* של *GDB* ולכן ניתן להשתמש בו בצורה נוחה. בשקופיות 66 – 58 יש הסבר לשימוש בדיבאגר של *Clion* למי שלא התעתק אליו עד היום (?!)

:strace

כל דיבוג קל לשימוש שמבצע מעקב אחר הקריאה שתוכנית מסוימת מבצעת לפונקציות של מערכת הפעלה. הפקודה *strace myProg* תראה את הקריאה לפונקציות של מערכת הפעלה, הארגומנטים וערכי ההוצאה שלהן. לפקודה יש מספר דגלים שימושיים:

- t** – כדי להציג את הזמן שבו בוצעה הקריאה לכל פונקציה
- T** – כדי להציג את משך הזמן שלקח לכל פונקציה ל clue
- e** – כדי להציג את סוג הקריאה
- o** – כדי לנטר את הפלט לקובץ
- s** – כדי לשנות את מגבלת התווים עבור מחרוזות

דוגמאות לפונקציות של מערכת הפעלה:

*file descriptor (fd) - int open(const char * pathname, int flags)* שהוא

סוג של *id* שיוגדר כמספר של הקובץ בפעולות הבאות עליו. יש 3 קבועים פתוחים תמיד:

fd = 0 → standard input, fd = 1 → standard output, fd = 2 → standard error

*ssize_t read(int fd, void *buf, size_t count)* – קורא *x* בתים מ-*fd* ל-*buf* כאשר *x* בין 1 ל-

count, ומוחזיר את *x* כדי שנדע כמה בתים הועתקו כבר מהקובץ.

*ssize_t write(int fd, const void *buf, size_t count)* – כותב *x* בתים מ-*buf* ל-*fd* כאשר *x* בין 1

ל-*count*, ומוחזיר את *x* כדי שנדע כמה בתים נכתבו כבר לקובץ.

הרצאה 2

User / Kernel Mode, System Calls

ל-CPU שני מצבים:

מצב ריצה של משתמש רגיל, ללא גישה ישירה לחומרת המחשב.

(נקרא גם *Kernel Mode*

- מערכת הפעלה ריצה במצב זה.

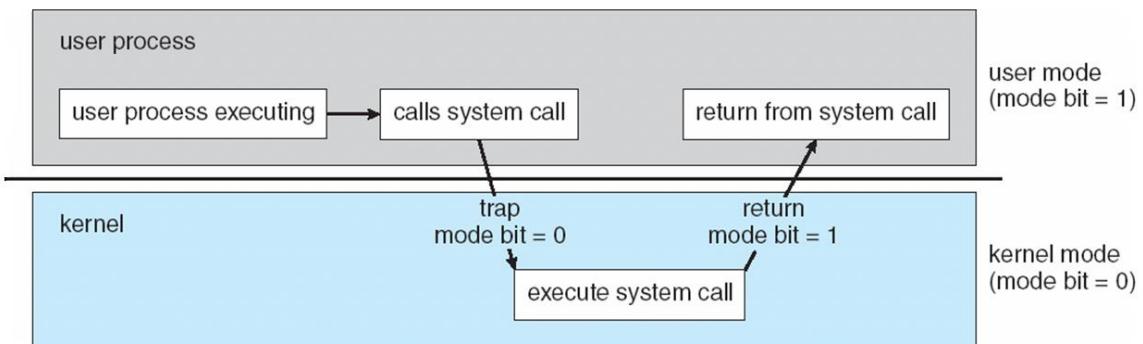
- יכול להריץ *Privileged Instructions* כדי לתקשר עם התקני חומרה.

:System Calls

מעבר ממצב של *User Mode* ל-*Kernel Mode* כדי לאפשר למשתמש לקבל שירותים מערכת הפעלה, נעשה באמצעות קרייה לפונקציות של מערכת הפעלה (*System Call*).
למרCHAT עין, ה-*System Calls* דומים לפונקציות רגוליות (למשל פונקציה של כתיבת קוד למסך) אך למעשה אלו שני דברים שונים. בקרייה לפונקציה רגילה אנחנו מרים פקודות אסמלבי של המשתמש. בקרייה לפונקציה של מערכת הפעלה, המשתמש מאבד שליטה בחוטוין על המחשב והוא עברת למערכת הפעלה. נרצה להשתמש בפונקציות של מערכת הפעלה כדי לבצע פעולות כמו גישה למערכת הקבצים, ניהול תהליכי ומבנה נתונים אחרים שלא ניתן לבצע דרך *User Mode*.
בו זמני, מערכת הפעלה בודקת שהפעולה שאנו רוצים לבצע היא פעולה חוקית אחרת היא לא תבצע אותה.

:Kernel Mode ל-User Mode

באIOR הבא ניתן לראות הדמייה לריצה של תוכנית כלשהי:



התוכנית מתחילה כ-*User Mode* עד שמתבצעת קרייה ל-*System Call*. התוכנית מחליטה לוותר על המעבד ולהעביר את השליטה למערכת הפעלה. מערכת הפעלה ריצה ב-*Kernel Mode* שם ניתן להריץ *Privileged Instructions*. להתחיל ההחלפה בין המצבים קוראים *Trap*.
כשהשליטה נמצאת אצל מערכת הפעלה היא בוחרת איך לטעוף את המשימות שהוטלו עליה, (כלומר, היא יכולה להחליט שנכוון יותר לבצע משימות אחרות לפני ה-*System Call* הנוכחי)
וכשהיא תסיים את הטיפול בפונקציית המערכת הרצiosa היא תחזיר את השליטה בחזרה לתוכנית,
במצב *User Mode*.

:Mode Bit

ה-*Mode Bit* הוא רגיסטר שנמצא בתוך ה-*CPU* והוא מיצג בכל רגע נתון את מצב המערכת. *Mode Bit = 0*, *Kernel Mode* כשהמערכת נמצאת ב-. *Mode Bit = 1*, *User Mode* כשהמערכת נמצאת ב-. *Mode Bit = 1* הרגיסטר נמצא בתוך ה-*CPU* כדי לוודא שלא מבוצעות הוראות לא מאושרוות אם 1.

:Exception

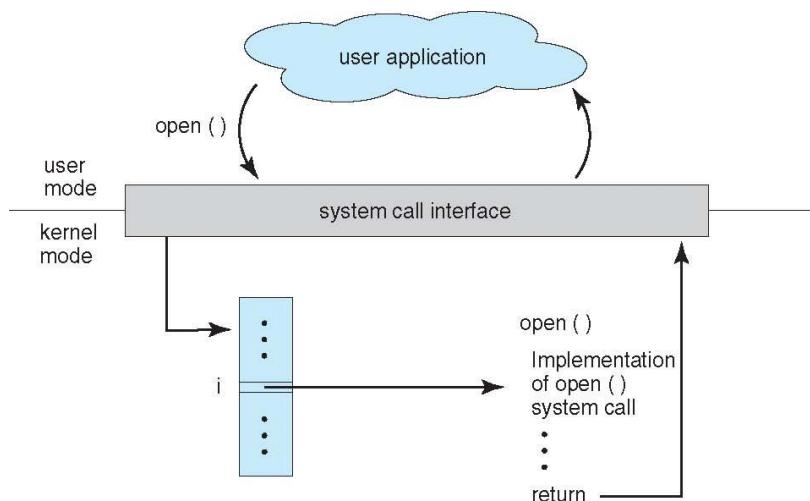
פעולה דומה ל-*Trap* המחליפה בין *Kernel Mode* ל-*User Mode* באופן לא מכון, למשל בביצוע פעולות לא חוקיות (כמו חילוק באפס), ואז מערכת הפעלה תתעורר ותטפל בשגיאה.

:System Calls

- *Process Control* – ניהול התהליכים במחשב, יצירה/החלפה של תהליכים וכו'.
- *File Management* – אבטורקציה שמערכת הפעלה מספקת לניהול מידע.
- *Device Management* – ניהול התקנים חיצוניים המוחברים למחשב.
- *Communications* – תקשורת בין אפליקציות על אותו מחשב או על מחשבים שונים.
- *Protection* – הגנה על המחשב, פקודות שמונעות מאפליקציה מסוימת לכתוב למרחיב הכתובות של אפליקציה אחרת. (גילהה)

Application Programming Interface – API

API הוא סוג של מיעוטם לקריאות *System Call*, ולכן במקום לקרוא ל-*System Call* באופן ישיר, יוכל לקרוא לפונקציות מה-*API* שעוטף אותן. למשל כשאנו משתמשים ב-*println*, *fprint*, *cout* אנחנו משתמשים ב-*API* של הפקודה *write* שהיא של מערכת הפעלה. *write* מקבלת בתים ולא מחוזת, וכן פחות נוח להשתמש בה. ה-*API* מאפשר גמ מעבר בין מערכת הפעלה אחת לאחרת, כי ה-*API* לא משתנה, אך ה-*System Call* יכול להשתנות בין מערכות הפעלה שונות.



:System Program

מערכות הפעלה מודרניות לרוב יגעו עם אפליקציות מובנות של המערכת.

- לעיתים משמש כմמשק עבור ה-*System Calls*.

- מעצב ומתוכן לחת שירות לאפליקציות אחרות.

נסתכל למשל על פקודה המערכת *echo*, שמקבלת מחuzeות וכותבת אותה בחזרה בשורה חדשה.

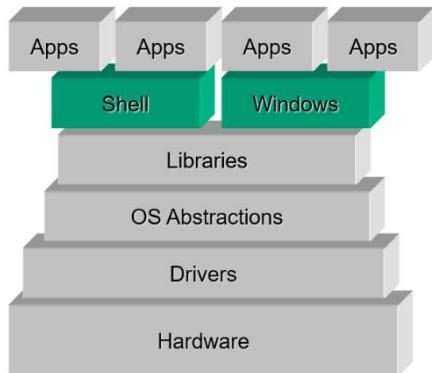
הפקודה כתובה בשפת C עם 274 שורות קוד, שבhem הפוקודה בודקת את הפורטטים ומבצעת פעולות

נוספות. הפוקודה *echo* רצה ברובה במצב *User Mode* אבל כמשמעותו לפונקציות של מערכת הפעלה,

מתבצע *Trap* (מעבר ל-*Kernel Mode*). לעומת זאת ש-*echo* חלק ממכלול המערכת, רובו פועל

במצב *User Mode* ויש מספר מעברים בין *User* ל-*Kernel*.

שכבות מערכת הפעלה



1. אפליקציות – השכבות העליונות, אפליקציות של המשתמש.

2. *User Interface* – שירותים של מערכת הפעלה כמו *shell* של

linux או *cmd* של *Windows*, או ממשק גרפי שמאפשר

למשתמש לבצע פעולות ללא כתיבת קוד ב-*cmd*. לעיתים נטעה

לחושב שאפליקציות מערכת הן אפליקציות רגילות ולהפוך. זה

תחום אפור, למשל דפדפן שmagיע עם רוב מערכות הפעלה כיום

הוא לא אפליקציית מערכת.

3. *Libraries* – ממשק שמערכת הפעלה מספקת כדי לאפשר

High Level Abstraction ל애플יקציות להשתמש בשירותים שלה.

:Low Level Abstraction – OS abstractions . 4

- אחסון וניהול קבצים

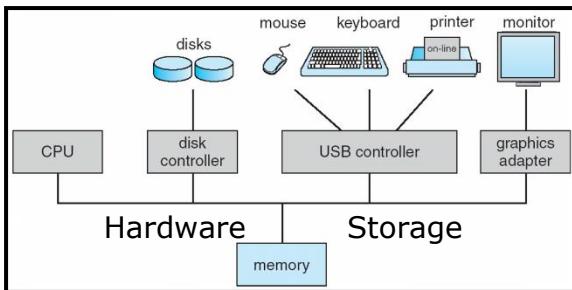
- מערכת הפעלה מספקת כלים להרצת מספר תהליכי CPU במקביל.

- תקשורת בין תהליכים ובין מחשבים.

5. *Drivers* – דרייברים הם חלק מערכת הפעלה והם מספקים ממשק שמאפשר גישה לחומרה.

6. החומרה עצמה.

:Bus



לרוב המרכזות יש *an Bus*, קו תקשורת המחבר בין הדיכרון והמעבד לבין התקנים חיצוניים כמו מקלדת / עכבר / מסך.

יתרונות:

- ורטיליות, ניתן להוסיף מכשירים חדשים בקלהות.
- ניתן להסיר התקנים חיצוניים או להעביר אותם למחשב אחר (המשתמש באותו סטנדרט של *an Bus*) בקלהות.
- מחיר זול.

חסרונות:

- צוואר הבקבוק של המערכת, כשהרבה מידע רוצה לעבור דרכו, מידע יכול ללקת לאיבוד, דברים משתבשים.
 - יוצר מגבלות על כמות התקנים החיצוניים שניתן לחבר ללא תקלות למערכת.
 - אין تعدוף בהעברת המידע.
- לרוב יש יותר מ-*an Bus* אחד ולכל אחד תפקיד שונה.

Interrupts

איך התקנים מתקשרים עם ה-CPU ומערכת הפעלה? בעזרת *Interrupts* :

פסקאות (*Interrupts*) הן כמו *System Call* אבל:

- לייצר הפסקה אין שליטה על זמן ההתרחשות.
- אפליקציות רגילות (*User Mode*) לא יכולות לבטל פסקאות, רק *Kernel Mode*.

למשל התקני חומרה שצויים תמייה / שירות מערכת הפעלה או התקני *I/O*.

ה-*Interrupt* מועבר ל-*CPU* על ידי שליחת סיגナル על גבי קו תקשורת מיוחד על ה-*Bus*.

מעבר זה מתבצע בעזרת חומרה מיוחדת שנמצאת במערכת (*IRQ*), (*Interrupt Request Line*) וטיפול המתבצע ב-*CPU* עצמו.

אופן הטיפול ב-*Interrupt* :

כאשר ה-*CPU* מקבל פסקה הוא עוצר את הפעולה הנוכחיית שלו כדי לטפל בה. ל-*CPU* יש טבלה שמתאימה בין כל פסקה לרץ' של פקודות ידועות מראש הפסקה הנוכחית. טבלה זו נקראת *Interrupt Vector Table* (באירור משמאלי). טיפול בפסקה מחייב שמירת הכתובת של הפקודה האחרונה שבוצעה, כדי שנוכל לחזור אליה לאחר הטיפול בפסקה. בדרך כלל אם מתקבלות מספר פסקאות במקביל, המערכת מטפלת באחת ומנטרלת את השאר. באחריות ההתקנים לשלוח פסקאותשוב. (קיימות פסקאות קריטיות שאפשר לדוחות ובמצב זה יהיה *Interrupt* בתוך *Interrupt* (Nested Interrupts))

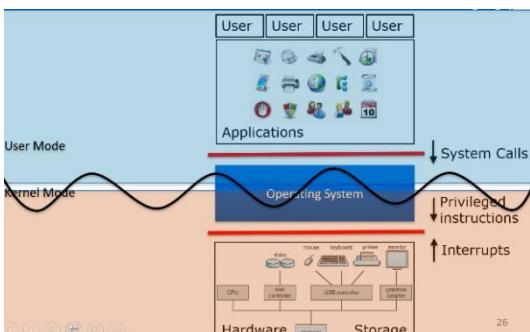
INT_NUM	Short Description PM [clarification needed]
0x00	Division by zero
0x01	Single-step interrupt (see trap flag)
0x02	NMI
0x03	Breakpoint (callable by the special 1-byte instruction 0xCC, used by debuggers)
0x04	Overflow
0x05	Bounds
0x06	Invalid Opcode
0x07	Coprocessor not available
0x08	Double fault
0x09	Coprocessor Segment Overrun (386 or earlier only)
0x0A	Invalid Task State Segment
0x0B	Segment not present
0x0C	Stack Fault
0x0D	General protection fault
0x0E	Page fault
0x0F	reserved
0x10	Math Fault
0x11	Alignment Check
0x12	Machine Check
0x13	SIMD Floating-Point Exception
0x14	Virtualization Exception
0x15	Control Protection Exception

שלושת הדרכים לתקשורת עם מערכת הפעלה:

1. *Interrupts (Hardware Interrupts)* . נסחיר חיצוני שאינו במעבד.
2. *Trap - System Calls (Software Interrupts)* . נגזר על ידי התוכנה, כולל בתוכו *Exceptions* (Software Interrupt) .
לעתים, שלושת הדרכים הנ"ל מקבלים שם כולל של *Interrupts*.

סיווג Interrupts:

- *Software VS Hardware* -
Trap, Exceptions - נגזר על ידי התוכנה, כולל בתוכו *Software Interrupt* -
Hardware Interrupts - נגזר מהחומרה, כולל פסיקות מהתקנים חיצוניים.
Internal VS External •
- פסיקות פנימיות נגרמות מהתוכנה ופסיקות חיצונית מהחומרה.
- *Synchronous VS Asynchronous* •
- *Synchronous* - פסיקות שמסונכרנות עם שעון המערכת (תוכנה, פסיקות שעון).
- *Asynchronous* – פסיקות חומרה שאין מסונכרנות עם שעון המערכת.
Maskable VS Non – Maskable •
- *Maskable* - פסיקות שנitin להשאות במידה ונשלחו פסיקות חשובות יותר.
- *Non – Maskable* – פסיקות דחופות שהייב לטפל בהן.
Periodic VS Aperiodic •
- פסיקות מחזירות - פסיקות המגיעות כל פרק זמן מסוים.
- פסיקות לא מחזירות - כל הפסיקות שהזכרנו.



נבחן מחדש את מצבי המערכת:

נשים לב כי מצבי המערכת *Kernel & User* לא באמת מופרדים לגמץ אחד מהשני. במצבים מסוימים הם מתממשים אחד עם השני. כל מערכת הפעלה בוחרת באופן שדרתי כמה פקודות ירצו ב-*Kernel Mode*-*User Mode* וכמה ירצו ב-*User Mode*-*Kernel*.

יתרונות לריצה ב-User Mode (כלומר כאשר Kernel קטן יותר):

- Kernel קטן יותר -> פחות traps -> תקורה נמוכה יותר -> יותר פעולות אמיתיות.
- Kernel קטן יותר -> פחות שימוש בזיכרון.
- Kernel קטן יותר -> פונקציונליות קטנה יותר -> שינוי בפונקציות שכבר לא חלק מהKERNEL לא יגרמו לקימפול מחדש של KERNEL (וקימפול מחדש של KERNEL זו לא פעולה פשוטה)

חסרונות:

- Kernel קטן יותר -> המשמש עבור "דרך" מערכת הפעלה פחות -> פחות הגנה.

תרגול 2

חזרה קצרה לגבי ה-CPU:

ה-CPU הוא החלק המרכזי במחשב, הוא זה שמבצע את הפקודות. הוא יכול להריץ פקודה אחת בלבד בכל רגע נתון והמידע שאיתו הואעובד שמור בתוך רגיסטרים. אין לו גישה ישירה לדיסק. כל תוכנית מתורגמת לשפת מכונה, המחולקת ל-3 סוג הוראות כפי שכבר למדנו: *Data Handling*, *Control Flow*, *Operations Arithmetique et Logique*. כל פעולה כזאת היא בסופו של דבר מספר בינארי. כדי להבהיר מידע/תוכנית מהזיכרון אל ה-CPU, נדרש לטען את המידע ל-RAM.

שימוש נכון בזיכרון

- כאשר אנחנו מבקשים לקרוא תא בזכרון, בлок זיכרון שמכיל את התא הרצוי מועתק ל-Cache.
(כלומר אנחנו לא מקבלים רק את התא הרצוי אלא בלוק שלם)
אם בקריאה הבאה יש שימוש בתא שנמצא בлок, אין צורך לקרוא שוב מה-RAM.
לכן, אם נרצה למשל לרווח על תאים של מטריצה המיוצגת על ידי מערכת של מערכים, יהיה יעיל יותר שהולאה הפנימית תróż על שורות ולא על עמודות, וכך נבצע הרבה פחות קריאות מה-RAM ל-Cache ונחסור בזמן ריצה.
- בהינתן מספר מבני נתונים, *vector*, *deque*, *list*, *vector*, זמן הריצה של חיפוש לינארי אחר אלמנט כלשהו בכל אחד מבני הנתונים הוא שונה:
vector - יהיה המהיר ביותר, כי הוא מנוהל כמו מערך, בлок בזכרון, וכך שהוא ברשימה בדוגמה הקודמת, המערכת צריכה לטען לפחות מקטע זיכרון מה-RAM ל-Cache.
list - מנוהל כמו רשימה מקושרת. כל איבר נשמר במקום אקריא בזכרון וכן זמן הריצה של חיפוש ב-list יהיה הגבוה ביותר. המערכת צריכה לטען מחדש בлок שלם מה-RAM ל-Cache עבור כל איבר ברשימה.
deque – מבנה נתונים שմשלב בין 2 המבנים הנ"ל. הוא מורכב מרשימה מקושרת של בלוקים. וכן זמן הריצה שלו בינם.

הגדרות:

- תוכנית** - קובץ הריצה (*executable file*). בפועל - קוד בשפת מכונה ששמור בדיסק.
- תהליך** - מופע של הריצה של תוכנית. **Process**
- כל *process* מקבל אישור בזכרון המ חולק לשני חלקים, הקוד וה-data.
- תוכנית שהמעבד מרים ברגע זה** (ורק אותו). **Active Process**
- Input/Output** - יכול להיות מידע המתקלט/מוחזר למשתמש, יכול להיות גם התקנים חיצוניים שהם קרא הקלט (מקלדת) ואליים נשלח הפלט (ה מסך).

Kernel Mode vs User Mode

ה-*Kernel* הוא החלק המרכזי, "הlibc" של מערכת הפעלה. יש לו שליטה מלאה על מה שקרה במחשב. כל תוכנית שרצה ב-*kernel mode* מוגדרת כ-*trusted* כלומר יש לה הרשות להכל. ה-*kernel* מחליט איזה תוכניות להריץ ומתי. כל תוכנית שלא רצתה ב-*Kernel Mode* היא *untrusted*. יש דברים שהוא לא יכול לעשות.

:*untrusted* – מריצה תוכניות שהן *User mode*

- יש פקודות שלא ניתן לבצע דרך דרך *user mode*, כמו *halt* (פונקציה שעוצרת את המערכת).
- תוכנית לא יכולה לגשת למקום בזיכרון שלא הוקצה לה. אם היא אכן ניגשת בטעות, נקבל *segmentation fault*.
- לא ניתן לגשת באופן ישיר לתקני חומרה.

OS - מערכת הפעלה

מערכת הפעלה היא התוכנה היחידה שרשאית לרוץ ב-*kernel mode* על המעבד. המטריה העיקרית של מערכת הפעלה היא להריץ תהליכים, והשאיפה היא שהקוד של מערכת הפעלה ירוץ כמה שפחות. כמו כן, על מערכת הפעלה לספק לתוכנות שירותים שימושיים מוצבי *kernel mode*.

כאשר *process* יצרך את אחד השירותיו של ה-*kernel*, הוא יפנה למערכת הפעלה באמצעות *system call*, ומערכת הפעלה תספק את השירות במידה והבקשה חוקית. *system call* הוא משק שמערכת הפעלה מספקת לתהליך שרך במחשב.

כאשר יש קריאה של *user mode system call* מ-*processor*:

- המצב של ה-*processor* משתנה ל-*kernel mode*.

(כלומר ה-*mode bit* משתנה למצב *kernel mode*, והמעבד יROUT בהתאם)

- ה-*kernel* יבצע את הבקשה, יוכל להיות שיבצע גם דברים אחרים עבור תוכנות אחרות.

- ה-*processor* יחזיר ל-*user mode*, והתוכנה תקבל את השילטה ב-*CPU*.

מעבר בין *user mode* ל-*kernel mode* הוא פשוט, כי מערכת הפעלה מחייבת לאבד את השילטה. אבל המעבר בין *user mode* ל-*kernel mode* הוא קצת יותר מסובך.ណון בהמשך.

五四יקות - Interrupts

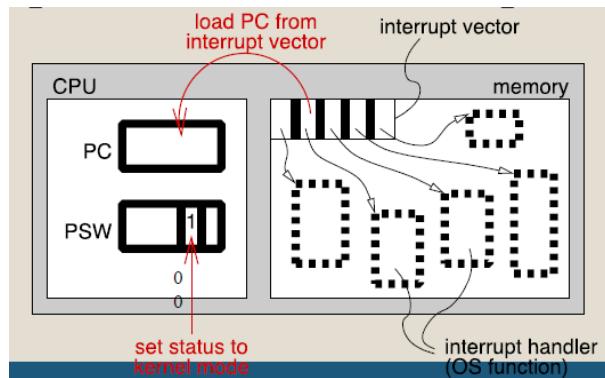
לכל רכיב חומרה במחשב יש את הפונקציונליות שלו, למשל דיסק שידוע להתנהל בכוחות עצמו, לקבל מידע, לקרוא מידע וכו'. *Interrupt* (פסיקה) הוא מנגנון שיכל לסייען בין התקני החומרה למערכת הפעלה. במחשב יש רכיב חומרה שנקרא *Interrupt Controller* שמחובר לכל התקנים החיצוניים ולמעבד. כשהתקן כלשהו צריך את המעבד הוא שולח אות ומספר, והוא-*Interrupt Controller* מעביר את הבקשה ביחיד עם המספר ל-*CPU*. כשהמעבד מקבל את הפסיקה, הוא צריך לעזוב הכל, לטפל באירוע שקרה ולהמשיך.

Hardware Interrupts (External Interrupts)

פסיקות המתקבלות מחומרת המחשב, למשל מקלדת, עכבר או השעון.

איך מתמודדים עם External Interrupts

כשה-*Interrupt Controller* מקבל פסיקה, הוא מקבל גם מספר שמייצג אותה. הוא מעביר את המספר זהה לפונקציה של מערכת הפעלה - *Interrupt Handler*. הפונקציה פונה ל-*Interrupt Vector* עם מספר הפסיקה (בפועל מייצג שורה בווקטור פסיקות), ואז היא מರיצה את קטע הקוד הרלוונטי.



סיכום התהליך של External Interrupt

1. רכיב חומרה שלוח פסיקה (אות אלקטרוניות למעבד) באמצעות ה-*Interrupt Controller*.
2. יצירת התוכנית הנוכחית, ושמירת המצב הנוכחי של התהליך האחרון שרך (כדי שנוכל לחזור אליו מוקם ולשחזר את ערכי הרגיסטרים במעבד, למשל ה-*PC*) זה קורה רק לאחר סיום ביצוע הפקודה הקודמת, אך לפני שמריצים פקודה חדשה.
3. העברת ה-*processor* ל-*Kernel Mode* באמצעות רגיסטר מיוחד שנקרא *PSW*.
4. רגיסטר ששומר מידע הנוכחי על ה-*CPU* ושם נמצא גם ה-*Mode Bit*.
5. בנוסף, השליטה עוברת ל-*Interrupt Handler* שמרי את ה-*Handler* המתאים לפי ה-*Vector Interrupt* (כפי שהסביר קודם) על מנת לטפל בפסיקה.
6. שחזור המצב הקודם. כמובן, שחזור הרגיסטרים ששמרנו בשלב 2.
7. העברת השליטה בחזרה ל-*User Mode* והמשך הרצת התוכנית.

Software Interrupts (Internal Interrupts)

פסיקות שהמערכת לא מצפה לקבל (חלוקת באפס, הוראה לא חוקית וכו') או בשם המלא - *Exceptions*. כشنשלחת שגיאה, הכתובת של ההוראה שיצרה את השגיאה נשמרת, לאחר מכן מערכות הפעלה נותנת ל-*Handler* הזדמנויות לתקן את הבעיה. אם יש *handler* מתאים, התוכנית חוזרת למקום שבו עצרה ומשיכה לroz. אם אין *handler*, המערכת הפעלה עצרת את התוכנית.

1. התמודדות עם *Internal Interrupt* זהה להتمודדות עם *External Interrupt* פרט לשלב 1.
- (כאן התוכנית לא שלוחת את הפסיקה אלא ה-*CPU*, קרה שהוא שאין יודע להתמודד איתה)

:Internal Interrupt – Trap

הוא סוג של *Exception* שמתרכז כחלק מריצה רגילה של התוכנית, אך הוא לא מייצג שגיאה. זו הדרך שבה ממומשת קריאה ל-*System Call*.

דוגמה ל trap שהוא לא System Call

פקודת האסמבלי *int3* זו פקודה שעוזרת בדיבוג וורמת לתוכנית לעצור בנקודת מסויימת (*breakpoint*). רק למערכת הפעלה יש את ההרשאות לבצע פקודה זאת.

הפקודה Man:

פקודה שנוננת את המידע הנחוץ בשימוש עם *System Call*. פקודה זו מחולקת לפי 3 קטגוריות:

1. *Man 1* - מידע על פקודות הרצה או *shell*.

2. *System Call - Man 2* (פונקציות המספקות על ידי הkernel)

3. *Man 3* - קראיות של ספריית *C* הסטנדרטית

ניתן לראות דוגמה לשימוש ב *system call* וידיבוג באמצעות "man" בשקופיות 44 – 38 בתרגול 2.

הערה: כשאנו עובדים עם *system calls*, חיבים לבדוק שערći ההחזקה שמוחזרים תקין.

(יעזר בתרגילים וירחוב בהמשך)

הרצאה 3

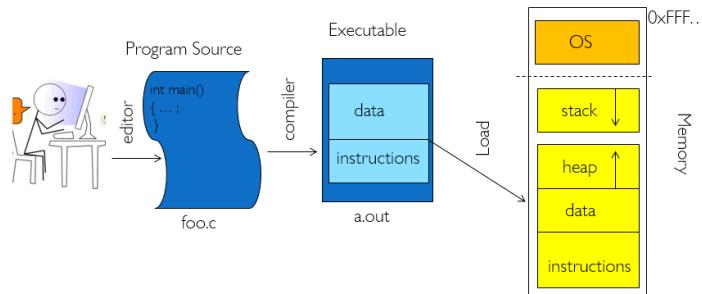
ניהול תהליכיים

שלבים להרצה תוכנית:

- המתוכנת כותב תוכנית - *foo.c*

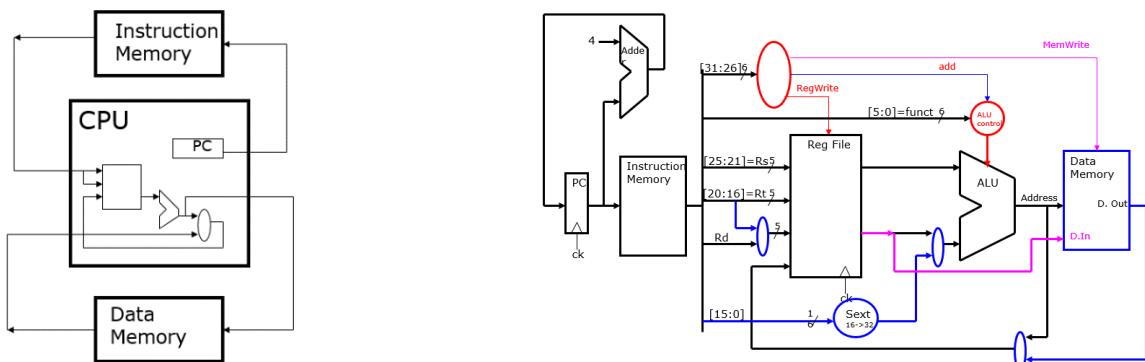
Data-Instructions ו- *Instructions* מיציר קובץ הריצה *a.out* עם *a*.

לאחר מכן יש *Loader* שמעלה את קובץ ההרצה לזיכרון כדי שנוכל להריץ אותו



מבנה ה-CPU:

כפי שראינו בנאנד / מבנה המחשב, ב-*CPU* יש את ה-*PC* (*Program Counter*) שמצוין לזכרון מיוחד (*Instructions Memory*) המכיל פקודות אסמבלי להרצה. ה-*PC* מצין בכל רגע נתון איזה הוראה להריץ וה-*CPU* מפרק אותה למספר寄存器ים שנמצאים בתוכו. בנוסף, רכיב ה-*ALU* שאחראי על החישובים האריתמטיים מבצע את החישוב ושומר את הפלט בזיכרון (*Data Memory*) ו/או בחלק מהרגיסטרים שבתווך המעבד. במחשב 32 ביט ה-*PC* יקפוץ בכל סיבוב ב-4 בתים להוראה הבאה, ובמחשב 64 ביט ה-*PC* יקפוץ ב-8 בתים. (ולעתיתים יקפוץ למקומות אחרים – תנאים ולולאות) (כמובן שמדובר בהפעטה של *CPU* ובפועל הוא מסובך הרבה יותר)



שלבי ההרצה מנוקודת המבט של המעבד:

- נבחר את ה-*PC* – *Instruction Register* (IR) מציבע עליו ונשמר אותו ב-
- ננתח את ההוראה שב-*IR*
- נבצע אותה (לעתיתים בעזרת寄存器ים נוספים)
- נכתב את תוצאה ההרצה לרגיסטרים או לזכרון (*Data Memory*)
- $PC = \text{Next Instruction}(PC)$
- נחזור על סדר הפעולות

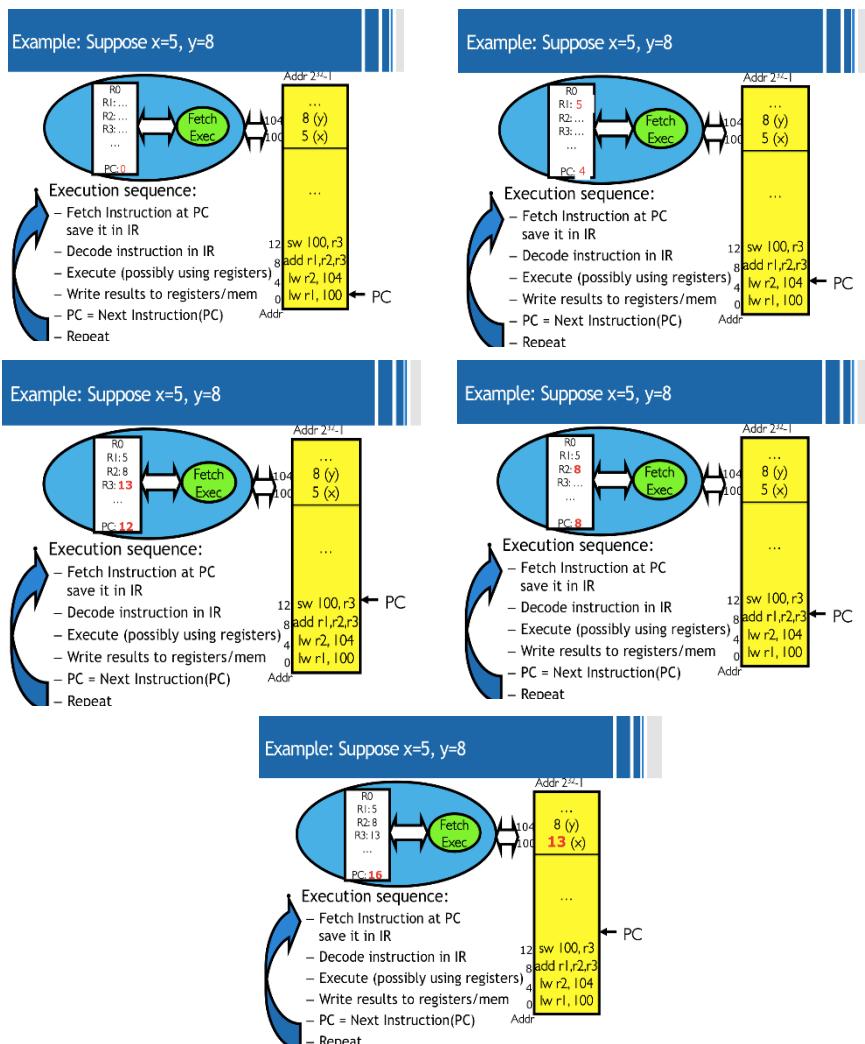
Context

נגיד ר מושג חדש ו חשוב. נניח שאנו מסתכלים על תוכנית שאין לנו מושג מה היא עשויה, מה הנטונים שנצטרכן כדי שனוכל לשחזר מה התוכנית עשתה עד לרגע מסוים?

- ערך הרגיסטרים במעבד:
 - PC -
 - כל שאר הרגיסטרים שראיםו עד עכשו ב-U-CPU
 - רגיסטרים נוספים כמו Stack Pointer, Heap Pointer וכו'
 - שמצביעים בדרך כלל לתא הראשון של סגמנטים חשובים מה נכתב לזיכרון (Data Memory)
 - קוד/טקסט, מידע, מחסנית, ערים
 - מידע פנימי של מערכת הפעלה שRELוונטי לתוכנית ונדרש עליהם בהמשך.
- כל אלה נקראים Context (הקשר), ונשתמש במושג זה הרבה בהמשך.

נראה דוגמה להרצת הפקודה $y = x + 8$ כאשר $x = 5, y = 8$.

ה-h-Context של ארבעת השקפים הראשונים נבדלים ביניהם רק ברגיסטרים של המעבד, ולעומת זאת h-Context של השקף החמישי נבדל מקודמי גם ברגיסטרים וגם בזכרון.



– תהליכיים – Processes

תהליך הוא מופע דינמי המיצג ביצוע של תוכנית מסוימת, בהקשר (*Context*) מסוים. כשרצחה להתייחס למספר תהליכיים נתייחו ל-*Context* של כל תהליך אויר להחלף ביניהם. תהליך מסוים לא מודע לתהליכיים אחרים, מבחינתו הוא רץ על המחשב בלבד. אם תהליך מסוים משתמש ב-*CPU* בזמן נתון, העריכים שלו נמצאים ברגיסטרים של ה-*CPU*. אחרת, אם הוא לא משתמש ב-*CPU*, ההקשר של התהליך ישמר בזיכרון במקום מיוחד.

Process VS Program

- תוכנית היא ישות פסיבית, תהליך הוא אקטיבי.
- תוכנית הופכת להיות תהליך כאשר קובץ הריצה נתען לזכרון.
- מבוצע על ידי *Loader*, רכיב של מערכת הפעלה.
- תוכנית זו רשימה של שורות קוד, לתהליך יש גם הקשר (ערכי רегистרים וערכי זיכרון)
- תוכנית יכולה להפעיל מספר תהליכיים

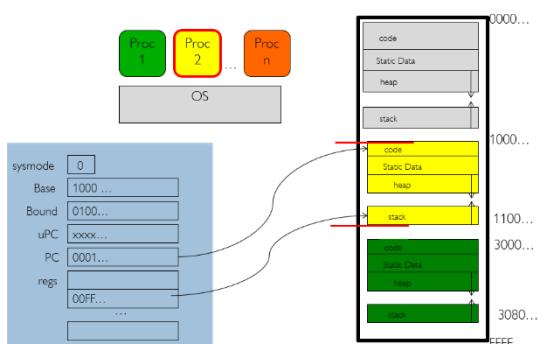
:Baking Analogy

תוכנית היא כמו מתבן להכנת עוגה:

- ה-*CPU* (שמבצע את פקודות התוכנית) הוא מי שאופה את העוגה.
- ה-*Data* (*Inputs*) אלה המצרכים של העוגה.
- ה-*Process* מיצג את התהליך האפייה.

:Address Space of Process

מרחב כתובות (*Address Space*) הוא סט רציף של כתובות "גיגישם" בזיכרון עבור תהליך מסוים. מוצג על ידי 2 פרמטרים, *base* ו-*bound*, באופן הבא: $[base, base + bound]$ מיוצגת כתובות *base* ו-*bound* (חוורתיות) שאין חריגה ממרחב הכתובות (ב-*CPU* יש 2 רегистרים נוספים *base* ו-*bound*). שמייצגים בכל רגע נתון את מרחב הכתובות הנוכחי עליו המעבד עובד. מסיבה זו, המעבד יכול להבדיל בין תהליכיים שונים, גם בין תהליכיים שרוצים במצב *User Mode*. כשתהליך ניגש לכתובת *x*, הוא מעוניין לגשת $base + x$. תהליך שמשתמש במעבד נקרא *Resident*. ונזכר שמעבד יכול להריץ פקודה אחת בלבד בכל רגע נתון, כלומר יש רק *Resident* אחד. מרחב הכתובות של מערכת הפעלה נקרא *Kernel Memory*, כל השאר נקרא *User Memory*.



נראה דוגמה: צד ימין זה הזיכרון, הצבעים (אפור, צהוב וירוק) מסמנים מרחבי כתובות שונים כאשר החלק האפור הוא מרחב הכתובות של מערכת הפעלה (*Kernel Memory*). הריבוע הכהול משמאלו מייצג את הרגיסטרים במעבד ולכן בדוגמה זו התהליך הנוכחי (ה-*Resident*) הוא התהליך הצהוב.

המעבר בין תהליכיים מתבצע על ידי שבירת הקשר של התהיליך הנוכחי בזיכרון. ככלומר נשמר את ערכי הרגיסטרים של המעבד במקום מסוים בזיכרון, ולאחר מכן שחרורם ישחרר את המצב הנוכחי ממנו התהיליך צריך להמשיך. אין צורך לשמר ערכים מהזיכרון כי לכל *process* יש מרחב כתובות שונה, ולכן מראש כל תהיליך עובד על סגמנטים שונים.

Process Control Block (PCB)

ה-*PCB* הוא מבנה נתוני שנמצא בתחום מערכת הפעלה (ב-*Kernel Memory*), והוא מכיל ומגדיר את כל המידע שמערכת הפעלה צריכה על התהיליך מסוים. ה-*PCB* מתעדכן בכל החלפת תהליכיים.

process state
process number
program counter
registers
memory limits
list of open files
...

עבור כל תהיליך, מערכת הפעלה שומרת את הפרמטרים הבאים:

– מצבו הנוכחי של התהיליך. *Process State*

(*Running, Ready, Waiting, Terminated* ווכ'

– מזהה ייחודי לתהיליך. *Process Number*

– מצביע להוראה הבאה שהטהיליך צריך לבצע. *Program Counter*

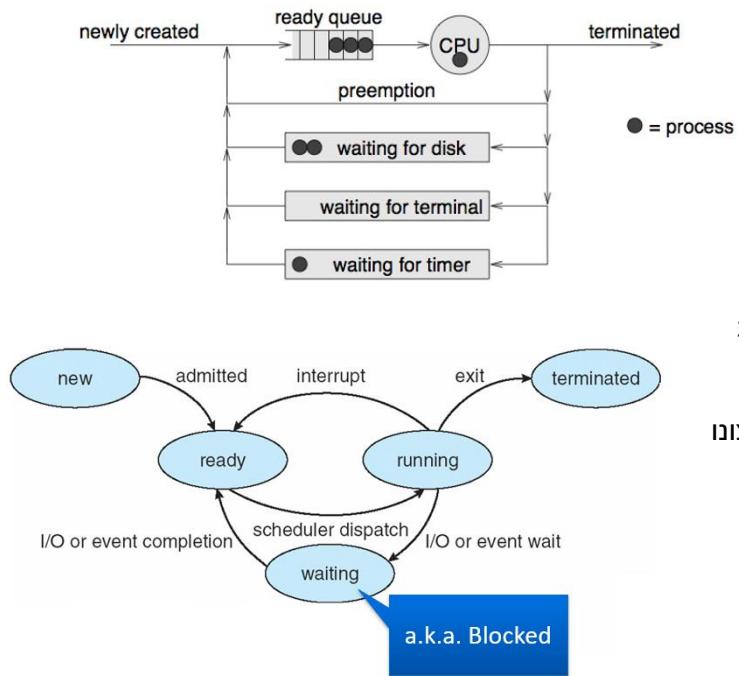
– שאר הרגיסטרים של המעבד. *Registers*

– גבולות (Bound) וה-*Base* – *Memory Limits* של התהיליך.

– הקבצים (*FD*) שמערכת הפעלה פתחה עבור התהיליך. *List of Open Files*

עוד מספר דברים שמערכת הפעלה מנהלת עבור התהיליך הספציפי כמו הגדרות *GUI*, תקשורת וכו'.

הדיagram משמאל מתארות את השלבים שתהיליך מסוים עשוי לעבור:



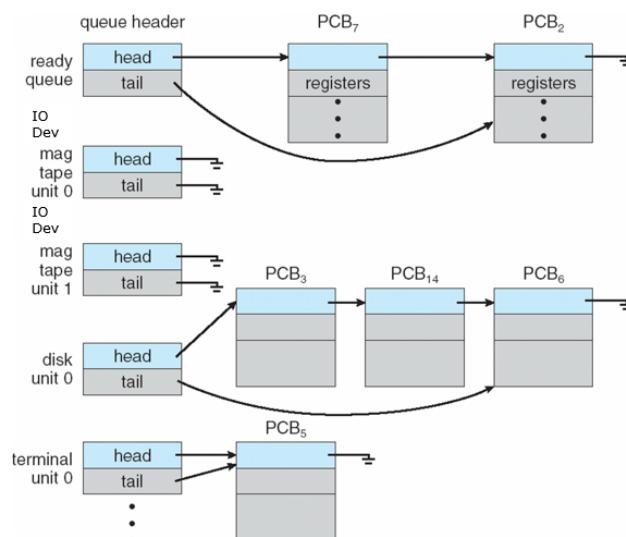
- ראשית התהיליך נוצר, והוא מוגדר כ- *Newly Created*.
- לאחר שכל מבני הנתונים הנדרשים עברו אוטחלו (כולל עדכון ה-*PCB* של מערכת הפעלה), התהיליך עובר למצב *Ready* והוא ממතין בתור ל-*CPU*.
- כשהמגיע תורו, כל עוד ה-*CPU* מרייך הוראות שלו, הוא מוגדר כ- *Running*.
- לאחר מכן, או שהוא מוחותר על ה-*CPU* מרצונו וועבר למצב *Waiting* (למשל כדי לגשת לזכרון או להחכות לקלט מהמשתמש) או שמערכת הפעלה תשזה אותו לטובות אופטימיזציה ואז הוא יփוך למצב *Ready* ויחזור ישר לתור.
- לבסוף התהיליך עובר למצב *Terminated*.

Scheduling

ה-*Scheduler* הוא רכיב של מערכת הפעלה והוא מחליט איזה תהליך יירוץ במעבד. הוא בוחר מתוך ה-*Ready Queue*, וב-*Time Sharing* (אם התהיליך לא מוגדר על המעבד מיווזמתו לפני הזמן שהוקצב לו, המערכת הפעלה תתעורר) נוהג לחלק את ה-*Scheduler* ל-2 ישויות נפרדות: ה-*CPU Scheduler* וה-*Dispatcher*. ה-*CPU Scheduler* מחליט איזה תהליך יירוץ ולכמה זמן ("המוח") וה-*Dispatcher* ("הכח"), דואג לעדכן את הריגיסטרים, מבצע את המעבר מ-*Kernel Mode* ל-*User Mode* וכו'. הערה: שניים משתמשים ב-*PCB*. הראשון משתמש במידע מה-*PCB* כדי לבצע את הבחירה, והשני משתמש ב-*PCB* כדי להעתיק מידע לריגיסטרים.

Ready Queue and Various I/O Device Queues

כל בлок *PCB*, המתאר את ה-*Context* של תהליך מסוים, נשמר בתור מיוחד לפי המצב שלו. למשל הבולוקים של תהליכיים במצב *Ready Queue* יהיו ב-*IO Dev* (磁带單元 0), *Disk Queue* (磁盘單元 0) וכן *terminal unit 0*.



כעת נוכל לתאר את תהליך ההחלפה של 2 תהליכיים באופן מפורט יותר:

- נשמר את ה-*Context* של התהיליך הנוכחי, כולל ה-*PC* וריגיסטרים נוספים.
- נעדכן את בлок ה-*PCB* של התהיליך עם המצב החדש שלו.
- נשאיר את הבלוק לתוך הרלוונטי (*Ready, Blocked/Waiting*)
- נבחר תהליך אחר להרצתה
- נעדכן את ה-*PCB* שלו ל-*Running*
- נעדכן מבני נתונים שונים של מערכת הפעלה
- נשחרר את ה-*Context* של התהיליך שבחרנו
- נמצא מ-*Kernel Mode* ל-*User Mode* חזרה

הערה: נשים לב שהפעולות הנ"ל הן חלק מהתקורה, אך הן פועלות הכרחיות.

מתי נחלף בין תהליכיים?

Interrups -

- שעון – תהליך השתמש בכל הזמן שהוגדר לו.
- *I/O*
- שגיאת זיכרון
- כתובות זיכרון נמצאת בזיכרון וירטואלי (נרחיב על כך בהמשך)
- קרייה לפונקציה של מערכת הפעלה
- שגיאות (*Exceptions*)

מתי נוצר תהליך?

- כשמערכת הפעלה עולה
- קרייה לפונקציה (*fork()*)
- כשהמשתמש מבקש ליצור תהליך (דאבל-קליק על אפליקציה או פקודת *Shell*)
- הריצה של מספר פקודות ברצף שבדרכם כל מיצירות תהליכיים (*Batch Job*)

דוגמה (חשובה) לייצירת תהליך עם הפקודה *(fork())*

```
int pid;
int status = 0;

pid = fork()
if (pid != 0)
{
    /* parent */
    .....
}
else
{
    /* child */
    .....
}
```

- נשימוש בפונקציה (*fork()* כדי לשכפל את התהליך הנוכחי שרצ).
- מערכת הפעלה משכפלת את בלוק-*PCB* של התהליך הנוכחי.
- נרצה לשכפל תהליך כדי לבצע פעולה שונה ממה-*State* הנוכחי.
- כדי לדעת איזה תהליך רץ כרגע במעבד, נשימוש בערך ההחזרה של הפונקציה (*fork()*, בדוגמה משמאלו הערך של *pid*).
- אם *pid* = 0 סימן שהתהליך המשוכפל (הילד) הוא זה שרצ,
- אחרת התהליך המקורי הוא זה שרצ. במקרה זו אנחנו יכולים לבצע פעולה שונה לכל תהליך.

נרחיב על (*fork()* בתרגול !

סיג널ים – Signals

סיגナル הוא נוטפיקציה (התראה) הנוצרת על ידי מערכת הפעלה ונשלחת לתהילר מסוים כדי להתריע על אירועים חשובים. הסיגナル גורם למעבד לעצור את מה שהוא עשו (לאחר סיום הרצת הפוקודה הנוכחית), מאלץ את התהילר להתמודד עם הבעה ולאחר מכן ממשיך כרגיל. סיגナル נוצר על ידי מערכת הפעלה, ומטופל על ידי התהילר, ואילו *Interrupt* נוצר על ידי החומרה או התוכנה (תליי בסוג הפסיקה) ומטופל על ידי מערכת הפעלה. ניתן לראות את סוגי השוני של סיג널ים באמצעות הפוקודה "man kill".

דוגמאות למקרים בהם נוצרים סיגלים:

- **kill מהמשתמש** - למשל כשהמשתמש לוחץ במקלדת על *Ctrl + C* או מקליד "kill pid" בטרמינל: נוצר *Interrupt*, ובין היתר מערכת הפעלה תשליח סיגナル לתוכנה כדי שתפסיק.
- **מערכת הפעלה** - ניתן לבקש למשל למערכת הפעלה שתעיר אותו בעוד 10 שניות (*Alarm*), ובמקרה זה מערכת הפעלה תשליח סיגナル לאחר 10 דקות. גם כאן זה קורה באמצעות פסיקה שגורמת למערכת הפעלה לשЛОוח סיגナル.
- **טהילר אחר שרצ בו זמן** (אם יש יותר מליבה אחת) – למשל אם בקוד של תהילר מסוים רשום לשלוח סיגナル לתהילר אחר.
- **Software Interrupt** – למשל כאשר פוקודה לא חוקית יוצרת התוכנית *SIGTRAP* למערכת הפעלה מקבלת את הפסיקה, מייצרת סיגナル ושולחת אותו לתהילר הבועתי.

דרכים ודוגמאות לשילוח סיגלים:

1. הדרך הנפוצה ביותר לשילוח סיגלים היא על ידי המקלדת, נראה מספר דוגמאות:
 - *Ctrl + C* – יוצר את הסיגナル ***SIGINT*** המשמש לסגירת התוכנית.
 - *\ + Ctrl* – יוצר את הסיגナル ***SIGQUIT*** המשמש לסגירת התוכנית.
 - *Ctrl + Z* – יוצר את הסיגナル ***SIGTSTP*** לעצירת התוכנית עד לSİיגןל ***SIGCONT*** שיופיע.
2. ניתן לשЛОוח סיגלים גם דרך Command Line:
 - על ידי הפוקודה "*kill [option] pid*" (process ID), כאשר ה-(*pid*) מייצג את תעודת הזהות של תהילר מסוים, ובמקרים רבים ניתן לשLOWח סיגナル אותו לתהילר. ניתן לרשום "*l* – *kill*" ולראות רשימה של כל הסיגלים שניתנו לשLOWוח.
 - הפוקודה "*fg pid*" ממשיכה את הרצת התהילר לאחר שעשינו אותו עם *Z + Ctrl*. על ידי שליחת הסיגナル ***SIGCONT***.
3. שליחת סיגלים מטהילר אחד לאחר:
 - ניתן לכתוב בקוד של תוכנית אחת את הפקודה *kill*, המקבלת *pid* של תהילר אחר אליו נרצה לשLOWח את הסיגナル, ואת מספר הסיגナル שנרצה לשLOWוח. מבוצע באמצעות הפוקודה: (*int kill (pid_t pid, int sig)*). (ניתן לשLOWח רק סיגלים ידועים מראש).
 - ניתן להגדיר *SIGUSR₁*, *SIGUSR₂* – סיגלים שתוכניות יוצרות לעצמן. כל תוכנית תהיה יודעת של הסיגלים ותפעל בהתאם.

פונקציות שימושיות לניהול סיגנלים

לכל סיגナル יש התנהגות דיפולרית שהתחילה מבע שביל לטפל באותו סיגナル (למשל יצאת מהתוכנית, להתעלם, לעזור וכו'). ניתן לשנות את התנהגות הדיפולרית, ולהחליט שהטיפול בסיגナル מסוים יהיה באמצעות פונקציה שהמשתמש כתוב, נראה מספר פונקציות שימושיות למטרה זו.

signal – הפונקציה `signal(signum,newHandler)` מקבלת מספר סיגナル ושם של פונקציה שנרצה להריץ **במקום** התנהגות ברירת המחדל. יש סיגנלים שלא ניתן לשנות, למשל *KILL, STOP*.

הערה: קרייה לפונקציה `signal` מבצעת שינוי זמני, כלומר השינוי יהיה תקף רק לסיגナル הבא שיגיע, ולא לכל הסיגנלים הבאים, שכן נדרש להשתמש בפונקציה `signal` בכל פעם מחדש.

```
#include<stdio.h>
#include <unistd.h>
#include <signal.h>

void catch_int(int sig_num) {
    //install again!
    signal(SIGINT, catch_int);
    printf("Don't do that\n");
    fflush(stdout);
}

int main(int argc, char* argv[]) {
    signal(SIGINT, catch_int);
    for (;;) {
        //wait for a signal
        pause();
    }
}
```

בדוגמת הקוד משמאל אנחנו מגדירים לסיגナル *SIGINT* את הפונקציה `catch_int` שתופעל בפעם הבאה שיתקבל הסיגナル (במקום לסגור את התוכנית, מה שמודדר כברירת מחדל) בפונקציה `newHandler`. אנחנו קוראים ל-*signal*'ים שוב, על מנת לוודא שהתנהגות זו תהיה "קבועה". ישן שתי פונקציות שניתן לה�试מש בהן *C-newHandler*:

ambil' שנצטרך לכתוב פונקציות חדשות:

SIG_IGN – התעלמות מהסיגナル הגרפי.
SIG_DFL – שחזור ההתנהגות הדיפולרית.

sigprocmask (Masking Signals)

מכיוון שיש סיגנלים מנוהלים באופן אינטגרטיבי, ניתן לבצע בו נקיול את הסיגナル *SIGINT* (למשל) תוך כדי ניהול סיגナル אחר (נקרא *race condition*, ריחוק על המושג בהמשך). נניח למשל שתהיליך מוחק קבצים שונים, ותוך כדי הפעולה מתתקבל סיגナル *SIGINT* שאמור לעזoor את התהיליך. במקרה זה ישארו קבצים שניים / קבצים פגומים. כדי להימנע מכך, משתמש ב-*Block Signals* - אם התקבלו שני סיגנלים ביחד, נחסום אחד מהם, וטפל בשני, ולאחר מכן נבטל את החסימה ונתמוך עם הסיגナル שחסמנו.

נעשה זאת בעזרת הפונקציה (`int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)`)

:*int how*

SIG_BLOCK – להוסיף סיגנלים לקבוצה.

SIG_UNBLOCK – לשחרר סיגנלים מוקבוצה.

SIG_SETMASK – לקבוע איזה סיגנלים יהיו בקבוצה.

:*const sigset_t *set*

- קבוצה של סיגנלים.

:*sigset_t *oldset*

- פוינטר לסת שדרסנו. (אם אין צורך נשלח *NULL*)

```

sigset(SIG_BLOCK, &set, NULL);
//blocked signals: SIGINT and SIGTERM

sigemptyset(&set);
sigaddset(&set, SIGINT);
sigaddset(&set, SIGTERM);
sigprocmask(SIG_SETMASK, &set, NULL);
//blocked signals: SIGINT and SIGTERM

sigemptyset(&set);
sigaddset(&set, SIGINT);
sigaddset(&set, SIGALRM);
sigprocmask(SIG_BLOCK, &set, NULL);
//blocked signals: SIGINT, SIGTERM, SIGALRM

sigemptyset(&set);
sigaddset(&set, SIGTERM);
sigaddset(&set, SIGUSR1);
sigprocmask(SIG_UNBLOCK, &set, NULL);
//blocked signals: SIGINT and SIGALRM

```

דוגמה לשימוש ב-*:sigprocmask*

- בחלק הראשון של הקוד הגדרנו *set* ויידאו שהוא ריק *.sigemptyset*.
- הוספנו ל-*set* את הסיגנלים *SIGINT, SIGTERM*, *SIG_SETMASK* השתמשנו ב-*sigprocmask*.
- כלומר הגדרנו שקבוצת הסיגנלים החסומים תכיל רק את הסיגנלים שנמצאים ב-*set* (בלי קשר למה שהוא הסיגנלים החסומים קודם).
- בחלק השני של הקוד רוקנו את הקבוצה *set* שוב, ומוספנו לה את הסיגנלים *SIGINT, SIGALRM*.
- בפקודה *sigprocmask* השתמשנו ב-*SIG_BLOCK*, כלומר הוספנו לקבוצת הסיגנלים החסומים (שמכילה כבר את *SIGINT, SIGTERM*) את הסיגנלים שנמצאים בקבוצה *set*, וכן הוסיףם *SIG_BLOCK*.
- בחלק השלישי של הקוד בפקודה *sigprocmask* השתמשנו ב-*SIG_UNBLOCK*, המכיל את הסיגנלים *SIGTERM, SIGUSR1*. במידה והסיגנלים שנמצאו ב-*set* היו חסומים, הם כבר לא יהיו חסומים. וכן הוסיףם החסומים בשלב זה הם *SIGINT, SIGALRM*.

:sigaction

*int sigaction(int sig, struct sigaction *new_act, struct sigaction *old_act)*

פונקציה זו מאפשרת לקבוע שלSIGןל מסוים יהיה *handler* מסוים, עד שנחליט אחרת. (כלומר השינוי הוא קבוע ולא חד פעמי כפי שראינו ב-*signal*).

sigaction חוסמת סיגנלים שהגדרנו מראש במהלך ה-*handler*. אם לא נקבע כלום, ההתחנות הדיפולטיבית היא שהסיגנל עצמו (עליו מפעלים את ה-*handler*) הוא זה שחשום.

```

#include<stdio.h>
#include <unistd.h>
#include <signal.h>

void catch_int(int sig_num) {
    printf("Don't do that\n");
    fflush(stdout);
}

int main(int argc, char* argv[]) {
    // Install catch_int as the
    // signal handler for SIGINT.
    struct sigaction sa;
    sa.sa_handler = &catch_int;
    sigaction(SIGINT, &sa, NULL);

    for ( ; ; )
        //wait until receives a signal
        pause();
}

```

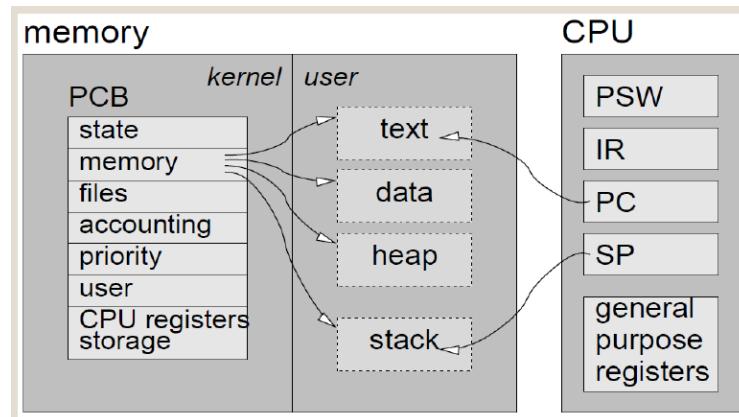
דוגמת הקוד משMAIL זהה לדוגמה של *signal* למעט השימוש ב-*sigaction* במקום *signal*. הגדרנו ב-*main* מבנה של *sigaction* ועדכנו את ה-*sa_handler* שלו עם מצביע לפונקציה *catch_int*. מכיוון *sigaction* מבצעת שינוי קבוע, לא קראנו לה שוב ב-*catch_int* (בניגוד ל-*signal*)

הערות נוספת:

- הפונקציה *Signal* כבר לא בשימוש, משתמש ב-*Sigaction*.
- למדנו בתרגול קודם שהפונקציה *strace* עוקבת אחר קריאות של תהליכי מסויים לפונקציות של מערכת הפעלה, מעכשו נוכל להשתמש בה גם לניטור סיגנלים.

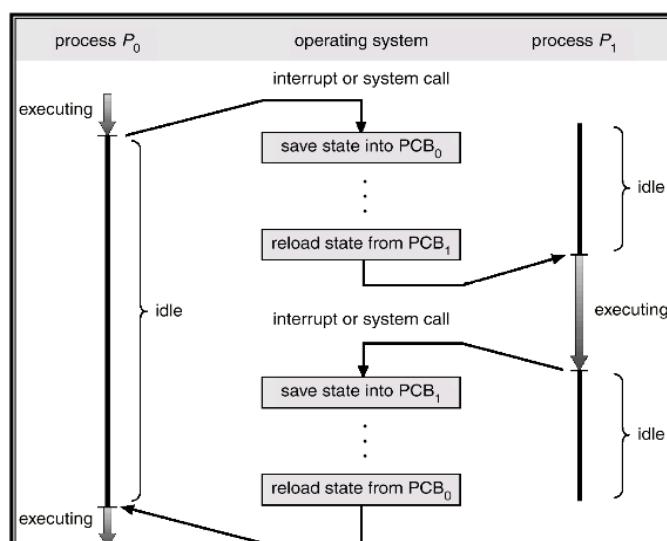
Threads

למדנו בהרצאה שבעזרת *Time Sharing* אנחנו יכולים להריץ מספר תהליכיים במקביל על ידי הריצת כל תוכנית לסירוגין. כדי לעצור הריצה של תהליך אחד ולהריץ תהליך אחר, מערכת הפעלה שומרת את כל הרגיסטרים מה-*CPU* במבנה נתונים ב-*Kernel Memory* הנקרא *PCB*, שבו שתוכל לחזור לנוקודה שבה התהליך עצר.



:Context Switch

ניתן לראות דוגמה להריצת שני תהליכים במקביל, כפי שהראנו עם שמירת המידע ב-*PCB*, כאשר כל תהליך "חושב" שהוא היחיד שרצה. באופן אידיאלי ככה היינו רוצים לעשות *multitasking*, אבל בפועל מערכת הפעלה מבצעת פעולות נוספות (שמירת המצב הקיימם ו切换回先前的状态) וזה ה-*overhead* שהוא יינו רוצים למזער ככל האפשר.



:Thread

(תהליכון) Chi בטור תחיליך ותהליך יכולם להיות מספר תהליכיונים. תהליכון יכול לזרז באופן עצמאי ללא תלות בתהליכיונים אחרים מכיוון שהוא מתחזק מחסנית משלה, ורגיסטרים של ה-CPU. שאר המידע של התהיליך (שה-Thread מקשר אליו) משותפים בין כל ה-*Threads*: קוד, זיכרון, קבצים פתוחים וכו'.

:Threads

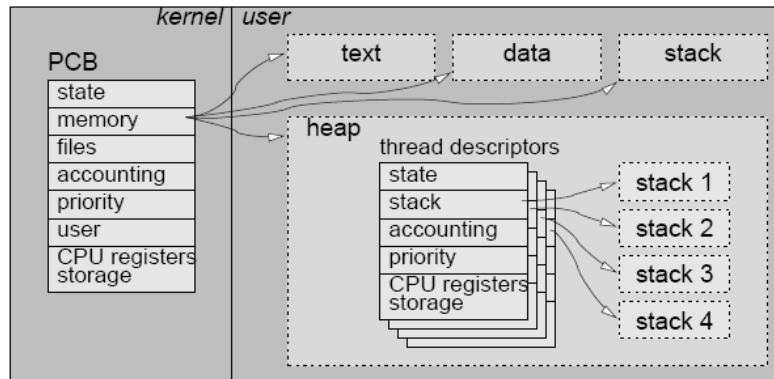
- **User Level Threads** – המשמש ממש אוטם ומערכת הפעלה לא יודעת על קיומם.
- **Kernel Level Threads** – תהליכיונים שמערכת הפעלה ממששת ומנהלת.

:User Level Threads

ממומש באמצעות ספירת Threads קיימת המכילה קוד ליצירה, עצירה, תזמן וחלפה של Threads. מערכת הפעלה רואה רק את התהיליך ולא מודעת לפעולות על תהליכיונים. لكن אם אחד מה手続きים חסום על ידי מערכת הפעלה (למשל קריאה ל-*System Call*, בקשה לקריאת קובץ ללא הרשות וכו'), כל התהיליך נחסם, מכיוון שהחלפה בין Threads נעשית ב-*User Space*. מנגד, הולות של ביצוע ה-*Context Switch* נמוכה מעלות *Context Switch* של מערכת הפעלה.

:Threads

בשימוש נכון, בכל רגע נתון רק תהליכון אחד יכול לשנות את המידע המשותף לכל תהליכיונים, שכן חלק מההגבלות ל-Threads יהיו מיותרות. כל Thread יוגדר באמצעות *Thread Descriptor*



החלפה בין תהליכיונים נעשית באופן הבא:

1. מפסיקם את ההרצתה של ה-Thread שרצ כרגע.

2. שומרים את המצב הנוכחי שלו במבנה נתונים.

3. קופצים ל-Thread אחר על ידי שחזור המצב שלו ששמרנו מראש.

ונכל לבצע את הפעולות הנ"ל בעזרת הֆונקציות *siglongjmp* ו-*sigsetjmp*.

- *Stack Context*, *sigsetjmp(sigjmp_buf env, int savesigs)* – השומרת את המצביע *env* ומאפשרת שילובו עם ה-*CPU* וריגיסטרים אחרים (*PC, SP, ...*). אם *savesigs* שווה ל-0, השומרת את המצביע *env*. אם שווה לאפס, השומרת את הריגיסטרים מה-*stack*, *heap*, *malloc* ועוד.
- כל מה שיכל להשנות באמצעות *Threads* אחריהם. ערך ההחזרה של הפונקציה מוסבר מתחילה בהמשך העמוד. חשוב להבין שהפונקציה *sigsetjmp* שומרת את המצביע הנוכחי של התהיליכון אך לא מבצעת החלפה בין תהיליכונים! לאחר הקראיה לפונקציה, נדרש לבצע את החלפה בעזרת *.siglongjmp*.
- *siglongjmp(sigjmp_buf env, int val)* – הפונקציה קופצת למקום השמור ומשחררת את המצביע *env*. הפונקציה תחזיר אותו למקום בקוד שבו קראונו לו-*jmp*. אם המצביע *val* נשמר אז הוא ישוחזר, ועוזר ההחזרה של הפונקציה יהיה *val* – פורמטור שהפונקציה מקבלת.

דוגמה להחלפה בין *threads* (הקוד המלא במודול, מומלץ לעבור ולהבין):

```

Thread 0:
void switchThreads()
{
    static int curThread = 0;
    int ret_val =
        sigsetjmp(env[curThread], 1);
    if (ret_val == 5) {
        return;
    }
    curThread =
        1 - curThread;
    siglongjmp(env[curThread], 5);
}

Thread 1:
void switchThreads()
{
    static int curThread = 0;
    int ret_val =
        sigsetjmp(env[curThread], 1);
    if (ret_val == 5) {
        return;
    }
    curThread =
        1 - curThread;
    siglongjmp(env[curThread], 5);
}

```

ברגע הקראיה לפונקציה *sigsetjmp*, ה-*IR* מצביע על פקודת הקראיה לפונקציה (*sigsetjmp*) ולן הערך של *PC* שנשמרו, מצביע לפקודת שבאה אחריו פקודת הקראיה.

(למשל במקרה של תמורה לעיל, ה-*PC* שנשמרו מצביע ל-*if* שבא אחריו *sigsetjmp*).

נבחן בין 2 מצבים שונים:

- התהיליכון קרא ל-*sigsetjmp* כדי לשמור את המצביע הנוכחי, וזה הוא צריך להריץ תהיליכון אחר.
- התהיליכון הוא תהיליכון משוחזר, כלומר הוא לא צריך לקרוא לתהיליכון אחר, אלא לבצע פעולות.

איך נבדיל בין המצבים? בשני המקרים ה-*PC* מצביע לשורת ה-*if* בדוגמה לעיל.

בשביל זה ערך ההחזרה של הפונקציה *sigsetjmp* יהיה 0 אם התהיליכון הנוכחי קרא לה (ואז נדרש להפעיל תהיליכון אחר), או ערך ההחזרה שונה מ-0 אם אנחנו תהיליכון משוחזר (ואז נדרש להריץ פקודות אחרות ולא לעבור לתהיליכון אחר).

:Kernel Level Thread

למערכת הפעלה יש *Threads Table* שעוקבת אחרי כל ה-*Threads*. אם אחד חסום, אחרים יכולים לרוץ. החליף בין *Kernel Level Threads* יקרה יותר מהחלפה בין *User Level Threads*.

Kernel Level Threads VS User Level Threads

תרונות:

<u>User Level Threads</u>	<u>Kernel Level Threads</u>
החלפה בין הריצה של <i>Threads</i> זולה יותר	חסימה נעשית ברמת ה- <i>Thread</i>
דברים לא יכולים לרווח באמת במקביל, ולכן אין בעיה של גישה למשאים בו זמן נתון.	מספר <i>Threads</i> יכולים לרווח באמת במקביל אם יש מספר ליבות (לא <i>Time Sharing</i>)
ניתן להתאים את ה- <i>Scheduler</i> לאפליקציה הספציפית של המשתמש.	ה- <i>Scheduler</i> כנראה יבצע החלטות יותר אינטלייגנטיות בנוגע ל- <i>Threads</i> ותהליכיים.

חסרוןות:

<u>User Level Threads</u>	<u>Kernel Level Threads</u>
לא ניתן להנות מההטבות של מספר ליבות	עלות גבוהה יותר עבור החלפה בין <i>Threads</i>
<i>Process</i> חסום אחד חוסם את כל <i>Thread</i>	צריך לשימוש לב למשאים משותפים
Poor OS Scheduling	

מה עדיף?

כאשר בוחרים שימוש צריך לבחון את הצורך של האפליקציה הספציפית.
 עבור אפליקציה המחייבת בין תהליכים לעיטים תכופות, נעדיף להשתמש ב-*User Level Threads*.
 עבור אפליקציה שיש לה הרבה תהליכים, או הרבה פעולות המשלבות *O/I*,
 נעדיף להשתמש ב- *Kernel Level Threads*.
הסבר על Ex2 ניתן לראות במצגת של תרגול 3, שקופיות 55-60.

שאלות מבחנים שהופיעו בתרגול:

7) כמה פעמים הקוד הבא ידפיס hello?

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

int main(int argc, char *argv[])
{
    sigjmp_buf jbuf;
    int i = 10;
    int ret_val = sigsetjmp(jbuf,1);
    if (ret_val == 0) {
        return 0;
    }
    i--;
    printf("hello\n");
    siglongjmp(jbuf,i);
    return 0;
}
```

- (א) 9 פעמים
- (ב) 10 פעמים
- (ג) 0 פעמים
- (ד) פעם אחת

2. הבדל אחד בין תהליך לבין kernel thread הוא

- א) עם רבים תהליכים ניתן לנצל רבים מעבדים, אבל עם kernel threads לא
- ב) עם kernel threads מונעים את הבעיה של חסימה כאשר אחד מבצע פעולה *O/I*, אבל הבעיה קיימת שימושים ברבי תהליכים
- (ג) תהליכים זוקרים לתיווך של מערכת הפעלה כדי לתקשר,kernel threads לאkernel threads רק קוד של מערכת הפעלה, ואילו תהליכים יכולים להריץ קוד משתמש kernel threads

(ד)

הרצאה 4

מתי תהליכי מסתים? (*Process Termination*)

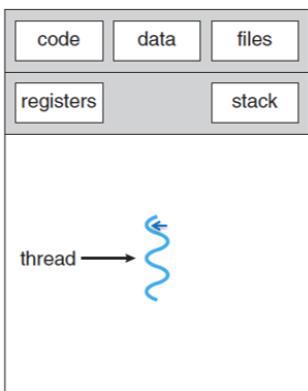
- *Normal Exit* – כשהתהליך בוחר לסיים את הריצה, יציאה רצונית.
- *Error Exit* – יציאה לאחר שגיאה שניית להთאושש ממנה, למשל ניסיון לפתח קובץ לא קיים, יציאה רצונית למשל ע"י קראיה ל-`exit()`.
- *Fatal Error* – סיום התוכנית לאחר שגיאה שלא ניתן להთאושש ממנה, יצאה לא רצונית, למשל לאחר שהתוכנית חילקה ב-0.
- *Killed by another process* – בתנאים מסוימים, תהליך יכול "להרוג" תהליך אחר על ידי שליחת סיגナル 9 – `kill`, יצאה לא רצונית.

תקשורת בין תהליכי (IPC) (*inter process communication*)

בהתנתק משימה גדולה, נרצה לחלק אותה באופן מודולרי למשימות קטנות כך שכל משימה קטנה תבוצע על ידי תהליך אחר. נרצה שכל תהליך יירוץ בזמן מסוים ומיידי פעם התהליכים יחליפו ביניהם מידע. בתהליך החלפת המידע יש להיזהר לא לדרכו אחד לשני את הזיכרון, ולכן מראש מידע יכול לעبور ביניהם ואיזה מידע לא. מערכת הפעלה מספקת לנו באמצעות *System Calls* את האפשרות להאזין, לשלוח ולקבל הודעות בין תהליכי, אך שימוש ב-*System Calls* מגדיל את התקורה. נתבונן בדרכים לתקשורת בין תהליכי:

- *Shared Memory* – הגדרה מראש של סגמנט זיכרון משותף בין מספר תהליכי, שם הם יכולים לכתוב ולקראן מידע.
- *Message Passing* – מנגן העברת הודעה מתהליך אחד לאחר דרך ערוץ תקשורת (למשל *Socket Interface*). מערכת הפעלה מתייחסת לתהליך זה כמו שליחת מסרים בין מחשבים שונים וכן התקורה של פעולה זו גדולה.

תקשורת בין תהליכי יוצרת בעיות סינכרוניזציה (נרחיב בעמודים הבאים).



- *Threads*

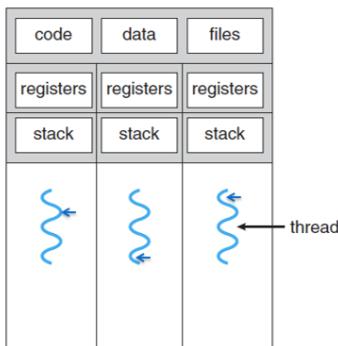
האיור משמאל מתאר תהליכי: לכל תהליך יש את המידע ב-*heap* – הקוד וה-*data*, הקבצים הפתוחים, הרגיסטרים, והמחסנית שדרוכה לבצע את הקוד. בהנחה שלכל תהליך יש *thread* יחיד, ניתן לדמות את ביצוע התוכנית ל"חוט מחשבה". החוט הכחול ידמה לנו את ה-*PC*, שיבצע פקודה אחר פקודה לפי הסדר שהתוכנית קבעה. החוט מייצג מסלול ייחודי של ביצוע פקודות התוכנית.

דוגמאות לתוכנות המבצעות מספר משימות בו זמן:

- דפדפן אינטרנט – מהזיר מידע מהרשת, ובו זמניון מציג על המסך תמונות ומידע שכבר נקרא.
- מעבד תמלילים *Word* – מעבד פקודות מהמקלדת, מציג על המסך גרפייה, ומריץ בדיקת איות.
- שרת אינטרנט – מעבד בקשות רבות משתמשים שונים בו זמניון ומקבל בקשות חדשות.

נבחן מספר דרכיים למימוש אפליקציה המבצעת מספר משימות במקביל:

- האפשרות הראשונה היא לא לאפשר מקבילות, כלומר האפליקציה לא תוכל לבצע יותר מפעולה אחת בו זמן. גישה זו בעייתייה כי למשל ב-Word, אם האפליקציה תוטר על ה-CPU בכל פעם שהיא צריכה לעבוד קלט שהגיע מהמשתמש היא ת Abed זמן CPU יקר וחשוב לטובות אפליקציות אחרות וכן היא לא תוכל לבצע פעולות חשובות נוספות באופן עיל.
- אפשרות שנייה היא פירוק האפליקציה למספר תהליכיים (Processes) קטנים יותר. זה אומנם יפתר את הבעיה הקודמת, אך CUT יש בעיות נוספות:
 - המשימות תלויות אחת בשנייה וכן נדרש לשתף הרבה מידע \Leftarrow שימוש רב ב-IPC.
 - אך ה-IPC זו פעולה יקרה, מבולגנת ואיטית שמגדילה את התקורה.
 - לא ניתן לשתף את הקוד בין התהליכים \Leftarrow שכפול הקוד לכל תהליך \Leftarrow בזבוז זיכרון.
- אפשרות נוספת (וטובה!) היא על ידי, שימוש ב-*Multi-threaded Process*.

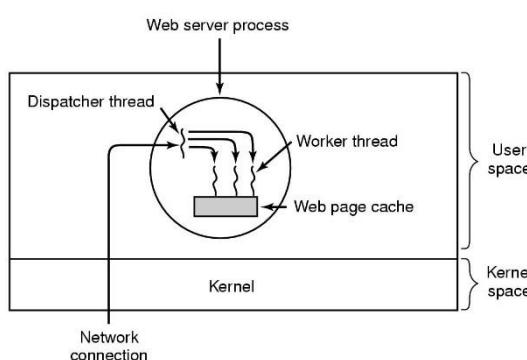


:Multi – threaded Process
חלוקת של תהליכיים לתהליכונים שונים. לכל תהליכון יש Registers מסוימים, וכי השווא ברתגול הקודם, הם חולקים את הקוד, משתנים גלובליים, המידע שנמצא ב-heap וקבצים משותפים. באופן זה, כל תהליכון ישמור על "חוט מחשבה" משל עצמו. תהליכונים אמורים חולקים את אותו קוד, אך הם יכולים לזרע על חלקים שונים שלו.

:Multi – Threaded Web Server

לשרת יש מעבד אחד, Web Server Process, ובתוכו הותהליקון Dispatcher Thread שתחזק ידו להאזין לרשota. בכל פעם שמתאפשרת בקשה חדשה ליצור Worker Thread חדש (כמו שתהליך יכול ליצור Thread, גם יכול ליצור Thread), תפקידו של ה-Dispatcher לטפל בה. لكن אם התקבלו 1,000 בקשות, ניצור 1,000 תהליכונים שייטלו בהן. (במה שמשך נראה דרכים לייעול) התהליכונים יעבדו במקביל, וכך ה-Dispatcher וה-Worker לא חוסמים אחד את השני. כל עוד המידע ב-Web Page Cache משותף, ניתן לטפל בכל הבקשות במקביל, וכיון שלכל אחד מהםRegisters ומחסנית אחרת, הם לא יפריעו אחד לשני. בכל רגע נתון, מספר התהליכונים יהיה מספר הבקשות מהשרת + ה-Worker Thread.

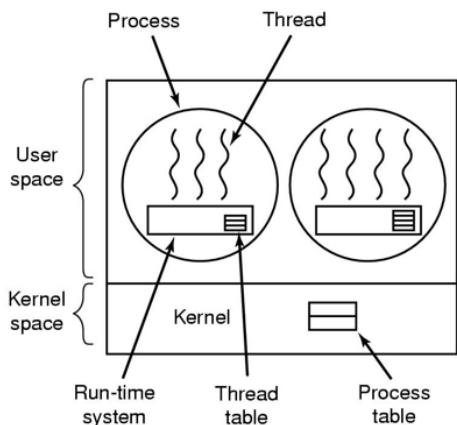
הערה: מכיוון שתהליכיים נוצרים מאותו תהליך, יוכל לוותר על מספר הגנות לטובות גמישות ויעילות.



:User Level Threads

כפי שראינו בתרגול, ניהול התהילכנים ממומש ברמת האפליקציה, מערכת הפעלה לא מודעת לקיוםם של התהילכנים, ושינוי התהיליכון שרצ ברגע נתון לתהיליכון אחר לא דורש הרשות של מערכת הפעלה. כמו כן תעדוף התהילכנים נקבע על ידי התוכנה.

A user-level threads package



הבעיות העיקריות של *User Level Threads* הן:

- כל *Call System* (למשל לטופת *I/O*) חוסם את כל התהיליך, ומכיון שהוא אחד הוא שמאפשר לתהילכנים לירוץ, אף תהיליכון אחר לא יירוץ בזמן זה.
- "תיכון מצב שבו *Thread* מסויים "ינצל" את העבודה שאינו "Time Sharing" לטובתו ולא יותר על ה-*CPU*, מה שמנע מתהילכנים אחרים לירוץ. בגיןוד למערכת הפעלה, תהיליכון לא יכול לכפות עצירה של תהיליכון אחר, ולא תמיד יש פסיקת שעון שעצרת את הריצה של התהילכנים.

:Kernel Level Threads

טהילכנים המנוהלים על ידי מערכת הפעלה. מערכת הפעלה מודעת לתהילכנים ומתיחסת אליהם באופן דומה לתהילכים, למעט העובדה שהם חולקים מידע ולכל אחד מהם יש רегистרים משלה עצמו. ככלומר ב-*Process Table* (שם נמצא ה-*PCB*) יהיה מצביע ל-*Threads Table*, טבלה המגדת את התהילכנים של אותו תהיליך, שם מערכת הפעלה תשמור רק את הרגיסטרים של כל תהיליכון, ולא את הקוד וה-*Data* של כל אחד. מערכת הפעלה תדוע לעשוות בرمת התהילכנים. ככלומר אם תהיליכון אחד יחסם, לא כל התהיליך יהפוך ל-*Waiting*, אלא רק התהיליכון הספציפי שנחסם. מעבר בין תהילכנים שונים מתבצע דרך מערכת הפעלה באמצעות *System Calls* וכן התקורת גודלה יותר.

חשוב להבדיל בין *Kernel Threads* לבין *Kernel Threads*:

- *User Mode - Kernel Level Threads* – תהילכנים המנוהלים על ידי מערכת הפעלה וריצים ב-*Kernel Space*.
- *Kernel Space - Kernel Threads* – תהילכנים המנוהלים על ידי מערכת הפעלה וריצים ב-*Scheduler*. לדוגמה

יתרונות לשימוש בתהיליכונים:

- יצירת *Thread* זו פעולה מהירה יותר (פ' 100 – 30) מיצירת תהיליך.
- עצירת *Thread* זו פעולה מהירה יותר מעצרת תהיליך.
- *Switch* בין שני תהיליכונים זו פעולה מהירה פי 5 מ-*Switch* בין תהיליכים.
- תהיליכונים משותפים *Data* וקבצים, ולכן הרבה יותר קל, זול ונוח לשתף מידע ולתCKER שר מבלי לעرب את מערכת הפעלה.

ההבדלים בין Threads ל-Proceses

<i>Processes</i>	<i>Kernel Level Threads</i>	<i>User Level Threads</i>
מוגן אחד מהשני, ודורש את מערכת ההפעלה בשבייל לתCKER ש	חולקים זיכרון משותף, תקשורת פשוטה בין אחד לשני, שימושי בהקשר של <i>Application Structuring</i>	
תקורה גבוהה: כל פעולה דורשת ביצוע <i>trap</i> , הרבה עבודה.	תקורה בינונית: פעולות דורשות ביצוע <i>Trap</i> , אבל מעט עבודה.	תקורה נמוכה: הכל נעשה ב- <i>User Space</i>
יש עצמאויות: אם תהיליך/טהיליכון נחסם, זה לא משפיע על האחרים.		אם <i>Thread</i> אחד נחסם, כל התהיליך נחסם,
יכולים לרוץ במקביל על מעבדים שונים.		כולם חולקים את אותו המעבד ולכן בכל זמן נתון רק <i>Thread</i> אחד רץ.
המערכת דורשת <i>API</i> ספציפי, התוכניות לא נידות (לא ניתנות להעברה ממוקם למקום)		ספריית <i>Threads</i> אחת זינה למספר מערכות.
ניהול תהיליכונים כללי לכולם		ניתן לנשל תהיליכונים באופן שמותאם יותר לאפליקציה הספציפית.

הערה: ה-CPU לא מודע לקיומם של *Threads* (לא *Kernel Level User Level*_threads)

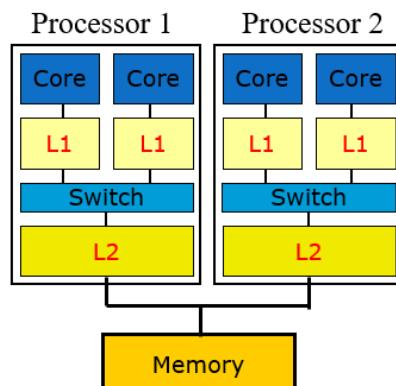
מקביליות אמיתית

מקביליות אמיתית בرمת תוכנה - Multi – core – processor :

עד כה ראיינו שמעבד יחיד מסוגל להריץ פקודה אחת בכל רגע נתון. נבחן בקצרה דרכיים למקביליות אמיתית בرمת התוכנה (ב-*CPUs* מתקדמים יש לעיתים מקביליות בرمת החומרה)
Multi – processor – על אותוلوح אם היו מספר *CPUs*, וכך נוכל להריץ מספר אפליקציות במקביל, כל אחת על *CPU* אחר. אם אפליקציות ירצו לתקשר ביניהן, התקורה תגדל ממשמעותית.
Multi – core CPU – יהיה *CPU* אחד עם מספר ליבות וכאן הוא יוכל לחשב מספר פעולות בו זמנית. מעבד עם מספר ליבות הוא נפוץ הרבה יותר ונמצא ברוב המעבדים החדשניים.

Multi – core processor

בקורס שלנו לא נתעמק בהבדלים החומרתיים בין *Multi – core* ל-*Multi – processor*,
 אלא נתמקד בהבדלים הלוגיים ביניהם. מערכת הפעלה חייבת להיות מודעת למספר המעבדים והlivbot
 במחשב. כל ליבה מבצעת הוראה בלתי תלולה אחרת. השוני הלוגי בין ליבות למעבדים הוא רמת הזיכרון
 שהם משתפים ועלות התקשורת ביניהם. בתמונה למטה יש 2 מעבדים, וכל מעבד 2 ליבות. לכל ליבה
 יש זיכרון L1 שקרוב אליה, ויש זיכרון L2 המשותף לכל הליבות של אותו מעבד, שהוא קצר יותר
 ائي ו יותר גדול, וכן גם התקשורת בין ליבות שונות איטית יותר. בנוסף, יש זיכרון המשותף לכל
 המעבדים שדרכו הם יכולים לתקשר אך הגישה אליו היא האיטית ביותר מבין הנ"ל.



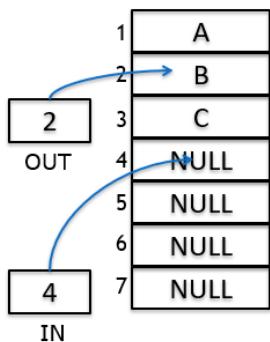
Two processors with two cores and shared memory

:Hardware – level Threads, Hyper – Threading

תהליכיונים שה-*CPU* מודע אליהם, וכן הוא יכול להחליף ביניהם מבלי לערब את מערכת הפעלה, ככלומר
 התקורה גמוכה יותר. מערכת הפעלה מבchinetta חושבת שיש *Core*-*m* אחד. ביום, בדרך כלל כל
Core מחולק לשני *Hyper Thread* שונים, וכן אם למחשב יש 4 ליבות, מערכת הפעלה תחשב שיש
 למחשב 8 ליבות.

Synchronization

נבחן את הבעיה הבאה: בהינתן שני תהליכי / תהליכי המעוניינים להדפס משוחה למסך, שניהם יוסיפו את מה שהם רוצים להדפס ל-*Spooler* – התור ממנו המדפסת מדפסה.



ה-*Spooler* משותף לכל התהליכים, نتيיחס אליו כל מערך עם 2 מצביעים:

- מצביע ראשון *IN* – מצביע למקומ הפניו הבא במערך.
(באיור, את הג'וב הבא נכתב בתא מס' 4)
- מצביע שני *OUT* – מצביע לג'וב הבא שציריך להדפס.
(באיור, ג'וב מס' 2)
- *NULL* מצבין שהתא ריק.

ה-*spooler* וה-*IN* משותפים לכלם. כדי להויף ג'וב ובצע את הקוד הבא:

```
spooler[IN] = job  
IN ++
```

נראה 2 דוגמאות הריצה:

- תהליך 1 מוסיף ג'וב וمعدכן את *IN*.
- יהיה *Context Switch*
- תהליך 2 מוסיף ג'וב וمعدכן את *IN*.

Process 1: Spooler[IN] = JOB1

Process 1: IN++

<Context Switch by OS>

Process 2: Spooler[IN] = JOB2

Process 2: IN++

במקרה זה לא יהיה בעיות. לעומת זאת במקרה הבא:

Process 1: Spooler[IN] = JOB1

<Context Switch by OS>

Process 2: Spooler[IN] = JOB2

<Context Switch by OS>

Process 1: IN++

<Context Switch by OS>

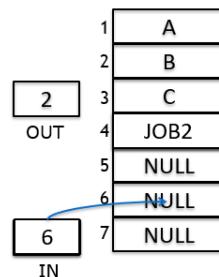
Process 2: IN++

- תהליך 1 מוסיף ג'וב.

- יהיה *Context switch*.

- תהליך 2 מוסיף ג'וב אך *IN* לא התעדכן עדין ולכן הוא דרש את הג'וב שהוסיף תהליך 1. בנוסס, בתא 5 נוצר רוח ויכשגע לתא הריך נראה *NULL*, נסיק כי כל התאים שאחורי ריקם ולכן כל העבודות שממוקמות אחרי ג'וב 2 אבודות.

כלומר יתקיים:



כל אלו בעיות הנוצרות בגלל כשל בסyncronization.

ניתן לראות דוגמאות דומות נוספות במצבה של הריצה 4 שkopiot 29 – 27.

ונגידר את הבעיה באופן פורמלי:

קריאה / כתיבה למשאב משותף ללא תיאום עם ישויות אחרות (תהליכיים, תהליכיונים, ליבות, מעבדים...).
 קטע הקוד שניגש למשאים המשותפים נקרא *Critical Section*, הוא יכול להיות הוראה אחת או קוד ארוך. המטרה היא למנוע מ-2 תהליכיים / תהליכיונים להריץ אותו כדי למנוע גישה כפולה למשאב. הפתרון הוא *Mutual Exclusion* – אלגוריתמים שתפקידם למנוע ריצה כפולה של אותו קוד.
הערה: *Mutual Exclusion* לא מספיק כדי לפתור את הבעיה.
 (למשל כאשר סדר ההרצה חשוב, נרחיב בהמשך)

בURITY הסינכרונייזציה היא בעיה ישנה שהוגדרה על ידי דיקוסטרה.

דיקוסטרה הגידר ומידל את המושג "קוד קרייטי".

בහינתן קטע קוד, נחלק את הקוד שלנו לקוד קרייטי ולשארית (כל מה שלא חלק מהקוד הקרייטי) *.Entry Code* וקוד יציאה *.Exit Code*.

הבעיה: כתיבת קוד כניסה וקוד יציאה כדי להבטיח את המאפיינים הבאים:

1. Mutual Exclusion (מניעה הדדית) - נרצה למנוע מ-2 תהליכיים

להריץ את קטע הקוד הקרייטי שלהם במקביל.

2. התקדמות - אם קיים תהליך שמנסה לגשת לקטע הקוד הקרייטי שלו,

קיים תהליך כלשהו (לא בהכרח זה שנייה, ולא בהכרח באותו זמן)

שיקבל גישה לקטע הקוד הקרייטי שלו. (כלומר לא יקרה מצב שתהליך

מנסה לגשת לקטע קוד הקרייטי שלו, אף תהליך לא יקבל גישה,

אחרת התוכנית לא תתקדם והפתרון לא יהיה פתרון טוב)

3. Starvation Freedom (מניעת הרעה) - אם תהליך רוצה להיכנס

לקטע קוד הקרייטי שלו, הוא מתישחו יכול להיכנס (אך יכול להיות שאחרים יכנסו לפניו).

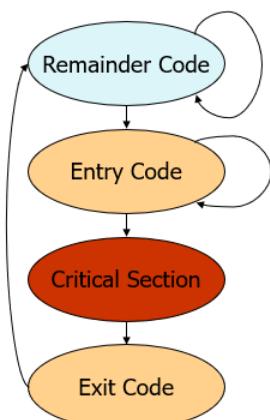
נשים לב שם תנאי 3 מתקיימ, תנאי 2 בהכרח מתקיימ.

4. כלילות - הקוד לא יהיה ספציפי למערכת מסוימת, האלגוריתם צריך לעבוד לכל מערכת בלי

תלות במספר המעבדים, ליבות וכו'.

5. אין לחסום בשארית – תהליך שרצ בשארית הקוד שלו (כלומר לא בקטע קוד הקרייטי), לא ימנע

מתהליכיים אחרים לגשת לקטע קוד הקרייטי שלהם.



נראה 2 פתרונות אפשריים. נניח שהמעבד לא יעצור את התוכנית בקטע קרייטי ונניח שזמן הריצה בקטע הקרייטי הוא זמן ריצה סופי.

פתרון ראשון (פתרון לא טוב):

- .*Enable Interrupts* ובעצם *Entry Code*, וב-*Exit Code*, *Disable Interrupts* נבצע ב-*CPU*.
- פתרון זה מספק את תנאים 1,2,3,5, אבל הוא לא מספק את תנאי 4 כי אנחנו יוצרים תלות בחומרה - אם יש יותר מ-*CPU* אחד הפתרון לא יעבד. הסיבה היא שלא ניתן לעשות *Disable Interrupts* ב-*CPU* אחד ל-*CPU* אחר.
 - לתהליכיים ברמת *User* לא צריכה להיות הרשות לבטל *Interrupt*.
 - ביטול *Interrupt* צריך לקרטות זמן קצר מאוד.

פתרון שני (עדין פתרון לא טוב):

נשתמש במשתנה גלובלי שיתאים בין 2 תהליכיים,

במיידת 0 Thread 0 ו-1 Thread 1 בדוגמה משמאלי.

במידה ו-0 Thread 0 מגיע לקטע קוד הקרייטי שלו, כל עוד $turn == 1$ סימן ש-1 Thread 1 מרים את קוד הקרייטי שלו ולכן הוא ימתין.

כש-1 Thread 1 יסימן, הוא יdag לשנות את *turn* ל-0,

ולכן 0 Thread 0 יצא מהלולאה ויכנס לקטע הקוד הקרייטי שלו.

כש-0 Thread 0 יסימן את קוד הקרייטי הוא יעדכן את *turn* ל-1, ואז 1 Thread 1 יוכל גם כן להריץ את קוד הקרייטי שלו.

כלומר יצרנו מעין חילוקה של תורות בין התהליכים.

אלגוריתם הנ"ל יש 2 מוגבלות:

- הוא מתאים רק לשני *Threads/Processes*. (נלמד אלגוריתמים מתקדמים יותר בהמשך)
- הוא משתמש במשתנה גלובלי *turn* שאנו מניחים שאף אחד לא ניגש אליו חוץ מהתהליכים.

פתרון זה מספק את תנאים 1,2,4 אך לא את 3 ו-5.

(למשל במקרה בו 1 Thread לא ישנה את *turn* להיות 0 בשום שלב כי הוא עוצר לפני)

הוכחה פורמלית לכך שמתקיים התנאי הראשון *Mutual Exclusion*:

נניח בשלילה ש-2 תהליכיים נמצאים בחלק הקרייטי בקוד שלהם בו זמןנית. בלי הגבלת הכלליות, בשלב

ש-0 Thread 0 עבר את *Entry Code* ונכנס למקטע הקרייטי שלו, $0 = turn$. המיקום היחיד *turn*-

יכול להשתנות ל-1 הוא כאשר 0 Thread מסיים את מקטע הקוד הקרייטי. כלומר $0 = turn$ לאורך הריצה

של מקטע הקוד הקרייטי של 0 Thread. נסיק מכך ש-1 Thread נמצא ב-*Entry Code* שלו כל עוד

Thread 0 נמצא ב-*Critical*, בסתיו להנחה.

הערה: צריך לדעת לנוכח הוכחות פורמליות כנ"ל لكن חשוב לעבור על ההוכחה.

תרגול 4

Thread Pools

עד עכשיו תיארנו תהליכיונים בהקשר של פיצול משימה גדולה לתתי משימות, וכל משימה היא תהליכון. אך מה לגבי משימות שקשה לפצל אותן לתחתי משימות? אין נוכן להשתמש בתהליכיונים ביעילות?

1. הצעה ראשונה: ניצור תהליכון לכל בקשה / קרייה לפונקציה. אך יש 2 בעיות עיקריות:

- ייצור תהליכון לכל בקשה יגרום לתקורה גדולה.
- מספר התהליכיונים הוא דינמי ויכול להיות גדול מאוד עד כדי עומס על המערכת, ניצול הזיכרון הקיים וקריסת התוכנית. בנוסף, נרצה להגביל את מספר התהליכיונים שניגשים לשאבי המערכת, למשל מאות תהליכיונים שפונים לדיסק הקשיח בו זמינים.

2. הצעה שנייה: Thread Pool Concept – אוסף של תהליכיונים המאותחלים עם אחת (למשל כשמערכת הפעלה עולה), ותוכניות יכולות להשתמש בהם לאורך הריצה. תוכנית תבקש מתהליכון פנוי לבצע פעולה, ואם אין תהליכון צזה היא תמתין.

Thread Pools in Client – Server Applications

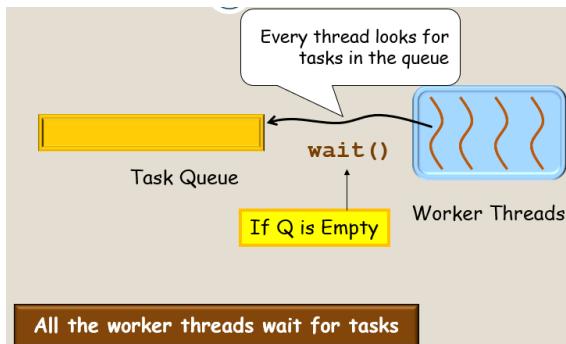
מודל Client – Server הוא מבנה אפליקטיבי המחלק משימות או עומס עבודה בין שרת ללקוחות שלו. השרת מצוי לפועל בזמנים של לקוחות (אפליקציות) בעזרת תהליכיונים קיימים. אם מגיעות יותר בזמנים ממספר תהליכיונים הפנויים, השרת מאלץ את הלקוח להמתין עד שתהליכיון מתפנה.

מימוש אפשרי:

בהתנחת מאגר (pool) של תהליכיונים, כל משימה מבקשת תהליכיון מתוך המאגר, ומחזירה אותו לאחר שהסתיימה. כאשר אין תהליכיונים במאגר, התהליכיון שהפעיל את המשימה ימתין עד שהמאגר לא ריק.

```
While(true) {  
    Wait for request r  
    Wait for thread t in the pool to be available  
    Run r on t  
}
```

המימוש הנ"ל בעיתוי – נרצה להימנע מהמתנה לתהליכיונים ריקים ולא לחסום תהליכיון בשביל זה.



פתרון אפשרי: נתזקק "טור משימות" (Tasks Queue). תהליכיון שמעוניין לבצע משימה יוסיף אותה לטור. תהליכיונים פנויים יבדקו אם יש משימות בטור שצורך לבצע, ואם לא הם ימתינו. ככלומר, העברנו את זמן המתנה מתהליכיון פועל לתהליכיונים פנויים (שלא מפריע לנו שהם ממתינים).

מה קורה לתהיליכון מתוך המאגר אם המשימה שהוא ביצע זרקה חריגה? או לחילופין המשימה ממחכה לקלט מ-*O/I* אך הוא לא מגיע? כדי לתחזק מאגר תהיליכונים תקין ופעיל, נצטרך לחסום זמן ריצה של קריאות *O/I* בעזרת *wait(time)* ולתפואו חריגות זמן ריצה כדי למנוע מהטהיליך להיסגר.

גודל המאגר:

איך נדע כמה תהיליכונים פעילים לתחזק ב-*Thread Pool* שלנו?

אם נתחזק מספר גדול של תהיליכונים:

- יכולה להיות תקורהגדולה של יצירת תהיליכונים (בעיה מינורית)
- כל תהיליכון צריך ذיכרון ומשאבים (למשל קבצים פתוחים).
- יכולה להיות תקורהגדולה בהקשר של *Context Switch*.

אם נתחזק מספר קטן של תהיליכונים:

- ניצולת נמוכה יותר – ככל שמספר התהיליכונים יותר קטן, יש יותר סיכוי שככל התהיליכונים מבצעים פעולות *O/I* (כמו גישה לדיסק), אז אף הטהיליך לא מעסיק את המעבד.
- זמן המתנה גבוהים יותר – משימות קצרות ימתינו זמן רב יותר למשימות גדולות שיטוימנו בכך נרצה שגודל המאגר יהיה מותאם למספר המשימות הצפוי.

Tunning the Pool Size

שינוי מספר התהיליכונים בזמן ריצה. לרוב נגידר גם מספר מינימלי ומקסימלי אפשרי של תהיליכונים. גישה נפוצה היא לבצע *Tunning* לפי העומס. כשייש עומס – מוסיף תהיליכונים ולהפר.

פתרונות אפשרי:

– הזמן הממוצע שתהיליך ממתיין לפעולות איטיות למשל *O/I* – *Blocked Time (BT)*

– הזמן הממוצע שלוקח ל-*CPU* לבצע פעולות של הטהיליך – *Service Time (ST)*

(ב-2 המקרים אנחנו לא מחשבים את הזמן שתהיליכון חיכה ל-*CPU* שיתפנה, כמו ב-*Queue (Ready Queue)*)

בממוצע, אם נתחזק $1 + \frac{BT}{ST}$ תהיליכונים נגרום לניצול מלאה של ה-*CPU*.

למשל אם $20 \text{ BT} = 5 \text{ ST}$ אז $1 + \frac{20}{5} = 5$ תהיליכונים יגרמו לניצול גבוה.

ניהול תהליוכנים

בזמן אתחול, ה-`main` מכיל תהליון דיפולטיבי בודד. כל שאר התהליוכנים צריכים להיווצר באופן ידני על ידי המתכנת. השתמש במחלקה `pthread` שמייצרת *Kernel Level Threads*, קלומר תהליוכנים שמערכת הפעלה מודעת אליהם. כדי לייצר תהליון חדש משתמש בפונקצייה:

```
int pthread_create(pthread_t *thread,
                  const pthread_attr_t *attr = NULL,
                  void *(*start_routine)(void *),
                  void *arg);
```

תהליון מסוים באמצעות הדרכים הבאות

- התהליון עצמו קרא ל-(`pthread_exit(void *status)`)
Return
- התהליון מסיים את הריצה בעזרת `pthread_cancel` כדי לבטל את התהליון שלו
- התהליון אחר קרא ל-`pthread_exit` כדי לשבור את התהליון שלו
- התהלייר שלו (שבתווך הוגדר התהליון) הסטיים על ידי קריאה ל-`exit`
או אחרי שהוא סיים את הריצה בהצלחה.

הערה: תהלייר יכול להסתיים לפני שכל התהליוכנים שלו סיימו וזה יגרום לסגירה של כל התהליוכנים מיד, ככלומר לא בטוח שכל התהליוכנים סיימו את הפעולות שלהם !

נראה דוגמה לייצרת 5 תהליוכנים

```
#define NUM_THREADS 5
void *PrintHello(void *arg) {
    int *index = arg;
    printf("\nThread %d: Hello World!\n", *index);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int res, t;
    for(t = 0; t < NUM_THREADS; t++){
        printf("Creating thread %d\n", t);
        res = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (res < 0){
            printf("ERROR\n");
            exit(-1);
        }
    }
}
```

ה-`main` מייצר 5 תהליוכנים שכל אחד מהם-Amor להדפס הודעה. אך יכול לקרות מצב בו ה-`main` יסתהים לפני התהליוכנים ואז לא בטוח שכולם הספיקו להדפיס את ההודעה.

Joining Threads

כדי לפטור את הבעיה הנ"ל נוכל להשתמש בפונקציה:

```
int pthread_join(pthread_t thread, void **argvalue_ptr);
```

במידה ואנחנו מרים את תחילICON B מתוך תחילICON A (בעזרת `pthread_create`)
קריאה לפונקציה `join` עם תחילICON B כפרמטר, תגרום לכך ש-A יהיה במצב
Blocked עד ש-B יסימם. כך נוכל להיות בטוחים שהתחילICON הסתיים בהצלחה.

המשך לדוגמה מעמוד קודם כדי לתקן את הבעיה:

```
// main thread waits for the other threads
for(t = 0; t < NUM_THREADS; t++) {
    res = pthread_join(threads[t], (void **)&status);
    if (res < 0) {
        printf("ERROR \n");
        exit(-1);
    }
    printf("Completed join with thread %d status = %d\n", t, *status);
}
```

קטע הקוד הנ"ל צריך להיות בהמשך ה-*main* מהדוגמה הקודמת, ולא במקומם. בקטע קוד הקודם ביצענו 5 קריאות ל-`pthread_create`, וعصיו ביצעו 5 קריאות ל-`join`, כדי לוודא שככל תחילICON סימן את הריצה. כשהולולה הנ"ל תסתיים זה סימן שככל התהילICONים ביצעו את עבודתם.

Mutex

- מצב שבו מספר תחילICONים או תחילICONים מנוטים בו זמינות להשיג גישה למשאב משותף. הראמו בעמודים קודמים בסיכון דוגמה ל-*Race Condition* (*Race Condition* בתרגול 3).

Mutex (Mutual Exclusion Algorithms)

אלגוריתמים/אובייקטים שתפקידם למנוע גישה לקטיעי קוד קרייטיים בו זמינות על ידי 2 תחילICONים. המטרה היא לא למנוע גישה לקוד עצמו אלא למשאים משותפים במקביל. בהרצאה ראיינו 2 דרכים (לא מוצלחות) למנוע את המצב הנ"ל, נראה אחת נוספת.

Mutex הוא אובייקט שניתן להיות ב-2 מצבים, נועל או פתוח. כשתחילICON מסוים ניגש לאובייקט הוא "נועל" אותו, וכל שאר התהילICONים יאלצו לחכות שהוא יסימם. לאחר מכן תחילICON נוסף יוכל לנעול אותו ולעבוד עליו, וכן הלאה.

תהליך שימוש ב-Mutex:

- יצרה ותחול של משתנה *Mutex*
- מספר תהליכיון מנסים לנעול את ה-*Mutex*
- רק תהליכיון אחד מצליח, והוא שולט על ה-*Mutex*
- התהליכיון שהצליח מבצע מספר פעולות
- לאחר מכן הוא משחרר את ה-*Mutex*
- תהליכיון אחר משתמש על המנעול וחוזר על התהליכיון
- לבסוף ה-*Mutex* נהרס

כדי ליצור *Mutex* נשתמש בטיפוס *pthread_mutex_t* ונראה כיצד לאתחל אותו לפני השימוש:

- זמן הרצה - *pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;*
- או באופן דינמי - *pthread_mutex_init(mutex, attr)* (בצורה זאת אפשר לבחור הגדרות שונות ל-*Mutex*). כדי להשתמש בברירת המחדל אפשר לשלו *NULL* במקום *attr*)

כדי למחוק אובייקט *Mutex* מהזיכרון נשתמש ב-(*pthread_mutex_destroy(mutex)*)

כדי לנעול *Mutex* נשתמש ב-(*pthread_mutex_lock(*mutex)*). קרייה לפונקציה תנסה לנעול את המנעול, ואם הוא כבר נועל תהליכיון יעבור למצב *Blocked* עד שהמנעול יפתח.

כדי לפתוח *Mutex* נשתמש ב-(*pthread_mutex_unlock(*mutex)*). תוחזר שגיאה אם ה-*Mutex* נועל על ידי תהליכיון אחר או אם ה-*Mutex* כבר פתוח.

הערה: נשים לב שהמחלקה *pthread* מימוש את *Mutex* כך שהוא מקיים רק את *Mutex* כבר פותח. מבין כל הפרמטרים שראינו בהרצאה.

<u>Thread 1</u>	<u>Thread 2</u>
<pre> <i>pthread_mutex_lock(&mut_counter);</i> int a = counter; a--; counter = a; <i>pthread_mutex_unlock(&mut_counter);</i> </pre>	<pre> <i>pthread_mutex_lock(&mut_counter);</i> int b = counter; b--; counter = b; <i>pthread_mutex_unlock(&mut_counter);</i> </pre>

הערה: יש לבדוק בנוסף את ערכי ההחזרה של הפונקציות.

Deadlocks

בתמונה הבאה יש דוגמה למצב שנקרא *Deadlock* וצריך לשים לב אליו. כל תהליכיון גועל מנעל מסוים, אחרי זה כל תהליכיון מנסה לנעול את המנעל של תהליכיון השני אך הם לא יכולים כי שניהם גועלים.

התוכנית לא תתקדם ממש לשום מקום ומשם מגע השם *Deadlock*.



Monitors (Conditional Variables)

מוניטור הוא מבנה המכיל **Conditional Variable** ו- **Mutex**. הוא מאפשר לתהילכנים לבצע *Mutual Exclusion* תחת תנאים מסוימים. כלומר תהיליכון יכול להפעיל מניעול רק אם תנאי מסוים הוא אמת, אחרת הוא מאפשר לתהילכנים אחרים לניעול את המניעול לפניו. בנוסף, מונייטור מאפשר מגנן של שליחת סיגナル לתהיליכון אחר כדי "להעיר" אותו. נראה דוגמאות בהמשך.

הספרייה *pthread* תוכננה לסוג ספציפי של סנכרון - *Mutual Exclusion* - **Thread Pool** – במקירם בהם יש *Buffer*, למשל ה- **Bounded Buffer**.

נניח שה- *Tasks Queue* ריק, נרצה שהטהילכנים יכבו את עצםם, ורק כשtagיע משימה חדשה התור יתריע להם שיש משימה חדשה. בעוד זו אנחנו מונעים מצב שבו התהילכנים מוחכים למשימות באופן תמידי וחוסכים פעולות *CPU* מיותרות.

זהו **Readers – Writers** – למשל אם יש לנו רשות מקוורת בתוכנית, נרצה למנוע מצב בו תהיליכון אחד כותב לרשימה בזמן שתהיליכון אחר קורא ממנה (ואז הוא יכול לקרוא חצי מידע ישן וחצי חדש), אך לא נרצה למנוע מ-2 תהילכנים לקרוא מידע.

זהו **Barrier** – יצירת מחסום כללי. למשל תהילכנים שMRIIZIM אלגוריתם *MergeSort* נרצה להתחיל בתהיליך האיחוד של נתבי רשימות רק לאחר שכל הרשימות פוצלו. במקרה זה נרצה להוסיף מחסום שיגרום לתהילכנים לאחד רשימות רק אחרי שכל שאר התהילכנים סיימו לפצל אותן.

כדי ליצור *CV* משתמש בטיפוס *pthread_cond_t* ונוהה חייבים לאתחל אותו לפני השימוש:

- **זמן היצרה** - *pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;*
- או **באופן דינמי** - *(pthread_cond_init(cond, attr);* המצב של ה-*CV* הופך *initialized*)

כדי למחוק אובייקט *CV* מהזיכרון המשתמש ב-*pthread_cond_destroy(cond)*.
כדי לניעול תהיליכון מסוים עד ש-*CV* הופך ל-*True* – *pthread_cond_wait(cv, mutex)* – *True* – *pthread_cond_wait(cv, mutex)* משמש להגנה על בדיקת התנאי. הפונקציה מניחה שלפני הקראיה התהיליכון נעל את התהיליך *mutex*, מצא שהתנאי לא התקיים, ולכן הוא מעוניין להיכנס למצב *wait*. הפונקציה *wait* גורמת לתהיליכון לשחרר את *mutex* ולעבור למצב *Blocked* עד שקריאה לפונקציה *pthread_cond_signal(cv)* (או *pthread_cond_broadcast(cv)*), הפונקציה *wait* מתחילה מתבצעת מהתהיליכון אחר כדי להעיר את התהיליכון הנוכחי. לאחר הקראיה השליטה לתהיליכון.

כדי לשולח סיגナル מהתהיליכון אחד לאחר שימוש ב-*(pthread_cond_signal(cv))*. הפונקציה בודקת אם יש תהילכנים המתאימים לתנאי *cv* הנוכחי. אם לא, היא פשוט חוזרת. אם יש תהילכנים המתאימים, היא מעירה את אחד מהם. לא ניתן להניח כלום על סדר הפעולה של התהילכנים. אפשר להשתמש בפונקציה (*pthread_cond_broadcast(cv)* או בollowah) כדי להעיר את כל התהילכנים המתאימים לתנאי ולא רק את אחד מהם.

דוגמאות

שימוש ב-Mutex לבעית CV (מוסבר ולא עיל, גורם ל- CV)

Thread 1:	Thread 2:
<pre> While (true){ mutex_lock(varMutex) if (canUseVar){ //Use var; canUseVar = false; break; } mutex_unlock(varMutex) } </pre>	<pre> mutex_lock(varMutex) init(var); canUseVar=true; mutex_unlock(varMutex) </pre>

שימוש ב-CV לטובות אותה בעיה

Thread 1:	Thread 2:
<pre> mutex_lock(varMutex) if (!canUseVar){ cv_wait(varCV, varMutex) } //Use var; mutex_unlock(varMutex) </pre>	<pre> mutex_lock(varMutex) init(var); canUseVar=true; cv_signal(varCV) mutex_unlock(varMutex) </pre>

שימוש ב-CV לטובות Bounded Buffer

- Data structures:
 1. Queue<T> q
 2. pthread_mutex_t qM
 3. pthread_cond_t qCV
- Reader Flow:
 1. pthread_mutex_lock(qM)
 2. If (q.empty())
 3. Wait (qCV, qM); //waiting for an element to be written
 4. Element e = q.pop(); //here we have both a lock and element
 5. Pthread_mutex_unlock (qM)
 6.

```

#define NTHREADS 5
pthread_mutex_t *lock;
pthread_cond_t *cv;
int ndone;

typedef struct {
    int id;
} TStruct;

main() { //Checking of return values omitted for brevity
    TStruct ts[NTHREADS];
    pthread_t tids[NTHREADS];
    int i;
    void *retval;
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cv, NULL);
    ndone = 0;
    for (i = 0; i < NTHREADS; i++)
        ts[i].id = i;
    for (i = 0; i < NTHREADS; i++)
        pthread_create(tids + i, NULL, barrier, ts + i);
    for (i = 0; i < NTHREADS; i++)
        pthread_join(tids[i], &retval);
    printf("done\n");
}

void *barrier(void *arg) { //Checking of return values omitted for brevity
    TStruct *ts;
    int i;
    ts = (TStruct*) arg;
    printf("Thread %d -- waiting for barrier\n", ts->id);
    pthread_mutex_lock(lock);
    ndone = ndone + 1;
    if (ndone < NTHREADS)
        pthread_cond_wait(cv, lock);
    else
        pthread_cond_broadcast(cv);
    pthread_mutex_unlock(lock);
    printf("Thread %d -- after barrier\n", ts->id);
}

```

Semaphores

סمفור זו הכללה של *Mutex*. סمفור מונע גישה למשאב משותף על ידי יותר מ-X תהליכיון במקביל.
(כלומר, *Mutex* הוא סمفור המונע גישה של יותר מתהליכיון 1 במקביל למשאב משותף)
נרצה להשתמש בסمفור במקרים בהם אם יותר מתהליכיון אחד ייגש למשאב זה יהיה בסדר,
אבל יותר מ-X תהליכיון ייגשו זה כבר יגרום לביעות (למשל עומס על המשאב).

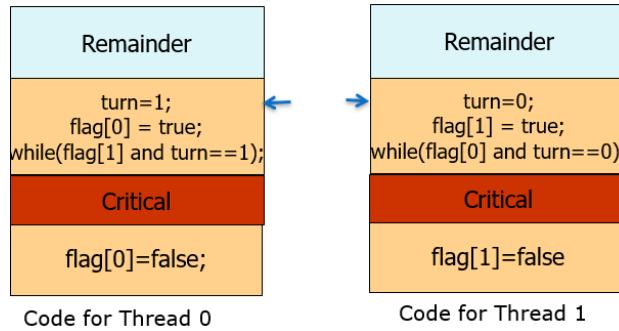
איך עובד סمفור?

הסמפור מאותחל עם ערך מסוים, *X*, ובכל פעם שתהליכיון ניגש לסמפור, הערך יורד ב-1.
שתהליכיון מסויים, הערך עולה ב-1. אם תהליכיון מנסה לגשת לסמפור כשהערך שווה ל-0,
הוא נחסם עד שתהליכיון אחר מסויים ומעלה את הערך.

איך משתמש בסمفור?

- סمفור הוא משתנה רגיל מטיפוס: `sem_t semaphore;`
- נאותחל אותו בעזרת: `int sem_init(sem_t *sem, int pshared, unsigned int value);`
- `sem` – מצביע לסמפור שצריך לאותחל.
- `pshared` – דגלו לשיתוף הסمفור בין תהליכיון שונים (לרוב לא נשתק וನשלח 0)
- `value` – הערך של הסمفור.
- נשחרר אותו בסיום בעזרת `int sem_destroy(sem_t *sem)`
- (הפונקציות `sem_init` ו-`sem_destroy` מוחזירות מספר שלילי אם הייתה שגיאה)
- כדי להקטין את הסمفור ב-1, נשתמש ב-`int sem_wait(sem_t *sem);`
- כדי להגדיל את הסمفור ב-1, נשתמש ב-`int sem_post(sem_t *sem);`

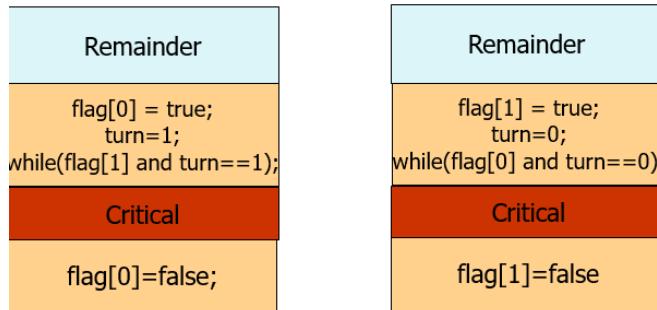
אלגוריתמים לפתרון Mutual Exclusion



במקרה בו *Thread 0* ירץ את השורה הראשונה $turn = 1$ ולאחר מכן יהיה *Context Switch* והתנאי בולולאה של *Thread 1* יהיה $flag[0] = false$ (כי $flag[0] = false$) ולכן יוכל לקטע קוד הקרייטי. לאחר מכן יבוצע *Context Switch* נסוף (*Context Switch* של *Thread 1* נושא השורה הראשונה של *Thread 0*), ולכן התנאי בולולאה של *Thread 0* יהיה $false$ וניכנס לקטע קוד הקרייטי ביחס עם *Thread 1*. כלומר לא מתקיים *Mutual Exclusion*.

האלגוריתם של פיטרסון

לעומת זאת, אם נחלף בין השורה הראשונה לשניה בכל תהליכיון:



מתקיים *Mutual Exclusion*, הוכחה:

נניח ש-2 התהליכיונים נמצאים בקטע קוד הקרייטי שלהם, ובלי הגבלת הכלליות ש-0 הוא הרראשון שעזב את *the-h-Entry Section* ונכנס לקטע קוד הקרייטי. לעומתו יצאנו מהלולה של *Thread 0* וכן או $flag[1] == false$ או $turn == 0$. נבחן את המקרים:
 1. *flag[1] == false* – במקרה זה *Thread 1* לא הירץ את השורה הראשונה של *the-h-Entry Section* שלו ולכן *Thread 1* יוכל לולאה שלו מתקיים $0 == turn == true$. אך גם $flag[0] == true$, ולכן תנאי הלולאה מתקיים ולכן *Thread 1* לא יוכל לקטע קוד הקרייטי שלו, סטירה.
 2. *turn == 0* – במקרה זה *Thread 1* ביצעה את שורה 2 ב-*Entry Code* שלו אחרי שבוצעה שורה 2 של *Thread 0* (אחרת $turn == 1$), ולכן גם שורה 1 של *Thread 0* בוצעה. לעומתו יממש *Thread 1* עד ש-0 $flag[0] == true \Leftarrow$ תנאי הלולאה של *Thread 1* מתקיים \Leftarrow הוא ימתין עד ש-0 $flag[0] == true$ \Leftarrow סטירה לכך. בסופי קטע הקוד הקרייטי ישנה את $[0] == flag[1] \Leftarrow$ סטירה לכך ששנייהם בקטע קוד הקרייטי ביחס.

מתקיים *No Blocking in the Remainder* כי הנקנו *sh-turn* ו-*flag* לא משתנים ב-*Thread*. **מתקיים**, *Starvation Freedom*, הוכחה:

נניח בשלילה ש-1 *Thread* מנסה לגשת לקטע קוד הקרייטי אך נעצר בשורה 3 לתמיד. במקירה זה 0 *Thread* נכנס לקטע קוד הקרייטי שלו מספר אין סופי של פעמים, אך בפעם השנייה ש-0 נכנס ל-*Entry Section* הוא ביצע $turn = true$ ולכן בפעם הבאה ש-1 *Thread* יקבל שליטה על ה-*CPU* תנאי הלולאה לא יתקיים והוא יכנס לקטע קוד הקרייטי שלו בסתריה. הערה 1: בדוגמה מהרצתה קודמתה הנחנו שקטע הקוד הקרייטי רץ ללא תלות והנחה זו תקפה גם כאן. הערה 2: הקוד הנ"ל תקף ל-2 תהליכיים בלבד (קשה מאוד להרחיב אותו יותר).

בנוספַּה הפתרון השתמש בלולאת *Wait Busy*, בהמשך נראה פתרון טוב יותר.

מתקיים, הוכחה: אם **מתקיים** *Starvation Freedom* גם *Progress*.

הגדירות

— מצב בו סדר הרצאה של רצף פקודות ב-*CPU* משנה את התוצאה הסופית. כלומר בהינתן רצף של פקודות, אם נשנה את סדר הרצאה שלהם נקבל תוצאות שונות. קשה מאוד לדבג *Race Condition* כי הוא יכול לקרוות במקרים מסוימים שקשה לשחרר.

— הוראה שמובוצעת בשלב אחד ולן הפעלת ההוראה לא תיעצר באמצעות הוראה אחרת. למשל פקודות קרייה / כתיבה לזיכרון (רגיסטר בודד. במקרים בהם מדובר על יותר מרגיסטר אחד זה לא מובטח). בנוספַּה פקודה כמו $1 + x = x$ היא לא אוטומית כי היא דורשת 3 פעולות. (קריאה הרגיסטר x בזיכרון, חישוב $1 + x$, כתיבת הערך החדש בזיכרון)

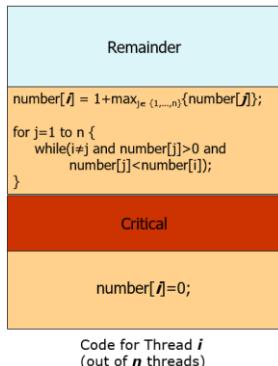
— טכנית בה תהליך רץ בלולאה כל עוד תנאי מסוים מתקיים, למשל: במקירה זה התהליך מבזבז זמן חישוב של המעבד (תקורה גבוהה) $while(flag[0] \text{and } turn == 0);$

Lamport's bakery algorithm

בעמוד קודם רأינו אלגוריתם המונע מ-2 תהליכיים לגשת לקטע הקוד הקרייטי שלהם בו זמןית. אלגוריתם המאפייה של למפורט פותר את הבעיה גם עבור X תהליכיים. האלגוריתם עובד באותו אופן כמו "טור מספרים" בקובעת חולים: הלקוח הראשון שmagiu לוקח מספר ופונה לדילפק לקבל שירות, הלקוח השני שmagiu לוקח את המספר הבא וממתין לתורו. באותו אופן מגיעים לקווות נוספים. הם לוקחים מספרים לפי הסדר, ומקבלים שירות בהתאם למספר שלהם. בכל רגע נתון רק הלקוח אחד מקבל שירות. מה קורה אם 2 לקוחות נכנסים בו זמןית לחנות? במצבות לא תהיה בעיה, אחד מהלקוחות יקח מספר והשני ימתין ויקח אחריו, אך במחשב יתכן מצב של *Race Condition*, 2 תהליכיים "יגיעו" לעמדת המספרים ביחיד ויקחו את אותו מספר לפני שהוא הספיק להתעדכן. בשקופיות 9-8 בסיכון הרצאה 5 יש אינימציה שמסבירה את הבעיה. נראה פתרון בעמוד הבא.

איך נמנש את האלגוריתם?

ניסוי 1



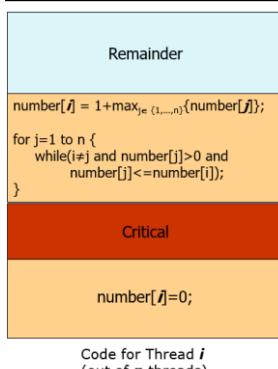
הגדנו מערך $number$ כך ש- $[i]$ יהיה המספר שהתחליכן ה- i מקבל: $1 + \text{מספר הגדל מבין כל התדרדים הקיימים}$.

(מקביל ללקוח שנכנס לקופת החולים ושאל "מי האחרון בתור?")

לאחר מכן יש לולהה שרצה כל עוד יש תחליכן עם מספר קטן מהמספר של התחליכן הנוכחי, לא כולל 0. אם יש צזה, נמתין לולהה. לאחרת לתחליכן הנוכחי יש את המספר הקטן ביותר ולכן תורו להיכנס לקטע קוד הקרייטי. בסוף קטע הקוד הקרייטי נעדכן את $0 = [i] = number$ כדי "לצאת מהטור".

בעיה: אם 2 תחליכונים קיבלו את אותו מספר (בגלל *Race Condition*), יתכן מצב בו שניהם יכנסו לקטע הקוד הקרייטי בו זמןית (שם דבר בלולאה לא מונע את המקרה הזה)

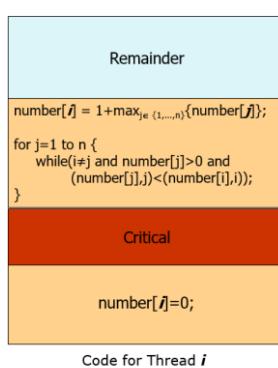
ניסוי 2



בניסוי זה הוספנו רק את הסימן $=$ לתנאי בלולאה, כדי לכלול את המקרה בו 2 תחליכונים קיבלו את אותו מספר. ככלומר נעצור בלולאה אם יש תחליכן עם אותו מספר כמו שלנו.

בעיה: 2 התחליכונים יתקעו בלולאה כי הם תמיד שווים. (*Deadlock*)
(גם כל שאר התחליכונים עם מספרים גדולים יותר יתקעו)

ניסוי 3

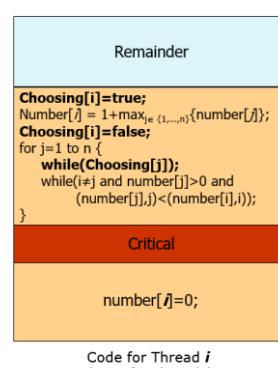


בניסוי זה אנחנו משווים בין תחליכונים בצורה ליקסוגרפית. קודם כל לפני המספר שלהם ואז לפि- tid שלהם. הבעיה הקודמת נפתרה.

בעיה: השתמשנו בפקודה max שהיא לא פיקודה אוטומטית. יתכן מצב בו

תחליכן עם $1 = tid$ קיבל את אותו מספר כמו תחליכן עם $2 = tid$, אך בזמן שתחליכון 2 משווה עצמו עם תחליכונים אחרים, תחליכון 1 עדין מחפש את המספר ולכן הוא עדין עם 0. במקרה זה תחליכון 2 נכנס לקטע הקוד הקרייטי ומיד לאחר שתחליכון 1 קיבל מספר הוא יכנס גם, מכיוון שאין תחליכון שקטן ממנו. בשקופית 13 במצגת של הרצאה 5 יש הדגמה מפורשת לבעיה, מומלץ לעבור עליה.

ניסוי 4 (פתרונות תקין שעונה על כל הדרישות !)



מוסיף מערך $Choosing$ כך ש- $[i]$ יהיה $Choosing$ true אם התחליכן ה- i

בוחרครיגע מספר. בaczורה זו תחליכון יכול להשוות מספרים רק אחד

בלולאה (j) $while(Choosing[j])$ שמנועת את הבעיה של הניסוי הקודם

(לא יקרה מצב בו תחליכון נכנס לקטע הקוד הקרייטי לפני שהוא השווה את עצמו עם כל תחליכון אחר וידע שהוא אכן הראשון להיכנס)

Read-Modify-Write Instructions

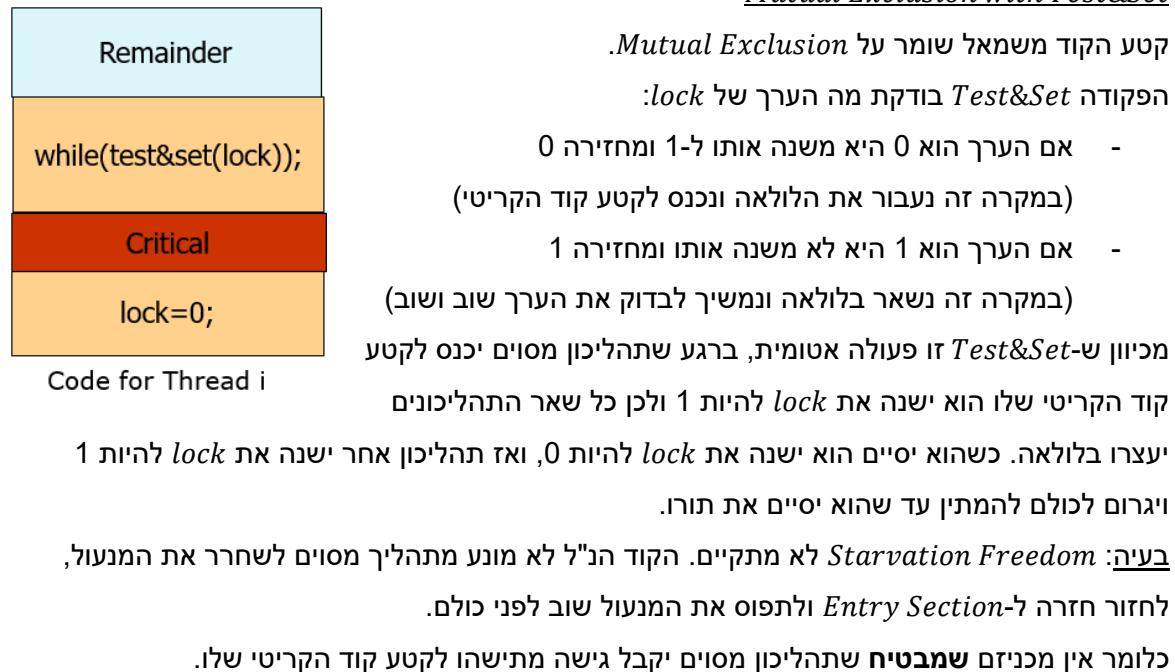
עד עכšíו הנחנו שפעולות אסםביי כמו קרייה וכתייה לזכרון הן פעולות אוטומיות, אך בין הפקודות יכול להיות *Context Switch*. במעבדים מודרניים יש מספר פעולות (Read – Modify – Write) (RMW) להיות מורכבות יותר הנתמכות ברמת חומרה כך שנitin להתייחס אליהן כפעולות אוטומיות:

Test&Set(&lock): $\{i = *lock; *lock = 1; return i;\}$

Fetch&Add(&p, inc): $\{val = *p; *p = val + inc; return val;\}$

Compare&Swap(&p, old, new): $\{if(*p \neq old) return false; *p = new; return true;\}$

Mutual Exclusion with Test&Set



Burn's Algorithm

האלגוריתם הבא פותר את הבעיה הנ"ל, אך נשים לב שהוא מסובך הרבה יותר (גם בכניסה וגם ביציאה) פקודות RMW אمنם עוזרות לסינכרוניzie אך הן יכולות גם לסייע.



עד עכשו ראיינו מספר פתרונות ל- *Mutual Exclusion* וכולן משתמשות בלולאות *Busy Wait* הגרומות לתקורה גבוהה. בנוסף, *Mutual Exclusion* זו בעית סינכרונייזציה אחת אך יש בעיות נוספות, כמו למשל *Contention* → *Coordination*, מצב בו יש תחרות בין מספר תהליכיונים על משאב משותף וצריך לסנכרן ביניהם. נראה בעיות נוספות בהמשך.

Synchronization Primitives

נדיר מבנה נתונים אבסטרקטי המומש על ידי מערכת הפעלה לביצוע סינכרונייזציה. לבניה הנתונים יהיה משך ברור ובו זמינות לא תינתן גישה למימוש הפנימי שלו.

Semaphore

סمفורי הוא מבנה נתונים פרימיטיבי המשמש לסנכרון. כפי שהגדכנו בתרגול 4, סمفורי מונע גישה למשאב משותף על ידי יותר מ-X תהליכיונים במקביל. יש לו שני>Status: ערך ורשימת תהליכיונים. בנוסף, ניתן לבצע 3 פעולות על הסمفורי:

- אתחול עם ערך רצוי, הקטנת ערך הסمفורי ב-1 (*(S)down*) והגדלת ערך הסمفורי ב-1 (*(S)up*) הפעולות הן פעולות אוטומיות ברמת תוכנה, כלומר זה מובטח על ידי מערכת הפעלה.
- כל עוד ערך הסمفורי גדול מ-0, כל תהליכון שմבקש להקטין את ערך הסمفורי מקבל גישה למשאב.
- אם הערך הוא 0 או פחות, הסمفורי מרדים את התהלייכון שפנה אליו ומקטין את הערך ב-1.
- ברגע שאחד התהליכיונים הרצים יותר על הגישה למשאב המשותף, הוא יגדיל את ערך הסمفורי ב-1, ובכך יתפנה מקום לתהליכון אחר להיכנס. הסمفורי פונה לרשיימת התהליכיונים הממתינים, יבחר אחד מהם ויריץ אותו. אופן פעולה זה מיותר את הצורך בלולאות *Wait – Busy Wait* מכיוון שהסמפור יdag להuir תהליכיונים שהגיעו זמן ו אין להם צורך לבדוק באופן אופני תדרם אם התפנה מקום.

נפתרו את בעית ה-*lock* בעזרת סمفורי:

Remainder	
down(lock)	lock.value= lock.value - 1 if lock.value < 0 then { add this thread to S.L; sleep();}
Critical	
up(lock)	lock.value= lock.value + 1 If lock.value≤0 then {remove a thread T from S.L; Wakeup(T);}
Code for Thread i	

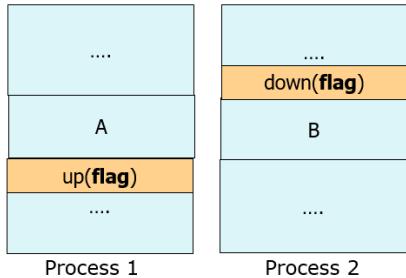
נדיר סمفורי משותף בשם *lock* המאותחל עם ערך 1.
אם $Value == 0$ סימן שיש תהליכון אחד בקטע קוד הクリיטי, ואם $X = Value$ סימן שיש X תהליכיונים הממתינים בתור. במקרה זה כל המאפיינים מתקיים אם רשיימת תהליכיונים ממומשת בעזרת תור *FIFO* (הראשון להיכנס הוא הראשון לצאת).

הערה: סمفורי עם ערך 1 נקרא לעיתים אובייקט *Mutex*.

Coordination

כפי שהזכרנו, בעית ה-*Mutual Exclusion* היא לא היחידה. נראה בעיה נוספת.

במידה ותהליך *A* כלשהו צריך לבצע את הקוד שלו לפני תהליך *B*, איך יוכל להבטיח סדר?



נגידיר סמפור בשם *flag* עם הערך 0. ל-*A* אין שום חסימה لكن הוא ירוץ כרגע. אם *B* ינסה להריץ את הקוד שלו לפני ש-*A* סיים לרוב, הסמפור ירדים אותו עד שהערך יגדל. בסוף הקוד של *A* יש קראיה ל-(*flag*) שיגדל את הערך ב-1, שכן הסמפור יעיר את *B* ויריץ את הקוד, אחרי ש-*A* סיים, כמובן.

סمفורי הוא לא פתרון קסם ! קל יותר להשתמש בסמפור אך עדין ניתן להגיע למצבים של

.Incorrectness-Deadlock, Starvation

תרגול 5

Concurrency pros and cons

הרעיון של מקביליות (*Concurrency*) הוא לחלק עבודה בין תהליכיונים שונים, כאשר כל תהליכון יעבוד על תט בעיה אחרת. נרצה שתהליכון אחד יריץ את הבעיה העיקרית ובמקביל לתהליכון אחר יעשה עבודות נדרשות ברקע. בצורה זו נוכל לשפר את ניצול ה-*CPU*. שימוש במקביליות יכול לייצר בעיות סינכרון כאשר תהליכיונים משתמשים במשאבים משותפים: *Deadlock* ומניעת הרעה.

בשיעור קודם הגדרנו מה הוא *Critical Section* – קטע קוד שבו יש גישה למשאבים משותפים, וכי למנוע שני תהליכיונים שונים להיכנס לקטע הקוד קרייטי שלהם במקביל נשתמש ב- *Mutual Exclusion Algorithms* - אלגוריתמים שנועדו למנוע בעיות סינכרון בין תהליכיונים שונים בזמן גישה למשאב משותף (כלומר בכניסה ל-*Critical Section*).

דוגמאות לבעיות סינכרון

דוגמה – int problem

שני תהליכיונים שמבצעים במקביל את הפקודה $A = X + 50$, כאשר X הוא משאב משותף.
את הפקודה נתרגם ל- 3 פקודות אסמלבי, בהנחה ש A, B רגיסטרים:

```
A = lw(&X) // get from memory to register  
B = add(A, 50) // add two registers  
sw(B, &X) // store the first register to the memory
```

במקרה זה, שלושת הפקודות הנ"ל יהיו קטעי הקוד הקרייטי

ובאיור הבא ניתן לראות מה יקרה אם יהיה *Context Switch* באמצע:

First thread	Second thread
<ul style="list-style-type: none">• $A = lw(\&X); //A=0;$• $B = add(A, 50); //B=50;$ • $sw(B, \&X); // X =50;$	<ul style="list-style-type: none">• $A = lw(\&X); //A=0;$• $B = add(A, 50); //B=50;$ • $sw(B, \&X); // X =50;$

At the end, $X=50$ (instead of $X=100$)

שני התהליכיונים אתחלו את A עם אפס. אם התהליכון הראשון היה מסיים לפני שההתהליכון השני התחיל, התהליכון השני היה מatkח את A עם 50, וההוספה הייתה מתבצעת כמו שצריך.

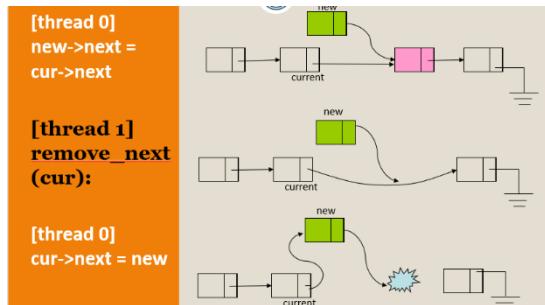
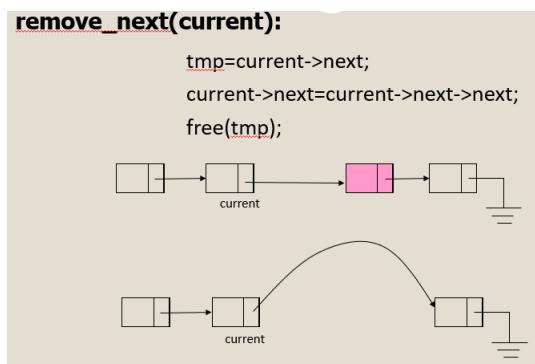
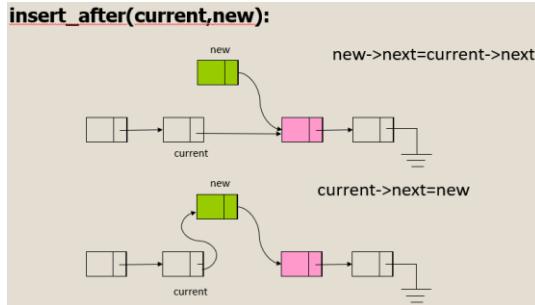
דוגמה נוספת – *Linked List Problem*

בහינתן הפעולות הבאות לניהול רשימה מקוורת:

- *insert_after(current,new)* – פעולה

שמתקבלת איבר ברשימה מקוורת *current*

ומוסיף אחריו איבר חדש *new*.



מתי יכולות להיות בעיות?

למשל אם מוסיפים איבר ובו זמניית מוחקים איבר

מאותו *Node current* (באיור משמאל). כפי שניתן

לראות, Thread 0 הספיק לבצע את הפקודה

הראשונה, וחיבור את *new* ל- *current* → *next*.

אבל לפני שהספיק לחבר את *new* ל- *current* → *next* ל-

, *current*, היה Thread 1. *Context Switch* בcut 1, נסה למחוק את האיבר שאחרי *current*,

ומכיוון שהפעולה הקודמת לא בוצעה במלואה, ימחק *Node* לא נכון במקום *new*.

בנוסך, כשנחזיר ל-0 Thread 0 וננסה לחבר את *current* → *next* ל-*new* לא נצליח

כי מחקנו את האיבר וلن גם את כל מה ברשימה אחריו.

The Critical Section Problem

בහינתן n תהליכיונים P_0, \dots, P_{n-1} , לא ידוע מה זמן הריצה של כל אחד מהם, ולא ידוע מה היא הפעולה שכל תהליך מבצע בקטע קוד הקרייטי שלו. נרצה בכל זאת למנוע את בעיית הסינכרון.

בהרצתה 4 (בעמוד 39) הראנו חמישה תנאים שצראים להתקיים בפתרון לבעית הסינכרון.

האלגוריתם של פיטרסון עבור 2 תהליכיונים

- קיימים שלושה משתנים משותפים: $turn$ ו- $want$.
- $[i]$ $want$ מצבן האם התהליכון ה- i רוצה להיכנס לקטע הקרייטי. (מאוחלים עם $false$)
- $turn$ מייצג את התהליכון שתורו להיכנס לקטע הקרייטי.
- כל תהליכון i משנה את $[i]$ ל- $true$ כדי לעדכן שהוא מעוניין להיכנס לקטע הקרייטי.
- הפקודה $i = turn = 1 - i$ מותרת על הזכות של תהליכון מסוים להיכנס לקטע הקרייטי לטובת התהליכון השני.
- התהליכון ה- i ימתין בלבד ריקה כל עוד $(i = 1 - turn == 1 - i) \&& turn == 1 - i$.
כלומר כל עוד התהליכון השני רוצה להיכנס גם הגיע תורו.
- כשהתהליכון השני יצא מהקטע הקרייטי, התנאי של הולאה נשבר והתהליכון ה- i יכול להיכנס לקטע הקרייטי.

האלגוריתם של פיטרסון מקיים את כל התנאים שהזכרנו מלבד הכלליות.

<pre>bool want[0] = false; bool want[1] = false; int turn;</pre>	
Thread 0 <pre>i = 0; want[i] = true; turn = 1-i; while (want[1-i] && turn == 1-i) { // busy wait } // critical section ... want[i] = false;</pre>	Thread 1 <pre>i = 1; want[i] = true; turn = 1-i; while (want[1-i] && turn == 1-i) { // busy wait } // critical section ... want[i] = false;</pre>

(ודוגמאות נוספות ל-*Test & Set* וэмפור שסוכמו כבר בהרצאה 5 בפיירוט)

בעיית סינכרון קלאסיות

בעיית הבאפר החסום - Bounded – Buffer Problem

בהינתן *Buffer* שיכול להכיל עד n רכיבים, ותהליכיונים מסווג *Producers* (שתיkeitם להוסיף רכיבים לבאפר) ומווסף *Consumers* (שתיkeitם להוציא רכיבים מהבאפר), נרצה לוודא ש-*Producer* לא יוסיף מידע לבאפר כשהבר אין מקום, ו-*Consumer* לא יוציא מידע מהבאפר אם הוא ריק. כדי לפתור את הבעיה נשתמש בסמפורים מהסוגים הבאים:

- ***Mutex*** – מאותחל ל-1, מאפשר גישה רק לתהליכון אחד (*Consumer* או *Producer*) בכל רגע נתון, מגן על הבאפר.
- ***FillCount*** – מאותחל ל-0, קובע כמה רכיבים יש בבאפר.
(רכיבים שה-*Consumer* יכול לקרוא מהם)
- ***EmptyCount*** – מאותחל ל- N , קובע כמה מקומות בבאפר פנויים לכתיבה.
(רכיבים שה-*Producer* יכול לכתוב אליהם)

ניתן לראות מימוש פשוט לתהליכיונים כנ"ל (*Consumer* ו-*Producer*) בקוד הבא:

Producer:	Consumer:
<pre>while (true) { produce an item down (emptyCount); down (mutex); add the item to the buffer up (mutex); up (fillCount); }</pre>	<pre>while (true) { down (fillCount); down (mutex); remove an item from buffer up (mutex); up (emptyCount); consume the item }</pre>

בעיית הפילוסופים הסודדים – Dining Philosophers Problem

הבעיה: חמישה פילוסופים יושבים בשולחן עגול שבמרכזו יש קערת של אורת. בין כל זוג פילוסופים יש צ'ופסטיק יחיד. הפילוסופים יודעים רק לאכול או לחוש, והם יכולים לאכול רק אם יש להם 2 צ'ופסטיקים. בנוסף, יש רק 5 צ'ופסטיקים لكن לא כולם יכולים לאכול במקביל, אבל בסוף הארוחה נרצה שכולם ישבעו. (כלומר נרצה להימנע מ-*Deadlock* והרעה) המשאב המשותף הוא קערת האורז (*Data Set*)
ומערך הסמפורים – הצל'ופסטיקים [5] המאותחלים ל-1. נרצה למצוא אלגוריתם לא סימטרי, רק ככה יוכל לפתור את הבעיה.



פתרון ראשון (ולא מוצלח)

```

While (true) {
    down ( chopstick[i] );
    down ( chopstick[ ( i + 1 ) % 5 ] );
    eat
    up ( chopstick[i] );
    up ( chopstick[ ( i + 1 ) % 5 ] );
    think
}

```

כל פילוסוף יבצע *down* לצ'ופסטייק במקומות ה-*i*, ולאחר מכן יבצע *down* לצ'ופסטייק במקומות ה- $5\% (i + 1)$, הוא יוכל לאכול באמצעות שני צ'ופסטייקים, וכשיסיים יעשה להם *up* על מנת שפילוסופים אחרים יכולים לאכול. הבעיה שאמם כל פילוסוף יבצע את הפוקודה הראשונה, ואז יהיה *Context Switch*, כל פילוסוף יקח רק צ'ופסטייק אחד ויחכה שימושו. ישחרר אחד נוספת זה לא יקרה ונגיע ל-*Deadlock*.

פתרון שני:

נוויף *Mutex* שתפקידו לתת רשות לפילוסוף ללקחת את הצ'ופסטייק, ואז או שהפילוסוף לא יוכל בכלל צ'ופסטייקים, או שיקבל את שניהם והוא יוכל לאכול.

פתרון שלישי:

פתרון אסימטרי – הפילוסופים במקומות הזוגיים והפילוסופים במקומות האי זוגיים יבצעו דברים שונים, ובכך הסימטריה תשבור.

אלגוריתם Lehmann Rabin:

```

repeat
    if coinflip() == 0 then           // randomly decide on a first chopstick
        first = left
    else
        first = right
    end if
    wait until chopstick[first] == false
    chopstick[first] = true          // wait until it is available
    if chopstick[~first] == false then // if second chopstick is available
        chopstick[~first] = true      // take it
    else
        chopstick[first] = false     // otherwise drop first chopstick
    end if
end repeat
eat
chopstick[left] = false
chopstick[right] = false

```

כל פילוסוף מטייל מטבע, אם קיבל 0 הוא ינסה ללקחת את הצ'ופסטייק שבצד השמאלי שלו, ואם קיבל 1 ינסה ללקחת את הצ'ופסטייק שבצד הימני שלו. נניח כי הפילוסוף קיבל 0 (בה"כ). הפילוסוף יחכה עד שהצ'ופסטייק משמאלו יהיה פנוי, וכשהוא יתרפה הוא יבדוק האם גם הצ'ופסטייק שמימינו פנוי, אם כן הוא יקח את שניהם ויתחיל לאכול, אחרת הוא יותר על שניהם ויתחיל את התהליך מחדש.

ב unifyת Readers – Writers

ב הינתן מבנה נתונים מסוותף בין תהליכיונים, קבוצה של תהליכיונים מסווג Readers (שיכולים לקרוא את הקובץ ולא יכולים לשנות אותו) וקבוצה של תהליכיונים מסווג Writers (שיכולים לקרוא או לכתוב לקובץ):

- קבוצת ה-*Readers* יכולה לקרוא במקביל מהקובץ
- רק *Writer* אחד יכול לקבל גישה לבני הנתונים בכל רגע נתון
- בזמן ש-*Writer* כותב צריך לחסום את ה-*Readers* לקרוא

פתרון ראשוני (ולא מוצלח)

נaddir שלושה סופוריים מסוותפים:

- *ReadCount* – מאותחל עם 0, מייצג את מספר הקוראים.
- .*readCount* – מאותחל עם 1, תפקידו להגן על ה-*ReadCount_mutex*
- מודוד שרק *writer* אחד יכול לקבל גישה בכל זמן נתון.

<pre style="font-family: monospace; margin: 0;"><i>Reader</i></pre> <pre style="font-family: monospace; margin: 0;"> while(true) { down(readCount_mutex); readCount++; if(readCount == 1) down(write); // lock from writers up(readCount_mutex) // reading is performed down (readCount_mutex); readCount--; if(readCount == 0) up(write); up(readCount_mutex); }</pre>	<pre style="font-family: monospace; margin: 0;"><i>Writer</i></pre> <pre style="font-family: monospace; margin: 0;"> while (true) { down (write); // writing is performed up (write); }</pre>
---	---

אך במקרה בו יש *Writer* שמעוניין לכתוב, יתכן מצב בו בכל רגע נתון יש *Readers* כלומר מניעת הרעה לא נשמרת.

פתרון שני:

נשתמש בסמפורים נוספים:

- סמפור **Read** – מאותחל עם 1, מאפשר בקשה לקרוא מהקובץ.
- סמפור **Integer WriteCount** – מאותחל עם 0, סופר את מספר הכותבים שמתנים לכתוב בקובץ.
- סמפור **.writeCount** – מאותחל עם 1, מגן על ה-**WriteCount_mutex**.
- סמפור **Queue** של סמפורים (שמאוחל עם 1), מבטיח שם **writer** אחד וכמה **readers** שמתחרים על הכניסה לקטע הקריאה, התחרות תהיה בין כותב אחד לקרוא אחד.

Writer

```
while(true) {
    down(writeCount_mutex)
    writeCount ++; //number of waiting writers
    if(writeCount == 1)
        down(read)
    up(writeCount_mutex)
    down(write);
    // writing is performed - one writer at a time
    up(write);
    down(writeCount_mutex)
    writeCount--;
    if(writeCount == 0)
        up(read)
    up(writeCount_mutex)
}
```

Reader

```
while(true) {
    down(queue)
    down(read)
    down(readCount_mutex);
    readCount++;
    if(readCount == 1)
        down(write);
    up(readCount_mutex)
    up(read)
    up(queue)
    // reading is performed
    down(readCount_mutex);
    readCount--;
    if(readCount == 0)
        up(write);
    up(readCount_mutex);
}
```

שאלות מבחנים שהופיעו בתרגול

3. כאשר מספר תהליכי חולקים גישה לבני נתונים משותפים, עלולות לגורן בעיות.

(a) להלן פתרון לביצוע הקטע הקריטי:

```
shared boolean flag[2] = {false};
shared int turn = 0;
do
{
    flag[i] = true;
    turn = i;
    while (flag[1-i] && turn==1-i);
    critical section
    flag[i] = false;
    remainder section
} while (true);
{}
```

מה דעך על השימוש הזה?

(ב) הוא לא מביא פתרון ביצוע המונעה ההודית במערכת עם שני תהליכי.

(2 נק) (2) אשר לפתרור את ביצוע הקטע הקריטי לשני תהליכי בהוספה וחווים בודדים (יש להוסיף אותם אם לדעתך אפשר).

(2 נק) (2) אפשר לפתרור את ביצוע הקיטוי לכל מספר של תהליכי בשני פקודה אחת בתכנית.

נשים לב ש- $i = turn$ ולא $1 - i$. וכך זה לא פיטרסון.

כאן יש שני *threads*, אף אחד לא ייכה בלהולאת ה-*while* – וישראל יכנסו

אם נסיף " $i - 1$ " כך שנקבל $i = turn = 1 - 1$, אז נקבל את פיטרסון ולכן תשובה ג' נכונה.

באופן כללי: תמיד להתחיל מלנסות את סדר הריצה הנאיivi

(הראשון מנסה וכשהוא בקטע הקריאה השני מנסה).

(ב) להלן פתרון לביעית קוראים כותבים. מה דעתך על המימוש הזה?

```
semaphore wrt_lock = 1;           READER:  
int rd_count = 0;  
if (rd_count==1)  
    down (wrt_lock);  
do_read();  
rd_count++;  
if (rd_count==0)  
    up (wrt_lock);  
  
WRITER:  
down(wrt_lock);  
do_write();  
up(wrt_lock);
```

- (נתק) א) הוא מבטיח את תכונת המוניה ההפוכה בין קבוצת קוראים לקבוצת כותבים.
(נתק) ב) הוא מבטיח את תכונת המוניה ההפוכה בין הכותבים.

למה סעיף א' לא נכון?

כי יכול להיות שקורא אחד יוסיף להגדיל את *rd_count* של *rd* מיד לאחר מכן יבצע *Context Switch* לקורא אחר שגם הוא יגדיל את *rd_count* של *rd*, כך שבשלב זה הערך של *rd* יהיה 2 וזה הערך שיראה כל אחד משני הקוראים, וכך אף אחד מהם לא ינסה למתפוס את *wrt_lock*. הם יעברו ל-*Critical Section* ועם זאת גם כותב שיופיע באותו זמן יוכל לנעול את המנוול ולהיכנס. גם אם לא היה *Context Switch* באותו מקום זה הסעיף לא היה נכון, מאחר שגם אם הקורא הראשון במאת מנסה למתפוס את *wrt_lock*, הקורא השני ימשיך לבצע הפעלה.

למה סעיף ב' לא נכון?

במצב הקודם, אם שני הקוראים יצאו לפני שהכותב יצא, הם ישחררו את המנוול (למרות שלא הם נעלו אותו – בסמפור זה אפשרי) ואז עוד כותב יוכל להיכנס. נניח שיש כותב שמבצע כתיבה (ולכן מחזיק את *wrt_lock*). לאחר מכן מגיע קורא, מגדיל את *rd_count* ל- 1 ומהכחה ש- *wrt_lock* ישחרר. לאחר מכן מגיעים שני קוראים שבגלל *context switch* בתוך *rd_count* מעדכנים את הערך של *rd_count* להיות 2 (ולא 3). הם שניהם נכנסים ל-*do_read()*. לאחר מכן אחד מהם מורד את *rd_count* ל- 1, הבא אחריו מורד ל- 0 ומשחרר את *wrt_lock*. כעת כותב חדש יכול לבצע *do_write()* (למרות שיש כבר כותב בפנים).

הסבר על Ex3 ניתן לראות במצגת של תרגול 5, שkopiot 41-48.

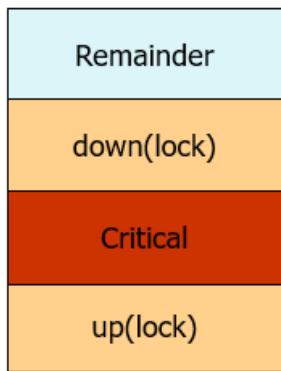
הרצאה 6

תזכורת: סמפור הוא מבנה נתונים עם 2 שדות - הערך שלו (מספר), ורשימה של תהליכיים או תהליכיונים שמחכים על הסטפורה. סמפור צריך להכיל 3 פונקציות - אתחול, הורדה, והעלאה, כאשר צריך למש את כל אחת מהfonקציות הנ"ל להיות פונקציה אוטומית. אם הערך של הסטפור הוא אפס או שלילי כל תהליך שנכנס להיכנס לסטפור יכנס לרשימת המתנה עד שתתהליך אחר יצא מהסטפור ויפנה למקום.

Down(S)	Up(S)	Init(S,v)
<pre>S.value= S.value - 1 if S.value < 0 then { add this thread to S.L; sleep();}</pre>	<pre>S.value= S.value + 1 if S.value≤0 then {remove a thread T from S.L; Wakeup(T);}</pre>	<pre>S.value= v</pre>

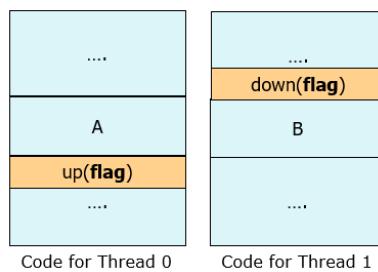
פתרון בעיות בעזרת סטפורים

בעיה ראשונה – Mutual Exclusion



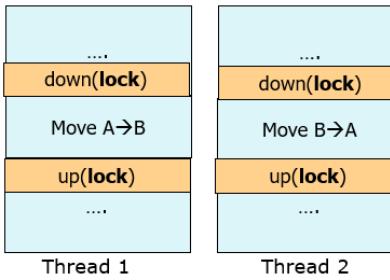
על מנת למנוע כניסה של מספר תהליכיונים לקטע קוד קרייטי, נגדיר סטפור משותף *lock* שיאותחל עם הערך 1. לפני הכניסה לקטע הקרייטי נעשה *down(lock)* – כדי להבטיח שתתהליכיונים נוספים יוכנסו להיכנס לקטע הקרייטי לא יכולו, וביציאה מהקטע הקרייטי נעשה *up(lock)* – כדי לאפשר לתהליכיון אחר להיכנס לקטע הקרייטי. נשים לב כי מניעת הרעבה מתקיימת רק אם הרשימה של התהליכיים *L* היא *FIFO* – *First In First Out*.

באוטו אוף, נוכל להגיד שקטע קוד מסוים יקרה לפני או אחרי קטע קוד אחר: נגדיר סטפור *flag* ונאותחל אותו עם 0. thread 1 לא יוכל להריץ את קטע קוד *B* (כי הוא יתקע ב-*(down(flag))*), עד ש-thread 0 יסיים לבצע את קטע קוד *A*, ועשה *(flag)up*. באוטו זה, קיבל כי קטע קוד *A* בהכרח יבוצע לפני *B*:

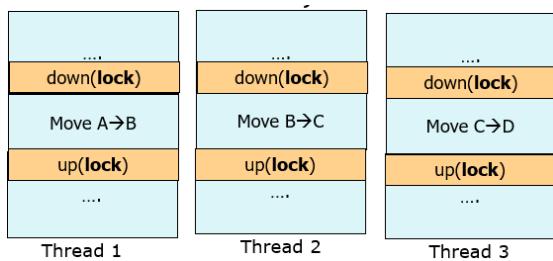


בעיה שנייה – העברת כספים בין 2 חשבונות:

בhininten שני תהליכיונים:

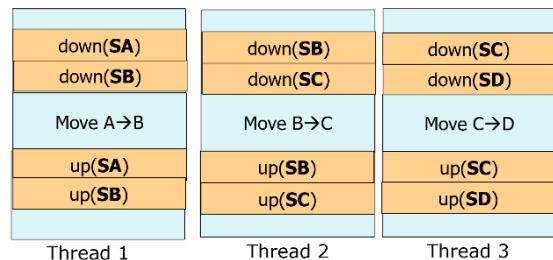


thread 1 המועוני להעביר כסף מחשבון A לחשבון B עם הפקודה – –*A*. thread 2 המועוני להעביר כסף מחשבון B לחשבון A עם הפקודה + +*A*. בשיעורים קודמים רأינו שפקודות אלה יכולות ליצור בעיות אם מבצעים אותן במקביל, ולכן נפתרו את הבעיה עם סמפור *lock* המאותחל ל-1, כפי שניתן למטה בAIR משמאל.



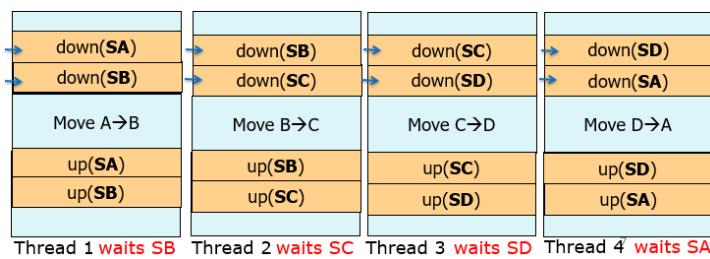
העברה כספים בין 3 חשבונות:

ננסה לפתור את הבעיה באותו אופן עבור 3 תהליכיונים. נגידר סמפור *lock* המאותחל ל-1 ובאותו אופן לפני כל פקודה העברה נעשה *down*, ואחריה *up*. פתרון זה אכן עובד, אבל הוא לא יעיל, הוא מונע מ-1 thread 3 ו-2 thread 1 לעבוד במקביל. (הם עובדים על חשבונות בנק שונים ולכן אין בעיה)

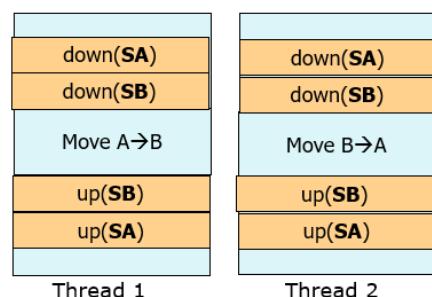


פתרון שני:

cut ננסה ליצור סמפור לכל חשבון בנק (*SA, SB, SC, SD*), NATCHL אותם עם 1 ובכל העברה נחסום אך ורק את החשבונות המשותפים להעברה כפי שניתן למטה בתמונה משמאל.



אמנם קל יותר לעבוד עם סמפורים, אבל סמפור הוא לא פתרון קסם. נסתכל על הדוגמה משמאל: אם כל אחד מהתהליכיונים יבצע רק את הפקודה הראשונה, ערכם של כל הסמפורים יהיה אפס, ויגיע ל-*deadlock*.

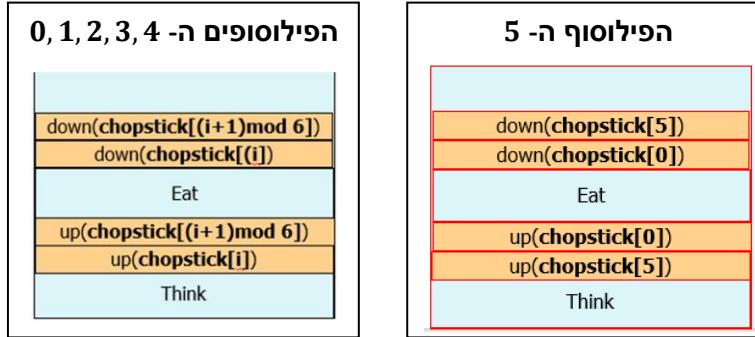


פתרון אפשרי לבעה כאשר יש רק שתי העברות (ולכן רק שני תהליכיונים), הוא סידור החסימות. אם תמיד נבצע קודם כל *down(SA)* ו ורק אז *down(SB)* לא נגיע למצב של *deadlock*.

בעיית הפילוסופים הסועדים – Dining Philosophers

את הבעה, ומספר פתרונות אפשריים שלה הזכרנו בתרגול 5 (עמודים 57-58 בסיכום) נשים לב כי בתרגול היו 5 פילוסופים וכךן יש 6 פילוסופים.

אחד הפתרונות היה לשבור את הסימטריה, למשל, פילוסופים 0,1,2,3,4 ירימו קודם את הצ'ופסטייק השמאלי, ורק אז את הימני, ורק פילוסוף 5 ירים קודם את הצ'ופסטייק הימני ואז את השמאלי. מהבינה אינטואיטיבית, שברנו את המתנה המוגלית וכך לא נקבל deadlock.



נכיה שיש progress באופן פורמלי, כלומר נוכיה שגם פילוסוף מנסה לקחת צ'ופסטייק, אך מתישחו, אוiziahu פילוסוף יצילח:

נניח שהפילוסוף ה- i מנסה להרים צ'ופסטייק אבל אף אחד מהפילוסופים לא מצליח לאכול (כלומר אף פילוסוף לא מצליח להרים את הצ'ופסטייק השני) ונפרק למקרים:
עבור $i = 0,1,2,3$:

אם הפילוסוף ה- i מנסה להרים את הצ'ופסטייק $1 + i$ ולא מצליח, בהכרח הפילוסוף $1 + i$ הצליח להרים אותו \rightarrow זה הצ'ופסטייק השני שהוא מרים \rightarrow סתירה.

אם הפילוסוף ה- i הצליח להרים את הצ'ופסטייק $1 + i$ אבל לא הצליח להרים את הצ'ופסטייק ה- i , אנחנו בהכרח ידעים כי הפילוסוף $1 - i$ לקח את הצ'ופסטייק ה- i ומחייב להרים את הצ'ופסטייק $1 - i$ וכן הלאה עד שנגיא לפילוסוף אפס. פילוסוף אפס לוקח את הצ'ופסטייק ה-1 ומחייב להרים את הצ'ופסטייק 0, אבל אם פילוסוף 5 לוקח את צ'ופסטייק 0, אז בהכרח פילוסוף 5 הצליח להרים את שני הצ'ופסטייקים \rightarrow סתירה.

עבור $i = 4$:

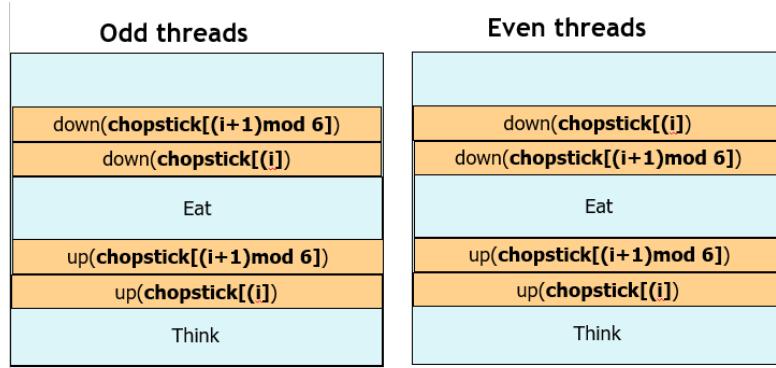
אם פילוסוף 4 ניסה לקחת את צ'ופסטייק 5 ולא הצליח, אז פילוסוף 5 לוקח אותו, וממתין לצ'ופסטייק 0, אבל אם פילוסוף 0 לוקח את צ'ופסטייק 0, אז הוא לוקח שני צ'ופסטייקים \rightarrow סתירה.

אם פילוסוף 4 ניסה לקחת את צ'ופסטייק 5 והצליח, וכעת הוא מחייב להרים את צ'ופסטייק 4, אך פילוסוף 3 לוקח את צ'ופסטייק 4 ומחייב להרים את צ'ופסטייק 3, וכן הלאה עד שנגיא לפילוסוף 0. פילוסוף 0 לוקח את צ'ופסטייק 1 ומחייב להרים את צ'ופסטייק 0, אם פילוסוף 5 לוקח את צ'ופסטייק 0, אז לפילוסוף 5 יש שני צ'ופסטייקים \rightarrow סתירה.

עבור $i = 5$:

אם פילוסוף 5 הצליח לקחת את צ'ופסטייק 5 אבל לא הצליח לקחת את צ'ופסטייק 0, אז פילוסוף 0 לוקח אותו, וזה הצ'ופסטייק השני של פילוסוף 0 \rightarrow סתירה. אם פילוסוף 5 לא הצליח לקחת את צ'ופסטייק 5, אז פילוסוף 4 לוקח אותו ומחייב להרים את צ'ופסטייק 4 ובאותו אופן כמו מקורה קודם \rightarrow סתירה.

הפתרון שהראנו אמן יבוד אבל הוא לא עיל, יתכן מצב בו רק פילוסוף אחד אוכל וכל השאר מוחכים. לכן, במקום לשבור סימטריה עם פילוסוף יחיד, נשבר סימטריה עם חצי מהפילוסופים, נחלק אותם לפי זוגיים ואי זוגיים, ונגדיר התנהגות באופן הבא:



מאפיינים של גלויים :deadlock

יתקיים אך ורק כאשר ארבעת התנאים הבאים קוראים:

1. **Mutual Exclusion** – יש משאב משותף / קטע קוד קרייטי.
2. **Hold and Wait** – תהליכיון מסוים נועל משאב אחד (למשל פילוסוף שנועל את הצלופסטיק הימני) ומחכה למשאב אחר (הצלופסטיק השמאלי) שתהליכיון אחר נעל.
3. **Non – Preemptive Allocation** – אם תהליכיון נועל משאב מסוים, רק הוא יכול לשחרר את אותו המשאב.
4. **Circular Wait** – כפי שראינו בדוגמה של הפילוסופים, הפילוסוף הראשון מחכה לשני, השני, השלישי לשישי וכן הלאה עד האחרון. אבל הפילוסוף האחרון מחכה לפילוסוף הראשון.

הערות:

- באופן כללי, מערכת הפעלה לא יודעת לזהות *deadlocks*.
- קורה כתלות ב-*context switch*, *deadlock* לא בהכרח נגלה זאת בכלל הריצה, מה שהופך את הדיבוג למאוד קשה.
- הדרך הנכונה לנעל משאים היא מספר שלהם לפי סדר מסוים, לנעל אותם מהגבוה לנמוך ולהחרר אותם מהנמוך לגבוה. בדרך זו נוכל למנוע *Circular Wait*.

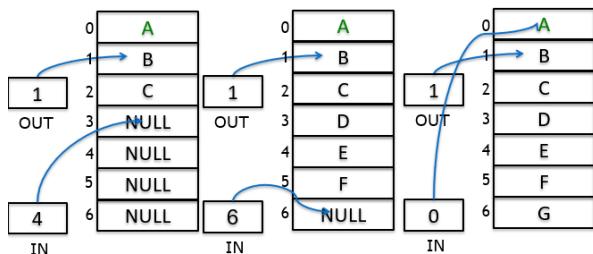
בעיה נוספת - Producer – Consumer

זכר ב-*spooler*: כאשר שני תהליכי רוצים להדפס משהו, הם כתובים *ל-spooler* (טור של הג'ובים שצורך להדפס), כאשר *Null* מייצג שאין ג'וב להדפס, *OUT* מייצג את הג'וב הבא שצורך להדפס, ו-*IN* מייצג את המקום הפנוי הבא בתור. כשנרצה להוציא ג'וב להדפסה נבצע את הקוד הבא:

```
Spooler[IN] = job  
IN ++
```

ראינו שהפעולה *+ IN* היא לא פעולה אוטומטית, ולכן יכולים להויזר בעיות אם מעודכנים את ה-*Spooler* על ידי שני תהליכי (*producers*) במקביל (למשל רוח בין העבודות, או דרישת עבודה). באותו אופן, יכולות להויזר בעיות אם יש שתי מדפסות (*consumers*) שמשתמשות ב-*OUT* במקביל. נשים לב כי הbeiter אליהם צוברים את הג'ובים הוא סופי, נתיחס אליו נאיל בAKER ציקלי, כלומר באפר שצמיגים לסופו, מתחילה למלא אותו שוב מההתחלתה.

דוגמה לבAKER ציקלי:

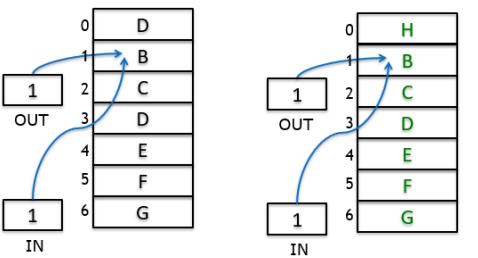


הסבר משמאל לימין: נכניס את הג'ובים אחד אחריו השני, נשים לב שהג'וב *A* במקומות ה- 0, כבר בוצע. לאחר מכן מילאנו את הAKER בג'ובים. כשהגענו לתא האחרון וראינו כי הוא מלא, חזרנו בחזרה להתחלה, לתא ה-0, שם נדרס את הג'וב שכבר הודפס.

איך נדע אם הAKER מלא או ריק?

נשימים לב בעיה הבאה, המצביעים *OUT*, *IN* שניהם מצביעים על תא מסוים 1, איך נוכל לבדוק האם הAKER מלא או ריק? כמו כן, איך נוכל לבדוק האם צריך לבצע את *B* או שכבר ביצעו אותו.

פתרונות ראשוני:



AKER is full

AKER is empty

מוסיף משתנה *counter* שסופר כמה ג'ובים נותר לבצע.

כשהAKER מלא ה-*counter* יהיה שווה ל-7, וכשהAKER ריק ייהו 0. כך נוכל לבדוק בין המצביעים. פתרון זה יעבד רק אם יש *producer* יחיד ו-*consumer* יחיד.

נראה שימוש אפשרי, נשים לב שפונקציות אלו חייבות להיות אוטומטיות, או מוגדרות כקטע קרייטי!!

Producer:

```
while(counter == OUT); \\\buffer is full, wait until a job is consumed
buffer[IN] = job;
IN = IN + 1 mod n;
counter ++;
```

Consumer:

```
while(counter == 0); \\\buffer is empty, wait until a job is produced
job = buffer [OUT];
OUT = OUT + 1 mod n;
counter --;
```

פתרון שני:

נוטר על התא האחרון בבאפר ואם המצביע IN נמצא תא אחד לפני המצביע OUT , נדע שהבאפר מלא.
גם כאן פתרון זה יעבד רק אם יש *producer* יחיד ו-*consumer* יחיד. נראה מימוש אפשרי:

Producer:

```
while((IN + 1 mod n) == OUT); \\buffer is full, wait until a job is consumed  
buffer[IN] = job;  
IN = IN + 1 mod n;
```

Consumer:

```
while(IN == OUT); \\buffer is empty, wait until a job is produced  
job = buffer [OUT];  
OUT = OUT + 1 mod n;
```

פתרון שלישי:

כעת נראה פתרון שייעמוד עבור יותר יותר מ-*producer*-ו*consumer* יחידים.
נשתמש בשלושה סטטוסים באופן הבא:

- **mutex** – מאוחחל ל-1, תפקידו להגן על קטע הקוד הקרייטי, כולם לדאוג ש-*producer* או *consumer* לא יצילחו לגשת לבאפר אם יש שם מישחו אחר שמבצע פעולה.
- **empty** – מאוחחל ל- n , סופר את מספר התאים הריקים.
- **full** – מאוחחל ל-0, סופר את מספר התאים המלאים.

נראה מימוש אפשרי:

Producer:

```
down(empty)  
down(mutex)
```

```
buffer[IN] = job;  
IN = IN + 1 mod n;  
counter ++;
```

```
up(mutex)  
up(full)
```

Consumer:

```
down(full)  
down(mutex)
```

```
job = buffer [OUT];  
OUT = OUT + 1 mod n;  
counter --;
```

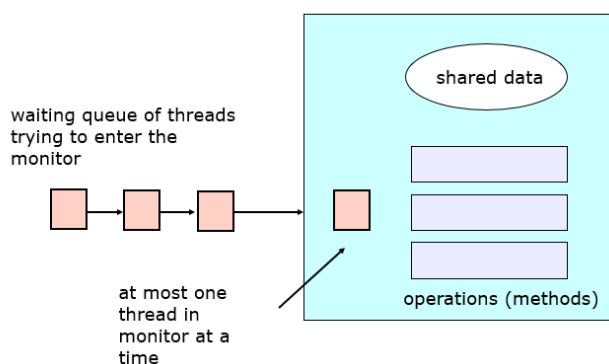
```
up(mutex)  
up(empty)
```

מונייטור

מבנה נתונים מובנה, המספק גישה מבוקרת למשאים משותפים כך שבעל פה רק תהליך/תהליכון אחד מקבל גישה למשאב. למשל, מוניטור מספק סינכרונייזציה בטוחה ומאפשר:

- מבני נתונים משותפים.
- מתודות (*procedures*) הפעולות על המשאים המשותפים.
- סינכרונייזציה בין תהליכיון המשמשים במתודות הנ"ל.

ניתן לגשת למידע המשותף רק מתוך המוניטור באמצעות שימוש במתודות המסופקות.



דוגמה:

בתמונה משמאל יש דוגמה למוניטור (בכחול) וטור של תהליכיון. יש מידע משותף ומוגדרות. בכל רגע נתון תהליכון אחד בלבד נמצא במוניטור, עושה מה שהוא צריך, וכשהוא מסיים המוניטור מכניס תהליכון אחר.

Mutual Exclusion

בכל רגע נתון רק תהליכון אחד מקבל גישה למשאב המשותף. אם תהליכון נוסף ננסה לקבל גישה הוא נחסם עד שהתהליכון שנמצא במוניטור יסיום, אך לא יתכו בעיות סינכרונייזציה.

Monitor

בהרצאה קודמת התחלנו לדבר על מוניטור. למוניטור יש 2 פונקציות:
 - `CDI להוסיף עבודה ל-Buffer Produce()`
 - `לקראם עבודה מה-Buffer ולחזור אותה Consum()`
הערה: ב-*Java* צריך לסמן את הפונקציות האלה כ-*synchronized* (מילה שמורה).

הזכירנו בהרצאה קודמת ש-*Mutual Exclusion* ניתן "בחןם" אך יש בעיה:
 כאשר *Consumer* קורא לפונקציה *Consume*, ניתנת לו עבודה מה-*Buffer*.
 מה קורה אם אין עבודה ב-*Buffer*? *Consumer* ימתין עד שתהיה עבודה, אך אף תהליך אחר לא יכול להיכנס למוניטור, גם לא *Producer* שיוסיף עבודה חדשה. במקרה זה לא תהיה התקדמות.

Condition Variable

כדי לפתור את הבעיה הנ"ל נשתמש ב-*Condition Variables*. במקום שהתהליכים ימתין בתוקן המוניטור, הוא ימתין ב"נקודות מפגש" עם תהליכי אחרים ויפנה את המוניטור עד שיגיע תורו.
 תהליכיים שונים שמתכוונים שם מזומנים על המוניטור עד שהמוניטור מסמן להם לחזור ולתפואו אותו.
 כדי ליצור מוניטור, חיבבים לשימוש ב-*Condition Variable* המאפשר 3 פעולה:

- `(Java c.wait() או () ב-c.wait(c))` – משחרר את המנעול של המוניטור
- `(Java c.notify() או () ב-c.notify(c))` – מעביר את התהליךון ל-*Condition Variable* עד שהוא יתירע לו
- `(Java c.NotifyAll() או () ב-c.broadcast(c))` – מעיר לכל היוטר תהליכי 1 שמתכוונים. אם אין תהליכיים ממתכוונים, הסיגנל יעלם.

פואודו-קוד עם *Condition Variable* אחד:

```
public synchronized void produce(char ch)
{
    while (COUNT == bufferSize)
        wait();
    store[IN] = ch;
    IN = (IN + 1) % bufferSize;
    COUNT++;
    notifyAll();
}

public synchronized char consume()
{
    while (COUNT == 0)
        wait();
    ch = store[OUT];
    OUT = (OUT + 1) % bufferSize;
    COUNT--;
    notifyAll();
    return ch;
}
```

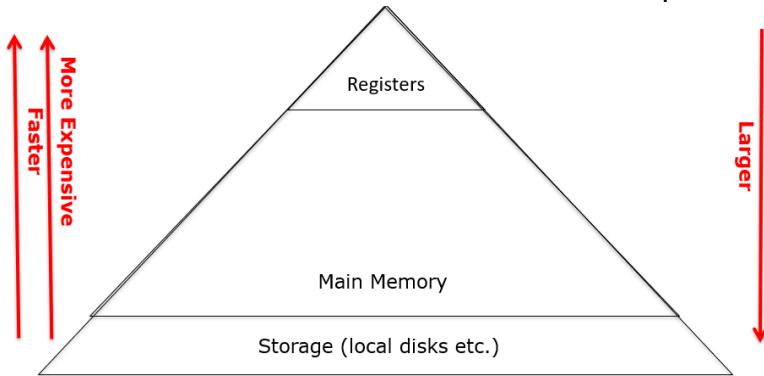
38

Shared Memory vc Message Passing Models

לאורך הקורס אנחנו מניחים ש-2 תהליכי / תהליכיים מתקשרים אחד עם השני דרך משבבים משותפים, לרוב זיכרנו משותף. דרך נוספת לתקשר היא ע"י שליחה וקבלת הודעות (*Messages*) גם בדרך זו ניתן בנסיבות דומות, אך הפתרונות שלהם מעט שונים. (לא נהיב על כך בקורס)

Memory Hierarchy

נסתכל על היררכיית הזיכרון במחשב:



כל שעולים בפירמידה, סוג הזיכרון יותר מהיר אך גם יותר יקר.
כל שירדים בפירמידה, יש יותר זיכרון מאשרו סוג.

RAM (Random Access Memory)

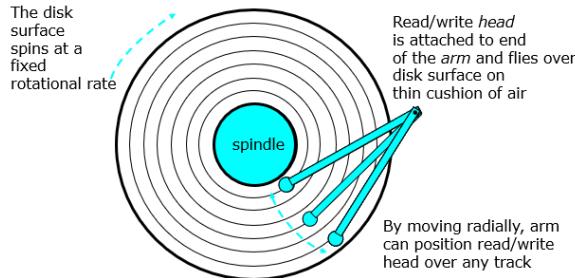
טכנייקת גישה ישירה לזיכרון שנמצאת בשימוש גם ברגיסטרים וגם בזכרון הראשי.

- **(Static RAM (SRAM))**
 - שומר על הערך בזכרון כל עוד יש זרימת חשמל
 - רגישות יחסית להפרעות כגון רעש חשמלי
 - מהיר יותר ויקר יותר מאשר DRAM
- **Dynamic RAM (DRAM)**
 - פעם ב-ms 100 – 10 צריך לעדכן את הערך בזכרון מחדש כדי שלא יעלם
 - רגישות להפרעות
 - איטי יותר וזול יותר מאשר SRAM
- טכנולוגיות נוספות שלא נרחיב עליהם בקורס:
 - SDRAM, SDR – SDRAM, DDR – SDRAM, NVRAM ועוד..

Memory	Single-chip Capacity	\$/chip	\$/MByte	Access speed (ns)	Watts/chip	Watts/MByte
DRAM	128MB	\$10-\$20	\$0.08-\$0.16	40-80	1-2	0.008-0.016
SRAM	9MB	\$50-\$70	\$5.5-57.8	3-5	1.5-3	0.17-0.33

התמונה לעיל מראה את ההבדלים בין DRAM ל-SRAM (עם נתונים מעודכנים ל-2010). ניתן לראות ש-SRAM יקר יותר אבל מהיר יותר. לרוב הרגיסטרים של המעבד יהיו מסווג SRAM, שאר זיכרון ה-RAM יהיה מסווג DRAM כי אנחנו צריכים הרבה ממנו.

Disk Operation



סוג זיכרון נוסף הוא הדיסק. (למשל דיסק קשיח)

הדיםק מסתובב ונitinן לקרוא ממנו מייד עזרת סיכה שיכולה לזרז רק כלפי פנים או כלפי חוץ, אך לא לאורר הדיסק. כדי לקרוא מידע שלא נמצא מתחתיה היא צריכה לחכות לשיבוב של הדיסק. מהירות הזיכרון מוגדרת לפי מספר הסיבובים שלו בדקה (RPM – Revolutions Per Minute)

כיום הארד-דיסקים מסתובבים בערך ב-*RPM* 3000 וזמן גישה להארד-דיסק שנחשב מהיר הינו *ms* 15 – 9. (איטי מאד יחסית ל-*DRAM*)

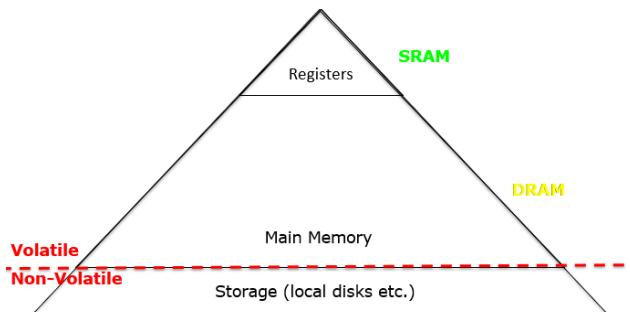
סוגי זיכרון נוספים הם למשל זיכרות *flash* או *SSD* שלרוב מבוססי *DRAM*, אך עדין הם הרבה יותר איטיים מה-*DRAM* של הזיכרון הראשי. בנוסף הם יקרים יותר מהארד-דיסקים רגילים אך דורשים פחות חשמל.

זיכרון נדייף (Non-Volatile) לעומת זיכרון לא נדייף (Volatile)

מה קורה כשנכחוה והחמל?

אם הזיכרון הוא נדייף, המידע נעלם (למשל *RAM*, *DRAM*, *SRAM* וכו')

אם הזיכרון לא נדייף, המידע נשמר (הארד-דיסקים, זיכרות *flash*, *ROM* וכו')



הערה: ב-*Storage* יש עוד היררכיות זיכרון, אבל בקורס שלנו נתיחס לכלן כרימה אחת משותפת.

ROM (Read – Only Memory)

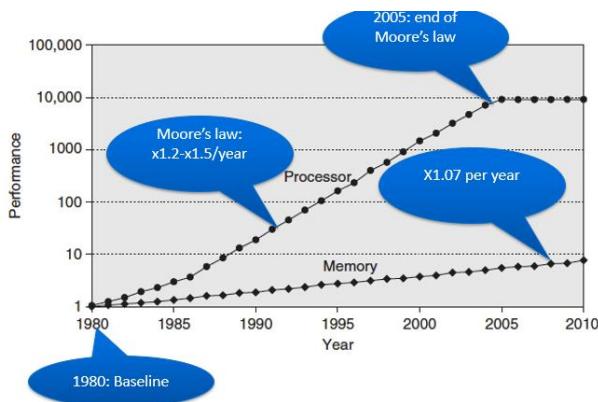
זיכרון לא נדייף, שם כלל למשפחה של זיכרות. למראות השם שלו, הוא כן ניתן לשינוי אך באופן איטי ובאופן יותר מסובך. ה-*ROM* יachsen מידע שלא מתעדכן באופן תדיר כמו בשאר הזיכרות, למשל:

Boot time code, BIOS (basic input/output system) -

graphics cards drivers, disk controllers bootstrap, firmware -

לא חלק מהיררכיית הזיכרון.

CPU vs DRAM



הגרף מראה את ההבדל בין שיפור מהירות ה-CPU לעומת זיכרון DRAM. מ-1980 ועד 2010 השיפור ב-CPU היה פי 10,000 לעומת פי 10 בזיכרון DRAM. אז איך ה-CPU מסתדר עם הבדלי המהירות? הוא לא צריך "להמתין" לזכרון כל פעם? בשילוב להגבר על הפעור שנוצר בשימוש בזיכרון!

Cache

2 עקרונות של זיכרון ה-Cache מנצל:

- Temporal locality

אם ניגשנו לכתובת מסוימת בזיכרון, סבירות גבוהה שנציגו לבקשת אליה שוב בעתיד הקרוב. (למשל בלולאות או בעדכון מידע בזיכרון)

- Spatial locality

אם ניגשנו לכתובת מסוימת בזיכרון, סבירות גבוהה שנציגו לבקשת גמ לשכנים שלה. (למשל מערכים או *Instructions*)

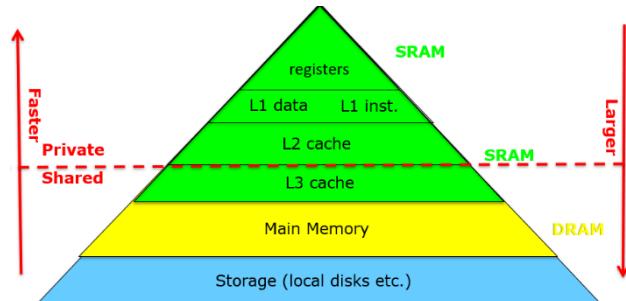
זיכרון ה-Cache מהיר יותר (SRAM) ונמצא פיזית קרוב יותר ל-CPU, אבל:

- הוא לא צריך להיות מהיר כמו הרגיסטרים ב-CPU

- (יש זיכרונות SRAM ב מהירותים שונות, זיכרון קטן יותר הוא מהיר יותר וקר יותר)

- לא טיפנו במקרים בהם יש יותר מעבד אחד (האם הזיכרון יהיה משותף? או חלק ממנו?)

לכן נוכל לחלק את ה-Cache לרמות נוספות בהיררכיה ולהשתמש בסוגי זיכרונות שונים לפי הצורך:



כל רמה בהיררכיה משתמשת ב-SRAM ב מהירות שונה. (הכי מהיר למעלה היכי "איטי" למטה). L1 ו-L2 הם *private*, כלומר רק למעבד הנוכחי יש גישה אליהם, לעומת L3 שמשותף בין כל המעבדים. גם את L1 בדרך כלל מפצלים ל-2 סוגים: אחד מנהל את *Data* והשני מנהל את *Instructions*.
כשהתבצע גישה לזיכרון (Main Memory), מקטע זיכרון שמכיל את התא הרצוי מועתק ל-L3, אם אנחנו ניגשים למידע זהה המן הוא עבר ל-L2, ובאותו אופן ל-L1 אם הוא ממש חשוב.
הערה: באותו אופן, ה-Cache מהווה Main Memory ל-Storage Cache וכן להלאה. כל קומה בהיררכיה מהווה Cache לרמה שמתחילה.

להלן טבלה שמסכמת את זמני הגישה לשוגי הזיכרון השונים בהיררכיה:

	Server	Mobile device
Registers	0.3 ns / 1 KB	0.5 ns / 0.5 KB
L1	1 ns / 64 KB	2 ns / 64 KB
L2	3-10 ns / 256 KB	10-20 ns / 256 KB
L3	10-20 ns / 2-4 MB	None
Main Memory	50-100 ns / 4-16 GB	50-100 ns / 256-512 MB
Storage	Disk: 5-10 ms / 4-16 TB	Flash: 25-50 ms / 4-8 GB

נדיר מספר מושגים:

- Cache hit – כשתוכנית צריכה לגשת לתא בזכרון, אם התא בזכרון נמצא ב-*Cache* – "הצלחנו", וכל הצלחה זאת נקראת *Cache hit*. נגדיר את *Hit rate* להיות $\frac{\text{מספר ההצלחות}}{\text{מספר הניסיונות}}$
- Cache miss – אם התא בזכרון לא נמצא ב-*Cache* – "פספסנו". נגדיר את *Miss rate* להיות $\frac{\text{מספר הפספסים}}{\text{מספר הניסיונות}}$
- Miss penalty – הזמן שלוקח להגיע למידע בהיררכית הזיכרון עד ל-*CPU*.

דוגמאות:

- נניח ש- $100\% Hit rate$, עברו 100 גישות לזכרון לקח לנו $1ns$ (גישה ל-*Cache* היא $1ns$)
 - עברו $100 Hit rate$, הינו 100 גישות ל-*Cache* בעלות של ns 100 ומתקומם 10 גישות דרשו מעבר ל-*L2* במהירות ns 10 לכל אחד. סה"כ $200 ns$.
- כלומר מהירות הזיכרון ירדה ב- 50% , זהה בהנחה שה-10 הנוספים נמצאים ב-*L2*!
במציאות, *L1* יש *Hit rate* של $97\% - 95\%$.

אפקט המימוש:

נתמך בשאלות הבאות: איך מומשת הגישה מהירה? מה אפשר לאחסן ב-*Cache*? *Cache*-*Cache*? מה מלא, מה נעדיף לדחוס? אם אנחנו משנים ערך ב-*Cache*, איך השינוי הזה מבוצע גם בשאר הזיכרון?

איך מומשת הגישה מהירה?

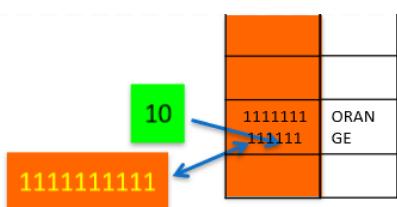
ה-*CPU* יודע לעבד רק עם כתובות של הזיכרון הראשי (כתובות באורך 32/64 ביט) מצד שני, ה-*Cache*-*Cache* מכיל פוחות זיכרון מאשר ה-*Main Memory* , אז איך עובד המיפוי ביניהם?
אם ה-*CPU* מבקש את הערך שנמצא בכתובת x באורך 32 ביט, لأن ה-*Cache* ניגש?
איפה הערך שנמצא בכתובת x מאחסן בתוך ה-*Cache* אם אורך הכתובות ב-*Cache* קטן יותר?
(לצורך פשטות ההסביר נניח שיש רק רמה אחת של *Cache*)

דוגמה:

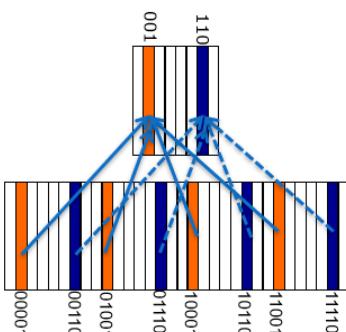


2 ביטים כי ה-*Cache* באורך 2 ביטים לאחר מכן נשמר את הערך הרצוי בכתובת 10 בזיכרון ה-*Cache*.

הבעיה? למשל המילה 1000000000000 תמורה לאותו מקום כי גם היא מתחילה ב-10. נזכיר שאחד העקרונות של זיכרון ה-*Cache* הוא *Spatial locality*, שבו אנחנו מעתיקים גם את השכנים של התא הרצוי וכן לרוב געתיק מספר כתובות עם אותה תחילה. איך נתמודד עם התנוגשיות?



נבחר לשומר רק אחד מהם, וכך להבחן מי בחורנו לשומר נctrue לשומר מידע נוסף – שאר הביטים ששמננו בצד (ה-*tag*). אם ה-*CPU* מעוניין בכתובת 101111111111, ניתן לכתובת 10 ואם ה-*tag* תואם לשאר הביטים, סימן שהגענו לערך הרצוי.



הבעיה? נctrue לשומר המון מידע "מיותר" – *Overhead* גבוהה. אם נפצל את המידע לפי 2 הביטים הימניים, נוכל למנוע הרבה התנוגשיות, כי במקרה רציף בזכרון הביטים הימניים משתנים בכל תא לעומת הביטים השמאליים, וכך גודיל את ה-*Cache Hit* שלנו. כדי לטפל בבעיית ה-*Overhead*, במקומ להעתיק תא זיכרון בודדים לא-*Cache*, געתיק מקטני זיכרון שלמים, ונשמר רק *tag* אחד שיזהה אותם.

כשנרצה לגשת לאחד מהתאים בבלוק שהעתיקנו, איך נדע לאיזה מהם לגשת? ננסה את הדרך שבה אנחנו מפצלים את הכתובת ובמקום לחלק ל-2 חלקים נחלק ל-3 חלקים:

10111111 11 111
tag index offset

בדוגמה שלנו, אם כל בלוק הוא בגודל 8, משתמש ב-3 הביטים הימניים (*offset*) כדי לסמן איזה תא מבין ה-8 אנחנו רוצים (למשל כבזה 111 מדובר בתא האחרון). 2 הביטים האחרים (*index*) מסמנים את התא ב-*Cache* שבו נמצא הבלוק, ושאר הביטים (*tag*) מסמנים אם זה הערך הנכון או לא.

הערה: לאורך ההסבר, כפי שאמרנו, השתמשנו בגודלים שרלוונטיים ל-*Main Memory* באורך 13 ביט ו-*Cache* באורך 2 ביט אך כמובן שהמספרים משתנים בהתאם לגודל הזיכרון וגודל ה-*Cache* בפועל. מעיטה והלאה לא يعنيו גודל הבלוקים אלא רק הרעיון הכללי שלהם.

תרגול 6

בעמוד 67 בסיכון ראיינו 4 מאפיינים הכרחיים לקיומם קיפאון (*Deadlock*), אך מה עושים במצב זהה?

אסטרטגיית ניהול *Deadlock*

- התעלמות – לא נטפל בבעיה כלל, גישה זו נקראת גם "גישת *Ostrich*".

גישה סבירה כאשר מצב של קיפאון הוא נדר ולבסוף עלות מניעת הבעיה היא יקרה.

טרייד-אוף בין נוחות לנכונות. *UNIX* ו-*Windows* פועלם לפי גישה זו.

- זיהוי והתחששות – נאפשר למערכת להיכנס במצב קיפאון אך נדע להתואושש ממנו במקרה צזה.

אסטרטגייה זו מוסיפה בזמן הריצה כדי לתחזק את המידע הדרוש להרצאת אלגוריתם לזיהוי קיפאון.

אלגוריתם הזיהוי מורכב מ-2 חלקים:

○ יצירת גרף "for" – *wait* : הגרף השמאלי

בתמונה הוא גרף של כל התהליכים והמשאים

המשותפים בתוכנית. כל עיגול בgraf הוא תחלה,

וכל ריבוע הוא משאב משותף. הגרף הימני הוא גרף

"*wait* – *for*" שנוצר על ידי מחיקת כל המשאים

המשותפים, והוא מציג את הקשרים בין התהליכים:

אם $P_i \rightarrow P_j$ אז P_i מחייב P_j . מספר עובדות בסיסיות:

▪ אם הגרף לא מכיל מעגלים, אין קיפאון.

▪ אם הגרף מכיל מעגלים ויש רק מופע אחד מכל משאב, יש קיפאון.

▪ אם הגרף מכיל מעגלים ויש יותר מmorphע אחד מכל משאב, יתכן מצב של קיפאון.

○ החלק השני הוא הפעלת אלגוריתם לזיהוי מעגלים בgraf מעט לעת:

זיהוי מעגל בgraf עם n קודקודים דרוש (n^2) פעולות.

כדי לצאת במצב של קיפאון נוכל לפעול לפי 2 גישות שונות:

נמחק את אחד התהליכים בمعالג. במקרה זה נדרש להריץ שוב את האלגוריתם לזיהוי

מעגלים כדי לוודא שakan פתרנו את הבעיה. (ואם לא, נמחק תחלה נוספת וכן הלאה).

גישה שנייה היא למחוק מראש את כל התהליכים בمعالג, אך זו פעולה יקרה בהשוואה

למקרים בהם היה מספיק למחוק תחלה בודד.

- הימנעות – הקצאת נתונים זהירה כך שהמערכת לא תגיע למצב של קיפאון. אסטרטגייה זו דורשת

של תחלה יזהיר מה המספר המקורי של מופעים מכל משאב מסו��ה הוא מתכוון לדריש לארוך

הריצה. ניתן למש אסטרטגייה זו באמצעות אלגוריתם הבנקאי (*Banker's algorithm*):

האלגוריתם מופעל על ידי מערכת הפעלה בזמן שתחלה מבקשת הקצאה. האלגוריתם מניע מקייפאון

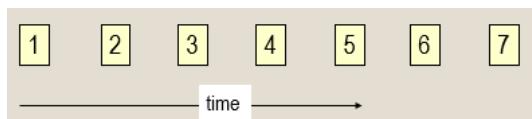
על ידי שלילה או דחיה של הבקשה אם הוא מזיהה שבקשת הבקשה יכולה לגרום לקיפאון. כאשר

תחלה חדש נכנס למערכת, הוא חייב להצהיר מראש לאיזה משאים הוא מעוניין לגשת בזמן הריצה.

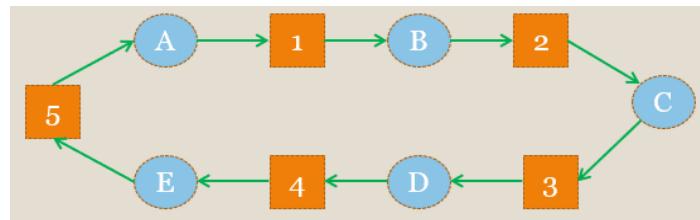
אם הרצת התחלה יכולה לגרום למצב של קיפאון, נדחה אותה. אחרת ניתן לגשת לו לרוץ.

ברגע שתחלה מקבלת את כל המשאים שהיא בקשת, הוא מחויב להחזיר אותם תוך זמן מוגבל.

- **מניעה** – שלילת אחד מארבעת התנאים ההכרחיים לקיום קייפאון.
- **שלילת Mutual Exclusion**: במקרים בהם נוכל להימנע משיתוף משאבים, נעשה זאת. כמובן שלא תמיד אפשר וכך אנחנו מוחפשים פתרונות לבעה...
- **שלילת Preemption No**: אם תהליך שמחזיק משאבים מסוימים מבקש משאב אחר שלא ניתן להקנות לו באופן מיידי, נרדם אותו עד שהמשאב יתפנה, ובו זמניית נחרר את כל המשאבים שהוא מחזיק. המשאבים יתווסף לרשותו של המשאב הנוכחי מעתה עבורם, והוא יופעל מחדש רק אם כל המשאבים שהוא צריך פנויים. (הישנים והחדשים) פתרון זה בעיתי במקרים בהם שחרור המשאבים יכול לגרום לבעות אחרות כמו למשל שימוש במדפסת. אם נעלית המדפסת תפסיק באופן פתאומי באמצעות הדפסת דף, תהליך אחר ימשיך להדפיס באמצעות הדף.
- **שלילת Hold and Wait**: נרצה להגיע למצב שבו בכל פעם שתהליך מבקש משאב מסווג, הוא לא מחזיק באותו רגע במסאב מסווג אחר. נעשה זאת בכך שנדרש מכל תהליך להקנות מראש את כל המשאבים שלו. במקרים בהם תהליכיים לא יודעים מראש איזה משאבים הם צריכים לא נוכל לפעול לפי גישה זו. בנוסף, ניצולת המשאבים פוחתת וייתכן מצב של הרעבה (*Starvation*). נ�性 זאת ב-2 שלבים: בשלב הראשון התהליך ינסה לנעל את כל המשאבים שהוא צריך, אחד אחרי השני. אם משאב מסוים לא זמין, הוא ישחרר את כל השאר וינסה שוב. בשלב השני, אחרי שהשלב הראשון הצלח, הוא יבצע את הפעולות הרצויות וישחרר את המנועלים. נוכל לנوع מצב של הרעבה בכך שנוסיף גם חותמת זמן (*Time stamps*): לפני שתהליך מתחיל לנעל משאבים, חותמת זמן ייחודית תקשורת אליו. אם הוקצתה חותמת זמן T_i לתהליך מסויים ולאחר מכן חותמת זמן T_j לתהליך אחר, אז $T_j < T_i$. כל משאב מקבל גם הוא חותמת זמן שמסמלת איזה תהליך מחזיק בו ברגע נתון, ובכך אנחנו נתונים עדיפות לתהליכיים "מבוגרים" יותר ומונעים הרעבה.
- **שלילת Circular Wait**: במספר את המשאבים ונדרש שככל תהליך יNeal אותם לפי הסדר:



אם נסתכל על דוגמת הצ'ופסטיקים משיעורים קודמים, אם הינו בוחרים את הצ'ופסטיקים לפי גישה זו, לא היה מצב של קייפאון. למה? נניח בשלילה שאכן היה מצב של קייפאון, אז גראף הקצאת המשאבים נראה כך:



(פילוסוף B מחזיק במשאב מס' 1 בזמן שפילוסוף A מעוניין במשאב מס' 1 וכן הלאה) נשים לב כי תהליך A מעוניין במשאב מס' 1 בזמן שהוא מחזיק את משאב 5, כלומר הוא קודם כל ניגש למשאב 5 בגין מօסכמה, סתירה.

Cache

ב

הרצאה

 למדנו על Cache ועל אופן המיפוי מצירון ה-RAM ל-*Cache* (מומלץ לעבור על זה עכשו שוב). שיטת המיפוי שראינו בהרצאה נקראת *Direct Mapping* (מיפוי ישיר), נראה דוגמה נוספת ומפורטת יותר, ולאחר מכן נסתכל על דרך נוספת למיפוי.

1. Direct Mapping (מיפוי ישיר)

עבור מחשב עם המפרט הבא:

$$Cache = 4 \text{ MB}, \text{ RAM} = 4 \text{ GB}, \text{ Word} = 4 \text{ byte (32 bit)}, \text{ Block} = 8 \text{ words}$$

לאן נמפה את הכתובת 5221 מה-RAM ל-*Cache*? איך נפרק את הכתובת ל-

- מספר המילים ב-*Cache* הוא $\frac{2^{22}}{4} = 2^{20}$ כי:

$$(4 \text{ MB} = 4 \cdot 1024 \text{ KB} = 4 \cdot 1024 \cdot 1024 \text{ Bytes} = 2^{22} \text{ Bytes})$$

$$\text{ומכיון שכל מילה היא } 4 \text{ בתים, יש } \frac{2^{22}}{4} = 2^{20} \text{ מילים.}$$

- מספר הבלוקים ב-*Cache* הוא $\frac{2^{20}}{8} = 2^{17}$.

(התא ב-*Cache* אליו ימופנה הערך בכתובת 5221 מה-RAM)

$$\log_2(2^{17}) = 17 \text{ (במקרה זה)}$$

- מספר הביטים שיסמננו את ה-*offset* (התא הרצוי מתוך הבלוק)

$$\log_2(8) = 3 \text{ (block size)}$$

- מספר הביטים שיסמננו את ה-*tag* (הערך שמצווד בשבלוק ב-*Cache* הוא אכן הבלוק הנכון) הוא $\log_2(\#blocks in cache) = 17 - 3 = 14$.

ולפי מה ש חישבנו לעיל, התא ה-5221 במחשב עם 32 ביט ימופנה באופן הבא:



כלומר אם נרצה את הערך בתא 5221 מה-RAM, נבדוק בתא מס' 00000001010001100 ב-

Cache אם ה-*tag* שווה ל-000000000000, נחזיר את המילה במקום 101 בבלוק.

אחרת, הוא לא נמצא ב-

Cache. או לרמה נוספת יותר ב-RAM.

¹ סרטון הסבר כולל דוגמאות של direct mapping

<https://www.youtube.com/watch?v=pSarQQTJbDA>

$2^x - way Associativity$

בשיטת מיפוי זו אנחנו מחלקים את ה-Cache לsets. כל סט מכיל 2^x בלוקים, וכל כתובות בזיכרון ממופה לשט בודד (באוטו אופן כמו Direct Mapping), אבל המידע יכול להיות מאוחסן בכל בלוק בסט זהה.

עבור אותה דוגמה מעמוד קודם, אם נמפה את ה-Cache לפי Cache – 16, יתקיים:

$$\text{מספר המילים ב-Cache} = \frac{2^{22}}{4} = 2^{20}$$

$$\text{מספר הבלוקים ב-Cache} = \frac{2^{20}}{8} = 2^{17}$$

$$\text{מספר הסטים ב-Cache} = \frac{2^{17}}{16} = 2^{13}$$

מספר הביטים של ה-Index הוא: Index

מספר הביטים של ה-Offset הוא: Offset

$\log_2(\#words in RAM) - index - offset = 30 - 13 - 3 = 14$ Tag

ולכן במקרה זה התא 5221 בזיכרון ה-RAM ממופה לפי החלוקת הבאה:

5221 = 

(ה-tag מכיל 4 ביטים נוספים לעומת הדוגמה הקודמת)

Fully associativity

במקרה זה הבלוק יכול להיות מאוחסן בכל מקום ב-Cache. אם גודל ה-Cache הוא 16 בלוקים, אז:

16 – way Associativity = fully associativity

נראה מספר שאלות שיבחרו את ההבדלים במיפוי

שאלה: עבור מחשב עם המפרט הבא:

Word = 4 byte, RAM = 2^{20} words, Cache = 2^8 words, block = 2^3 words

ועבור מיפוי way – 4. נניח שאנו רוצים לגשת לתא 209715 ב-RAM,

מהן כל הכתובות האפשריות ב-Cache בהן התא 209715 יכול להימצא מבין הכתובות הבאות?

- a. 10101101 b. 11000011 c. 11011011 d. 00110011

תשובה: נחשב באותו אופן כמו בדוגמאות קודמות, ונקבל את המיפוי הבא:

209715 = 

מכיוון שיש לנו 4 בלוקים אפשריים בהם הערך יכול להימצא (way – 4), נדרש לבחור 2 ביטים שיסמן באיזה בלוק מדובר, ולכן התא 209715 יכול להיות באחת מהכתובות הבאות:

195 (11000011) – answer b

203 (11001011)

211 (11010011)

219 (11011011) – answer c

או באף אחת מהן! (כלומר הביטים שמנסנים את הבלוק האפשרי נמצאים בין ה-index ל-offset).

ולכן התשובות הנכונות הן b ו-c.

שאלה: עבור מחשב עם המפרט הבא:

$$Word = 4 \text{ byte}, \ RAM = 2^{20} \text{ words}, \ Cache = 2^8 \text{ words}, \ block = 2^3 \text{ words}$$

עבור מיפוי cache – hit – 4. נניח שיש לנו cache של הכתובת 209715 מה-*RAM*

בתא ה-194 ב-*Cache*, מה הערך בתא ה-194 ב-*Cache*-*hit*?

a. 209715

b. 209714

c. The value at physical address 209714

d. We cannot determine for sure

תשובה:

נתון שהוא *Cache – hit* על הכתובת 209715, אך כל הבלוק שמכיל את התא זהה נמצא ב-*Cache*:

00110011001100110000

00110011001100110001

001100110011001100010

001100110011001100011

001100110011001100100

001100110011001100101

001100110011001100110

001100110011001100111

(נתון שגודל *block* הוא 8 ולקן *offset* בגודל 3 ביטים, שיכולים לייצג את המיקום בבלוק)

בנוסף:

$$209715 = \underbrace{00110011001100}_{tag} \underbrace{110}_{index} \underbrace{011}_{offset}$$

$$209714 = \underbrace{00110011001100}_{tag} \underbrace{110}_{index} \underbrace{010}_{offset}$$

כלומר גם 209714 נמצא ב-*Cache* (כי הוא באותו בלוק),

ולכן התא ה-194 ב-*Cache* הוא הערך מהתא 209714 ב-*RAM*, כלומר התשובה היא C.

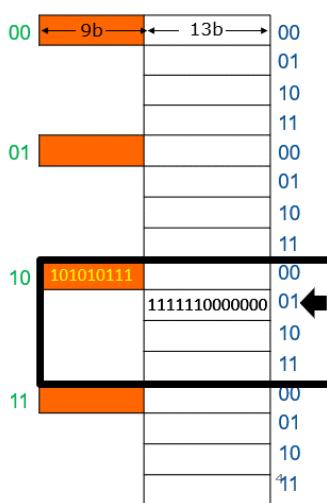
הרצאה 8

בברצאות קודמות העלנו את השאלות הבאות:

- איך נמשח *Fast Access* (גישה מהירה) ל זיכרון ה-*Cache*?
- אילו רכיבים נשמר ב-*Cache*? אילו רכיבים נמחק מ-*Cache*?
- אם רכיב השתנה, איך נשנה אותו בכל שכבות הזיכרון ביעילות? דיברנו בהרחבה על האפשרות הראשונה, כתע נראה מספר דוגמאות. כפי שראינו, כדי למשח גישה מהירה גם בזיכרון ה-*Cache*, נוכל למפות כתובות של *Main Memory* לכתובות תואמות ב-*Cache* בעזרת *.tag, index, offset*. כלומר, נפרק את הכתובת ל-3 חלקים: *Address Translation*

דוגמה Direct Mapping w/Blocks

בහינתן מחשב עם הנתונים הבאים:



Main Memory Address (RAM): 13 bit

Cache Block Size : 4 → 2 bits

Cache Size : 4 blocks → 2 bits

נרצה לגשת לכתובת 10010101111001. כפי שראינו, נחלק אותה באופן הבא:

101010111 10 01
tag index offset

ונבדוק האם tag של בלוק 10 הוא 101010111?

- אם לא, Cache Miss - ולכן נחפש במקום נמור יותר בהיררכיה.
- אם כן, Cache Hit - נחזיר את המילה בPOSITION 01 בבלוק 10 (ניתן לראות באיזור משמאלי).

Set Associativity

ניתן לחבר מספר זיכרונות *Cache* בivid, ולחפש בכל אחד מהם את הכתובת הרצiosa.

כל בלוק יכול להיות ב-*Cache* כלשהו מתוך קבוצה של *n* זיכרונות *Cache*.

כולומר כל כתובות יכולה להיות ממופea למספר כתובות, ובכך נפחית את הסיכוי להתנגשות.

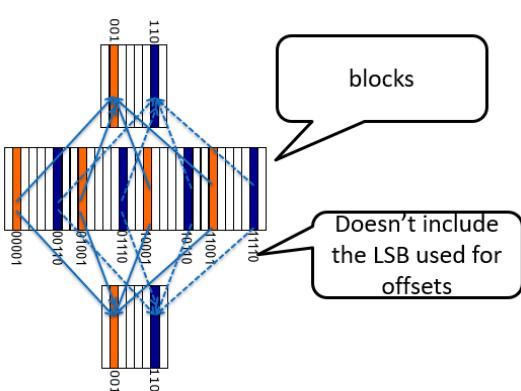
למשל אם הזיכרון הוא 2 – Way Set Associative (*Cache* 2 זיכרונות מחוברים ביחד):

כל בלוק יכול להיות באחד מבין 2 זיכרונות – *Upper Cache* ו- *Lower Cache*, ולכן:

- יהיו פחות התנגשויות (*Cache Conflicts*) ויתר גמישות.

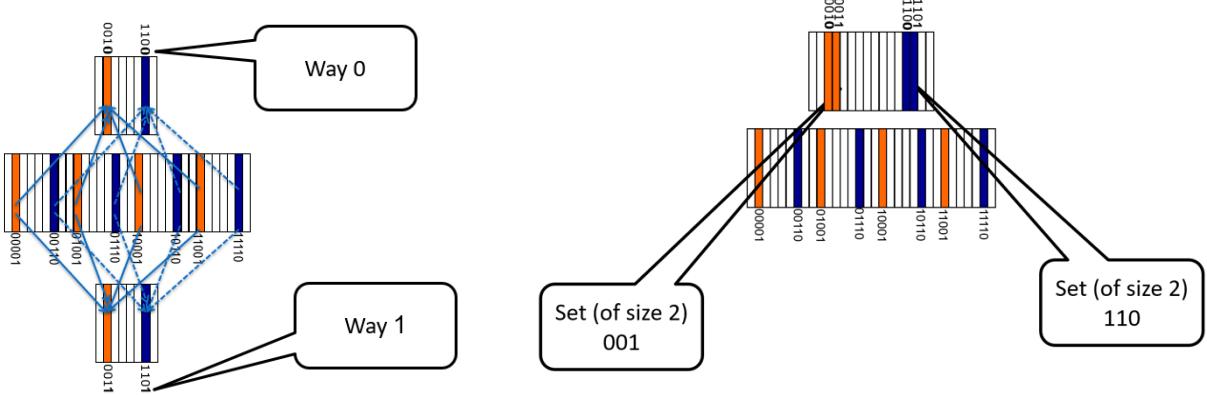
- החיפוש יעשה בשני מקומות שונים.

- החיפוש נעשה במקביל ברמת החומרה.



2-Way Set Associativity

שנחפש כתובת, נחפש קודם ב-*way*-0 ולאחר מכן ב-*way*-1. באיר משמאל הבלוק **0110** והבלוק **1101** הם שני בלוקים שונים שהאינדקס שלהם הוא 110. נוכל לאחד אותם ולקבל את האיר מימין:



דוגמה: איך נתייחס לכתובת 178 (10110010)?

Memory: 256 words (word = 8 bit)

cache: 32 words; 2 – Way set associative

Block: 8 byte (8 words)

ב-*cache* בגודל 32 מילימ' עם שיטת מייפוי *way* – 2 יש 4 בלוקים,

לכל *way* יש שני בלוקים, וכן נציגר בית בודד שיציין את הבלוק.

über בлок בגודל 8, נציגר 3 ביטים שיציינו את *offset*,

ולכן ל-*tag* נשארו $4 - 3 - 1 = 0$ – 8 ביטים, כולם:

tag index offset

נבחן את המקרים:

- אם ה-*tag* של בלוק 00 הוא 1011, יש *cache hit* ונחזיר את התוכולת של הכתובת 00010.
- אם ה-*tag* של בלוק 01 הוא 1011, יש *cache hit* ונחזיר את התוכולת של הכתובת 01010.
- אחרת, *Cache Miss*.

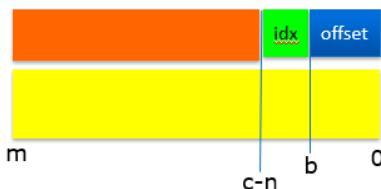
贊記憶體的存儲方式 – *n – Way Set Associative Cache*, 其中 *n* 可能有 *n* 個可能的存取位置。

贊記憶體的存儲方式 – *Fully Associative Cache* – 每一個可能的地址都可以存取。

חישוב *index, tag, offset*

בהתאם להגדלים הבאים:

RAM: 2^m bytes, *cache* 2^c bytes, *block* 2^b bytes, $2^n \rightarrow$ way set associative



עבור כתובות X , נוכל לחשב את *index, tag, offset* באופן הבא:

offset: $X \bmod 2^b$

index: $(X \div 2^b) \bmod 2^{c-n-b}$

tag: $X \div 2^{m-c+n}$

עבור $2^n - 2^a$: כל ש a גדול, הגודל של ה- *tag* גדול.
לכן יש יותר overhead (אבל פחתות *cache conflict*).

סיבות ל-*Cache Miss*

- . *Cache Miss (cold start)* -
- . *Capacity* -
הפתרון – גודיל את ה-*Cache*.
- . *Conflict (Collision)* -
הפתרון – גודיל את ה-*Cache* או גודיל את ה-*Associativity*.
- . *Coherence (Invalidation)* -
ה-*Cache* עם הערכים שלו, וכך יכול לגרום ל-*Cache Miss* עבור התהילך השני.
- . *Policy* -
אם המדיניות (מי מכניסים ומי מוציאים) לא אופטימלית.

עד עכשיו ענו על השאלה הראשונה – איך נמפה כתובות בשיל למשגישה מהירה.
כעת נעבור לשאלת השנייה, אילו רכיבים נשמר ב-*Cache*? אילו רכיבים נמחק מה-*Cache*?

כתיבה וקריאה:

Cache Misses יכולים לקרות גם בכתיבה וגם בקריאה. רוב זיכרונות ה-*Cache* שומרם את המילים האחרונות שקרהנו, בגלל ההנחה שכנראה נצטרך אותן שוב, או מילים שקרובות אליהן. במקרה זה, אם חיפוש ב-*Cache* מוביל ל-*Cache Miss*, תבוצע החלפת הבלוק כלשהו ב-*Cache* לטובות הבלוק שהיפשנו.

Spatial locality – אם קראנו מילה מסוימת כנראה שנרצה לקרוא מילים שקרובות אליה בקרוב.

Temporal locality – אם קראנו מילה מסוימת, כנראה שנרצה לקרוא אותה שוב בקרוב.

בכתיבה קיימות שתי אפשרויות:

1. נכתב את הערך ל-*Lowest Memory* (בhirerchity ה-*Cache*), איפה שהבלוק קיים.
(אין צורך ליבא את כל הבלוק מחדש *Cache* אלא רק לעדכן את המילה)
 2. בדומה לקריאה, נתען את הבלוק הרלוונטי ל-*Highest Level Cache* ונכתב בו את הערך.
- הערה: בפועל פעולה הכתיבה מורכבת יותר, צריך לשמור על עקביות של המידע בהיררכית ה-*Cache*.

Block Replacement

כעת נבחן מספר אפשרויות למחיקת ערכים מהזיכרון. זיכרון ה-*Cache* מלא כל הזמן במידע. כשייש בפעולות קריאה, צריך ל揖א בלוק חדש ל-*Cache*, וכך צריך לבחור *Victim* שאותו נפנה לטובת הבלוק החדש. נרצה שה-*Victim* יהיה הבלוק שאמנו צואו אותו, כמוות הפספוסים בעtid תהיה מינימלית. ה-*Victim* צריך להיות מאותו סט אסוציאטיבי (כלומר מtower המיקומות אליו הם ניתנים למופת את הבלוק החדש). הקורבן שנפננה מה-*Cache* יכול להיבחר באופן רנדומלי, אבל גודיף לבחור קורבן שלא ייגשים אליו לעיתים קרובות (כדי למנוע/לדוחות העברה שלו ל-*Cache* מחדש).

נראה מספר אלגוריתמים / גישות לבחירת הקורבן (*Replacement Algorithm*)

Optimal Algorithm (לא מציאותי):

לפי גישה זו, נחליף את הבלוק שלא השתמש בו בעtid להכי הרבה זמן (הבלוק שלא נעשה בו שימוש, או שהשימוש בו הכי רחוק). מכיוון שלא ניתן לחזות את העtid, השתמש בגישה זו כדי להשוות תוצאות של אלגוריתמים אחרים לפתרון זה, האופטימלי.

NRU – Not Recently Used:

נכזה להעיר לפי תוצאות העבר. לכל בלוק יש *Reference Bit* המציין אם השתמשנו בו לאחרונה. 1 יצביע שכך, 0 אחרת. בכל פעם שניגש לבלוק נשנה את הביט ל-1 ונדיר *Clock Interrupt* לפרק זמן מסויים שישנה את הביט בחזרה ל-0. בנוסף, אם הביטים של כל הבלוקים הם 1, נאפס את כולם ל-0. כשנចטרף לבחור *victim*, נבחר בלוק באופן רנדומלי מתוך הבלוקים עם בית 0. מדיניות זו קלה למימוש, לא צורכת הרבה משאבים (ביט אחד), אך אין בה סדר (חלוקת גסה לשתי מחלקות) והביצועים בהתאם.

FIFO – First In First Out

נמיין את הבלוקים לפי הסדר בו הם נכנסו ל-cache, נאחסן את הסדר ברשימה מקוشرת. ה-*victim* יהיה הבלוק הראשון (זה שנמצא הכי הרבה זמן ב-cache). החישוב: הבלוק הישן ביותר עלול להיות הבלוק שנשתמש בו הכי הרבה (למשל ה-*instruction*).

Belady's Anomaly

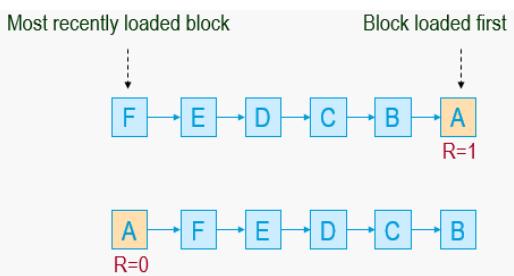
נראה התנהגות "מוזרה" בשימוש עם ה-cache. נניח ונחנו טענים ל-cache בלוקים של זיכרון בסדר הבא:
 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
 נשים לב איך יראה ה-cache כאשר יש לו 3 בלוקים, או 4 בלוקים.

Cache with 3 blocks	1	2	3	4	1	2	5	5	5	3	4	4
	1	2	3	4	1	2	2	2	5	3	3	
	1	2	3	4	1	1	1	2	5	5		
Cache with 4 blocks	1	2	3	4	4	4	5	1	2	3	4	5
	1	2	3	3	3	4	5	1	2	3	4	
	1	2	2	2	3	4	5	1	2	3		
	1	1	1	2	3	4	5	1	2	2		

באיור הנ"ל, כל עמודה אפורה מייצגת שינוי ב-cache, וכל עמודה לבנה משמעותה שהיא לא השתנה. היינו מצפים שב-cache עם 4 בלוקים יהיו פחות שינויים מאשר ב-cache עם 3 בלוקים, אבל להפתענו, הוספה בלוק ל-cache, לא הורידה את מספר השינויים אלא רק הגדילה אותו.

Second Chance FIFO

בדומה ל-*NRU*, לכל בלוק יש *Reference bit*. כשבאים אליו או כשניגשים אליו הוא משתנה ל-1.



האלגוריתם פועל כמו FIFO אבל אם $R = 1$, נפעל באופן הבא:

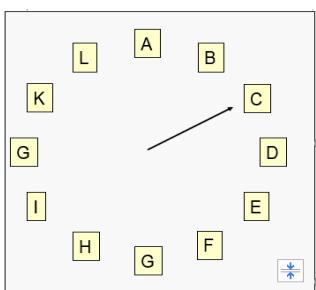
- נשנה את R להיות 0.
- נזיז את הבלוק לסוף התור.
- נועבר לבלוק הבא ונפעל באותו אופן.

פתרון זה לא יעיל בגלל שהוא מזיז בעקבות בלוקים ברשימה.

ניתן למש את האלגוריתם גם בעזרת שעון:

ונסדר את הבלוקים במעגל ונגיד ר"מ "מחוג" שמאכיע על הרראשון בתור (באיור C). אם $R_c = 1$ נשנה את R_c ל-0, נזיז את המחוג שיציביע ל-D, וכך הלאה.

מימוש זה נוח ומסודר יותר מרשימה מקוشرת רגילה.



LRU – Least Recently Used

ה-*victim* יהיה הבלוק שלא היה בשימוש הכי הרבה זמן מבין כל הבלוקים בזיכרון. גישה זו משתמשת בהנחה שבлокים שהשתמשו בהם לאחרונה הם הבלוקים שסביר להניח שנרצה להשתמש בהם שוב.

דומה ל-*NRU* אבל במקומם לנחל בית יחיד, נשמר את זמן השימוש בבלוקים השונים.

LRU יקר וקשה למימוש. ניתן לשמר *Timestamp* לכל בלוק ואז לבחור את הזמן המינימלי, אבל בחירת המינימום זו פעולה יקרה. במידה ונתחזק רשיימה ממוינת, נדרש להשיקע זמן ומשאבים במילון הרשימה מחדש אחרי כל גישה לבlok.

list overhead: $\log_2(n)$ bits/block

הערה: *LRU* הינה השיטה הכי נפוצה, ויש הוכחה שברוב המקרים היא טובה (קרובה לאופטימלי עד כדי קבוע). אם *LRU* נכשלת, לרוב משתמשים בה או בוויריאציה עליה שמהונדת כדי שתעבד טוב יותר.

מימוש *LRU* עם מטריצה:

בשביל לדעת מה הבלוק שאוטו צריך להוציא נשמר מטריצה ריבועית בגודל מספר האיברים ב-*cache*, בכל פעם שניגשים לבlok ב-*cache* משנים לו-1 את כל הערכים בשורה המתאימה, ול-0 את כל האיברים בטור המתאים. באופן זה יוצא שהשורה עם הערך הבינארי ה-1 נמוך מתאימה למספר הבלוק ב-*cache* שאוטו צריך לפנות.

ניתן למימוש ברמת החומרה, אבל זה דורש *n bit/block*.

ניתן לראות בדוגמה הבאה:

0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	0	1	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	1	1	1	0	0	1	1	1	0	1	1	0	0
2	0	0	0	0	0	0	0	0	1	1	0	1	1	1	1	1	1	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
(a)				(b)				(c)				(d)				(e)			
(f)				(g)				(h)				(i)				(j)			

תרגול 7

בשעה הראשונה של התרגול הייתה חזרה על אלגוריתמי החלפה (*Replacement Algorithm*) של מדריכנו בהרצאה (מומלץ לעבור עליהם שוב). לאחר מכן הוצגה השאלה הבאה:

<p>נניח כי Cache הוא בגודל 4 בלוקים, כל בלוק בגודל 256 מילוט זיכרון וממוצע read-allocate,fully associative. כל מילוט זכרון בגודל 4 בתים. גודל כל משתנה הוא בוגדל מילה.</p> <p>א. (12 נק') ניזג כל בלוק ע"י מספר. עבור סדרת הגישות הבאה (משמאלי לימין) צייר מי היו ה-victims הראשונים (הблוקים שייפנו הראשונים מה-cache) בכל אחד מהאלגוריתמים בטבלה.</p> <p style="text-align: center;">→</p> <p style="text-align: center;">1,2,3,4,1,3,2,1,5,2,3,6,5,3,2,1</p> <p>כאשר יש יותר מקום רק אחד ב-cache, האיבר יוכנס למקום בעל הכתובת הנמוכה ביותר.</p>
--

פתרונות:

	Victim 1	Victim 2	Victim 3
LRU	4	1	6
FIFO	1	2	3
Second Chance FIFO	1	4	5

ב-*LRU*, בgal שניגשנו ל-1,2,3,-1, גרמנו לכך ש-4 הוא הבלוק שהשתמשנו בו הכי פחות, ולכן הוא הראשון לעוף ובמקומו נכנס 5. באותו אופן 2,3 גרם לכך ש-1 הוא הבא בתור לעוף.

ב-*Second Chance FIFO* צריך לזכור שלאחר הכניסה 1,2,3,4 הביט עבר כולם בעל ערך 1 והפינטרא (המצביע על האיבר הבא שאמור לצאת מה-*Cache*) מצביע על 1 (כי לפי *FIFO* הוא הראשון לצאת).

לאחר מכן הגיעו ל-1,3,2,-1 אין שניי (הbeit עבר כולם כבר 1). כאשר הגיע ל-5, המצביע עבר על כל אחד מהבלוקים שכבר ב-*Cache*, כולם עם בית 1 לכן הוא מאפס לכולם את הביט ווחזר חזרה ל-1. מכיוון שהbeit שלו כעת הוא בעל ערך 0, אז 1 הקורבן הראשון ו-5 נכנסו במקומו. כעת הפינטרא מצביע על 2 אשר 2 ו-3 מגיעים,beit שלהם הופך להיותשוב עם ערך 1. כאשר הגיע 6, הפינטרא שמצוין הגיע על 2 מאפס אתbeit שלו וועבר ל-3, מאפס אתbeit שלו גם כן וועבר ל-4, ולאחר מכן מוציא את 4 שהוא הקורבן השני כיbeit שלו עם ערך 0 ו-6 נכנס במקומו. הפינטרא כעת מצביע על 5. כאשר 3,2 מגיעים הערך שלbeit שליהם הופך להיות 1, ואחר אחד מגיע,beitם לכל הבלוקים מתאפסים והפינטרא חוזר ל-5 ומוציא אותו, הוא הקורבן השלישי, ו-1 נכנס במקומו. מי שזקוק להסביר נוספת והפינטרא חוזר עידן שעלה למודול ב-05/07 אפשר לראות בדקה 31: פתרון מפורט.

Memory Management

כפי שאנו כבר יודעים, ה-*CPU* יודע לעבוד רק עם רגיסטרים וה-(*Main Memory (RAM)*). אם אנחנו צריכים ערכים מהדיסק הקשיח למשל, נדרש לטען אותם קודם ל-*RAM*. בנוסך, ה-*CPU* יודע לבצע את הפעולות הבאות:

- *Data Handeling* – שינוי ערך של רגיסטר (*set*), אחסון ערך מרגיסטר לזיכרון *RAM* (*store*)
 - טיענת ערך מזיכרון ה-*RAM* לרוגיסטר (*load*)
 - *Arithmetic Operations* – פעולות אריתמטיות על רגיסטרים (פעולות *Bitwise*, השוואה וכו')
 - *Control Flow* – בקרת זרימה באמצעות רגיסטרים (קפיצה וקפיצה מותנית)
- כדי להריץ תוכנית למשל, נהיה חייבים לייבא אותה לזכרון ה-*RAM* כדי שנוכל להריץ אותה.

Physical Address

הכתובת שבה ישב ערך מסוים בזכרון ה-*RAM* נקראת **הכתובת הפיזית** שלו. מרחב הכתובות הפיזי תלוי בגודל הדיסק, למשל עבור זיכרון *RAM* בגודל 8GB, אורך כתובות בזכרון היא 33 כי:

$$8GB = \underbrace{2^3}_{8} \cdot \underbrace{2^{10}}_{1GB=1024MB} \cdot \underbrace{2^{10}}_{1MB=1024KB} \cdot \underbrace{2^{10}}_{Bytes} = 2^{33} Bytes$$

כלומר 8GB הם 2^{33} Bytes, וכך ניתן ליצג את כל מרחב הכתובות בזכרון של 8GB, נציגו 33 בתים - $(8GB)_{log_2}$. (בקורס זה אנחנו עובדים עם בתים ולא ביטים)
הערה: כתובות פיזיות נקראות גם "כתובת אמיתית" או "כתובת בינארית".

VAS (Virtual Address Space)

ב-*Nand2Tetris* (או מבנה המחשב) בנינו מחשב והרכנו עליו תהלייר בודד - למשל המשחק שבניינו. התהלייר השתמש בכל מרחב הכתובות כי הוא התהלייר היחיד שרצץ, ולא נדרש להתעסק עם חלוקת מרחבי כתובות לתהליכים שונים. בפועל יש המונע תהליכים שעובדים עם הזיכרון בו זמנית. מערכת הפעלה צריכה לוודא שכולם עובדים עם הזיכרון ביתאום, והוא עושה זאת באמצעות *VAS*.

VAS הוא מרחב כתובות וירטואליות שמערכת הפעלה מקצה לתהלייר מסוים. במערכת הפעלה של 32 סיביות, גודל שטח הכתובות הוירטואלי הוא 2^{32} Bytes. ובמערכת הפעלה של 64 ביט 2^{64} . לדוגמה, מצביע לפונקציה או משתנה הוא למעשה כתובות וירטואלית. בנוסך, למדנו שלכל תהלייר יש מחסנית, *heap* ואגמנטים נוספים שמוגדרים רק עבורה, כל אלו נמצאים במרחב הוירטואלי של התהלייר !

MMU

נניח שימושה מסוים קובל כתובות וירטואליות, אך הכתובת ממופה לכתובת פיזיות בזכרון ה-*RAM*? *Memory Management Unit (MMU)* הוא רכיב חומרה שאחראי על המיפוי. תוכניות שרצות על המחשב מתנהלות עם כתובות וירטואליות בלבד, הן לא מודעות לכתובות הפיזיות שלhn ו מבחינותן הן היחידות שרצות על המחשב ולכן כל הזיכרון לרשותן.

Segmentation

הגיע הזמן שנבין מה השגיאה המוכרת "Segmentation Fault" אומרת ! כל תוכנית שרצה על המחשב היא אוסף של סגמנטים, כאשר סגמנט הוא יחידת זיכרון לוגית. פילוח זיכרון (*Memory Segmentation*) היא פעולה המחלקת את הזיכרון הראשי (*RAM*) לשגמנטים, למשל:

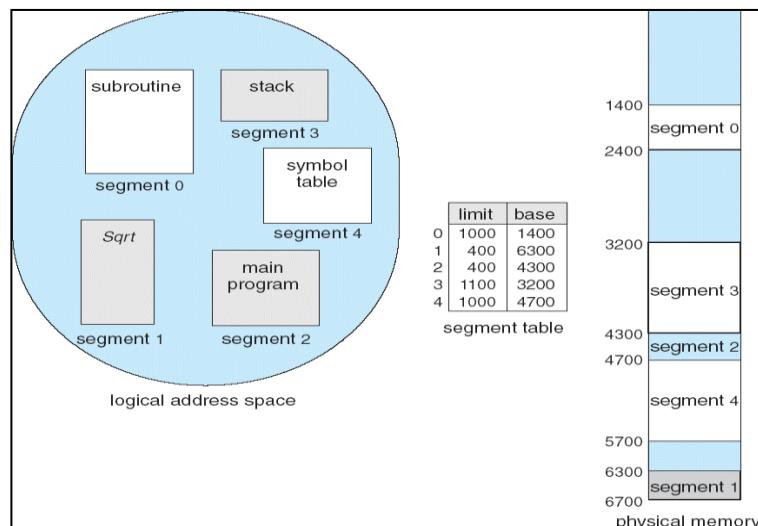
main program, function, object, local & global variables, stack, symbol table, arrays

איך רכיב החומרה *MMU* מפיה את הכתובות הווירטואליות לכתובות הפיזיות?

נראה מספר הגדרות בארכיטקטורת סגמנטים

- כל כתובות וירטואלית מורכבת מזוג $\langle segmentNumber, offset \rangle$ (מספר סגמנט והאינדקס בתוכו) אמרנו שבזיכרון *RAM* בגודל 64 כל כתובות וירטואלית מורכבת מ-33 בתים, אז חלק מהכתובת תציג את ה-*offset* והחלק השני את מספר הסגמנט.
- *Segment Table* – טבלה שמערכת הפעלה מייצרת לכל תהליך, וכל שורה בה מצינית סגמנט שהוקצה עבור התהליך. כל שורה מכילה את הפרמטרים הבאים:
 - *base* – הכתובת הפיזית בה מתחילה הסגמנט בזיכרון.
 - *limit* – מסמן את האורך של הסגמנט. לכן, *offset* נחשב תקין אם הוא קטן מ-*limit*.
 - *Validation Bit (also called present – bit)* – בית שמציע אם הסגמנט נמצא בזיכרון הפיזי (*b-RAM*) או שהוא נמצא בוירטואלי, אז נדרש לטען אותו מהdisk לזכרון לפני שנוכל לגשת אליו. (0 אומר שהוא בזיכרון הוירטואלי, 1 אחרת)
 - *read/write/execute privileges* – ביטים נוספים המצביעים אם אפשר לקרוא / לכתוב / להריץ את הסגמנט זהה.
- כל תהליך מחזיק גם ברגיסטרים הבאים:
 - *SegmentTable Base Register (STBR)* – מצביע למקום ה-*ST* בזיכרון.
 - *SegmentTable Length Register (STLR)* – מצין את מספר הסגמנטים ששימוש על ידי התוכנית. מספר הסגמנט *s* של התהליך הוא מספר חוקי אם $STLR < s$.

שגיאת Segmentation Fault נזרקת אם פועלות התרגומים מכתובות וירטואלית לכתובות פיזית נכשלת.

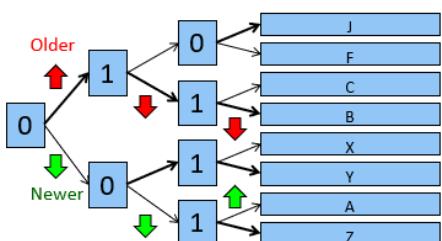


מציג CUT אלגוריתם החלפה נספף - *Pseudo-LRU* – ללא הספקנו להציג בהרצאה קודמת.

Pseudo-LRU

נזכר שאלגוריתם *LRU* (Least Recently Used) בוחר את ה-*victim* שלו להיות הבלוק שלא היה בשימוש הכי הרבה זמן. כפי שכברذكرנו, מימוש *LRU* הוא פשוט, הוא דורש שימוש בחומרות זמן ושימוש בראשימה ממינית או רשיימה מקוורת, מה שיכל לגרום לתקורה גבוהה. מסיבה זו, מעבדים רבים (בינהם?) ממשים אלגוריתם מקרוב לאלגוריתם *LRU* שמשיג תוצאות דומות בסביבות גבוהה.

(כלומר האלגוריתם לא תמיד יבחר את הבלוק ה-*LRU* Least Recently Used אלא בערך) נסביר את השימוש בעזרת האירור משמאלו. נבנה עץ שהעלים שלו הם הבלוקים בזיכרון, וכל קדקוד בעץ שאינו עלה מכל בית 0 או 1 המצביע לאיזה בן של הקדקוד ניגשנו בפעם האחרון שעברנו דרכו.



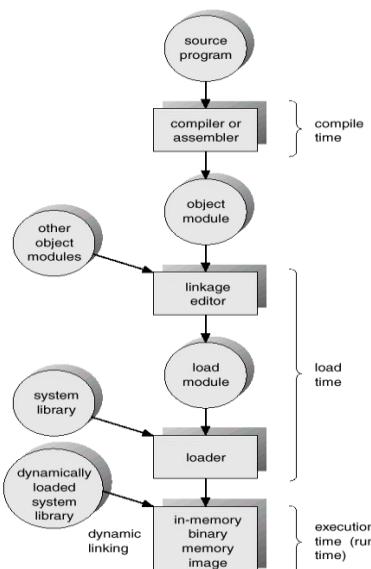
- 0 אם ניגשנו לבן התחתון.

- 1 אם ניגשנו לבן העליון.

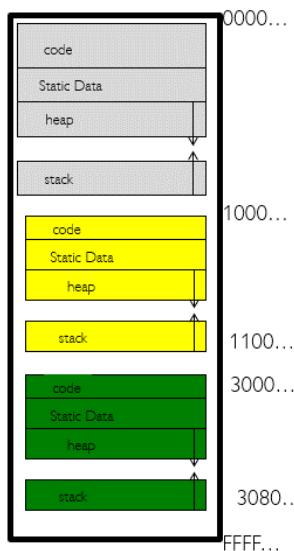
וזו יכול להשתמש בערכאים כדי למצאו בלוק שלא ניגשנו אליו הרבה זמן. נסתכל על האירור משמאלו כדי למצאו בלוק צזה: נתחל בשורש, הביט שלו הוא 0 لكن ניגשנו לאחרונה לבן התחתון שלו. מכיוון שאנו מוחפשים בבלוק שלא ניגשנו אליו לאחרונה, נרצה להמשיך עם הבן העליון שלו. הגיענו לקדקוד עם בית 1, ככלומר ניגשנו לאחרונה לבן העליון שלו, لكن נמשיך עם הבן התחתון שלו. הגיענו שוב לקדקוד עם בית 1 ולכן ניגש לבן התחתון, *B*, והוא הבלוק שאנו מוחפשים כמו שלא ניגשנו אליו הרבה זמן. העז הנ"ל מתאפשר למשל עבור סדר הגישות הבא: *A, F, X, Z, B, C, Y, J*. ככלומר האלגוריתם מצא ש-*B* הוא בלוק ישן למרות ש-*J* הוא הבלוק הכי ישן, אך אכן מדובר בקיורוב מסויף עבורנו בתמורה לתקורה נמוכה: $\frac{1}{n}$ ביטים לכל בלוק. (כי לעצם בינהי עם *n* עליים יש *n* – *n* צמתים פנימיים)

Memory Management

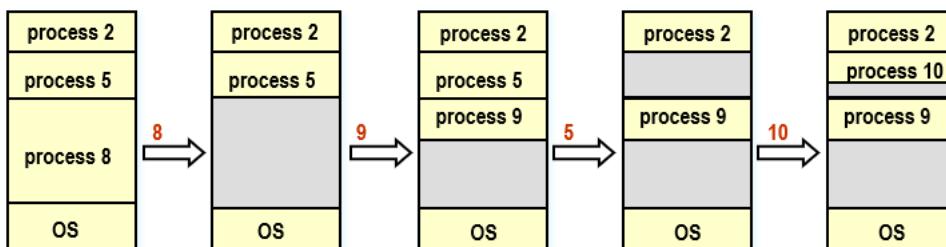
למדנו בתרגול האחרון שכל תהליך משתמש בזיכרון המחשב תחת הנהנה שהוא התהליך היחיד שרצ על המחשב. הוא מתנהל עם כתובות זיכרון וירטואליות (*MMU*, Logical Addresses) (רכיב חומרתי במחשב) אחראי על תרגום הכתובות הווירטואליות לכתובות אמיתיות (*Physical Addresses*), כדי לאפשר למספר תהליכי לroz בו זמן. תרגום הכתובות נעשה בשלב *Load & Execute*, מכיוון שהכתובות של חלק מהתוכניות ידועות רק לאחרטעינת התוכנית כולה לזיכרון הראשי. בעמוד 22 ראיינו דוגמה מופשטת בה לכל תהליך יש מרחב כתובות משלו שמוגדר בעזרת 2 משתנים: *Bound*-*I Base*, וה-*MMU* מתרגם כתובות וירטואלית *x*, לכתובת אמיתית *x + base*.



ניהול זיכרון



- **Internal Fragmentation** – בזמן הקצתה זיכרון לתהילך מסוים, נרצה לשמר מקום נוספים צמוד לזכרון שהקצתנו, כדי לאפשר לתהילך להקצות עוד זיכרון מבלי שנ לצורך להעתיק אותו למקום אחר. ככלומר נשאיר חורים קטנים בזיכרון שלא ניתן להשתמש בהם בין ה-*stack* ל-*heap*.
- (המילה *Fragmentation* בשם היא בגלל שיש שבריריז זיכרון לא מונצחים, ו-*Internal* בגלל שהם נמצאים בתור הזיכרון שהקצתנו לתהילך כלשהו)
- **External Fragmentation** - כתובנית רצה, היא מקבלת מקטע רציף של זיכרון אותו היא מפנה בסיום הריצה. לאחר זמן, נוצרים חורים קטנים בין מקטעי זיכרון שונים שכולים ביחד להוות מספיק מקום עבור זיכרון לתהילך חדש, אבל כל חור קטן זה לא מספיק בשביל תהליך שלו. באior הבא ניתן לראות למשל את החור בין process 9 ל-*process 10* שיכל למנוע מאייתנו למקם תהליך שצריך את 2 החלקים האפורים כדי לרצוף:

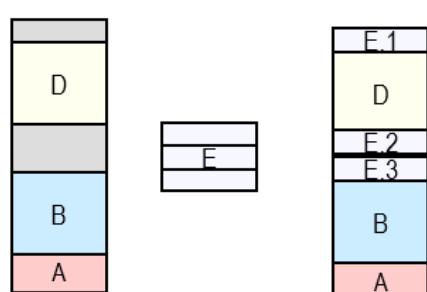


لتופעה זו קוראים *External Fragmentation*, והוא אחת הביעות הנפוצות בניהול זיכרון. באופן כללי, *Fragmentation* היא תופעה בה יש מספיק זיכרון פנוי למקם תוכנית חדשה, אבל הזיכרון לא רציף ולכן לא ניתן להקצות אותו.

פתרון אפשרי ל-External Fragmentation – Compaction – (לא פתרון טוב):

הזהה וסידור מחדש – כישיש חורים קטנים בין בלוקים של זיכרון, נזיז את הבלוקים הקיימים ונאחד את כל החורים במקומות אחד. הבעיה: פעולות ההזהה היא יקרה. כמו כן כדי להעתיק זיכרון ממקום אחד לאחר אנחנו חייבים להשתמש במקומות נוספים שלא בהכרח קיימים.

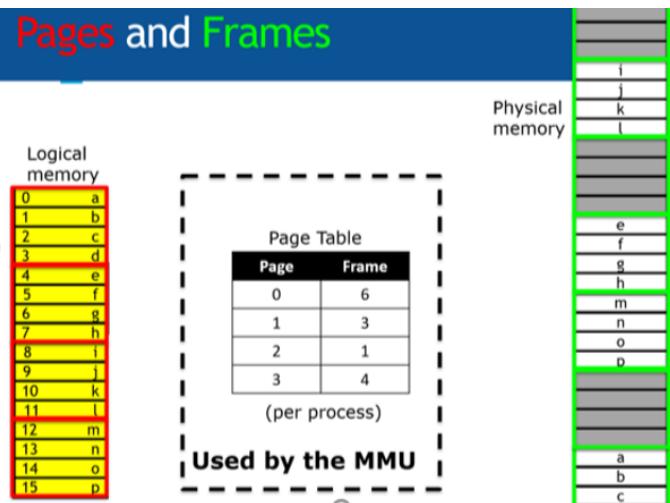
הפתרון הנפוץ – שימוש בדףים (Pages):



מחלק את התהילך שאנוינו מעוניינים להריץ לדפים (מקטעי זיכרון) קטנים יותר בדרך כלל בגודל $4KB$ (4), וכל דף זהה נמקם בחור אחר בזיכרון. שימוש בדףים מחייב מייפוי בין כל דף לבין מיקומו בזיכרון. (רוב מערכות הפעלה משתמשות בפתרון זה)

דף – חלוקת הזיכרון של תהליך למספר מקטעים זיכרון.
מסגרת (frame) – חלוקת הזיכרון האמתי למסגרות.

Pages and Frames



icut כדי לאפשר לתוכנית לróż כמו שצרכן נצרך למפות את הדפים (מהזיכרון הלוגי) למסגרות (מקטעים בזיכרון האמיתי). הדפים והמסגרות יהיו באותו הגודל.

על מנת לבצע את העימוד נוצר טבלה – *Page Table*, שתפקידה לשמר באיזה *frame* נמצא כל דף. נשים לב שככל תהילך יחזיק טבלה צזו, וכך ה-MMU ידע לתרגם את הכתובות שלו.

:Paging

מערכת הפעלה שומרת את המיקומים של המסגרות (*frames*) הפנויות. כשתהליך מבקש מקום ל-n דפים, מערכת הפעלה מחפשת n מסגרות פנויות ומקצת לו אותן. ככלمر היא מוחקת אותן מרשימת המסגרות הפנויות, מעטיקה את המידע לזכרון האמיתי ויוצרת את ה-*Page Table* של התהליך. כאשר תהיליך מסוים, המסגרות שהוקטו לו חוזרות לרשימה המסגרות הפנויות.

גודל אופטימלי של דף

מערכת הפעלה מקצתה תמיד מספר שלם של דפים לתהליך מסוים (היא לא תקצת דף וחצי למשל), ולכן ככל שנגדיל את גודל הדף, ה-*Internal Fragmentation* יגדל גם הוא כי אם תהיליך צריך דף וחצי הוא יקבל 2 דפים. חצי מהדף זה זיכרון שלא נחוץ לתהליך, ותהליכים אחרים לא יוכל להשתמש בו. מנגד, ככל שנקטין את גודל הדף, טבלת ה-*Page Table* תגדיל ולפניהם התקורה גדלה. עברו:

p – page size, s – process size, e – size of the entry in the page table

$$\text{התקורה היא: } \frac{se}{p} + \frac{p}{2}$$

$\frac{s}{p}$ זה מספר הדפים שתהליך צריך (כזכור מספר השורות בטבלה), ונכפיל את זה בגודל כל שורה בטבלה.

$\frac{p}{2}$ הוא ממוצע של חצי דף ריק שנשאר לכל תהיליך.

על מנת למצוא את גודל הדף האופטימלי נגזר את $\frac{se}{p} + \frac{p}{2}$ לפי ק ונחפש נקודת מינימום. נקבל $\sqrt{2se}$.

עבור bit b , $e = 64$, $s = 1MB$, $p = 4KB = 2^{12}$, וזה הגודל האופטימלי.

דוגמה לתהיליך תרגום כתובות

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	
9	
10	
11	
12	e
13	f
14	g
15	h

Page	Frame
0	6
1	3
2	1

נניח שהאIOR הצחוב משמאלי מຕאר את ה-*זיכרון הווירטואלי*, האIOR האפור-לבן את ה-*זיכרון האמייתי* והטבלה את ה-*Page Table*. נרצה למצוא את הכתובת האמייתית של *g*.

ראשית נתרגם את 6 ליצוג הבינארי 00110, ולכן: $\underline{001} \quad \underline{10}$ *page offset*

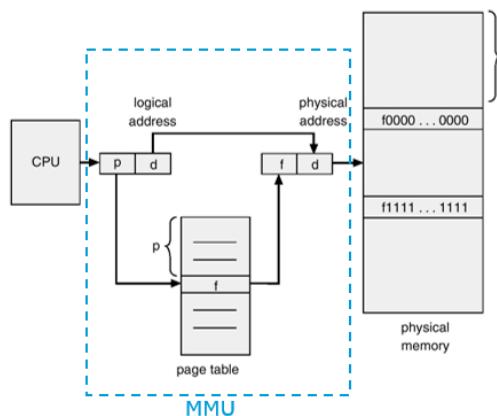
נחפש לפי ה-*Page Table* את השורה בטבלה. ה-*Page* הוא (1 ביצוג עשרוני), לכן נלך לשורה מספר 1 ב-*Page Table*. בשורה מספר 1 רשום שהמגראת המתאימה היא מסגרת מספר 3 (011 ביצוג בינארי) ולכן:

$\underline{011} \quad \underline{10} \Rightarrow \text{physical address} = 01110$ *frame offset*

(4 ביצוג עשרוני). כלומר הכתובת האמייתית של *g* היא 14.

ארQUITקטורת ה-MMU

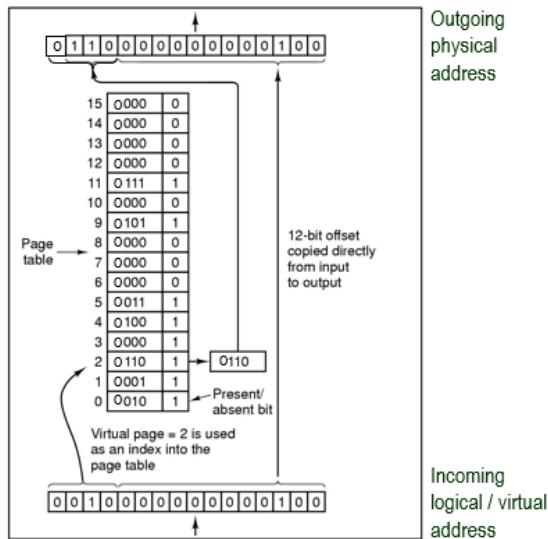
למשל עבור הוראה *d|p*, ה-*CPU* מעביר ל-*MMU* בקשה לתרגום הכתובת של *d*. ה-*MMU* הולך לאינדקס *p* בטבלה ומוחזיר את *f* – הערך של ה-*frame*.



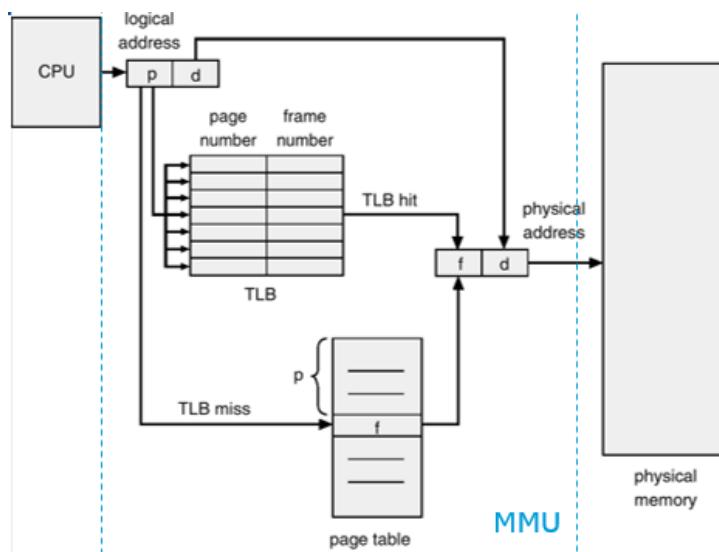
דוגמה של פעולה ה-MMU עבור offset 12 ביטים

כל גישה לזיכרון דורשת Mai-Teno בפועל 2 גישות זיכרון (פעם אחת ל-*Page Table* ופעם אחת ל-*זיכרון* האמתי). פעולה זו מגדילה את התקורה בצורה משמעותית. בשайл לפטור את הבעה הזאת, יש במחשב *זיכרון* נוסף - *TLB* – *Translation Lookaside Buffer* – *Cache* – *Cache Hit*. ואמ' יש *Cache Hit* חסכנו גישה איטית ל-*זיכרון*.

למסגרות שראינו לאחרונה אנחנו עושים לבקש אותם שוב. בזמן תרגום כתובות, נחפש קודם ב-*Cache* וב-*Cache* נרשם מיפוי של עמוד למסגרת, ולא רק אינדקסים.



ה- *TLB* יהיה לרוב *64 Entries*, קטן – *Fully Associative* או בעזרת חומרה מיוחדת. (כמובן הוא יהיה מהיר מאוד). הארכיטקטורה נראה כך:



Virtual Memory

נרצה לדעת ברגע נתון איזה זיכרון גדול יותר: האמתי או הווירטואלי. אם יש הרבה תהילכים שרצים במקביל במחשב, ומרקך של כל תהיליך חושב בכל הזיכרון לרשותו, יכול להיות שהזיכרון הלוגי יהיה הרבה יותר גדול מהזיכרון הפיזי. (כלומר סך מקומות הזיכרון שהתהליכים הקצו לעצםם גדול מהזיכרון בפועל) במצב זה נדרש להשתמש בזכרון וירטואלי. במקרה זה רק חלק מהתוכנית נתען לזכרון האמיתי, ורוב *the-data* של התוכנית נשאר בדיסק. כצפוי, נתען את המידע מהדיסק לזכרון.

- במצב זה הזכרון הלוגי יכול להיות גדול בהרבה מהזיכרון הפיזי.
- יותר תהילכים יכולים לזרז במקביל ולהשתמש בזכרון הראשי.
- אפשר הרצה עיליה יותר של תהילכים (פחות מידע לטען מראש).

הבעיה: אם המידע הנחוץ לא נמצא בזכרון הראשי, התהיליך צריך ליבא אותו מהדיסק מה שגורם לו לוותר על ה-*CPU* (כי אין ל-*CPU* מה לעבד) ובנוסף מדובר בפעולה איטית.

כדי לדעת אם דף נמצא בזכרון האמיתי או על הדיסק, נסיף לטלבת *Page Table* עמודה שתכיל בית 0 או 1.

אם 0 - העמוד לא נמצא בזכרון הראשי, ולכן הערך בשורה

של המסגרת לא רלוונטי.

(יכול להיות שיש שם ערך יSEN אבל הוא חסר משמעות)

מכיוון שהוא Cache נדרש לענות על 3 השאלות הבאות:

- איך נמשג גישה מהירה? תשובה: משתמש בתרגום כתובות ונייעזר ב-*TLB*.
- אילו רכיבים צריכים להיות ב-*Cache*?
- אילו רכיבים צריך לפנות מה-*Cache*?

?Cache - מה נשמר ב- Demand Paging

אם חיפשנו דף מסוים ב-*Cache* והוא לא שם, מערכת הפעלה תטען אותו לזכרון. פעולה הטעינה נקראת *Page Fault* והוא פעולה יקרה - אם נדרש לעשות אותה הרבה פעמים גודיל את התקורתה. לכן, מערכת הפעלה מנסה לנחש מראש לאילו דפים נדרש לגשת בעtid (בניחס מושכל), והוא מעלה אותם לזכרון. אם הניחוש שלו היה טוב, נחסך הרבה זמן. אם לא, נדרש לעשות *Page Fault*.

Page Fault הוא סוג של *trap* הקשור כשהוראה מבקשת עמוד בזיכרון שלא נמצא. במקרה זה התהיליך חייב לוותר על ה-*CPU* בזמן שהעמוד נתען לזכרון (העמוד יכול נתען ולא רק כתובות ספציפיות) וכן התהיליך עובר מצב *running* למצב *waiting*. הדיסק מעתיק את תוכן העמוד לזכרון באמצעות DMA. אחרי שהעמוד הרצוי עולה לזכרון וה-*scheduler* נותן לתהיליך את השליטה ב-*CPU*, צריך לשנות את הבית ל-1, ולבצע את הפקודה הרצiosa. (בשאיפית 42 בסיכון הרצאה 9 יש הדמיה מפורטת לתהיליך)

אילו רכיבים נפנה מהזיכרון?

נستخدم באלגוריתמים כמו *NRU*, *FIFO*, *LRU* ושאר האלגוריתמים שלמדנו עבור *cache* רגיל. בנוסף, יש אלגוריתמים נוספים שייחודיים ל-*Paging* ו-*Working Set*: *WSClock*.

המבחן של *Page Fault*

ונסה לחשב את המבחן של *Page Fault* כדי להבין אם הפתרון של דפים הוא פתרון טוב. נגידו: *Page Fault* – הזמן שלוקח לגשת לזיכרון בפועל, כש-ז' או ההסתברות ל-*Page Fault* – כמה האטנו את המערכת כתוצאה מכך שלא שמרנו הכל בזיכרון הפיזי.

$$\text{Effective Access Time} = p \cdot (\text{Page Fault Time}) + (1 - p) \cdot (\text{Memory Access Time})$$

$$\text{Slowdown} = \frac{\text{Effective Access Time}}{\text{Memory Access Time}}$$

עבור הנתונים הבאים: $\text{Page Fault Time} = 25 \text{ ms}$, $\text{Access Time} = 100 \text{ ns}$
נקבל כי:

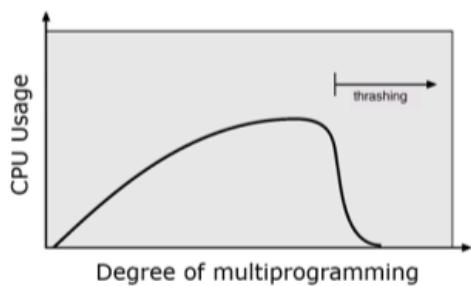
$$p = 0.001 \Rightarrow \text{slowdown} = 250$$

$$p = 0.0000004 \Rightarrow \text{slowdown} = 1.1$$

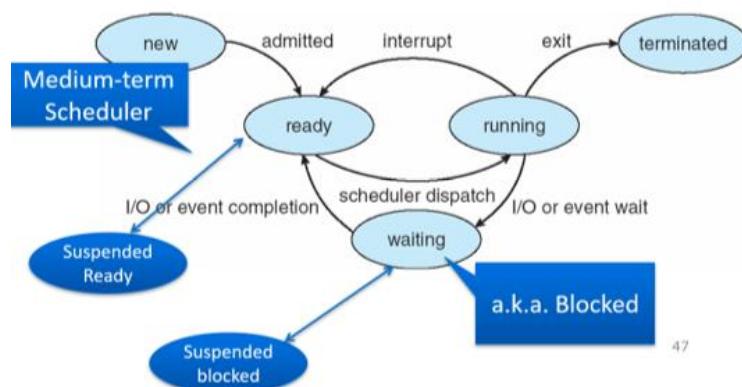
בבחירה נכונה של *Replacement Algorithem* נצליח להגיע ל- $p = 0.0000004$.

Thrashing

נזכיר ב-*CPU Usage* – הזמן שבו ה-*CPU* מבצע *Instructions*. ככל שיש יותר תהליכי ש्रצים במקביל, ה-*CPU Usage* עולה גם הוא. אבל נשים לב לתופעה מעניינת בשם *Thrashing* – שבתתהליכיים עוסקים בחילוף דפים מהדיקן לזכרון, דורסים את הזיכרון אחד לשני במקום לבצע הוראות ובכך הם גורמים לצניחה ב-*CPU Usage*. בגרף הבא נראה שככל שיש יותר תהליכי ש्रצים, ה-



CPU Usage עולה, אבל בנקודת מסוימת הוא צונח משמעותית. *Thrashing* קורה כאשר הגודל של הלוקאלים יותר גדול מהגודל של כל הזיכרון (לדוגמה השילוב של כל התהליכים שעובדים במקביל חורגת מהקיבולת של הזיכרון). הפתרון הוא להגביל את מספר התהליכיים שרצים במקביל, וכך אשר מסpter התהליכים שרצים במקביל חורג, נעצור חלק מהם, ונחליף בין התהליכים הרצים בין ה-*Suspended* בקצב יותר איטי. ככלומר ניתן ליצור תהליכי חדשים אבל לא להריץ אותם במקביל. נוכל לראות זאת בדיאגרמת המצבים הבאה:



תרגול 8

לקריאה המבחן, יש מספר דברים בסיסיים שצורך לשנות בהם ברמה גבוהה:

$$1 \text{ Byte} = 8 \text{ bits}, \quad 1 \text{ KB} = 2^{10} \text{ bytes}, \quad 1 \text{ MB} = 2^{20} \text{ bytes}, \quad 1 \text{ GB} = 2^{30} \text{ bytes}$$

чисובים בסיסיים:

$$\frac{4\text{GB}}{8\text{KB}} = \frac{2^{32}}{2^{13}} = 2^{32-13} = 2^{19}$$

כמה מספרים ניתן לייצר בעזרת 8 ביטים? 2^8 מספרים (כל המספרים בין 0 ל- $2^8 - 1$)
מהו הערך הדצימלי של 1101? $1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 = 13$

Memory Management

Virtual and Physical address spaces

תהליכיים לא יכולים לדעת את כתובות הזיכרון איתן הם יעבדו בזמן קומPILEציה. מקטע זיכרון ממופים עבורם בזמן ריצה ולכן הם צריכים לעבוד עם כתובות וירטואליות. למדנו שפועלות התרגומים מתבצעת בעזרת מערכת הפעלה וה-MMU. כדי לבצע את פעולה התרגום אנחנו מחלקים את התהילר לSEGMENTים, מקטע זיכרון כגון *code, heap, stack* וכו' וכל SEGMENT מקבל מקטע זיכרון רציף בזיכרון האמתי. בהינתן כתובות וירטואליות *X* בסegment *Y*, נטרר:

- קיבל את כתובות הבסיס של הסegment *Y* בזיכרון הפיזי (*Y_PHYSICAL*)
- לבדוק שגודלו של הסegment גדול מ-*X* (אחרת לזרוק שגיאת segmentation)
- להיכנס לכתובת הפיזית *X + Y_PHYSICAL*.

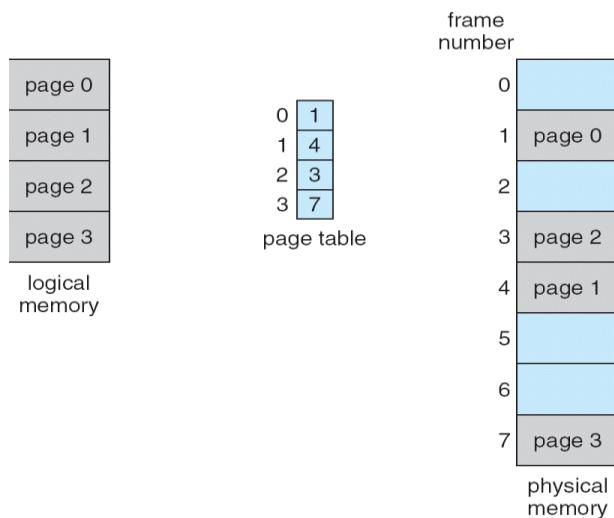
התהילר מתבצע בעזרת כתובות וירטואליות של *n* ביטים. הביטים הראשונים מייצגים את מספר הסegment, ואחר הביטים מייצגים את *offset* בתוכו. בנוסף יש Segment Table שתפקידיה למפות עבור כל סegment את כתובות הבסיס שלו + הגודל.

External Fragmentation

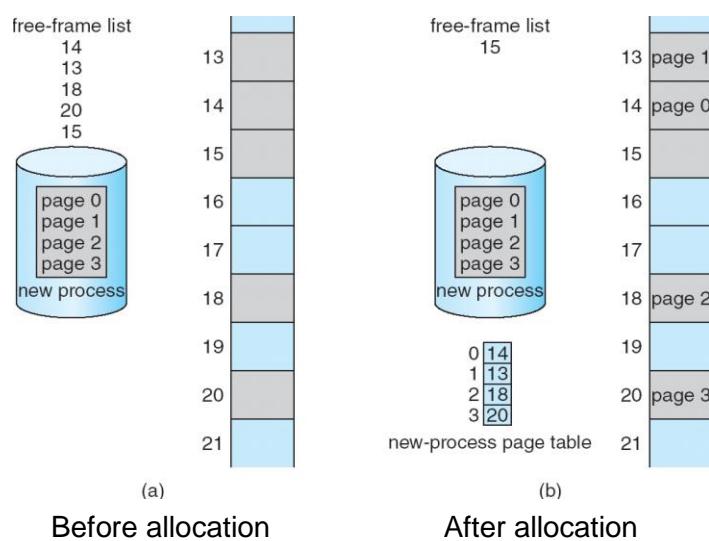
בהרצתה תיארנו את בעיית *the-hole* *External Fragmentation* וראינו שהפתרון מבוצע בעזרת דפים.

- מרחב הכתובות הלוגי של התהילר מסויים יכול להיות לא רציף.
- הזיכרון הפיזי מחולק לבLOCKים בגודל קבוע הנקראים מסגרות (הגודל הוא חזקה כלשה של 2).
הזיכרון הלוגי של התהילר מחולק גם לבLOCKים בגודל קבוע הנקראים דפים (pages).
- יש צורך במקבב אחר כל המסגרות הפנויות.
- כדי להפעיל תוכנית עם *n* עמודים, צריך למצוא *n* מסגרות פנויות ולטעון אליהן את התוכנית.
- צריך להגדיר Page Table ששומרת את המיפוי בין דפים למסגרות כדי לתרגם כתובות לוגיות.
- במקרה זה אין את בעיית *the-hole* *External Fragmentation* אבל יכולה להיות בעית *Internal Fragmentation* (הוגדר בהרצאה).

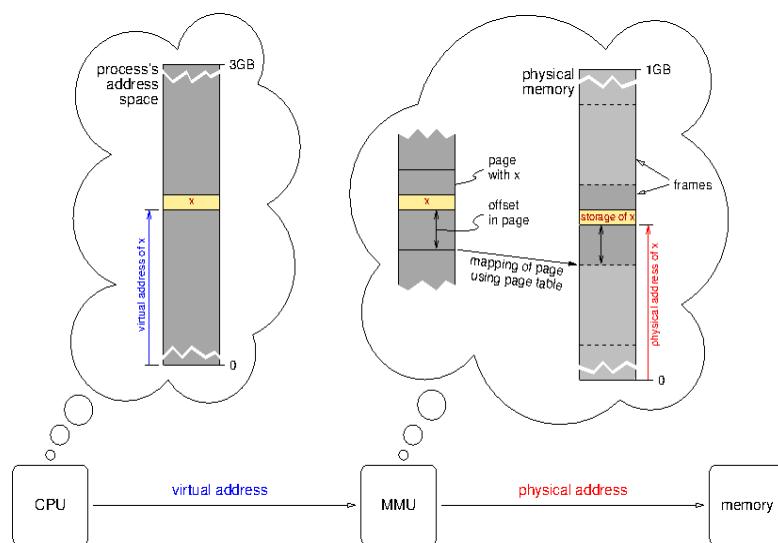
דוגמה לשימוש בדפים ומסגרות:



דוגמה לתחילה הקצתת מסגרות:



תחילה תרגום כתובות ע"י ה-MMU- עבור ה-CPU



Address Translation Scheme

כתובת SHA-*CPU* מעבד מחולקת ל-2 חלקים:

- אינדקס ב-*Page Table*, המכיל את כתובת הבסיס של כל דף בזיכרון האמיתי.

- משלב עם כתובת הבסיס כדי להגדיר את הכתובת בזיכרון האמיתי.

עבור מרחב כתובות לוגי בגודל 2^m מילימ ודף בגודל 2^n מילימ:

<i>page number</i>	<i>page offset</i>
<i>p</i>	<i>d</i>
$m - n$	n

בדומה ל-*Segment Table*, גם בשבייל ה-*Page Table* נוצרה לשמר 2 רגיסטרים נוספים:

- *Page Table-Base register (PTBR)* - מצביע ל-

- *Page Table-length register (PRLR)* - גודל ה-

מידע נוסף שנשמר ב-*Page Table*

- *Valid bit* – בית המסמך האם הדף הנוכחי מופה למסגרת בזיכרון הראשי או לא.
- *Modified* (נקרא גם *Dirty bit*) – מסמן האם הדף עבר שינוי מאז שהוא נטען לזיכרון הראשי.
- אם כן, נדרש לעדכן את הדיסק עם אותו שינוי, ואם לא אין צורך לכתוב אותו מחדש סתם.
- *Used bit* – יכול להיות יותר מביט אחד, שומר מתי ניגשנו לדף בפעם الأخيرة (*timestamp*) (משמש למשל את אלגוריתם השעון, אלגוריתם *LRU* – *Page Replacement*)
- *(read-only, read-write)* – בית שמצוין הרשות (*Access Permissions*)

Page Replacement

אם ניגשנו לדף עם *Valid bit* כבוי, הדף לא מופה לשום מסגרת בזיכרון הראשי ולכן במקרה זהה יש חריגה בשם *Page fault* ומערכת הפעלה תחפש מסגרת פנوية להעביר אליה את הדף:

- אם יש צאת – הדף יטען מהדיסק למסגרת.
- אם אין צאת, מערכת הפעלה תבחר דף מהטבלה (*victim*), תעדכן אותו בדיסק אם יש צורך,

תעביר את הדף החדש במקום ה-*victim* ותעדכן את ה-*Page Table* בהתאם.

כדי לבחור את ה-*victim*, מערכת הפעלה משתמשת באלגוריתם *Page Replacement* (באוטו אוף כמו ה-*Segment Replacement*). למשל אלגוריתם השעון (בעזרת ה-*Used bit*).

Structure of the Page Table

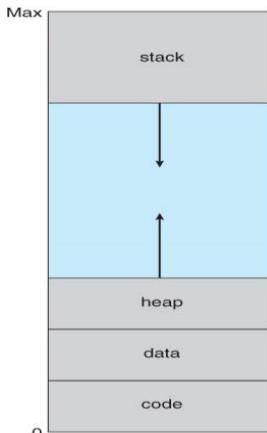
כל שמספר התהיליכים גדול, הזיכרון המוקצה ל-*Page Tables* גדול בהתאם.

נראות 2 דרכי לפטור את הבועה הزادת ולהקטין את התקורה:

Hierarchical Page Tables -

Inverted Page Table -

על השני נרחב בתרגול הבא.



(נקרא גם Hierarchical Page Tables)

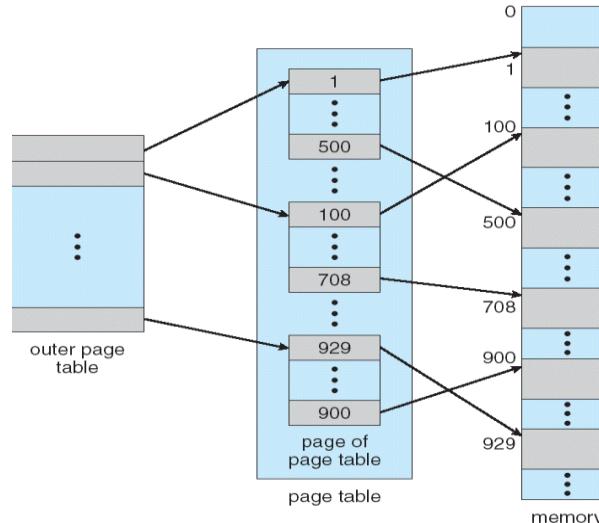
התמונה משמאל מתארת באופן מופשט את הסגמנטים שתהיליך מוקצה לעצמו בזמן ריצה. אמרתו שההיליך רץ בהנחה שכל הזיכרון עומד לרשותו, אך אם נמפה בעזרת *Page Tables* את כל הזיכרון הוירטואלי שלו אנחנו מוקצה ונתחזק טבלאות מיותרות למרחבי כתובות שלא בשימוש. לרוב, ההיליך לא ישתמש בכתובות זיכרון שנמצאות בין-ה-*Stack*-ל-*Heap* (מה שמסומן בכחול), וכך אין צורך לתחזק *Page Tables* למקטע זהה. לעומת זאת, הפתרון הוא להקצות טבלאות רק לפי דרישת, ולא להקצות אותן מראש לכל הזיכרון הוירטואלי.

Two – Level Page Table היא

כדי למנוע הקוצאות של *Page Tables* מיוחרים, נוכל להשתמש בטבלה ראשית שתשמור מצביעים ל-*Page Tables* הפעילים של הטעילים. כל *Page Table* נמצאת גם היא ב-*Frame* (אחד או יותר) בזיכרון הראשי, וכך כל תא בטבלה החדש בעצם שומר את מספר המסגרת שבה נמצאת ה-*Page Table*, וכך כל פעם שהתוכנית תקיצה לעצמה מקטעים נוספים, היא תקיצה טבלאות חדשות והטבלה הרלוונטית. בכל פעם שתהיליך תקיצה לעצמה מקטעים נוספים, היא תקיצה טבלה נוספת וטהילה הריאטיבית תटעדכן בהתאם. בקרה זו אנחנו אומנים מייצרים טבלה נוספת, אבל לרוב מקצים פחות טבלאות ברמה השנייה ומקטינים את התקורה משמעותית.

הערה: באותו אופן יכולות להיות רמות נוספות של *Page Tables* בנוסף ל-2 רמות.

בדוגמה הבאה יש 2 רמות:



Two – Level Paging Example

כתובת לוגית (על מחשב עם 32 ביט ודף בגודל 1KB) מחולקת ל:

- Page Offset - 10 ביטים.

- (32 – 10 = 22 – Page Number - 10 ביטים.)

מכיוון שה-Page Table גם כן שומר בדף, מספר ה-Page Table מחולק ל:

- Page Offset - 10 ביטים.

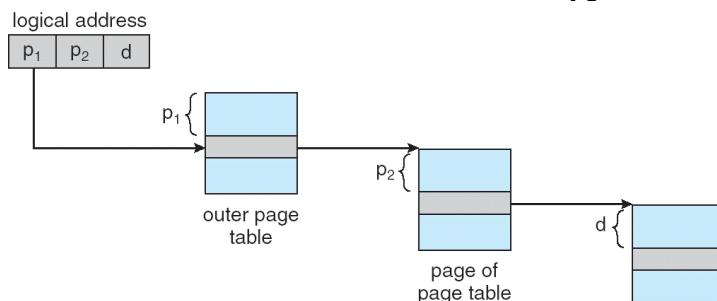
- (22 – 10 = 12 – Page Number - 10 ביטים)

ולכן הכתובת הלוגית מורכבת באופן הבא:

<i>page number</i>	<i>page offset</i>	
p_1	p_2	d
12 bits	10 bits	10 bits

כאשר p_1 מצין את האינדקס בטבלה החיצונית (ברמה הראשונה),

ו- p_2 את האינדקס בטבלה ש- p_1 מצביע עליה.



:Ex4

בתרגיל 4 נממש ממשק לזכרון וירטואלי לפי *Hierarchical Page Tables* עם מספר שרירותי של רמות בעזרת זיכרון פיזי מודומה. נממש את 2 הפונקציות הבאות:

- *VMread(virtualAddress, *value)* – קוראת מילים מהזיכרון הוירטואלי.

לטור **value*. מחזירה 1 עבור הצלחה ו-0 עבור כישלון.

- *VMwrite(virtualAddress, value)* – כתובת את המילה *value* לזיכרון הוירטואלי.

מחזירה 1 עבור הצלחה ו-0 עבור כישלון.

שאלות חזרה על תרגום כתובות

Question A - עברו מחשב עם זיכרון (אמיתי) בגודל 8GB, גודל דף של 8KB וכל שורה ב-*Page Table* היא בגודל 4 bytes. כמה רמות של *Page Tables* צריך כדי למפות מרחב כתובות וירטואליות באורך 46 ביטים, אם ניתן לאחסן כל *Page Table* בדף אחד?

תשובה: דבר ראשון נרשום את הנתונים באופן הבא:

$$RAM: 8GB = 2^{33} \text{ bytes}, \text{ Page: } 8KB = 2^{13} \text{ bytes}$$

מכך שגודלו *Page* הוא בגודל *Frame*, גם גודל כל *Frame* הוא 2^{13} .

$$\text{לכן מספר ה-RAM-frames} = \frac{2^{33}}{2^{13}} = 2^{20}.$$

מכך שכל שורה ב-*Page Table* היא בגודל 2^2 bytes וכל טבלה היא בגודל 2^{13} bytes יש $2,048 = 2^{11} = \frac{2^{13}}{2^2}$ שורות בכל טבלה, וכך כל טבלה יכולה להכיל 2,048 דפים.

כל דף מכיל 2^{13} בתים ולכן אם רמה אחת אנחנו יכולים למפות $2^{24} = 2^{11} \cdot 2^{13}$ בתים.

(2^{11} טבלאות שכל אחת ממפה 2^{13} בתים) זה לא מספיק, אנחנו צריכים למפות 2^{46} בתים שכן נctruck רמה נוספת. עם רמה נוספת נוכל למפות $2^{35} = 2^{11} \cdot 2^{11} \cdot 2^{13}$ בתים (2^{11} טבלאות שכל אחת מחייבת 2^{11} טבלאות שכל אחת ממפה 2^{13} בתים, סה"כ 2^{35} בתים). זה גם לא מספיק אך עם רמה שלישית קיבל: $2^{46} = 2^{11} \cdot 2^{11} \cdot 2^{13}$ כפי שרצינו. לכן התשובה היא 3.

Question B - רשום את כל השדות של שורה בודדת בטבלה, כולל ב-*Page Table Entry* (PTE)

תשובה: נתון שגודלו PTE הוא 4 בתים (32 ביטים), שתפקידם למפות כתובות וירטואליות לכתובות

אמיתית. לכן כל שורה צריכה להכיל מצביע ל-*Frame* הרלוונטי בזיכרון האמתי, וביתים נוספים כגון

$$\frac{\text{RAMSize}}{\text{FrameSize}} = \frac{2^{33}}{2^{13}} = 2^{20}$$

מסגרות בזיכרון הראשי, שכן נדרש 20 ביטים שימפו אותן. זה משאיר לנו 12 ביטים וחופשיים לשאר המידע זהה מספיק.

Question C – ללא שימוש ב-*Cache*, כמה פעולות זיכרון נדרשות כדי לקרוא מילה בודדת של 32 ביטים?

תשובה: ללא חומרה מיוחדת, נדרשות 3 פעולות חיפוש ב-*Page Tables* אחר הכתובת האמיתית בזיכרון

(כי יש לנו 3 רמות), ולאחר מכן עוד פעולה אחת כדי לגשת לזיכרון האמתי.

Question D – כמה זיכרון אמתי נדרש עבור תהליך עם שלושה עמודים של זיכרון וירטואלי?

(לדוגמה, *code*, *data*, *stack* אחד-אחד?).

תשובה: דרישים שישנה עמודים (8KB כל אחד), לכן סה"כ נדרש 48KB של זיכרון אמתי.

עמוד אחד נדרש לטבלה בrama הראשונה, עמוד אחד לטבלה בrama השנייה, עמוד אחד לשלייתית ועוד 3 עמודים למקטעים עצם. נשים לב שלא נדרשנו לטבלאות נוספות כי גישת ה-*Hierarchical Page Tables*.

התבונן בקטע קוד הבא שמאפס מערך של מספרים מסוג integer (כל integer הינו בגודל 4 בתים).

```
for (int i=0; i<2^29; i++) {
    numbers[i] = 0;
}
```

הנחות:

- למחשב זיכרון פיזי בגודל 2^{32} בתים המוחולק לمسגרות בגודל 2^{12} בתים. שימושו לבן כל מסגרת מכילה עד 2^{10} איברים מסוג integer.
- איברי המערך מוקצים בצורה רציפה בתוך כל דף ומתחלים מתחילה הדף הראשון.
- הקוד כולם נכנס לדף אחד.
- שיטת החלפת דפים הינה paging - demand. נמצא ברגיסטר.
- בהתחלת הזיכרון מכיל רק את טבלאות הדפים והן נשארות תמיד בזיכרון (התעלמו מ-ms-page faults על טבלאות הדפים).

א. (6 נק') נניח שהחלפת דפים מתבצעת במדיניות LRU. כמה page faults יהיו במהלך האלגוריתם אם מוקצים לתהיל' 2^{12} מסגרות בזיכרון (לא כולל טבלאות דפים)? נמק.

מערך `numbers` הוא בגודל 2^{29} איברים מסוג integers כלומר בגודל 2^{19} דפים. בתחילת הריצת התוכנית אף דף אינו בזיכרון ולכן יש להביא את כל הדפים הנ"ל לזכרון. כמו כן, צריך להביא לזכרון את דף ה-`code` (ע"פ הנתון הקוד הוא בדף בודד). הבאת כל דף גוררת `page fault` וכן סה"כ מספר ה-`page-faults` יהיה $2^{19}+1$.

מדיניות ה-LRU משנה רק לגבי הקוד (ברגע הבאת דף חדש מהמערך, לא השתמש יותר במלוא כל הדפים הקודמים). מכיוון שאנו משתמשים בקוד כל הזמן הוא ישאר בזיכרון כל הזמן ולא יגרור page faults נוספים.

ב. (6 נק') נניח שתחלפת דפים מתבצעת במדיניות Second Chance FIFO. כמה page faults יהיו במהלך האלגוריתם אם מוקצים לתהיל' 2^{12} מסגרות בזיכרון (לא כולל טבלאות דפים)? נמק.

כל ה-`page fault` מסעיף א' יקרו גם במקרה זה. יתרה מזאת, לאחר האיטרציה ה- $1-2^{12}-2^{12}$ יתמלא הזיכרון המוקצה לתהיל' באופן הבא: דף אחד לקוד, ו- $1-2^{12}$ דפים למערך. מכיוון שלא פנינו או ניסינו לפנות דפים עד כה, bit reference של כל הדפים הוא 1. וכך האלגוריתם יתן הגדמות שנייה לכל הדפים. ככלומר, למעשה, יאפס את bit-reference של כל הדפים, ולבסוף יוציא את הדף הראשון (הקוד) מהתור. מכיוון שצריך לשימוש בקוד, יתווסף עוד `page fault` להכנסתה מחודשת של הקוד.

תופעה זו תקרה כל $2^{12}-1$ איטרציות וכן סה"כ יתרוספו 2^7 . $\lceil 2^{19}/(2^{12}-1) \rceil = 2^{19}+2^7+1$ page faults, והוא סה"כ מספר ה-`page-faults`

נתון מחשב עם מבנה הזיכרון הבא:

- זיכרון לוגי לכל תהליך 2^{32} בתים.
- זיכרון פיזי בגודל 2^{21} בתים, המוחולק למסגרות בגודל 2^2 בתים כל אחת.
- L1 cache בגודל 2^6 בלוקים, גודל כל בלוק הוא 2^6 בתים, ממושך 4-way set associative.
- טבלת דפים היררכית בעומק 2, כאשר הטבלה הראשונה בהיררכיה (בשורש העץ) היא בדיק בגודל דף יחיד. כל שורה בטבלאות היא באורך 4 בתים.

ג. (6 נק') מהו הגודל (בדפים) של טבלת הדפים של כל תהליך?
כמה דפים בטבלת הדפים מכילים רק רשותות לא-חוקיות
 $(bit=0)$? התיחסו למספר המקסימלי והמינימלי של דפים
כאלן.

נתון כי גודל טבלת הדפים הראשונה הוא דף אחד, כלומר $2^{12} = 2^{10} / 4 = 2^{10}$ בתים או 2^{12} רשומות. כאמור מספר הטבלאות ברמה השנייה הוא 2^{10} . גודל הזיכרון הלוגי הוא $2^{32} = 2^{20} / 2^{12}$ דפים, ולכן מספר הרשותות הנדרשות בכל טבלה ברמה השנייה הוא $2^{20} / 2^{10} = 2^{10}$ רשומות. וכך נסכל טבלה ברמה השנייה היא בגודל דף אחד. ולכן, גודל טבלת הדפים הוא $1 + 2^{10}$.

יתכן ואף דף של התהיליך לא הובא לזכרון הפיזי, וכך טבלה ברמה השנייה גם לא הובאה לזכרון, ולכן לכל הרשותות יש $valid = 0$, כלומר מספר הדפים המקסימלי המכילים רשומות לא חוקיות הוא $1 + 2^{10}$ (כל הדפים). התקבלו גם תשיבות המתיחסות לכך שמספר דפים בודדים תמיד בזיכרון (למשל של הקוד). כל הדפים הללו יכולים להיות מוצבעים מטבלה אחת ברמה השנייה, ולכן במקרה זה מספר הדפים המינימלי המכילים רשומות לא חוקיות הוא $1 - 2^{10}$ (כולם מלבד הרמה הראשונה וטבלה אחת ברמה השנייה).

המספר המינימלי של דפים המכילים רשומות לא חוקיות תלוי בגודל הזיכרון הפיזי. סה"כ יש 2^9 מסגרות בזיכרון הפיזי, ולכן לכל היוטר 2^9 רשומות עם $valid = 1$.

המספר המינימלי מתקיים כשל רשותה כזו נמצאת בטבלה אחרת ברמה השנייה וסה"כ המספר המינימלי של דפים המכילים רשומות לא חוקיות בלבד הוא $2^{10} - 2^9 = 2^9$.

הערה: התשובה הتعلמה מהמקום שתופסות טבלאות הדפים בזיכרון הפיזי.
במקרה ומתייחסים למקום זה התשובה היא 2^8 .

הנתונים הבאים מתייחסים לסעיפים ד' וה':
 לשם הפשטות, נניח כי התוכן של איבר באינדקס 0 בכל טבלת הדפים מתkowski מהיפור של הביטים שלו והוספה ביטים 0 משמאלו במידת הצורך.
 למשל, זיהוי טבלת הדפים בעלת אינדקסים באורך 2 כשתוכן כל איבר הוא 10 ביטים

אינדקס	תוכן
00	00000000011
01	00000000010
10	00000000001
11	00000000000

אנחנו רוצים לגשת כתובות לוגית 1111111111111111000 (20 אפסים ואחריהם 12 אחדים).

ד. (7 נק') צין באילו כתובות (אחת או יותר) ב-L1 cache יתכן וימצא התוכן של הכתובת הלוגית.

גודל כל בלוק הוא 2^6 , גודל כל set הוא $4=2^2$, ולכן מספר ה-set האפשריים הוא $4^4=2^6$. ככלומר כל כתובה מכילה 6 ביטים ל-offset, 2 ביטים ל-way ו-4 ביטים ל-.index

התרגום של כתובות ל- cache נעשה מה-LSB של הכתובת הפיזית. מכיוון שגודל כל דף בזיכרון הפיזי הוא 2^{12} , אזי 12 הביטים ה-LSB הם offset (בזיכרון הפיזי) וליקן ערכם זהה בכתובת הפיזית ובכתובת הלוגית. (COLUMN 1 במקרה זה).

ומכאן ארבעת הכתובות האפשריות הן:
111111111111, 1111101111111, 11111001111111

ה. (7 נק') צין באילו כתובות (אחת או יותר) בזיכרון הפיזי יתכן וימצא התוכן של הכתובת הלוגית.

על פי סעיף ג', 10 הביטים הראשונים מוקצים לטבלה ברמה הראשונה ו-10 הביטים אח"כ מוקצים לטבלה ברמה השנייה. 12 הביטים האחרונים מוקצים כאמור ל-offset בתוך הדף.

נביט בטבלה הראשונה. התוכן של אינדקס 0000000000 הוא 1111111111. אך הטבלה ברמה השנייה הרלוונטי היא במסגרת 1111111111. נביט באינדקס 0000000000 של טבלה זו והתוכן יהיה 111111111111.0000000000. וליקן הכתובת הפיזית (שהינה באורך 32 ביט) היא 0.00000000001111111111111111 (22 אחדים מימין, 10 אפסים משמאלי).

התקבלו גם תשובות המניחות שאורך הכתובת הוא 21 (גודל הזכרון הפיזי). במקרה זה התהילך לא משתנה מלבד התוכן של אינדקס 0000000000 בטבלה השנייה: 1111111111 (9 פעמים 1) והכתובת הפיזית היא 11111111111111111111 (21 פעמים 1).

שאלת הופ-אף:

בاهינתן מחשב מודרני עם זיכרון של bit 64 (כלומר מרחב הזיכרון הלוגי הוא בגודל $bytes = 2^{64}$), גודל דף / מסגרת - $4KB = 2^{12} bytes = 2^3 bytes = 8 bytes = 2^{12} bit = 64 bit$, מה המספר המינימלי של דףים בטבלת הדפים שכל הרשומות שבhem הם *invalid*?

פתרון:

גודל הזיכרון הלוגי הוא 2^{64} וגודל כל דף הוא 2^{12} لكن מספר הדפים בזיכרון הלוגי הוא $2^{52} = \frac{2^{64}}{2^{12}}$.
 ככלומר בטבלת הדפים, שטוחה لأن כל דף ממופה בזיכרון האמתי,
 צריכה להכיל 2^{52} שורות כדי שתוכל למפות כל דף.
 מכך נסיק שהגודל בטבלת הדפים הוא: $2^{52} = 2^3 \cdot 2^{52}$ שורות שכל אחת בגודל 2^3).
 כדי לשמר את בטבלת הדפים בזיכרון האמתי נוצרה למפות גם אותה לדפים,
 וכן נוצרה 2^{43} דפים מתוך הזיכרון האמתי לטובת בטבלת הדפים.
 גודל הזיכרון האמתי הוא $2^{35} bytes = 2^{23} frame$, וגודל כל *frame* הוא 2^{12} לכן מספר ה-*frames* הוא 2^{23} .
 לסתום: יש 2^{52} דפים, 2^{23} מסגרות ובנוסף בטבלת הדפים היא עצמה בגודל 2^{43} דפים.

כעת נוכל לענות על השאלה:

המספר המינימלי של דפים שכל השורות בהם הם *invalid* שווה למספר כל הדפים פחות המספר המksamימי של דפים שלפחות שורה אחת בהן היא *valid*. אם שורה היא *valid*, סימן שהקצתנו דף למסגרת, ושמרנו את המיקום בשורה הזאת. יש סה"כ 2^{23} מסגרות, אך יש לכל היותר 2^{23} שורות *valid* בכל הטבלה. מספר הדפים המksamימי עם שורת *valid* מתקבל אם נשמר בכל דף רק שורת *valid* אחת. יש סה"כ 2^{43} דפים בטבלת הדפים ולכן המספר המינימלי של דפים שכל השורות בהם הם *invalid* הוא $2^{23} - 2^{43}$ (חיסור ! לא חילוק). שזה כמעט 2^{43} כלומר הרוב המוחלט של הדפים הם דפים שכל השורות בהם הם *invalid*.

הדוגמה הנ"ל מדגימה לנו את הבעייתיות בטבלת דפים גדולה – רובה ריקה ולכן יכולה לתפוס את רוב הזיכרון לחינם. פתרונות אפשריים:

- הרחבות בתרגול קודם. - *Hierarchical Page Tables* -

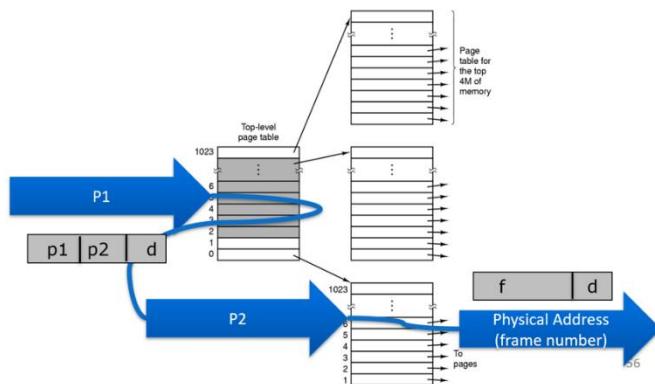
Hashed Page Tables -

Inverted Page Tables -

Hierarchical Page Table

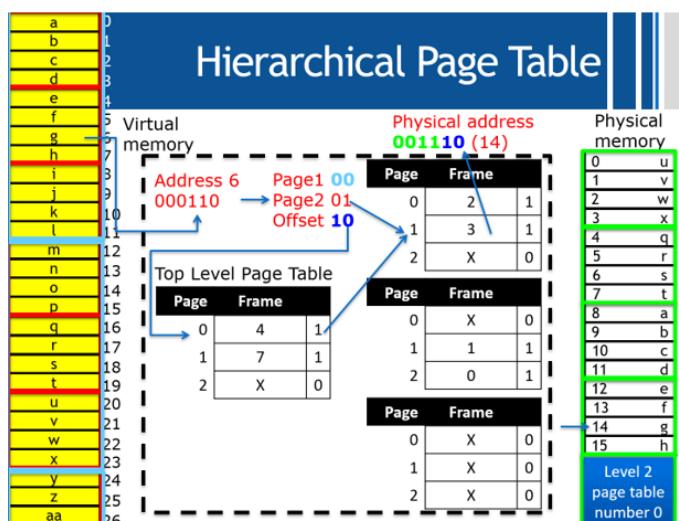
בשיעורים קודמים רأינו שה-MMU מקבל כתובת לוגית (כתובת שמכילה את מספר הדף וה-offset) ומתרגם אותה לכתובת פיזית (כתובת שמכילה את מספר ה-frame ו-offset בתוכו).
 בתוכו) ומתרגם אותה לכתובת פיזית (כתובת שמכילה את מספר ה-page ו-offset בתוכו).
 ב-*hierarchical page table* עם 2 היררכיות, נחלק כל דף לשני חלקים: p_1 ו- p_2 .
 (עבור טבלה עם 3 היררכיות נחלק את $page$ ל- p_1, p_2, p_3 , וכך הלאה)
 במקומ להציג טבלה אחת גדולה למיפוי כל הכתובות, נציג טבלה ראשית (*Top Level Table*)
 וכל שורה בטבלה זו תכיל מספר *frame* שבתוכו יש טבלה נוספת.
 גודל כל טבלה כזו הוא גודל של דף בודד. במקורה כזה, כתובות וירטואליות מורכבות מ:

- $page1/p1$: מספר השורה בטבלה הראשית (*Top Level Table*)
- $page2/p2$: מספר השורה בטבלה המשנית (*Second Level Table*)
- ה-*offset* בדף הרצוי.



דוגמה למיפוי כתובות וירטואליות לכתובת פיזית:

ניעזר בתמונה למטה, ונבדוק מה הכתובת האמיתית של הכתובת הווירטואלית 6. המספר 6 ביצוג בינארי הינו 00 01 10, لكن בטבלה הראשית נלך לתא 0 (00) שמצוין על 4. בטבלו יש טבלה page1 page2 offset
 נוספת, נלך לתא 1 (01) שמצוין על 2-*page*. בתוך 2 *page* נלך לתא ה-2-*frame*.
 בתא זהה מופיע המספר 3 -> 0011, אך הכתובת הפיזית היא frame offset



דוגמה נוספת:

עבור מחשב עם זיכרון של *bits* 32, גודל כל דף הוא *4KB* ולכן *offset* הוא 2^{12} .

יש $2^{20} = \frac{2^{32}}{2^{12}}$ מסגרות בזיכרון הראשי. נסתכל למשל על הכתובת הירטואלית:

(20 ביטים שמצינים את מספר הדף 5,500, ו-12 ביטים נוספים המציינים את *offset* בדף) אנחנו שואפים לחלק את טבלת הדפים לטבלה ראשית וטבלאות משנהות כדי לא להקצת זיכרון מיותר,

לכן נחלק את הייצוג הבינארי של 5,200 ל-2 חלקים: $\underline{\underline{0000000101}} \underline{\underline{0001010000}} \quad \begin{matrix} 5 \\ 80 \end{matrix}$

ונקבל כתובת וירטואלית עם $5 = p_1$, $80 = p_2$ ו- $d = offset = d - p_1 - p_2$. p_1 מציין את *offset* בטבלה המשנית, p_2 מציין את *offset* בטבלה הראשית, d מציין את *offset* של המסגרת בזיכרון המקורי,

ו- d נשאר אותו דבר כי הוא מציין את *offset* של המסגרת בזיכרון המקורי)

נכיח ב��וף:

- עמוד 5,200 שמור במסגרת 1,000.
- הטבלה הראשית (*Top Level Table*) שמורה במסגרת 17.
- הטבלה המשנית (*Second Level Table*) שמורה במסגרת 700.

כעת כדי למצוא את מספר המסגרת בזיכרון המקורי מ透ה כתובת הירטואלית שקיבלנו:

1. נלך למסגרת 17 (שם נמצאת הטבלה הראשית) ונקרה ממנה את שורה 5. נתון שהטבלה

המשנית שמורה במאגרת 700 لكن שורה 5 בטבלה הראשית מכילה את המספר 700.

2. נלך למסגרת 700 (שם שמורה הטבלה המשנית) ונקרה ממנה את שורה 80.

שורה 80 מכילה את הערך 1,000 (שם עמוד 2,500 שמור לפני הנתון)

3. ולכן הכתובת המקורי היא:

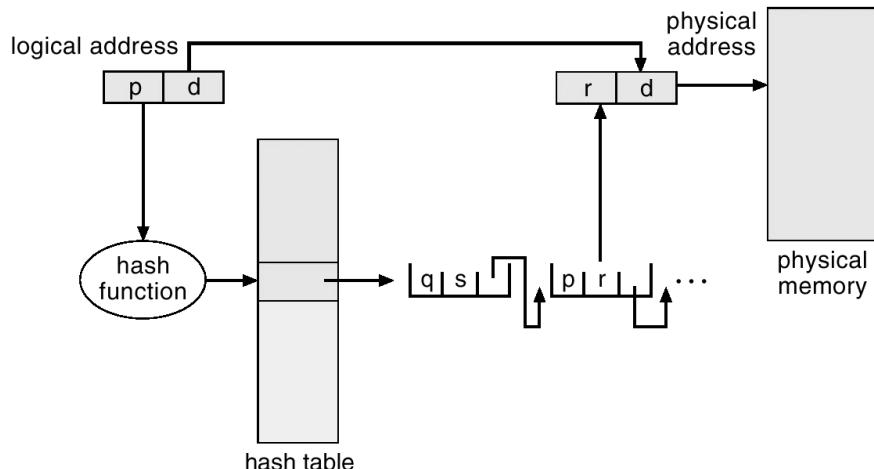
1000	<i>d</i>
20 bits	12 bits

Memory Accesses and Page Fault

ראינו שעבור 2 רמות של טבלאות דפים, כל פקודת גישה לזכרון מתורגמת ל-3 גישות זיכרון:
גישה לטבלה הראשית, גישה לטבלה המשנית וגישה לתא הרצוי. ראיינו גם שאין צורך לשמור טבלאות דפים ריקות (טבלאות *invalid*) בזיכרון או בדיסק הקשיח. אבל באיזה שלב מתבצעת הקצהה של טבלה חדשה? متى טבלה צריכה להיות *valid*? בזמן תרגום כתובת וירטואלית שדורשת גישה לטבלאות *invalid* יקרה - *Page Fault* - גישה לעמוד שלא נמצא על הזיכרון המקורי (ובמקרה זה גם לא בדיסק). בזמן *Page Fault* מערכת הפעלה תזהה שהטבלה הזאת לא קיימת בדיסק, ולכן היא תקצת מסגרת חדשה שכולה אפסים (*Zero Frame*) ותמשיך כרגע.

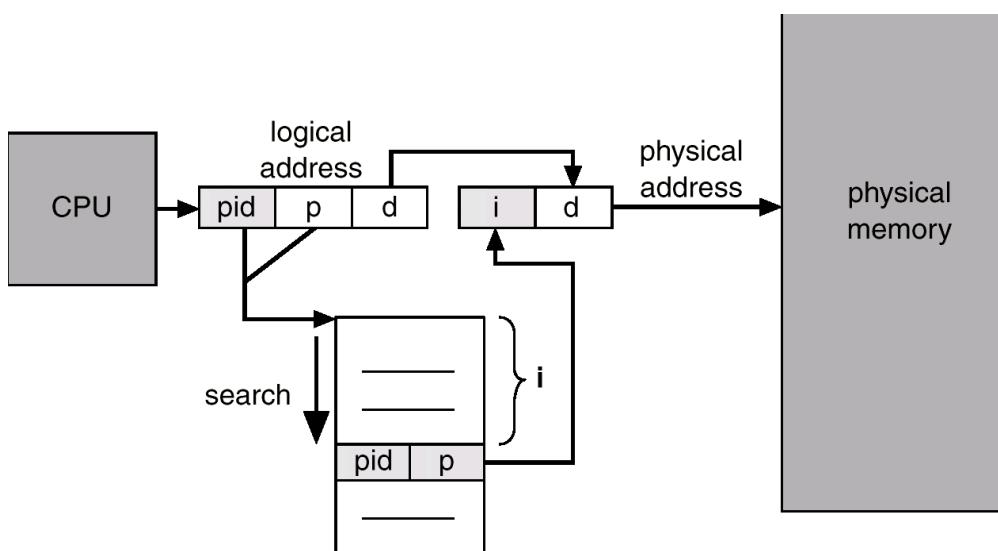
Hash Page Table

דרך נוספת להתמודד עם הבעה שטבלת הדפים גודלה מאד והזיכרון הפיזי קטן יחסית קטן, היא שימוש בטבלת גיבוב. כפי שניתן לראות באIOR, בהינתן כתובות לוגית $p|q$ נפער פונקציית גיבוב על p ונקבל ערך. יתכן שמספר כתובות מופו באותו ערך, لكن נחזיק בכל תא רשימה מקושרת כך של איבר ברשימה מכיל גם את $page$ המקורי וגם את $frame$ המתאים לו. נחפש ברשימה את p ונחזיר את z (הערך ש- q מומפה אליו). במקרים בהם הרשימות ארוכות מאוד, התקורה תגדל (נוצרף לעבר על כל הרשימה עד $Page Fault$ ולכן זו לא דרך נפוצה במיוחד לתרגום כתובות).



Inverted Page Table

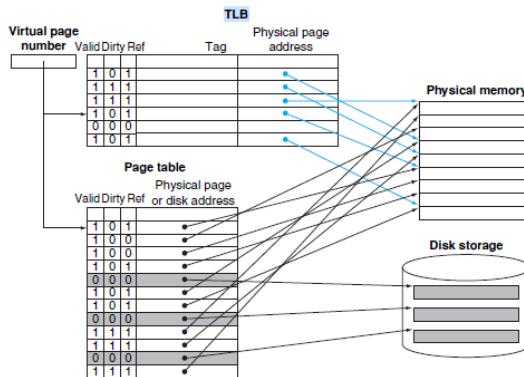
לפי גישה זו לא נשמש כלל בטבלת דפים. נחזיק טבלה אחת שמספר השורות שלה הוא כמספר המסגרות שיש בזיכרון הפיזי, וכל שורה i בטבלה תשמר 2 משתנים: pid של התהיליך שמויה למסגרת ה- i - בזיכרון, ומצביע לעמוד הרלוונטי (ה- pid חשוב כי עד עכšíו לכל תהיליך היה טבלת דפים משלו, עכšíו מדובר על טבלה אחת ששומרת את המידע לכל התהיליכים ביחד). בזמן תרגום כתובות, נורץ לאורך הטבלה ונחפש שורה שמכילה את pid וה- $page$ המתאים לכתובות. כשם נמצא אותה נחזיר את i (מספר השורה בטבלה) שמצוין את מספר המסגרת בזיכרון הפיזי שמכיל את העמוד הרצוי. שימוש זה יעיל רק אם הזיכרון הפיזי מאד קטן.



לסייעם – ניהול זיכרון

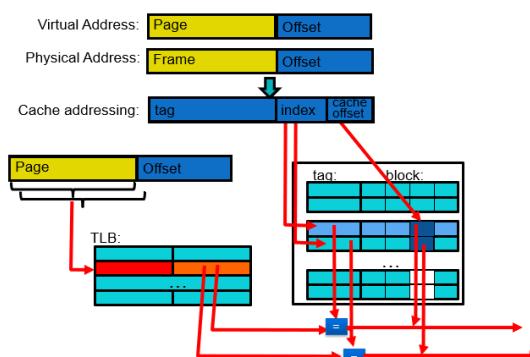
הראינו שיש 2 דרכי ייעולות לתרגום כתובת לוגית לכתובת אמיתית:

- באמצעות TLB - חיפוש העמוד ב-TLB ואם הוא שם חסכנו פעולה חיפוש יקרה. ברוב המקרים העמוד אכן יהיה ב-TLB אך מדי פעם נדרש לעבור לשלבים הבאים בהיררכיית הזיכרון.
- באמצעות תרגום הכתובות: פירוק הכתובות הלוגית ל $offset, m_2, m_1$, וחיפוש בטבלאות השונות. לאחר שנמצא את הכתובות הפיזית הרצוי, נחפש את הכתובות הפיזית ב-Cache. (זיכרון Cache שעבד עם כתובות פיזיות!) אם קיבלנו Cache Hit חסכנו גישה לזיכרון הפיזי. אם קיבלנו Cache Miss, ניגש לזכרון הפיזי וניבא את הבלוק הרלוונטי לתוך ה-Cache. (כפי שראינו בהרצאות קודמות)



Parallel TLB Cache Access

אם בזמן תרגום כתובת לוגית לכתובות פיזית קיבלנו *TLB Miss*, זמן הריצה יהיה גדול בכל מקרה ואין לנו מה לעשות נגד זה. אבל ברוב המקרים יש *TLB Hit* ובמקרה זה נוכל לקצר את זמן הריצה עוד יותר. כפי שכבר למדנו, אחרי תרגום כתובות בעזרת ה-TLB אנחנו ניגשים ל-*Cache*, מפרקם את הכתובות הפיזית ל-*index, tag, offset* ובודקים האם התא הרצוי נמצא ב-*Cache*. אם נדאג שגודל ה-*tag* במחשב יהיה גדול *page*, נוכל לבצע את הבדיקה ב-TLB וב-*Cache* במקביל (ברמת החומרה), כי במקרה הזה ה-*index* אותו אנחנו צריכים לחפש ב-*Cache* הוא חלק מה-*offset* של הכתובות הווירטואליות, וה-*offset* הוא לא משוה שמשתנה בין כתובות וירטואליות לכתובות פיזיות لكن לא נctrיך לחכות שפעולות התרגומים תסתיים לפני חיפוש ב-*Cache*. ניגש ל-*index* הרצוי ול-*offset* בתוכו, ובסוף פעולה התרגום נctrיך רק לוודא שה-*tag* של הבלוק ב-*Cache* שווה לפולט של פעולה התרגום. אם כן, יש *Cache Hit* וחסכנו גישה לזיכרון CI היא בוצעה במקביל לפעולות התרגום. **הערה:** נזכיר שגם ה-*Cache* הוא *associative* *Cache* *a*, נctrיך לבדוק *a* כתובות שונות.



הפתרון	הבעיה
תרגום כתובות: <i>base + bound , page tables</i>	נקודות המבט של תהילך שונה מהמציאות (תהליך חושב שהוא היחיד שרצ' וכל הזיכרון לרשותו)
<i>compaction, paging</i>	<i>External fragmentation</i>
<i>virtual memory + swapping</i>	הזיכרון הלוגי יותר גדול מהזיכרון הפיזי

שאלות מבחנים

נתון מחשב עם מבנה הזיכרון הבא:

- זיכרון לוגי לכל תהילך 2^{32} בתים.
- זיכרון פיזי בגודל 2^{32} בתים, המחולק למסגרות בגודל 2^9 בתים כל אחת. עובד בשיטת demand paging.
- טבלת דפים היררכית בעומק 3 (שורש, צמתים פנימיים, ועלים), כאשר כל הטבלאות בגודל דף אחד בדיק. כל שורה בכל טבלה היא באורך 4 בתים.
- טבלת דפים היררכית בעומק 3 (שורש, צמתים פנימיים, ועלים), כאשר כל הטבלאות מלבד הטבלה בשורש העץ (כלומר כל הטבלאות בצתמים הפנימיים ובעלים) הן בגודל דף אחד בדיק. כל שורה בכל טבלה היא באורך 4 בתים.

בשאלה זו, מילה = 1 בית

א. (5 נק') מהו הגודל (בדפים) של טבלת הדפים ברמה הראשונית בהיררכיה (שורש העץ)?

תשובה:

גודל כל דף: נתון שגודל מסגרת הוא 2^9 בתים לכן גם גודל דף הוא 2^9 .

מספר השורות בכל דף: כל שורה בכל טבלה היא באורך 4 בתים, לכן יש $2^7 = \frac{2^9}{2^2}$ שורות בדף.

מספר המסגרות והדפים הכללי: גודל הזיכרון הפיזי חלקי גודל מסגרת $2^{23} = \frac{2^{32}}{2^9}$.

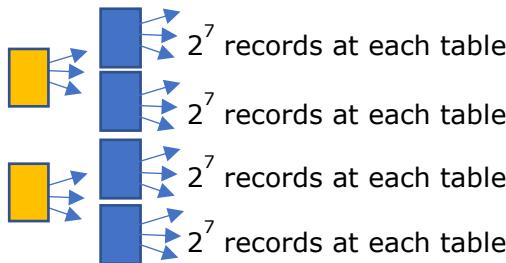
מספר טבלאות הדפים בעליים: אם יש סה"כ 2^{23} דפים ומסגרות, ציריך 2^{23} שורות בכל טבלאות

הדף בעליים שימפו ביניהם. כל דף מכיל 2^7 שורות לכן ציריך $2^{16} = \frac{2^{23}}{2^7}$ טבלאות דפים.

מספר טבלאות הדפים בקדוקודים הפנימיים: $2^9 = \frac{2^{16}}{2^7}$ (כדי למפות 2^{16} טבלאות).

גודל הטבלה הראשית: ציריך 2^9 שורות בטבלה הראשית (כדי למפות את הטבלאות המשניות),

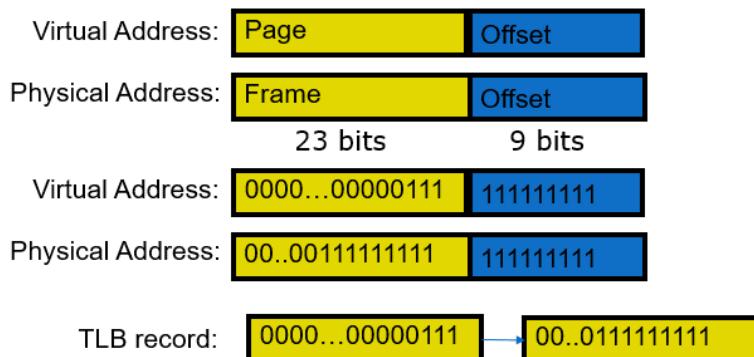
כל עמוד מכיל 2^7 שורות, לכן הטבלה הראשית היא מגודל 4 $= 2^2 = \frac{2^9}{2^7}$



דוע כי כתובת לוגית מקבלת 20 אפסים ואחריהם 12 אחדים) ממופה לכתובת פיזית (12 אפסים ואחריהם 20 אחדים). כמו כן ידוע כי כתובת זאת היא האחרונה שנקראת.

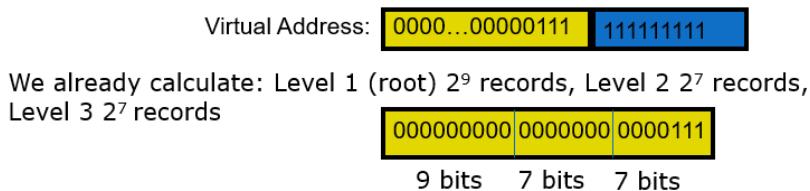
ב. (4 נק') מהנתונים לעיל, צין אילו רשומות קיימות בודדות ב-TLB? צין את כל הפרטים האפשריים על רשומות אלו (גם אם לא ניתן לדעת את כל השdots). נמק את תשובתך.

תשובה:



ג. (5 נק') מהנתונים לעיל, צין אילו רשומות קיימות בודדות בטבלאות הדפים? צין את כל הפרטים האפשריים על רשומות אלו (גם אם לא ניתן לדעת את כל השdots)... התיחס לטבלאות דפים בעומק 1,2,3 וنمוק את תשובתך.

תשובה:



Level 1: Record 000000000 is valid (we don't know what's its value)

Level 2: Record 0000000 is valid in one table (we don't know which table or what's the value)

Level 3: Record 0000111 is valid in one table (we don't know which) and its value is the physical frame number (

00..0011111111

)

ד. (5 נק') צין באילו כתובות (אחת או יותר) ב-L1 cache יתכן וימצא התוכן של הכתובת הלוגית. נמק.

תשובה:

Virtual Address: 0000...00000111 1111111111

Physical Address: 00..0011111111 1111111111

- L1 cache בגודל 2^7 בלוקים, גודל כל בלוק הוא 2^3 בתים, ממומש write back-read allocate-way set associative-2-way.

Physical Address: 00..0011111111 111111111111

Cache of size 2^7 blocks \rightarrow 7 bits addresses

Of which 3 LSB are for offset

1 for way (2-way: either 0 or 1)

3 left for index (from the left)

rest is tag – irrelevant to the question

1110111
Or
1111111

ה. (5 נק') האם ניתן לדעת באילו כתובות ב-L1 cache ימצא התוכן של הכתובת הלוגית 0000000000001111111111111111111100? (12 אפסים, 18 אחדים, ואחריהם 2 אפסים)? אם כן, צין את כל הכתובות האפשריות ונמק. אם לא, הסבר את תשובהך.

תשובה:

Virtual Address: 0000...00000111 1111111111

We can only know if they are in the same block. But Our **virtual address** is very far from this virtual address and therefore the answer is no.

ה. (5 נק') האם ניתן לדעת באילו כתובות ב-L1 cache ימצא התוכן של הכתובת ~~הכתובת הלוגית~~ 0000000000001111111111111111111100? (12 אפסים, 18 אחדים, ואחריהם 2 אפסים)? אם כן, צין את כל הכתובות האפשריות ונמק. אם לא, הסבר את תשובהך.

תשובה:

Our Physical Address: 00..0011111111 111111111111

New Physical Address: 00..0011111111 111111100

Same block, and therefore, the only change is in the offset:

1110100
Or
1111100

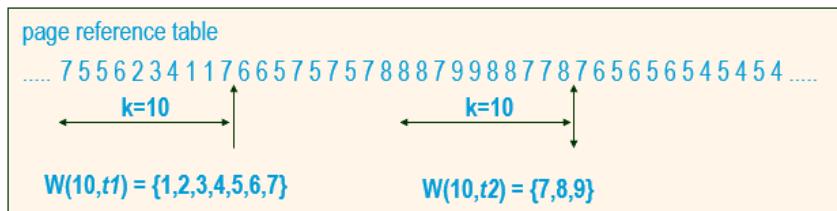
It must be in either ones because our physical address is the last one read, and the cache is read-allocate.

Paging-לייחודיים Replacement Algorithm

כשידרנו על זיכרון וירטואלי, אמרנו שלא כל הדפים של תחילך מסוימים נתונים לזכרון, ויתכן מצב של *Page Fault*. במצב זה אנחנו בוחרים *victim* (דף מהזיכרון) ומחליפים אותו בדף החדש שלא היה שם. בנוסף לאלגוריתמי *Replacement* שכבר רأינו (*NRU, FIFO, LRU*), נוכל להשתמש גם באלאגוריתמים ייחודיים *Paging-לייחודיים*.

Working Set

נסתכל על k הדפים האחרונים שתהילך מסוימים ניגש אליהם ב- k הגישות האחרונות לזכרון. נגיד אוטם *C-Working Set* ובכל פעם שנרצה לפונת דף, נפנה דף שלא נמצא ב- k גודל ה-*Working Set* יכול להיות קטן מ- k (כי לרוב אנחנו ניגשים לאוטם דפים מספר פעמים) ה- k צריך להיבחר בקפידה: k קטן יכול לגרום להרבה *Page Faults* ו- k גדול יכול לגרום לכך שנתפוא קבוצה גדולה מדי של דפים ולא נשאיר מספיק דפים לפינוי.



למשל אם תחילך ביקש דפים לפי הסדר באירוע הנ"ל, $t_1 = k = 10$, אז בפרק הזמן הראשון t_1 , הדפים הרלוונטיים היו $\{1,2,3,4,5,6,7\}$ ובפרק הזמן השני t_2 , הדפים הרלוונטיים היו $\{7,8,9\}$. כדי למש את האלגוריתם הנ"ל נזכירמתי ניגשנו לכל דף בפעם الأخيرة, להעיף את הדף הכי ישן ולהוסיף את הדף החדש שניגשנו אליו בנקודת הזמן הנוכחיית, כלומר:

$$w(k, t) = w(k, t - 1) \setminus \{reference\ t - k\} \cup \{reference\ t\}$$

Working Set, Second Chance FIFO, NRU - WSClock

כל יותר לעבוד עם *Working Set* לפי זמן, כלומר לפי k msec. מנגנון זה מטרתו, כי מערכת הפעלה בכל מקרה משתמש בפסיקות שעון. במקרה זה נסדר את הדפים של התהיליך בצורה מעגלית (כמו שעון) ונשמר לכל אחד מהם זמן גישה אחרון וביט R שמצוין אם ניגשנו לדף זה או לא. אם ניגשנו לדף מסוים, נעדכן את ה- R שלו להיות 1. בכל פרק זמן מוגדר נעדכן את ה- R של כל אחד מהדפים להיות 0, אבל עבר דפים שה- R שלהם היה 1, נשנה את זמן הגישה האחרון להיות זמן הגישה הנוכחי.

בנוסף, נגדיר "מחוג" במעגל שיצביע בכל שלב על אחד הדפים.

אם נדרש לעשות *page fault* נפעיל באופן הבא:

1. נבדוק את הדף שהמצביע מצביע עליו.
2. אם $1 = R$ נעביר את המצביע לדף הבא ונחזור לשלב הקודם.
3. אחרת, $0 = R$: נבדוק אם הדף נמצא ב-*Working Set* על ידי הבדיקה הבאה: $< (current virtual time) - (time of last use) > k$ – אם $R = 0$ וגם הדף נמצא ב-*Working Set*, נקדם את המצביע לדף הבא ונחזור לשלב 1.
4. אם הדף *dirty page* (כלומר הביט *modified* שלו דלוק), סימן שניינו את תוכן שלו ולכן נקדם את המצביע ונחזור ל-1 תוך כדי שנעתיק את תוכן הדף לדיסק.
5. אחרת, נפנה את הדף, ונעדכן את המצביע לדף הבא.
 - אם לא נמצא מועמד מתאים – מפנים את הדף הישן ביותר (אפילו אם הוא *dirty*) אם הדף הישן ביותר הוא *dirty*, נעתייק אותו במקביל לדיסק (תהליך העתקה לדיסק ירוץ במקביל למעבד בעזרת DMA).

פינוי Dirty Page

בזמן פינוי *Dirty Page* (דף שביצעו בו שינויים מאז הפעם האחרונה שהוא נכתב לדיסק) או *Clean Page* (דף שרק קראנו ממנו ולא עשינו בו שינויים), נזכיר שהעתק ששמור בדיסק והעתק ששמור בזיכרון הראשי שונים זה מזה, ולכן נעדכן את הדיסק אם ביצענו שינויים בעותק של הזיכרון הראשי.

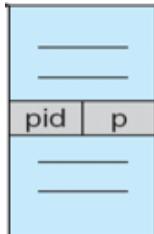
Global VS Local Paging

כשתהיליך p_1 גורם ל-*Page Fault*, האם מערכת הפעלה צריכה לבחור *victim* מתוך אחד הדפים של p_1 או לבחור דף אחר? חלק מערכות הפעלה מושמות *Local Paging* (כלומר יבחרו *victim* מתוך הדפים של p_1), וחלק יממשו *Global Paging*, כלומר דף מבין כל הדפים של כל התהיליכים.

תרגול 9

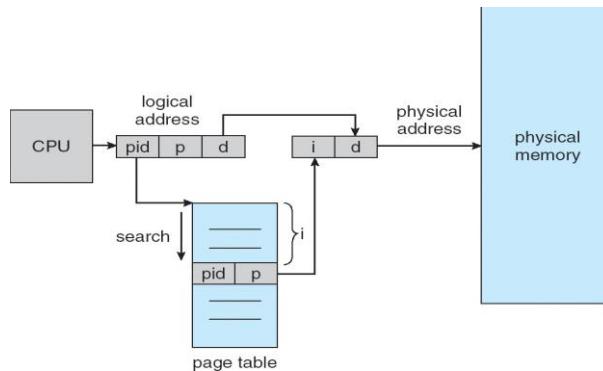
בתרגול קודם הרחכנו על עלייה בזיכרון הזרקן הנדרשת לטבלאות *Inverted Page Tables* הדפים ככל שיש יותר תהליכיים במחשב. עכשיו נרחיב על פתרון נוסף:

Inverted Page Tables



לפי פתרון זה, יש רק טבלה אחת בגודל הזיכרון הפיזי, שמשותפת בין כל התהליכיים. כל שורה i בטבלה מכילה את כתובתה הירטואלית של העמוד שמאוחסן בכתובת הפיזית i , בנוסף למידע על התהיליך שהעמוד שייר ל. למשל בטבלהี้ משמאלי, i הוא מספר העמוד $-pid$ והוא מספר התהיליך שהעמוד שייר ל. פתרון זה מקטין את גודל הזיכרון הדרוש לשמירת טבלאות דפים, אבל מגדיל את זמן החיפוש מכיוון שהוא על כל השורות בטבלה עד שנתקלים בשורה עם המידע הרצוי.

(שורה שמכילה את מספר העמוד הרצוי **וגם** את מספר התהיליך אליו הוא משוייך)

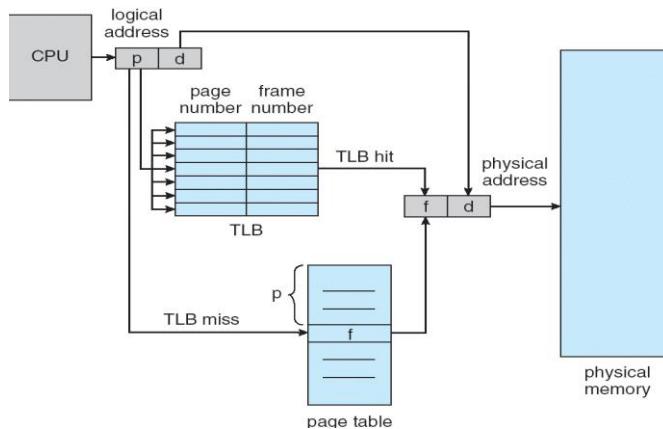


מייפוי כתובות וירטואליות לכתובות פיזיות מוסיף לתקורה בכל גישה לזכרון. אפילו עם רמת *Paging* אחת, צריך 2 גישות לזכרון: 1 לטבלת הדפים ו-1 לתא הרצוי. (לרבות מדובר ביוטר מ-2 גישות זיכרון) לכן, ה-CPU משתמש ב-*Cache* נוסף בשם *TLB* (לרוב *SRAM* - מהיר מאוד) שומר את המיפויים האחרונים שהיו בשימוש (בנהנזה שנצטרך אותם שוב) ובכך חוסף גישות מיותרות לזכרון האיטי.

ה-*Cache* יהיה בדרך כלל **fully associative** עם 64 שורות.

Fully associative \Rightarrow *Index size = 0* \Rightarrow *tag = page number (only tag and offset parts)*
חלוקת מה-*TLBs* שומרים (*ASIDs*) בכל שורה, כלומר מזהים ייחודיים לכל תהיליך, כדי לאפשר הגנה על מרחב הכתובות של כל תהיליך.

במצגת של תרגול 9, שkopiot 12-10 יש סיכום לכל הנושא של *Address Translation* שכדי להביע.



בחלק זהה של התרגול נתמך באופן בו שומרים מידע בדיסק – באמצעות קבצים. קובץ הוא יחידה לוגית של מידע, סוג נתונים מופשט (*ADT*). יש לו שם, תוכן (בדרך כלל רצף של בתים), מידע נוסף (תאריך יצירה, גודל וכו'), ונitin לבצע עלייו פעולות (קריאה, שינוי שם וכו'). קובץ הוא מידע מסווג *volatile* – *non*, כלומר יכול להשתמש בו, לモות, ואז תהילך אחר יכולשוב להשתמש בו. הוא לא תלוי בתהילך מסוים. תהילך יכול להשתמש בו, לモות, ואז תהילך אחר יכולשוב להשתמש בו. כל קובץ מחזיק בנוסף לתוכן שלו גם במידע נוסף, *Metadata*, כגון:

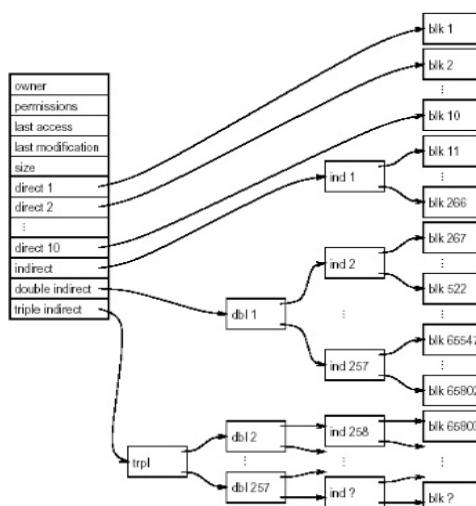
- גודל הקובץ וה-owner שלו
- איזה הרשות יש לו (קריאה, כתיבה, הרצה)
- חותמות זמן שמציניות זמן יצירה / זמן שינוי
- מיקום (האם נמצא על הדיסק או לא? אם כן, איפה?)
- סוג (קובץ בינארי, קובץ טקסט וכו')

inode

inode הוא מבנה נתונים במערכת קבצים מסוג *Unix*, שמאחסן את ה-*metadata* של קובץ (או תיקיה). הוא לא כולל את שם הקובץ כי שם הקובץ מגיע מהפיטה אחרת שאנו חווים – תיקיות, והתפקיד שלහן זה לתת שמות לקבצים שימושיים בתוכם. בנוסף ל-*metadata* שמצוינו לעיל, ה-*inode* מכיל בתוכו גם מצביעים לבЛОקים הרלוונטיים בדיסק ששומרים את תוכן הקובץ:

- מספר מצביעים ישירים לבLOCKים הרלוונטיים (כ-10 מצביעים).
- אם הקובץ לא צריך להשתמש בכלם, חלקם יהיו *Null*.
- "מצביע עקי" אחד (*One single indirect pointer*), שמצוין לבLOCKים נוספים של הקובץ.
- "מצביע עקי כפול" אחד (*One double indirect pointer*), שמצוין לבLOCKים עקיפים של הקובץ.
- "מצביע עקי משולש" אחד (*One triple indirect pointer*), שמצוין לבLOCK שכולו מצביעים עקייפים (double indirect pointers) (single indirect pointers).

מצביע לבLOCK שכולו מצביעים עקייפים כפולים (double indirect pointers)



דוגמה:

- עבור *inodes* של 32 ביט (כל מצביע בגודל 4 בתים) ובЛОקם בגודל 4,096 בתים:
- המצביעים הקיימים (כ-10)אפשרים לשמר קובץ בגודל של עד $48KB$.
 - המצביע העקיף, המכיל 1,024 מצביעים ישרים, אפשר לשמר עוד $4MB$.
 - המצביע עקיף כפול, המכיל 1,024 מצביעים עקיפים, אפשר לשמר עוד $4GB$.
 - המצביע עקיף משולש אפשר לשמר עוד $4TB$.

כלומר לא ניתן להחזיק קובץ בווד ששולן יותר מ- $4TB + 4GB + 4TB = 8TB$,

גם אם יש מספיק מקום פנוי בדיסק!

תיקיות

תיקיות מאוחסנות באוטו אופן כמו קבצים, הם מסוג *directory*, והתוכן שלהם מכיל מיפוי של קבצים ותיקיות ל-*inode*. (ב-*Linux* אפשר לראות את מספרי *inodes* של קבצים באמצעות *i - ls*). כפי שהזכרנו, קבצים לא מאוחסנים את השמות שלהם כחלק מה-*metadata* שלהם, אלא התיקיה מאפשרת לתת שמות לקבצים שנמצאים בתוכה.

Superblock

Superblock הוא בלוק ראש שמאתחל רק כשאנחנו מפרטים את הדיסק. הוא מכיל את גודל הדיסק, רשימה של בלוקים פנויים, רשימה של *inodes* שלא בשימוש ועוד.. בעזרת המידע זהה אפשר להקצות בלוקים חדשים לאחסון מידע.

inodes allocation

גודל *inode* בווד נקבע מראש בזמן פרטוט הדיסק, וכך גם מספר ה-*inodes* הכללי, מה שאומר שמספר ה-*inodes* הוא מוגבל! יכול ל��ות מצב שבויה לנו מספיק מקום פנוי בדיסק, אבל לא יוכל ליצור קובץ חדש כי אין *inode* פנוי להקצות עבורו. ה-*Superblock* שומר רשימה קצרה של *inodes* פנויים, ולא רשימה של כל ה-*inodes* הפנויים. אם תחליך ציריך *inode* חדש, הקרナル יכול להשתמש ברשימה הזאת כדי להקצות לו *inode* חדש. כשי-*inode* משוחרר מהזיכרון, המיקום שלו נשמר ב-*Superblock* רק אם יש מקום פנוי ברשימה. אם רשימת ה-*inodes* הפנויים ריקה, הקרナル יחפש בדיסק *inodes* פנויים ויוסיף אותם לרשימה.

Data Blocks allocation

לכל קובץ קיים יש *inode* שהוקצה לו. כשתחליך כותב נתונים לקובץ, הוא צריך לכתוב את המידע לבLOCKים שה-*inode* מצביע עליהם ואם אין אלה (המקום בBLOCKים הקיימים נגמר, או שעדיין לא הוקטו לו בLOCKים), מערכת הפעלה צריכה להקצות BLOCKים חדשים מהdisk. הבלוק הבא שיוקצת הוא הבלוק הזמין הבא ברשימה שנמצאת ב-*Superblock*.

אחסון וגישה למידע

כדי לשנות תוכן בבלוק של קובץ קיים, נדרש לטען את כל הבלוק ל זיכרון ה-*RAM*, לשנות את החלק שאנו רוצים, ואז לכתוב את כל הבלוק מחדש לדיסק. כמובן מדובר בעולה יקרה מאוד, ולכן לטען בלוקים של זיכרון ל-*RAM*, לבצע שינויים רק ב-*RAM* ופעם בכמה זמן לעדכן גם את הדיסק. בכלל, כשאנו קוראים מהדיסק או כותבים אליו, אנחנו עובדים עם בלוקים שלמים ולא עם בתים. הפתרון הנ"ל יכול לאירוע לביקורת במקורה של הפסקת חשמל (כי אז חלק מהשנים לא באמת עברו לדיסק), או למשל אם המשתמש נתק את הדיסק לפני שהוא עדכן (דיסק אונקי) וכו'

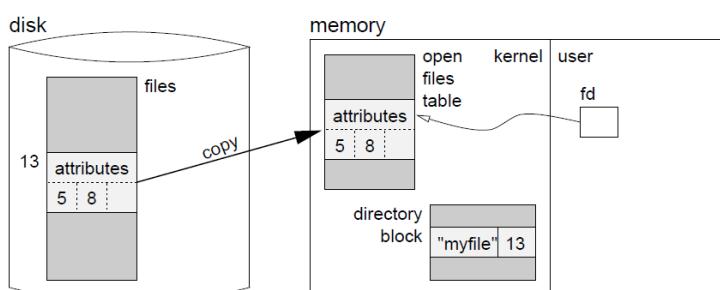
File Descriptor (FD)

- כשאנו עושים (*fd = open("myfile", R)*), והפונקציה רצתה בהצלחה, אנחנו מקבלים בחזרה *FD* – מספר אי שלילי שמצוין את האינדקס בטבלה שנקראת *file descriptor table*. (יש צאת לכל תהליך!) ה-*FD* מאפשר לנו גישה לקובץ או למשאבי קלט-פלט אחרים כמו *network socket* (ולמד בהמשך).
- תהליכיונים חולקים ביניהם את טבלת-*FD* (וה-*file offset*) (זה-*FD* נוצר, והוא תמיד יציב לאותו קובץ).
 - ב-*Unix*, כל דבר הוא קובץ.
- יש מספר *FD* מוגדרים מראש:

File	File Descriptor	POSIX Symbolic Constant
Standard Input	0	STDIN_FILENO
Standard Output	1	STDOUT_FILENO
Standard Error	2	STDERR_FILENO

תהליך פתיחת קובץ:

1. המשתמש מריץ את הפונקציה (*fd = open("myfile", R)*)
2. מערכת הקבצים קוראת את התיקייה הנוכחית ומוצאת ש-*"myfile"* מיוצג על ידי המספר 13 (*(myfile.inode == 13)*)
3. מערכת הפעלה קוראת את שורה 13 מרשימה הקבצים שמתחזקת על הדיסק, ומעתיקה את תוכן של שורה 13 לטבלה שמנהל מערכת הפעלה - *open files table*.
4. הרשות המשמש נבדקות, ומספר ה-*fd* שיוחזר למשתמש מיועד להיות מספר השורה בטבלה ה-*file descriptor table* שיש לכל תהליך.
5. ה-*FD* מאפשר למערכת הפעלה למנוע מהמשתמש לגשת למקומות שהוא לא רשאי לגשת אליו. הכל מתנהל בעזרת פונקציות של מערכת הפעלה.



למה מערכת הפעלה צריכה לתחזק את ה-*open files table*?

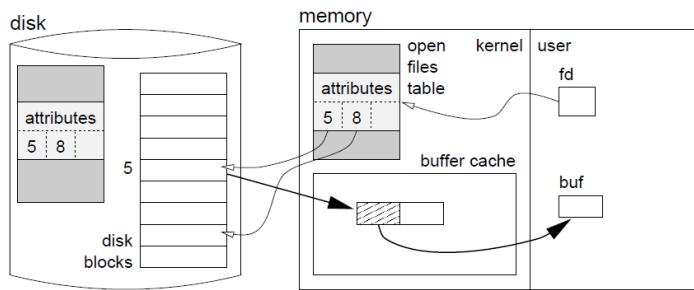
- כדי לבדוק הרשותות

- כדי לאחסן את ה-*offset* (המיקום הנוכחי שאנו חנו קוראים מהקובץ)

כל הפעולות על הקובץ מבוצעות דרך ה-*FD*.

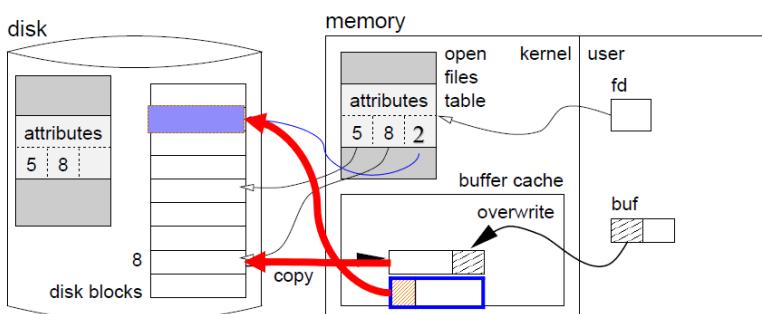
דוגמה לתהיליך קרייה מקובץ (לפי הצורך למטה):

1. המשמש מריצ' את הפוקודה *read(fd, buf, 100)*
2. הארגומנט *fd* מצין את מספר השורה הרלוונטי ב-*file descriptor table*.
3. מערכת הפעלה ניגשת לרשימה הבלוקים שמכילים את תוכן הקובץ.
4. היא קוראת את בלוק מס' 5 (דוגמה שלנו) ל-*Buffer cache* (את כל הבלוק)
5. 100 בתים מתוך הבלוק מועתקים לזכרון המשמש למיקום ש-*buf* מצביע עליו



דוגמה לתהיליך כתיבה לקובץ:

נניח שאנו רוצים לכתוב 100 בתים, שמתחילה בbite-ה-000, בקובץ מס' 8. כל בלוק בדיסק מכיל 1,024 בתים ולכן המידע שאנו רוצים לכתוב אליו נמצא בסוף הבלוק השני של הקובץ ובתחילה הבלוק השלישי בקובץ. נכתוב 48 בתים לסופ' בלוק מס' 8 (הблוק השני בדוגמה שלנו), ואת שאר המידע צריך לכתוב בבלוק השלישי. הבלוק השלישי עדין לא קיים ולכן נזכיר בלוק חדש מרשימה הבלוקים הפנויים וכותבים אליו. לבסוף נעדכן את הבלוקים החדש'ם בדיסק.



הערה: ה-*Buffer Cache* קרייטי לשיפור ביצועים (הוא שומר את כל הבלוק ב-*Cache* בהנחה שבקריאה

הבאנה כנראה שנרצה לקרוא את המשך הבלוק). אבל מנגד הוא יכול לגרום לבעיות אמינות מהסיבת

שכבר הזכרנו – יכול להיות שזמן הפסקת חשמל או הוצאה דיסק אווןקי, המידע לא נכתב לדיסק עדין.

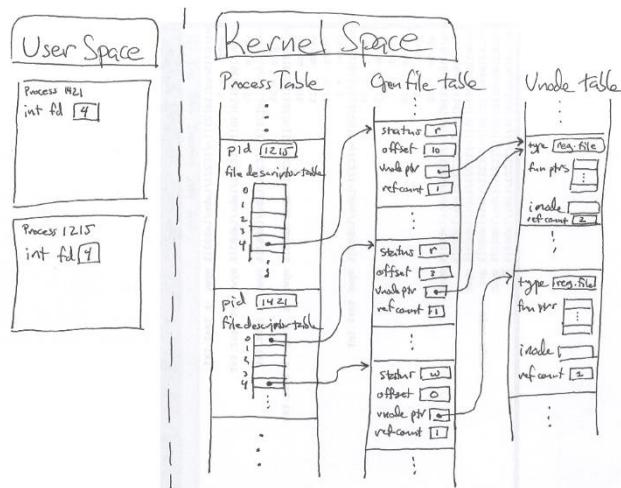
הערה 2: הפקציה *read* מקבלת כתובת *buffer* כדי לשמור את המידע בזיכרון של המשתמש,

אבל לא מאפשרת לקבוע את ה-*offset* בקובץ. מערכת הפעלה היא שאחראית על שמרנת ה-*offset* הנוכחי בקובץ. אם בכלל נרצה לגשת למיקום שרירותי בקובץ, משתמש בפקציה *seek*.

Main OS file tables

נסכם את הטבלאות שמנוהלות את מערכת הקבצים:

- טבלה שטיפה *inodes* לקבצים. כל קובץ יכול להופיע רק פעם אחת בטבלה.
- כל שורה בטבלה זו משתנה בכל פעם שקובץ נפתח. היא מכילה מצביע *inode*, ומיקום בתוך הקובץ (*offset*). יכולות להיות מספר שורות שמצוינות לאוות *inode*, אם המשמש פתח למשל את אותו קובץ מספר פעמים.
- טבלה נפרדת לכל תהליך (שנשמרת ומנויהת ע"י ה kernell). כל שורה בטבלה מצביעה על ערך בטבלה הקבצים הפתוחים (*open files table*). כל *fd* שהפונקציה *open* מחזיר לנו, הוא האינדקס הרלוונטי בטבלה שמצביע מידע על הקובץ הרצוי.



שאלות מבחנים

(6) כמה פעולות קריאה וכתיבה של בלוק מהדיסק צריך לבצע בהרצאת התקنية הבאה (הנה שהקובץ קיים אבל ריק, גודל כל מדיריק בлок אחד, ושם דבר לא נמצא בזכרון מראש, ולא צריך לגשת לדיסק כדי להקצות בЛОקים):

```
fd = open("/x/y/z/foo", O_CREATE);
write(fd, &buf, 13);
close(fd);
```

.א. 5

.ב. 8

.ג. 11

.ד. מספר אחר

תשובה: (המבחן: (http://www4.huji.ac.il/exams/67808_2012_2_2_1.pdf))

- * הדגל *O_CREATE* יוצר את הקובץ במידה והוא לא קיים. מדיריך == תיוקיה.
- ה-"/" בתחילת הכתובת של הקובץ *foo* הוא שם של תיוקיה (התיקייה הראשית במחשב). לכן יש לתיוקיה הזאת *inode*. ניגש ל-*inode* זהה, ובתוכו ניגש לבlok שהוא מצביע עלייו (נתון שכל תיוקיה בגודל בлок 1) ככלומר כבר היינו 2 גישות. בблוק זהה יש מצביע ל-*inode* של *x*, ניגש אליו ובתוכו ניגש לבlok שהוא מצביע עלייו. (עוד 2 גישות) באותו אופן גם *y* ו-*z* מצביעים כל אחד עוד 2 גישות. סך הכל 8 עד עצשו. עכשו נשארה גישה נוספת מתוך *z*, ל-*inode* של *foo*. הינו צריכים 9 גישות זיכרון כדי לפתח את *foo*.
- פעולות *write* תיגש ל-*first direct pointer* (כי הקובץ ריק) ותכתב אליו 13 בתים. (גישה 10)
- ולבסוף *close* ניגש שוב לקובץ כדי לעדכן למשל תאריך שינוי אחרון, גודל הקובץ וכו'. (סה"כ 11 גישות)

שאלה 3: מערכות קבצים (24 נקודות)

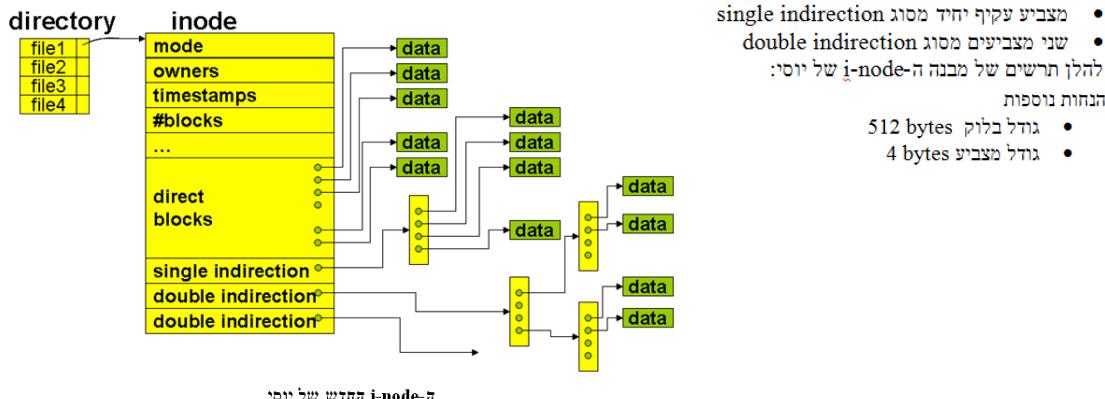
השאלה הבאה מתייחסת לבניה המנתנים node-i המשמש לאחסון נתונים על הדיסק, שנלמד בהרצאה על ממשן מערכות קבצים. (התרשים מהרצאה מופיע בעמוד הבא)

ויסי, הכותב את מערכת הפעלה **Xosix**, נגש למשן מערכת הקבצים. מאחר ונראה לו מסובך למשן והערך כי רב הקבצים במערכת היו קטני, החליט להרשות במאכביים ישירים.

- לפיך המיפוי לבלוקי הקובץ מה-node-i שיצר נראה כך:
- 24 מאכביים ישירים לבלוקי הקובץ מספרים מ-0 עד 23

- מצביע עקי חיד מסוג single indirection
- שוי מאכביים מסוג double indirection
- להלאן תרשימים של מבנה ה-node-i של ויסי:

- גודל בלוק 512 bytes
- גודל מאכבע 4 bytes



מבחן: הטכניון, תשס"ה מועד א'

קובץ: שאלה 3 בקובץ "Tech_2005_Moed a_solution.doc"

(4 נקודות)

- A. ישנו נתונים רבים המופיעים ב-node-i הנ"ל, כגון mode, owners, timestamps ועוד. אולם הוא אינו כולל את ה-offset בקובץ. כיצד ניתן זה מוחזק?

תשובה:

נתון זה נמצא ב-file object שנמצא ב-Open Files Table

(3 נקודות)

B. מהו הגודל המקסימלי של קובץ (בלוקים) שנitinן להצביע אליו באמצעות node-i זה, וכמה בלוקים סה"כ יתפוס קובץ זה על הדיסק (כולל index blocks, אולם לא ה-node-i של הקובץ)? יש להציג חישוב מפורט ולהסביר. הגודל המקסימלי הוא _____ בלוקים. קובץ זה יתפוס _____ בלוקים על הדיסק.

תשובה:

מספר המצביעים לבלוקים שאפשר להכיל בבלוק אחד במערכת של ויסי הוא $128 = \frac{512}{4}$,

לכן מספר הבלוקים שאפשר להצביע עליהם מהתור inode בודד הוא:

24 בלוקים מהמצביעים הישרים + 128 בלוקים מהמצביע העקי הבודד

+ 2 מצביעים עקיפים כפולים שכיל אחד מצביע על 128^2 שכן סה"כ:

$$\text{Number of data blocks: } 24 + 128 + 2 \times (128)^2 = 32,920$$

כלומר inode בודד יכול להצביע ל-32,920 בלוקים לפחות.

יתפוס (לא כולל inode) צריך לספר גם את הבלוק שהמצביע ההפוך מצביע אליו + בלוק שמצויב ל-

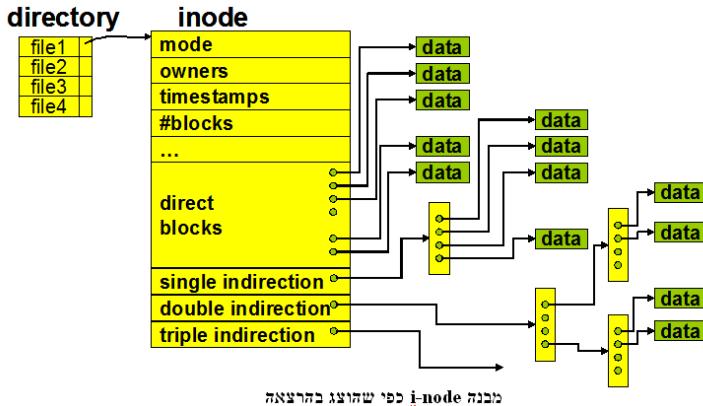
128 בלוקים שמצויבים ל-128 בלוקים (פעמיים), סה"כ:

$$\text{Total number of blocks: } 32,920 + 1 + 2 \times (128 + 1) = 33,179$$

ולכן קובץ צזה יתפוס 33,179 בלוקים על הדיסק.

ג. מהו האודל המקסימלי של קובץ שניון להציגו אליו לפי יציג ה-node-i ש널מד בהרצאה? (בכל node-i יש 10 מוחונים ישירים, וכן מצביעים ל-3 רמות של indirect). ראו תרשימים מצורף). כמה בЛОקים סה"כ יתפוז קובץ זה על הדיסק (כולל index blocks (אולם לא ה-node שגודלו של הקובץ)? יש להציג ויחסם מפורט ולהסביר. הגודל המקסימלי הוא _____ בЛОקים. קובץ כזה יתפוז _____ בLOWקים על הדיסק.

(3 נקודות)



תשובה: (הנימוק דומה לשיעיף הקודם)

$$\text{Number of pointers in an indirect block: } \frac{512}{4} = 128$$

$$\text{Number of data blocks: } 10 + 128 + (128)^2 + (128)^3 = 2,113,674$$

$$\text{Total number of blocks: } 2,113,674 + 1 + (128 + 1) + (128^2 + 128 + 1) = 2,130,317$$

בשאלות הבאות הניחו את הנתונות הבאות:

- השאלה מתייחסות לsworth ניהול הקבצים הנהוגה ב-Linux.
- בכל directory יש מספר קטן של קבצים.
- open() אינו מבצע prefetch (כלומר הקיראה של הנתונים עצם תבצע רק כאשר מבצעים read)

בספריה /usr/yossi ישנו קובץ בשם myfile שגודלו 32k
ד. כמה בLOWקים יתפוז הקובץ במערכת ההפעלה של יוטי? (כולל Index blocks, אולם לא ה-node שגודלו של הקובץ?)
הקובץ יתפוז _____ בLOWקים.

תשובה:

$$\text{Number of data block} = \frac{32\text{KB}}{512\text{B}} = 64 \rightarrow \text{all directs} + 40 \text{ indirects} \rightarrow \underbrace{64}_{\text{data}} + \underbrace{\frac{1}{indirect1}}$$

הקובץ צריך 64 בLOWקים, על 24 מתוכם נקבע עם המצביעים הישירים, ועל ה-40 הנוספים נקבע בעזרתו מצביע עקי (שדורש בLOW נוסף), لكن בסה"כ הקובץ יתפוז 65 בLOWקים.

ה. כמה בLOWקים יתפוז אותו קובץ לפי יציג ה-node-i שראיתם בהרצאה? (לפי הנתונים מסעיף ג')

תשובה:

$$\text{Number of data block} = \frac{32\text{KB}}{512\text{B}} = 64 \rightarrow \text{all directs} + 54 \text{ indirects} \rightarrow \underbrace{64}_{\text{data}} + \underbrace{\frac{1}{indirect1}}$$

הקובץ צריך 64 בLOWקים, על 10 מתוכם נקבע עם המצביעים הישירים, ועל ה-54 הנוספים נקבע בעזרתו מצביע עקי (שדורש בLOW נוסף), لكن בסה"כ הקובץ יתפוז 65 בLOWקים.

ג. נטונה קריית המערכת הבאה:

`open(fd, "/usr/yossi/myfile", O_RDONLY);`

הנicho שהקובץ אליו נגשים לקרוא קיים, והוא קובץ רגיל (לא soft/hard link).

מה המספר המונומלי של גישות לDick שיתבצעו ע"י קריאה זו?

נימוק:

תשובות:

0 גישות, אם כל הנתונים כבר בזיכרון הראשי ולכן לא נדרש גישה לדיסק.

____ מה המספר המקסימלי של גישות לDick שיתבצעו ע"י קריאה זו?

נימוק:

תשובות:

7 גישות לכל היתר. מtbodyות גישות ל-*-i* /usr , /usr/yossi , /usr/yossi/myfile לפתחה. (לפי אותו הסבר בעמוד 122)

File Systems

אפליקציות יכולות לאחסן מידע בזיכרון הראשי (*RAM*) אבל לא תמיד זה הפתרון הנכון:

- גודל הזיכרון מוגבל לגודל הזיכרון הווירטואלי ולכך לא תמיד מספיק.
- המידע נמחק ברגע שהאפליקציה מסיימת את הריצה.
- המידע לא זמין עבור תהליכיים אחרים.

נרצה לשמר מידע בזיכרון לטווח ארוך שייעמוד בקריטריונים הבאים:

- מסוגל לאחסן מידע רב.
- המידע צריך "לשוחד" גם אחרי שתהיליך מסוים משתמש בו (ולא להימחק).
- המידע צריך להיות נגיש למספר תהליכיים.

פתרון:

- נוכל לאחסן מידע על הדיסק (או בכלל זיכרון *volatile – non volatile*) ביחידות שנקראות "קבצים".
- קובץ זמין תמיד, ורק הבעלים שלו יכול למחוק אותו.
- קבצים מנוהלים על ידי מערכת הפעלה.

File Systems

- מערכת קבצים זו הדריך בה מערכת הפעלה מממשת ומנהלת קבצים.
- היא מכילה ממשקים שונים כמו יצירה, קריאה, כתיבה, חיפוש, מחיקה ועוד..
- השימוש עצמו כולל שיטות להקצת זיכרון, שיთוף קבצים, ניהול מרחב הזיכרון ועוד..
- פעולות של מערכת קבצים הן *System Calls* כדי לאפשר למערכת הפעלה לבצע פעולות מיוחדות ברמת חומרה, במקרה זהה בשבייל הדיסק.
- התשובות מהדיסק מגיעות דרך פסיקות (*Interrupts*).

File Attributes

מידע ספציפי המנווה על ידי מערכת הפעלה:

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

שם הקובץ – לטובת המשמש.

מזהה – מספר ייחודי המיצג את הקובץ

עבור מערכת הפעלה.

סוג הקובץ

מקום – מצביע למיקום של הקובץ בדיסק.

גודל נוכחי של הקובץ

הגנה – הרשותות קראיה, כתיבה או ררצה

תאריך ושעה – לטובת מעקב שימוש

זהות מנהל הקובץ – לטובת הגנה

File Extensions

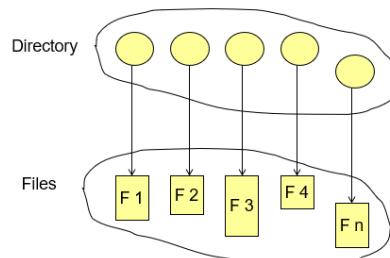
שם קובץ מחולק ל-2 חלקים, החלק השני הוא סימנת הקובץ.
ב-*UNIX* הסימנות לא הכרחיות (אבל הקומפיילר של שפת *C* עלול להתעתק עליהן כי הן שימושיות עבורה) ו-*Windows* מנסה לשמר משמעות לסימנות – למשל בכך שהיא משיכת אפליקציות לשימוש מסוימות.

Directories

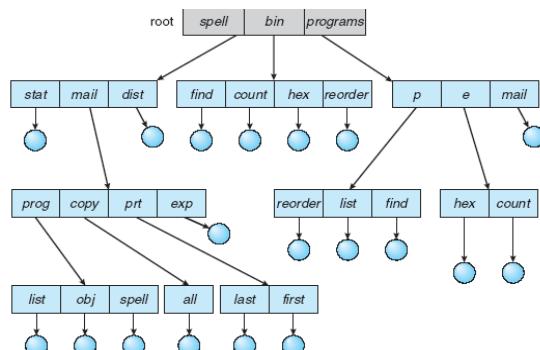
קבצים מאורגנים בתיקיות לטובות:

- יעילות – מציאת קובץ בקלות
- מתן שמות – נוח עבור משתמשים.
- 2 משתמשים יכולים להשתמש באותו שם עבור קבצים שונים.
- לאותו קובץ יכולים להיות מספר שמות.
- קיבוץ – קיבוץ לוגי של קבצים לפי מאפיינים (למשל "משחקים", "מסמכים" וכו')

תיקיה היא סוג של טבלת סמלים הממפה קבצים לשמות רצויים. המבנה שלה צריך לאפשר חיפוש, יצירה, מחיקה, שינוי שמות קבצים ועוד..



בפועל, תיקיה היא עצם שכל קדקוד פנימי בו הוא תיקיה נוספת, וכל עלה הוא קובץ:



Path Names

כדי לגשת לקובץ, המשתמש צריך לעבור למספרה שבה הקובץ נמצא או לציין את הנתיב (*path*) שלו.

נתיב יכול להיות מוחלט (*absolute*) או יחסי (*relative*):

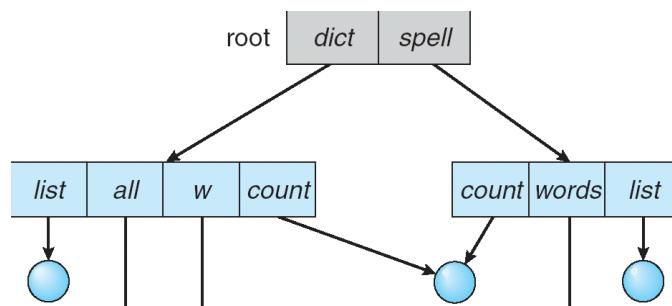
- *Absolute*: נתיב לקובץ החל מהתיקייה הראשית במחשב.
- *Relative*: נתיב לקובץ החל מהתיקייה הנוכחיית.

ברוב מערכות הפעולה יש שני ערכיהם מיוחדים:

- "...". כדי לציין את התקינה הנוכחיית, ו- "..". כדי לציין את תיקיית האב.

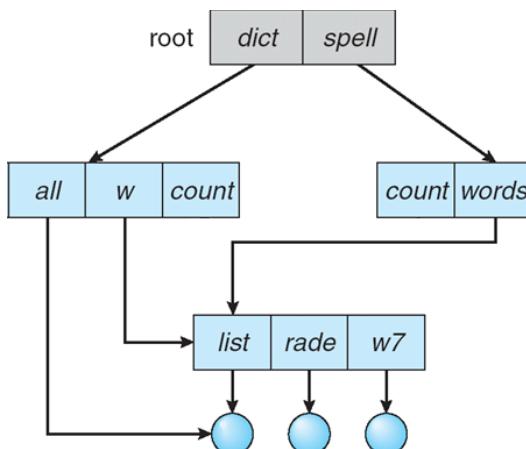
Hard Link & Symbolic Link

— möglichst 2 Symbole für dasselbe Objekt:



- Die Datei `dict/count` enthält ein Hard Link zu `spell/count` mit dem gleichen Inhalt.
- Wenn der Name des Hard Links geändert wird, ändert sich dies nicht für das Original.
- (Wenn dies geschieht, ist es ein Hard Link)
- Es kann kein Hard Link für eine Datei erstellt werden, die leer ist.
- Prüfung der Hard Links (Hard Links) zeigt, ob es mehrere Wege gibt, um zu einer Datei zu gelangen.
- (Beispiel: Wenn es zwei Wege gibt, um zu einer Datei zu gelangen, dann ist es ein Hard Link)
- Ein Hard Link kann nur auf einer Plattform wie UNIX erstellt werden.
- Ein Hard Link kann auf Windows erstellt werden.

— möglichst 2 Symbole für dasselbe Objekt:



- Die Datei `spell/words` enthält ein Symbolic Link zu `dict/w/list`.
- Wenn der Name des Symbolic Links geändert wird, ändert sich dies nicht für das Original.
- (Wenn dies geschieht, ist es ein Symbolic Link)
- Ein Symbolic Link kann nur auf einer Plattform wie UNIX erstellt werden.
- Ein Symbolic Link kann auf Windows erstellt werden.

File System Mounting

הfonקציית *Mount* ב-Linux מאפשרת לחבר בין מערכות קבצים שונות. למשל:

`mount("/dev/usb0","/mnt",0)`



העץ השמאלי מייצג את מערכת הקבצים שלנו לפני השינוי, והעץ הימני אחריו השינוי.

הוספנו את מערכת הקבצים **/dev/usb0** (עץ של תיוקיות וקבצים) לקדקוד **/mnt** במערכת הקבצים שלנו.

בעזרת *Mount* אפשר לצרף למערכת הקבצים גם מערכות קבצים מרוחקות כמו למשל תיוקיה בשרת

מרוחק. אחרי פקודה **Mount**, הגישה תתבצע כמו גישה לכל תיוקיה אחרת במערכת הקבצים.

(בפועל, כל פעולה במחשב המקומי מתורגם להודעות שנשלחות למחשב המרוחק).

המחשב המרוחק מקבל את ההודעות, מבצע אותן ומחזיר את הפלט למחשב המקומי)

Windows אפשר לעשות אותה פעולה באמצעות [coni רשות](#).

File Protection

מערכות הפעלה צרכות לאפשר למנהל/ויצר הקובץ לנוהל את הרשאות שלו - מה אפשר לעשות עם הקובץ ועל ידי מי. פועלות על הקובץ הן למשל קריאה, כתיבה, הריצה, הוספה לסופי הקובץ (*Append*), מחיקה ועוד.. נוכל לחלק הרשות למשתמשים בודדים או לקבוצות שלמות:

- משתמשים בודדים – ניהול גישות לפי משתמשים (локליים של המחשב או משתמשי רשת)
- קבוצות – ניהול גישות לפי קבוצות, כל משתמש שישיר לקובץ רלוונטי יקבל הרשות.

Linux Access Rights

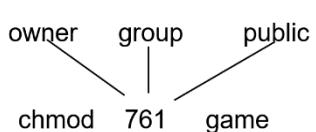
בלינוקס יש 3 סוגי גישה – קריאה (*read*), כתיבה (*write*) והריצה (*execute*), ו-3 מחלקות משתמשים - *Owner, Group, Public*. נוכל לבחור הרשות רצויות לכל מחלקה:

a) owner access	7	\Rightarrow	RWX 1 1 1
b) group access	6	\Rightarrow	RWX 1 1 0
c) public access	1	\Rightarrow	RWX 0 0 1

בעזרת הפקודה *chmod* אפשר לשנות הרשות לקובץ מסוים:

למשל הפקודה *chmod 761 game* בלינוקס תאפשר למנהל

(*owner*) של הקובץ *game* הכל (קריאה, כתיבה והריצה), לקובץ (group) רק קריאה וכתיבה, ולגישה פומבית רק הריצה.



נשים לב שיש רק מנהל (*owner*) אחד ורק קבוצה אחת. זה אומנם קומפקטי מבחינה בתים בזכרון, אבל לא תמיד מספיק לצרכים שלנו. לעומת זאת, ב-Windows לכל קובץ יש *Access Control List* (*ACL*), רשימה המפרטת לבדוק לאיזה משתמש יש הרשות ואיזה מה שמאפשר ניהול מוחלט אבל דורש יותר בתים.

File Access

בעבר, גישה למידע בדיסק הייתה רק גישה רציפה (*Sequential Access*):

- קרייה של כל הבטים/רשומות מההתחלתה
- לא ניתן לקפוץ למקומות ספציפיים בקובץ (ביעילות).
- אפשר לקפוץ קדימה או לחזור אחורה אבל הפעולות האלה יהיו יקרות יחסית.

במערכות הפעלה כיום, גישה למידע בדיסק מאפשרת גם גישה אקראית (*Random Access*):

- קרייה של בתים בכל סדר רצוי
- הכרחי למסדי נתונים

נענו על השאלות הבאות:

- מה מבנה הנתונים הנכון לאחסן מיקומי הקובץ בדיסק?
- איך נמפה את הקבצים בדיסק?
- איך נממש גם גישה רציפה וגם גישה אקראית?

Fragmetation (as in memory)

מצב זה מצב בו יש מספיק זיכרון להקצות לקובץ, אבל הזיכרון לא רציף. במקרים בהם הזיכרון לא מאפשר מחיקה של קבצים (כמו למשל DVD), לא יקרה מצב של *Fragmentation*.

פתרונות אפשריים:

- *Compaction* – סידור מחדש של המידע כדי לאחד את הזיכרון. פעולה זו יקרה מאוד ולפעמים אין מספיק מקום בזיכרון כדי לבצע את הסידור.
- פיצול הקבצים לבlokים (בדרך כלל בגודל 4KB). (באוטו אופן כמו *pages* בזיכרון הראשי)
 - כל הפעולות מבוצעות על בלוקים ולא על בתים.
 - כדי לקרוא חלק מהבלוק, ניבא את כלו לזכרון וನשלוף את הערכים הרלוונטיים.
- (למשל קריאה לפונקציות *getc()* ו-*putc()* שקוראותתו תז בודד בכל קריאה לפונקציה, מייבאות בלוק שלם ל-*Buffer* וקוראות ממנו)
- כדי לשנות ערכים בבלוק, ניבא את כלו לזכרון, נשנה אותו, ונכתוב את כלו מחדש!
- בדיקן כמו מיפוי בין *frame* ל-*page*, כאן אנחנו ממפים בין סקטורים (יחידות הזיכרון בדיסק) לבין בלוקים. (בדרך כלל $BlockSize \geq SectorSize$)
 - מעתה נתיחס לקובץ zusätzlich של בלוקים.

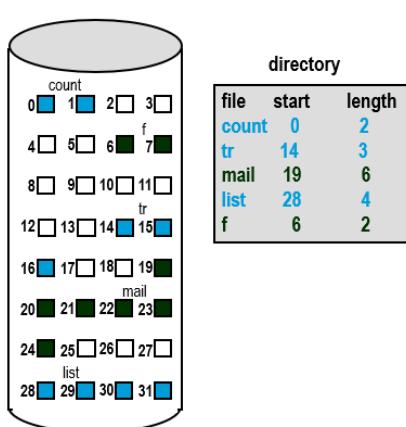
איך קבצים/בלוקים מאורגנים בזיכרון? כדי לענות על השאלה הזאת נctrיך להחליט על אופן פריסת הבלוקים בדיסק, באיזה מבנה נתונים משתמש לניהול המידע, ואיך ננהל את המקום הריק בדיסק (כדי שנוכל להקצות מקום לקבצים חדשים). איך כדי לענות על השאלות האלה, נctrיך לדעת איך ממומשת הגישה לקבצים, איפה הם מאוחסנים (דיסק, DVD, SSD וכו'), איפה ה-*meta-data* ישמור ואיך, ומהן נסוכן של מערכת הפעלה (תמייה בגרסאות ישנות, הגנה, שיינטוף מידע וכו')

פעולות על הדיסק

בעמוד 73 הסבכנו איך מתבצעת הקריאה מהדיסק.

- כל פעולה קריאה קוראת סקטור שלם.
- עבור דיסק של PRM 7200, זמן סיבוב שלם (*Rotational time*) הוא כ-*4ms*
- זמן הzzת הסיכה (*Seek time*) הוא כ-*12ms* – 8 – 0.25*ms*
- זמן העתקת המידע (*Transfer Time*) הוא כ-*0.25ms*

ונכל להימנע מזמן הzzת הסיכה ומזמן סיבוב הדיסק (חו"ץ מהפעם הראשונה עד שהסקטור הראשון יגיע לסיכה) אם הסקטורים יהיו מסודרים באופן רציף בדיסק. אם לא ניתן לשמור את המידע באופן רציף, ננסה לפחות לשמור אותו על אותו מסלול (ובכך להימנע מזמן הzzת הסיכה)



הකצת זיכרון רציפה

יתרונות:

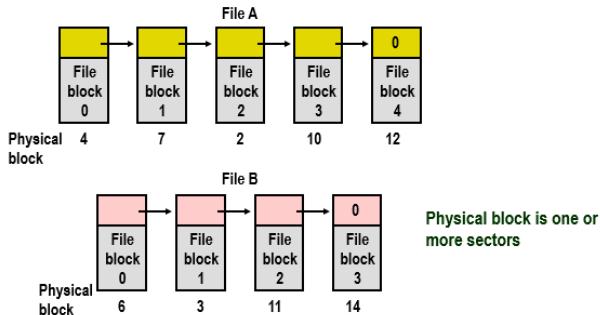
- פעולה פשוטה יחסית. צריך רק נקודת התחלתה (מספר בלוק) ואורך (מספר הבלוקים)
- גישה מהירה, מספר נמוך של חיפושים (*Seeks*)

חסרונות:

- קובץ דינמי עם גודל משתנה דורש העתקה של המידע
- למיקומים אחרים ככל שהוא גדול
- בזבזני מקום, *Fragmentation*

לבדוק אם כדאי לשמור בלוקים ברשימה מקושרת

עבור קובץ A, נשמר את הבלוקים שלו ברשימה מקושרת:

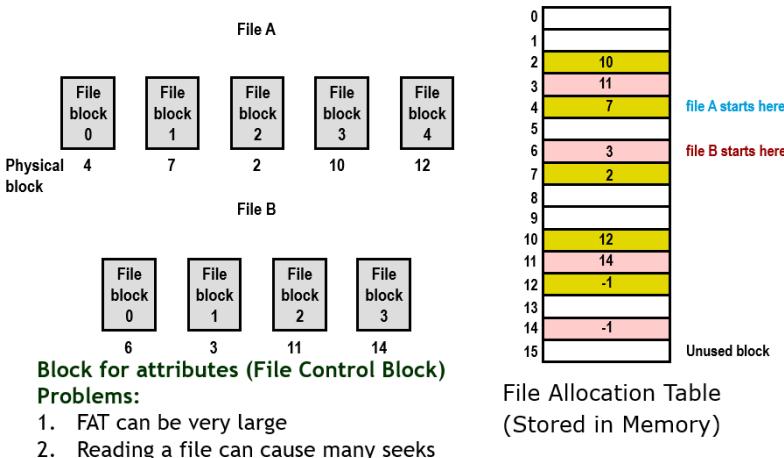


בעיות בגישה זו:

1. לא ניתן למשוך *Random Access* בצורה יעילת (נצרך לעبور על כל הרשימה עד שנמצא את הבלוק הרצוי)
2. כל בלוק כבר לא יכול חזקה שלמה של 2 (למשל 4KB) כי המצביעים ידרשו כמה בתים.

פתרונות אפשרי לבעיה הנ"ל

נשמר טבלה נפרדת (*FAT: File Allocation Table*) שתשמור את המצביעים:

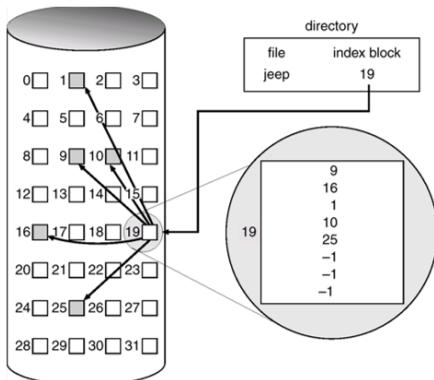


כלומר הבלוקים "מרחפים" בזיכרון, ומה שמחבר ביניהם זו הטבלה. הבלוק הראשון של קובץ *A* (File block 0) נמצא בשורה מס' 4 בטבלה, ובשורה הזאת נמצא המצביע על הבלוק הבא בזיכרון (7). באותו אופן בשורה 7 בטבלה יש מצביע ל-2 כי זה הבלוק השלישי וכן הלאה. הבלוק האחרון של קובץ *A* הוא בלוק 12, ולכן שורה 12 בטבלה מכילה את המספר 1. בצורה זו המצביעים לא תופסים מקום בבלוקים, והגישה לבlokים מהירה יותר מאשר מקוشرת.

בעיות בגישה זו:

1. הטבלה יכולה להיות גודלה מאוד !
2. קריאת קובץ יכולה לגרום להרבה הזזות של המוחט בדיסק (Seeks) אלא אם כן כל הטבלה נמצאת בזיכרון הראשי. אף עבורי דיסק בגודל $20GB$ (כיוון הדיסקים הרבה יותר גדולים), ועבור בלוקים בגודל $4KB$, יש 5 מיליון שורות בטבלה *FAT*. אם כל שורה היא 4 בתים, צריך $20MB$ בזיכרון ה-*RAM* רק בשbill הטבלה.

מיפוי בלוקים באמצעות מצביעים



אפשר להציג לכל קובץ בלוק שיכיל מצביעים לבlokים אחרים שמאחסנים את תוכן הקובץ. למשל באיזור שמאל, הבלוק 19 מכיל מצביעים לבlokים 10, 9, 16, 1, 10 ו-25 לפי הסדר. כל שאר המצביעים ריקים (-1).

נניח שכלי בלוק בגודל $4KB = 4,096 bytes$, אז כדי לקרוא את הביט 7,000 של הקובץ נוכל לפנות ישיר לבלוק 16 (הблוק השני של הקובץ), כי הביטים 8192 – 4096 נמצאים שם. בתרגול למדנו שב-*UNIX* קבצים ממומשים בדרך זאת

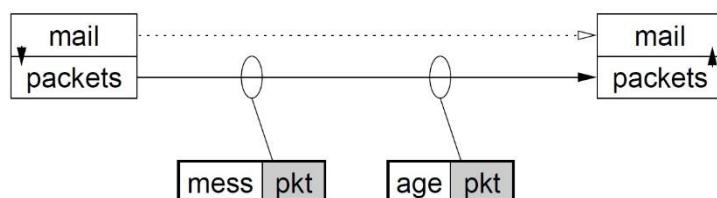
בעזרת *inode*. בנווט על מצביע עקייף יחיד/כפול/משולש, ושנוכל למצוא את הבלוק הרצוי של קובץ מסוים בעזרת 3 גישות לדיסק לכל היותר.

פרוטוקול תקשורת – *Communication Protocol*

ויקיפדיה: פרוטוקול תקשורת הוא נוהל לתקשורת. כמובן, אוסף של כללים המגדירים את אופן בקשת וקבלת נתונים במערכת תקשורת מסוימת וכללים לייצוג המידע, איות, אימות, ותיקון שגיאות לצורך העברת המידע בערוצ תקשורת. פרוטוקול מוכר ופשוט הוא שיחת טלפון הכוללת כללים מוסכמים: הרמת השופורת, קריית "הלו", הצד מניגד עונה ב"שלום" (זהו שלב האימות) ולאחר מכן יסביר את מהות התקשורת ותחילה העברת המידע. לפני ניתוק השיחה יפרק האנשים ב"בי" או "להתראות". אולם ישנה גמישות, ואין בהכרח צורך בפרוטוקול קשיח ומוחלט, וכך לא כל שיחת טלפון מתנהלת על-פי הפרוטוקול המדויק הנ"ל. אך כאשר מדובר בראש תקשורת בין מחשבים, שימוש בפרוטוקולים מודיקים הכרחי על-מנת שהצדדים יבינו זה את זה ויוכלו לספק שירותים זה לזה.

בזמן שליחת הודעה ממחשב אחד לשני יתכנו הפרעות בדרך, ביטים מסוימים בהודעה יכולים להשתנות. נניח למשל שאנו חnage שולחים קובץ גדול בין 2 מחשבים. יש אפשרות להזהות אם הייתה הפרעה בדרך וחילק מהביטים השונים, אך לא ניתן לדעת מה בדיק השתנה. במקרה זה ההודעה לא רלוונטית (כי אי אפשר לסמור על המידע שעבר), וכן אנחנו נותר על כל הודעה ! בזאת זמן ומשאבים לחינם. מסיבה זו, נרצה להעביר הודעה קטנות בלבד. אם ביט מסוים השתנה באחת ההודעות, לוותר עליה (ולהעביר אותה מחדש) יפריע לנו פחות מאשר הודעה גדולה.

כדי בכל זאת לאפשר שליחה של הודעות ארוכות, הצד של שליח ההודעה יש רכיב שמקבל הודעה ארוכה ומפרק אותה לתתי הודעות קטנות – פקודות (*Packets*). הצד של מקבל הודעה יש רכיב נוסף שודיע להרכיב את ההודעות הקטנות בחזרה להודעה אחת גדולה. אחרי פירוק הודעה לפקודות, נוסיף לכל הודעה שכבת מידע נוספת המכילה את מספר הודעה:



גם כאן יכולות להיווצר בעיות כמו פקטה שנעמלמה או פקודות שהגיעו בסדר שונה. כדי לפתור את הבעיה הנ"ל נستخدم ב-*End to end Control* – מנגנון שתפקידו להוסיף שכבת אמינים לתאගת שההודעות שיגיעו לצד השני יגיעו בסדר שבו רצאים וכך כן יסיה למונע אובדן של פקודות.

ניטוב – *Routing*:

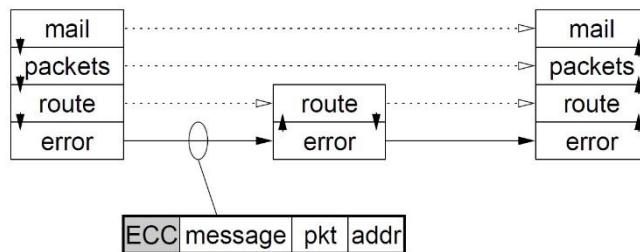
לרוב, מחשבים לא מחוברים ביניהם בכבליŞir, ולכן כൺשלחת הודעה היא תעבור בין כל מיין מחשבים בדרך שיהו תחנות ניטוב ויעבירו אותה להלאה. לשם כך נצטרך להוסיף שכבה נוספת להודעה – כתובת הנמען. כשמחשב מקבל הודעה, אם היא לא מיועדת אליו הוא ידע לאן להעביר אותה.

:Error Correction

כדי לזרות שגיאות, נוסיף להודעה גם *Error – Correction Code (ECC)* – מגנון נוסף שתפקידו לזרות האם ההודעה השתנתה במהלך הדרך. לעיתים ניתן גם לתקן – לרוב זה לא יקרה.

:Protocol Stack

הא מונח *protocol stack* הוא מושך המתיחס למימוש של פרוטוקולים על פני רשתות תקשורת שונות. הצד השולח מוסיף להודעה כותרות מסוימות (*headers*) והצד מקבל משתמש בהן בכל שכבה ולאחר מכן מוחק אותן. בדרך זו, כל שכבה יכולה לתקשר באופן ישיר עם השכבה המקבילה לה בצד השני.



(ה-*Internet Protocol Suite (TCP/IP)* – protocol stack שה האינטרנט משתמש בו:

פרוטוקול זה מורכב מ-4 שכבות, כאשר כל שכבה מספקת שירותים לשכבה שמתוחתיה, וונעצתה בשכבה שמعلיה. כמו כן, לכל שכבה יש פרוטוקול משלها.

שם השכבה	תיאור (מטרה)	פרוטוקול
<i>Application</i>	תקשורת בין תהליכיים. המשמש משתמש בפרוטוקולים אלו בצורה ישירה.	<i>HTTP/S</i> <i>SSH</i> <i>FTP</i> <i>DNS</i>
<i>Transport</i>	תקשורת <i>End – to – end</i> עבר או פליקציות.	<i>TCP, UDP</i>
<i>Network/Internet</i>	הוספה כתובת של נמען לכל פקטה, בתהליך זה מתבצע הפירוק והוספה כתובת ה- <i>IP</i> .	<i>IP</i> – 4 מספרים המייצגים כתובת, מופרדים בנקודות וכל מספר באורך 8 ביטים.
<i>Link/Physical</i>	העברה מיידית בין מחשבים סמוכים אחד לשני.	<i>802.11 WiFi, Ethernet</i>

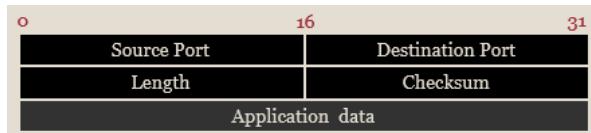
Transport layer

שכבה ה-*Network (IP)*, שנמצאת מתחת לשכבה ה-*Transport*, היא זו שודאגת להעברת הודעות אבל היא לא דואגת לאמינות המידע: שפקיות לא ילכו לאיבוד, ישוכפלו, יוחלפו או יעברו בסדר לא נכון. שכבה ה-*Transport* מספקת את האמינות, והפרוטוקולים העיקריים שלו הם *TCP* ו-*UDP*.

(User Datagram Protocol (UDP))

בפרוטוקול זה, לכל פקטה מתווסף מידע מינימלי (*Headers*) באופן הבא:

- כתובות *IP* מפנה את הפקטה למחשב מסוים, אבל כדי שהפקטה תדע לאן לשלוח בתוך המחשב (לאיזה תחılır למשל) היא צריכה לציין גם את ה- *Destination Port* אליו היא מיועדת. התהılır בצד השני מАЗן לפורט וברגע שהפקטה תגיע הוא ידע להשתמש בה. בנוסוף ל- *Destination Port* נשלח גם את ה-*Source Port* כדי שהטהılır יוכל לקבל את הפקטה, ידע לאן לשלוח תשובות בחזרה.
- *Packet Length and Checksum* – פרוטוקול *UDP* מאפשר בדיקת אמינות מינימלית (לעומת *TCP* שנראה מיד). לפני שליחת ההודעה, נוסיף את האורך שלה (*Length*) וערך נוסף הנקרא *Checksum*. בגדול, מדובר בתוצאה של פעולה מתמטית כלשהי (למשל קסור) על כל הבטים בהודעה, וכן מחשב היעד יכול להריץ את אותה פעולה על ההודעה שהוא קיבל ולהשוות בין הפלטים. אם הפלט זהה, המידע (כל הנראה) לא נפגם בדרך.



- פרוטוקול *UDP* לא מודד שהפקטות שנשלחו הגיעו לידי, ולכן יוכל להשתכוף, לילכת לאיבוד, או להגיע בסדר לא נכון. הוספה מנגן לוודא הגיעו מגדיל כל הודעה ומוסיף לתקורה.

(Transmission Control Protocol (TCP))

גם בפרוטוקול זה נוסיף לכל פקטה *Headers*, אך כאן גם נודע שההודעות אכן הגיעו לצד השני. בנגדוד ל-*UDP*, ב-*TCP* יש תיאום בין שני הצדדים, כלומר מוסכם מראש כמה פקודות צrüיכות להגיע, באיזה סדר, וכו'. כמו כן, כל צד מחזק *buffer* שנועד לקבל טעויות שmagיעות ולודא שהצד השולח לא שלח יותר מידע ממנו שנitinן לקבל.

- מספור הפקטות ושליחתן בסדר מסוים. מאפשר לוודא שהמידע מתקיים בסדר נכון.

- וידוא שלח יותר מידע מאשר ניתן לילכת לאיבוד כי אין איפה לאחסן אותו. פרוטוקול זה מבטיח שפקיות לא ילכו לאיבוד, לא ישוכפלו, ולא ישולחו בסדר לא נכון. כפי שאמרנו, כדי להבטיח זאת אנחנו מוסיפים יותר כותרות ומגדילים כל פקטה, וכן יהיו מצבים שנעדיף להשתמש ב- *UDP* כמו למשל ביוטיוב, משחקי רשות ועוד.

² מילקfidah: ניתן להסביר את המונח פורט באמצעות האנלוגיה הבאה: נניח שכתובות IP היא כתובות של בניין מגורים. אם מכתב נשלח לכתובות מסוימת ללא מספר דירה, לא ניתן לדעת למי הוא שייך. לכן על שלוח המכתב לציין פרט לכתובות (IP) את מספר הדירה (Port) .

לסייעם – ההבדלים בין TCP ל-UDP

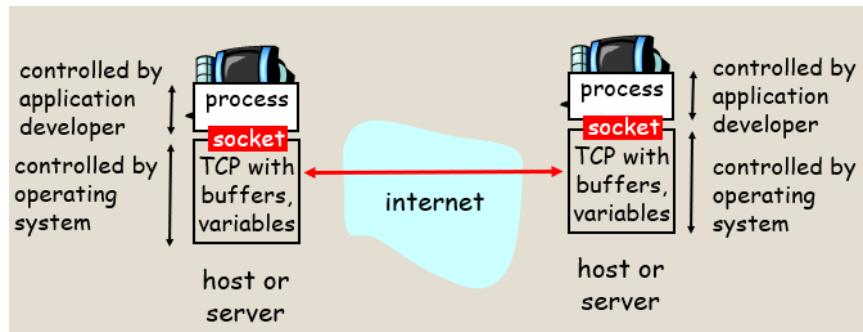
TCP	UDP	
יש	אין	אמינות
<i>Connection oriented</i>	<i>Connectionless</i>	סוג החיבור
יש	אין	<i>Flow control</i>
גבוה	גמור	<i>Latency</i>
<i>HTTP, HTTPS, FTP, SMTP, Telnet, SSH</i>	<i>VOIP, Most games</i>	APPLICATIONS

Sockets

תקשרות בין מחשבים ממומשת באמצעות *sockets*. *socket* הוא ממשק שנוצר על ידי אפליקציה מסוימת במחשב (נקרא גם *Host*), המנוהל על ידי מערכת הפעלה ומאפשר תקשורת עם הצד השני. דרך ה-*socket* אפשר לשלוח ולקבל מידע והוא ממומש לפי פרדיגמה של *Client-Server*. *Server* רץ ראשון, וכל שאר המתחברים אליו (מקבלים ושולחים מידע ממנו). *Client* להגדיר את ה-*socket* להשתמש בתקשרות אמינה או לא אמינה (*TCP* או *UDP*).

דוגמה לייצירת תקשורת בין תחאים לפי פרוטוקול TCP:

פותח תוכנה שימושי לקבל הודעות (ה-*Server*), מייצר ממשק של *socket* ובוחר שהתקשרות תבוצע בעזרת פרוטוקול *TCP*. מערכת הפעלה מנהלת את התקשרות בפועל.



לפני שנוצר חיבור בין *Client*-*Server*, צריך ליצור *socket* כללי שמאזין לבקשת תקשורת. *Client* שרצה להתחבר ל-*Server* צריך לשלוח פקודה שمبיאה לפתח ערוץ תקשורת ביןיהם. *Client*-*Server* יוצר קשר עם *Server*-*Client* באמצעות ייצור *socket* משלו בפרוטוקול *TCP*, המציג את כתובת ה-*IP* ומספר הפורט הרצוי של התהיליך ב-*Server*. כשה-*Server* מקבל בקשה לפתיחת ערוץ תקשורת, נוצר *socket* חדש לטובת התקשרות ביניהם. ה-*socket* הראשי מאשר להאזין לבקשת, בaczera זו ה-*Server* יכול לתקשר עם מספר *Clients*, ובעזרת *Source Port* הוא יודע להבדיל בין *Clients* שmaguiim מאותו מחשב.

Streams

stream הוא רצף של נתונים או בתיים, שנכנס או יצא מתהליך.
 – מידע שמתתקבל. למשל מהמקלדת.
 – מידע שנשלח. למשל מוניטור או socket.
 – מידע שמשלח. למשל output stream

נראה דוגמה לתהליך שרך אצל Client-ה-socket מחוברת למחשב שבו רץ התהליך.

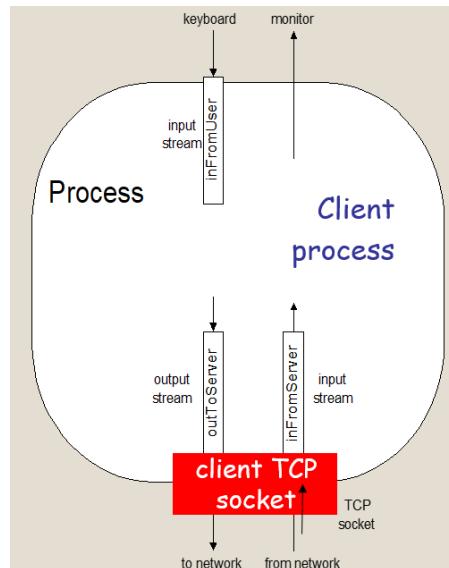
1. קורא שורות מה-*Client* (InFromUser stream) Standart Input-*Client*.

2. משלוחו ל-*server* מה-*socket* (outToServer stream).

3. ה-*server* קורא שורה מה-*Socket*.

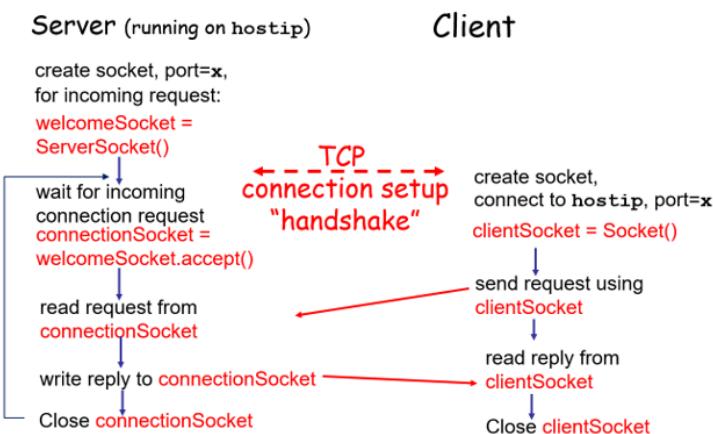
4. ה-*server* הופך את האותיות בשורה ל-Upper Case ושולח אותה בחזרה ל-*Client*.

5. ה-*Client* קורא ומדפיס את השורה מה-*Socket*.



דוגמה נוספת:

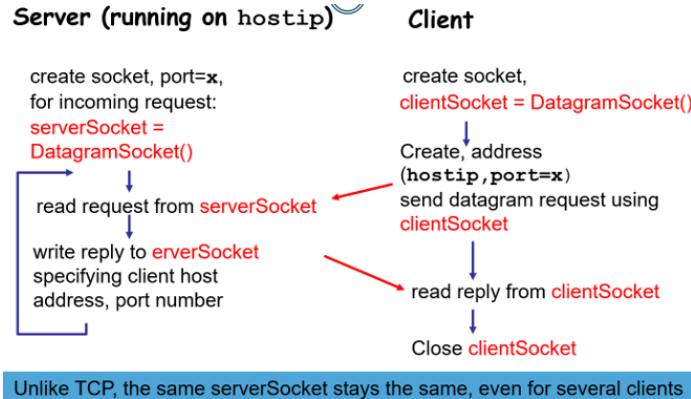
בדוגמה זו ניתן לראות איך Server-ו-Client הפעלים במחשבים שונים, יוצרים Sockets ומתחברים ביניהם לפי פרוטוקול TCP.



דוגמה ליצירת תקשורת בין תחאליכים לפי פרוטוקול UDP:

ב-*UDP אין* "Connection State" בין ה-*Client* ל-*Server*, כלומר:

- אין תקשורת מלאה בין שני המחשבים, השולח לא מודוד שהצד השני קיבל את המידע.
 - השולח מכרף במפורש כתובת IP ופורט של היעד לכל פקטה.
 - ה-*Server* חייב להלץ את כתובת ה-IP והפורט של השולח.
- כפי שנאמר קודם, יתכן שמידע יאביד, ישוכפל, או יגיע בסדר שונה. בדוגמה הבאה ניתן לראות איך *Client* ו-*Server* הפעלים במחשבים שונים, יוצרים *Sockets* ומתקשרים ביניהם לפי פרוטוקול *UDP*.



Socket's Address

:Struct sockaddr

מערכת הפעלה מספקת לנו *struct* *struct sockaddr* לשמר כתובת שאליה אפשר להתחבר. ה-*struct* מכיל את השדות:

- *sa_family* – קובע מה יהיה סוג הכתובת שמוגדרת ב-*struct*.
לרוב נשתמש במקרו *(IP)* *AF_INET*.
- *sa_data[14]* – הכתובת של היעד, ומספר הפורט של ה-*socket*.

```

struct sockaddr {
    unsigned short sa_family;
    char sa_data[14];
};

```

פורמט הכתובת של socket

- *.sockets* – שימוש ב-*UNIX pathname* על מנת לזהות *socket*.
- שימוש ל-*(IPC)* *Inter – process communication*.
- *AF_INET* – שימוש בפרוטוקול אינטרנט *IP*, קלומר *numbers* – *numbers* – 4 byte – *numbers* – *numbers* – *numbers* – *numbers* – *port* – *numbers* – *numbers* – *numbers* – *numbers*.
- המאפשר יותר מ-*AF_INET socket* ייחיד במחשב אחד.

```

struct sockaddr_in {
    short           sin_family;
    unsigned short   int sin_port;
    struct in_addr   sin_addr;
    unsigned char    sin_zero[8];
};

struct in_addr {
    uint32_t   s_addr;
};

```

:struct sockaddr_in

בניגוד ל-*struct* הקודם, כאן נפרק את המידע לפורט, כתובת, *-o sin_zero* – ריפוד כדי שנוכל לעשות *casting* למבנים אחרים (במידה ונctrar).

זה מקל על תהליך חילוץ המידע מה-*sockets*.

הערות על המשתנים:

- ניתן לעשות *casting* בין מצביע של *struct sockaddr_in* למצביע של *struct sockaddr* (כי הסטרuktאים באוטו הגודל).
- *.memSet()* ציריך להיות מאוחחל עם אפסים באמצעות הפונקציה (*sin_zero*).
- *.AF_INET* ציריך להיות מאוחחל עם *sa_family* (*sin_family*).
- *.Network Byte Order* חייבים להיות מסודרים לפי *sin_addr* ו- *sin_port*.

קיימות שני סוגי של Byte Ordering

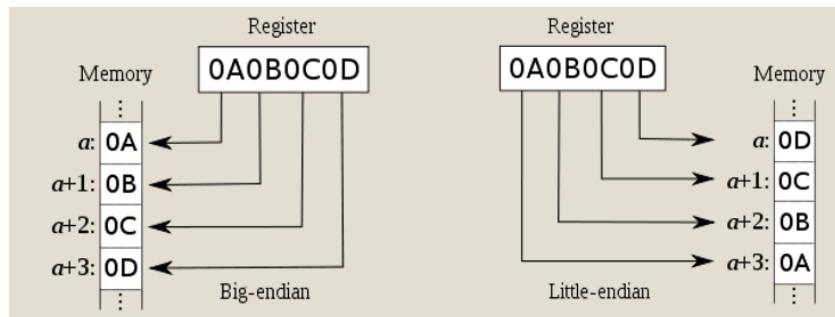
- *Little Endian (Host Byte Order)* – מהבית הימני ביותר לבית השמאלי ביותר.

- *Big Endian (Network Byte Order)* – מהבית השמאלי ביותר לבית הימני ביותר.

במחשבים ביתיים (*hosts*) לרוב משתמשו ב-*Little Endian*.

ורשותן תקשורת לרוב משתמשו ב-*Big Endian*.

הערה: כדי להפוך סדר קראות לארון לאנטוקציית המטרה.



פונקציית ההמרה:

ניתן להמיר בין שני סוגי *short* ו-*long*, להלן פונקציות ההמרה:

- *.Host to Network Short – htons()*
- *.Host to Network Long – htonl()*
- *.Network to Host Short – ntohs()*
- *.Network to Host Long – ntohl()*

לא לשכוח להעביר את הבטים *Network Byte Order* לפני שנעלמה אותם לרשף.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
struct sockaddr_in my_addr;
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(3490);
inet_aton("10.12.110.57", &(my_addr.sin_addr));
memset(&(my_addr.sin_zero), '\0', 8);
```

הfonקציית inet_aton()

- מקבלת כתובת IP V4 בתור מחורזת, כולל 4 מספרים ונקודות ביניהם, ומחליפה אותם למספר בינארי שייצג את הכתובת (לפי *Network Byte Order*).
- בנויגוד לרוב הfonקציות הקשורות ל-socket, הfonקציה זו מחזירה ערך שונה מאפס כשהיא מצלייה, ואפס כשהיא נכשלת.

הfonקציית getpeername()

```
int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);
```

תיאור: מחזירה את הכתובת של הצד השני אליו ה-socket מחובר. (שם העמית)
ערך החזרה: 0 עברו הצלחה, 1 – במקרה של שגיאה.

ארוגומנטים:

- ה-socket FD של ה-socket – מבצע יוזם – מבצע יוזם struct sockaddr – שומר את המידע על הצד השני שמחובר.
- .sizeof(struct sockaddr) – מודד את האורך של addr. נדרש להיות מאוחROL עם addrlen – אם הערך לא גדול מספיק, הfonקציה תגדיל אותו ערך.

Domain Name Service (DNS)

פרוטוקול בשכבה האפליקציה שהמטרה שלו היא לתרגם *domain*³ לכתובת *IP*. הפונקציה *(gethostname()* מחזירה את שם המחשב שהתוכנית רצה עלי (שם של-*host* שMRIIZ את הפוקודה). נרץ את הפונקציה *(gethostbyname(const char *name)* עם הפרמטר - שם המחשב שקיבלו מהפונקציה הקודמת, ונקבל את כתובתה ה-*IP* של המחשב שהתוכנית רצתה עלי. הפונקציה מחזירה מצביע ל- *struct hostent* במקרה של הצלחה, ו- *NULL* במקרה של שגיאה.

```
#include <netdb.h>
struct hostent*
gethostbyname(const char *name);
```

```
struct hostent {
    //Official name of the host
    char *h_name;
    //Alternate names
    char **h_aliases;
    //usually AF_INET
    int h_addrtype;
    //length of each address
    int h_length;
    //network addresses for the host in
    N.B.O
    char **h_addr_list;
};

#define h_addr h_addr_list[0]
```

Struct hostent
זה מכיל:
.host – שם ה-*h_name* –
.***h_aliases* – שמות נרדפים.
.AF_INET – סוג כתובת, לרוב יהיה *h_addrtype* –
. – אורך הכתובת.
.***h_addr_list* – רשימה של כתובות, לרוב נסתפק באיבר הראשון ברשימה.

דוגמה:

בקטע קוד הבא אנחנו מגדרים *struct hostent* שמתתקבל על ידי הפונקציה *gethostbyname*. כמו כן, ניתן לראות שימוש במקerro *h_addr* שמחזיר את האיבר הראשון ברשימה.

```
int main(int argc, char *argv[]) {
    struct hostent *h;
    if (argc != 2) {
        fprintf(stderr, "usage: getip
address\n");
        exit(1);
    }
    if ((h=gethostbyname(argv[1])) ==
NULL) {
        fprintf(stderr, "gethostbyname ");
        exit(1);
    }
    printf("Host name : %s\n", h-
>h_name);
    printf("IP Address : %s\n",
           inet_ntoa(*((struct in_addr *)h-
>h_addr)));
}

return 0;
}
```

³ שם ייחודי של אתר בראשת האינטרנט, שבדיל אותו משאר האתרים הנמצאים בראשת. למשל *google*, *www.google.com*

שלבים להקמת Server חדש – *Socket Programming*

נראה כעת 4 שלבים להקמת תקשורת באמצעות *Sockets*.
שלבים 1-3 הם שלבי אתחול, ושלב 4 מבצע את התקשרות בפועל.

שלב ראשון – יצירת *socket* חדש

כדי ליצור *socket* חדש נשתמש ב:

```
int socket (int domain, int type, int protocol);
```

ה-*domain* מגדיר משפחה של פרוטוקולים מתוך קבועים של המחלקה (*sys/socket.h*). למשל:

- *AF_UNIX* בשביל תקשורת לوكאלית (במקרה זה ה-*protocol* חייב להיות 0)
- *AF_INET* לתקשורת IPv4 (בין רשתות שונות)
- *AF_INET6* לתקשורת IPv6 (לגרסה 6)

ה-*type* מגדיר את הסמנטיקה בה הפרוטוקול משתמש, והוא יכול להיות אחד מה הבאים:

- *SOCK_STREAM* – מציין שהמידע יגיע ל-*socket* כרצף של תווים לפי סדר (*TCP*).
- *SOCK_DGRAM* – מציין שהמידע יגיע ביחידות הנקראות *datagrams*, בלי לוודא שהמידע הגיע (*UDP*).
- *SOCK_RAW* – מאפשר למשתמש להגדיר את הפרוטוקול כרצונו, הוא יקבע איך לפרק לפקודות, איך לשולח, מה לשולח וכו'. לא מומלץ !

ה-*protocol* מגדיר באיזה פרוטוקול *Socket* עובד מ בין משפחת הפרוטוקולים שהגדכנו ב-*domain*:

- לרוב יש רק פרוטוקול אחד במשפחה ה-*socket*, ולכן 0 כארוגמנט.

שלב שני – *Binding an Address*

ניתן ל-*socket* כתובת *IP* ופורט שדרכו הוא יכול להקשיב לבקשתו. נשתמש ב-*system call*:

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

נשים לב שהפונקציה () *socket* מחזירה לנו *FD*, את אותו *FD* שקיבלנו נשלח ל () *bind* בתור הpermeter .*sockfd* הpermeter *const struct sockaddr *addr* מכיל את כל המידע של הכתובת, והpermeter .*struct socklen_t addrlen* מציין את האורך של ה-*addrlen*.

שלב שלישי – *הקשבה*

נרצה להגדיר כמה בקשות לחברים ניתן לקבל במקביל מ-*Clients*. אם נהיה עוסקים בחיבור כלשהו,

בקשת חיבור חדשה תחכה בתור עד שנוכל להתמודד אליה. נעשה זאת באמצעות הפונקציה () *listen*

שמגדירה מה מספר הביקושים המקיים שניית להתמודד אליו במקביל לפני שנדרה בקשות חדשות.

```
int listen (int sockfd, int backlog);
```

הpermeter *sockfd* – כמו קודם, ה-*FD*.

הpermeter *backlog* – כמה *Clients* שיוכולים לוחכות בתור עד שנתחיל לדוחות בקשות חדשות.

דוגמה:

בדוגמה זו נראה איך משתמש בשלושת הצעדים הראשונים שראינו קודם.

```

int establish(unsigned short portnum) {
    char myname[MAXHOSTNAME+1];
    int s;
    struct sockaddr_in sa;
    struct hostent *hp;
    //hostnet initialization
    gethostname(myname, MAXHOSTNAME);
    hp = gethostbyname(myname);
    if (hp == NULL)
        return(-1);
    //sockaddr_in initlization
    memset(&sa, 0, sizeof(struct sockaddr_in));
    sa.sin_family = hp->h_addrtype;
    /* this is our host address */
    memcpy(&sa.sin_addr, hp->h_addr, hp-
        >h_length);
    /* this is our port number */
    sa.sin_port= htons(portnum);

    /* create socket */
    if ((s= socket(AF_INET, SOCK_STREAM, 0)) <
        0)
        return(-1);

    if (bind(s , (struct sockaddr *)&sa , sizeof(struct(
        sockaddr_in)) < 0) {
        close(s);
        return(-1);
    }

    listen(s, 3); /* max # of queued connects */
    return(s);
}

```

שלב רביעי – המתנה לבקשתן:

בשלושת השלבים הקודמים ביצענו פעולות אתחול, בשלב זה נבצע את התקשרות עצמה.

עשה זאת באמצעות הפונקציה:

*int accept(int sockfd, struct sockaddr *cli_addr, socklen_t *cli_addrlen)*

הfonkzia *accept* מחזירה *socket* חדש שמחובר למי שהתקשר.

דוגמה:

```

int get_connection(int s) {
    int t; /* socket of connection */

    if ((t = accept(s,NULL,NULL)) < 0)
        return -1;
    return t;
}

```

The Client

אחרי שיצרנו לנו שם וכתובת, נשימוש בפונקציה *connect()* כדי להתחבר ל-*Server*.

```
int connect (int sockfd ,const struct sockaddr *serv_addr,socklen_t addrlen);
```

הפרמטר *socket* הינו ה-*FD* של ה-*Server*, שאר הפרמטרים הם הפרטים של ה-*Client*.

דוגמה:

נראה דוגמה לחיבור של *Client* ל-*Server*:

```
int call_socket(char *hostname, unsigned short
                portnum) {

    struct sockaddr_in sa;
    struct hostent *hp;
    int s;

    if ((hp= gethostbyname (hostname)) == NULL)
    {
        return(-1);
    }

    memset(&sa,0,sizeof(sa));
    memeply((char *)&sa.sin_addr , hp->h_addr ,
            hp->h_length);
    sa.sin_family = hp->h_addrtype;
    sa.sin_port = htons((u_short)portnum);

    if ((s = socket(hp->h_addrtype,
                    SOCK_STREAM,0)) < 0) {
        return(-1);
    }

    if (connect(s, (struct sockaddr *)&sa , sizeof(sa))
        < 0) {
        close(s);
        return(-1);
    }

    return(s);
}
```

העברת מידע בין סוקטים:

כעת כשחיברנו את ה-*sockets* אחד לשני, נרצה להעביר ביניהם מידע. לשם כך נשתמש בפונקציות *(read ו-write)* שאנו מכירים כבר מעובודה עם קבצים. את המידע נקרא בלולאה (כי קיבל אותו מוחולק לפקודות), ולכן נשמר משתנה *counter* שסופר את מספר הבטים שקראו עד כה. נניח שכמויות הבטים שנרצה לקרוא היא *n*, אז נקרא ערכיהם בלולאה עד ש-*n == counter*. *counter* – *n*.

בדוגמה הבאה, אם *buf > br* סימן שקראו מידע, אך נעדכן את ה-*counter* ואת ה-*buf* (שמיצין לאיפה נרצה שהמידע יכנס)

```
int read_data(int s, char *buf, int n) {
    int bcount; /* counts bytes read */
    int br;      /* bytes read this pass */
    bcount= 0; br= 0;

    while (bcount < n) { /* loop until full buffer */
        br = read(s, buf, n-bcount))
        if ((br > 0) {
            bcount += br;
            buf += br;
        }
        if (br < 1) {
            return(-1);
        }
    }
    return(bcount);
}
```

:Server has multiple FD

כפי שאמרנו, *Server* יכול לעבוד עם מספר *FDs* (*Sockets*):

- אחד (או יותר) שמקשיבים לחיבורים חדשים.

- *FDs* שנוצרו בעזרת הפונקציה *(accept)* על ידי ה-*Socket* שמאזין.

נניח שיש לנו *Socket* אחד שמאזין ועוד 2 *Sockets* שנוצרו בעזרת *(accept)*. אנחנו צריכים מצד אחד להמשיך להאזין ל-*Socket* הראשי ובו זמינות להאזין גם ל-2 הנוספים. נניח שהחלטנו להאזין ל-*Socket* הראשי, ע"י קרייה *accept*. אם אין *Client* שմבקש ליצור חיבור חדש, הפונקציה *(accept)* תרדיד את התהליך שקרה לה עד שיגיע *Client* חדש ומערכות הפעלה תעיר אותו. במקרה זה לא יוכל להאזין ל-2 הסוקטים הנוספים ובפועל לא יוכל לתת שירות ליותר מ-*Socket* אחד בו זמינות.

כדי להתמודד עם הבעיה, ה-*Server* יכול להשתמש ב-*Multiple Threads*. למשל להגדיר *Socket* לכל תהליכיון, וכונראה ליצור גם *Threads Pool* ליחסור במספר הThreads הנדרש, אבל במקרה בו יש יותר סוקטים מתורדים, אנחנו באזזה בעיה כי יכול להיות שכולם הילכו לישון ולא נתקדם לשום מקום. אופציה נוספת היא להשתמש בפונקציה *(select)*, שעלייה גורחיב בעמוד הבא.

הפונקציה `:select()`⁴

```
int select (int nfds, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, struct timeval *timeout);
```

- הפונקציה (`select`) מקבלת רשימה של *FDs* ומבצעת את תהליך ההאזנה אחריהם בשבילנו. היא מרדימה את התהליך שקרא לה ומעירה אותו רק אם:
- יש *FD* חדש שМОן לקריאה / כתיבה.
 - ה-*timeout* (אחד מהארוגומנטים) עבר לפני שהיא שינוי ב-*FDs*.

ארגוניים:

- *nfds* – צריך להיות מאותחל עם מספר ה-*FD* המקסימלי מבין כל ה-*FDs* ששלחנו לו + 1.
 - בכל סט, כל ה-*FDs* שהמספר שלהם קטן מ-*nfds* ייבדקו.
 - סט של *FDs* שהפונקציה צריכה לבדוק אם הם במצב *ready* לקריאה.
 - סט של *FDs* שהפונקציה צריכה לבדוק אם הם במצב *ready* לכתיבה.
 - סט נוסף של *FDs* שלא רלוונטי אליו כרגע.
 - מגדר את הזמן המקסימלי שהתהליך שקרא ל-`select` מעוניין לישון.
- (כלומר כשה-*timeout* עבר, מערכת הפעלה תעיר את התהליך גם אם אף *FD* לא מוכן)

הערות:

- אם אין לנו צורך בעקבות אחריו סט מסוים (למשל ה-*exceptfds*) נשלח *nullptr* במקום
- *FD* נחשב מוכן לקריאה אם הקריאה ל-*read* עלייה לא תחסום את התהליך.

ערך החזרה:

במקרה של הצלחה, יוחזר מספר ה-*FD* שМОן לקריאה/כתביה. אם הגבלת הזמן נגמרה לפני שהיא שינוי עם אחד מהסתוקטים, יוחזר 0. במקרה של שגיאה יוחזר -1.

כמו כן, הפונקציה משנה את הקבוצות שהיא קיבלה ומשאירה בכל קבוצה רק את ה *fds* המוכנים.

כדי ליצור ולתחזק סט של *FDs* נשתמש בפונקציות הבאות:

- *fd_set* – מייצג סט של *FDs*.
- *FD_ZERO(fd_set *fdst)* – מאתחלת את הסט *fdst* להיות קבוצה ריקה.
- *FD_CLR(int fd, fd_set *fdst)* – מורידה מהקבוצה איזשהו *fd*.
- *FD_SET(int fd, fd_set *fdst)* – מוסיף לקבוצה איזשהו *fd*.
- *FD_ISSET(int fd, fd_set *fdst)* – בודקת האם *FD* מסוים נמצא בקבוצה או לא.

⁴ לקריאה נוספת על <https://www.mksssoftware.com/docs/man3/select.3.asp> :`select`

```

MAX_CLIENTS = 30;
fd_set clientsfds;
fd_set readfds;

FD_ZERO(&clientsfds);
FD_SET(serverSockfd, &clientsfds);
FD_SET(STDIN_FILENO, &clientsfds);

While (stillRunning) {
    readfds = clientsfds;

    if (select(MAX_CLIENTS+1, &readfds, NULL,
               NULL, NULL) < 0) {
        terminateServer();
        return -1;
    }

    if (FD_ISSET(serverSockfd, &readfds)) {
        //will also add the client to the clientsfds
        connectNewClient();
    }

    if (FD_ISSET(STDIN_FILENO, &readfds)) {
        serverStdInput();
    }

    else {
        //will check each client if it's in readfds
        //and then receive a message from him
        handleClientRequest();
    }
}

```

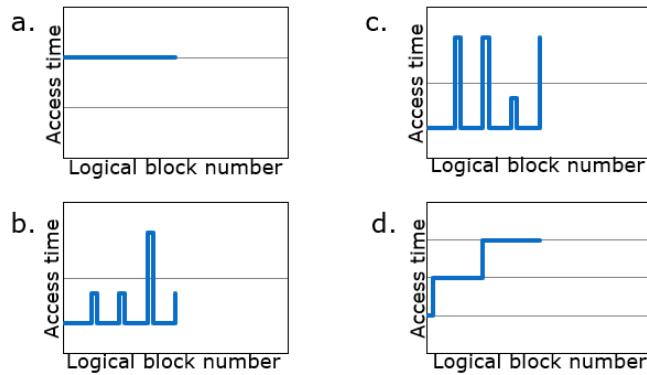
סיכון של Stream Socket

<u>Server Side:</u>	<u>Client Side:</u>
1. <i>socket()</i>	1. <i>socket()</i>
2. <i>bind()</i>	2. <i>connect()</i>
3. <i>listen()</i>	3. <i>read()/write()</i>
4. <i>accept()</i>	
5. <i>read()/write()</i>	

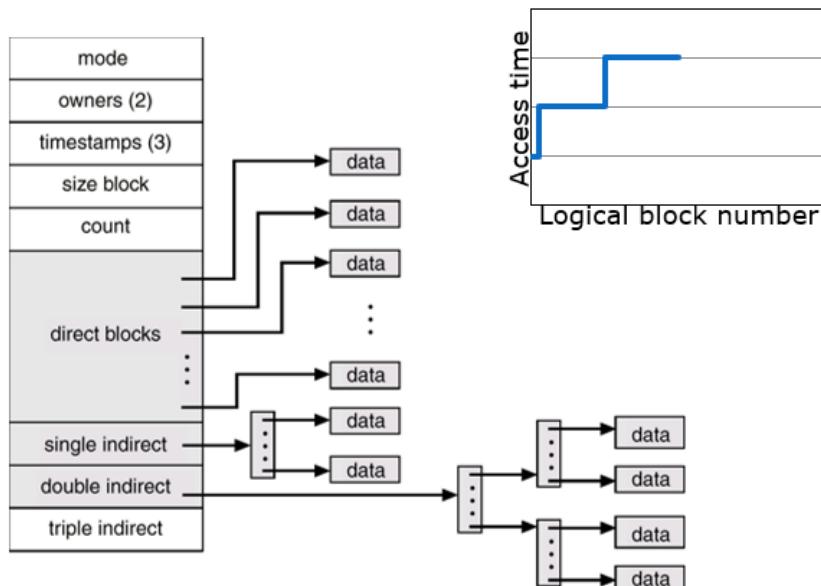
לקריאה נוספת: <https://www.abc.se/~m6695/udp.html>

שאלת הופ-אפ:

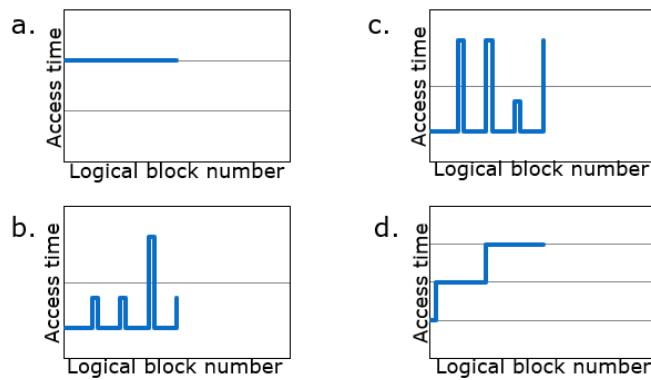
בහינתן מערכת קבצים המשתמשת ב-*nodes – i buffer cache*. ונניח שיש לנו קובץ עם הרבה בЛОקים (לוגים) ממוספרים ... 0,1,2 ... איזה מהగורפים הבאים מתאר היטב הכי טוב את זמן הגישה לבלוק בפעם הראשונה כפונקציה של המספר (הלוגי) של הבלוק?

פתרונות:

נזכיר במבנה של ה-*nodes – i cache*, מכיוון שהנחנו שאין *cache*, ל-12 הבלוקים הראשונים נגיע בהצבעה שירה, לאחר מכן נצטרך לגשת לבלוקים ב-*single indirect* (יכולים להיות 1024 כאלה), ולכל אחד מהבלוקים האלה נדרש שתי גישות לדיסק: הראשונה בשביל ליבא את הבלוק של המצביעים והשנייה בשביל ליבא את הבלוק של *data* (לא נוכל לשמר את הבלוק של המצביעים, כי אין לנו *cache* וכן נדרש ליבא הכל כל פעם מחדש). לאחר מכן נצטרך לגשת עם *double indirect*, ככלומר בכל פעם שנרצה לקרוא *data* נדרש לקרוא בלוק של מצביעים, ולאחר מכן עוד בלוק של מצביעים ורק אז נגיע ל-*data*, וכך התשובה היא **d**.

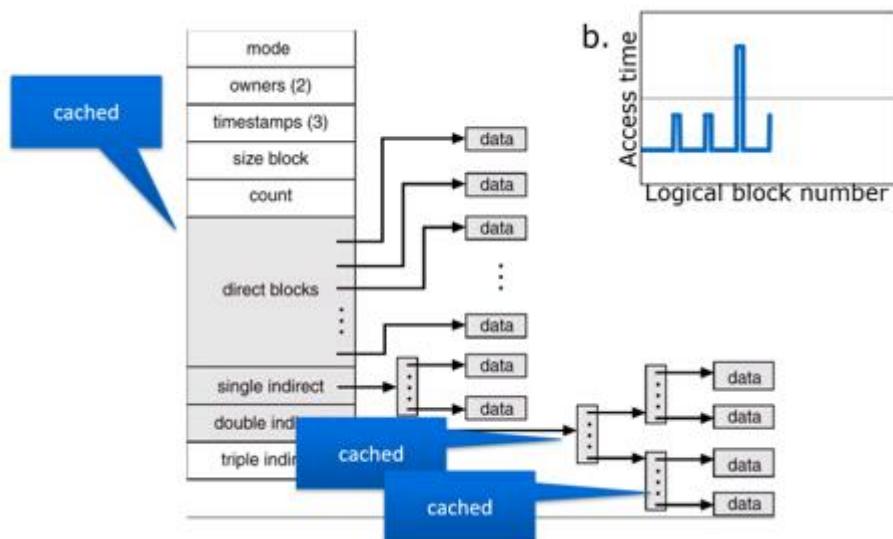


זיכרון מהיר בין הדיסק, *cache*, או כל דבר שנמצא בין ה-*main memory* לבין ה-*disk cache* משמש אותנו לשמר דברים שקרנו לאחרונה מהדיסק, והדיסק משמש מקום לאחסן מידע של הזיכרון הוירטואלי שאין מספיק מקום בשביולו ב-*main memory*. מידע שקרנו לאחרונה מהדיסק (מתוך קבצים) נשמר ב-*buffer cache* ב-*main memory* או ב-*disk cache* (לרוב יכול אלפי בלוקים כדי לחסוך בגישות לדיסק) لكن נוכל לשאול את השאלה מחדש: בהינתן מערכת קבצים המשמשת ב-*i – nodes buffer cache* אינסופי. נניח שיש לנו קובץ עם הרבה בלוקים (לוגים) ממושפרים ... 0,1,2 ... איזה מהגרפים הבאים מתאר היטב הזמן הטוב את זמן הגישה לבlok בפעם הראשונה כפונקציה של המספר (הлогי) של הבלוק?



פתרונות:

כשנרצה ליבא את ה- *data* בפעם הראשונה, נצטרך ליבא גם את ה- *i – nodes cache*, וגם את הבלוק של ה- *data* (המהמורה הראשונה הקטנה). כשנ策טרך ליבא *single indirect* גם נצטרך ליבא פעם אחת את הבלוק הרלוונטי (המהמורה השנייה הקטנה), וכשנ策טרך *double indirect* נצטרך ליבא פעם אחת את הבלוק הגדולה (המהמורה הגדולה), אך התשובה היא *b*.



Open files of a process

בשיעורים קודמים דיברנו על מימוש מערכת קבצים - איך מערכת הפעלה מנהלת את הכתיבה והקריאה מהדיסק. [בעמוד 120](#) למדנו שלכל תהיליך יש גישה לרשימת הקבצים שהוא פתוח, ומערכת הפעלה מנהלת מאחוריו הקלים את כל הקבצים הפתוחים במחשב בעזרת *FDs*. נותר לנו להבין איך מערכת הפעלה מתייחסת לקבצים, איפה היא שומרת אותם, איך היא פותחת קובץ ועוד..

איך פותחים קובץ?

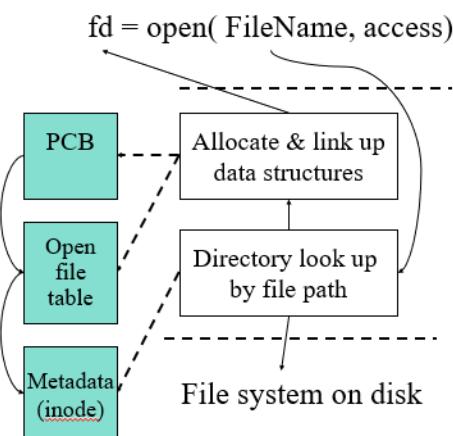
לפני קריאה/כתיבה לקובץ, רוב מערכות הקבצים ידרשו קריאה ל-*(system call) open()*. לאחר הקריאה, מערכת הפעלה תוסיף שורה חדשה ל-*file descriptor table* שהטהיליך מנהל (לכל תהיליך יש טבלה עצמאית, וב-*PCB* של התהיליך יש מצביע לטבלה הזאת), למשל:

	File name	permissions	access date	pointer to disk block
11	Test.c	rw		
12	Mail.txt			

מספר השורה בטבלה הוא ה-*FD* של הקובץ (בדוגמה לעיל, ה-*FD* של *Test.c* הוא 11). בכל פעם שהטהיליך ביצע פעולות על ה-*FD* עם הערך 11, מערכת הפעלה תבין שמדובר ב-*Test.c*. ככלות המילוי אחד פותח את אותו קובץ, יכולות להיווצר בעיות סנכרון - למשל שטהיליך אחד יקרא מהקובץ בזמן אחר כותב אליו, [בעמוד 33](#)ذكرנו את ה-*IPC* שפותר את בעיות הסינכרוניזציה ועביר את האחריות על בעיות הסינכרון לתהיליך עצמו.

מערכות הפעלה כמו לינוקס שומרות לרוב שלוש טבלאות לניהול קבצים: הטבלה הראשונה היא הטבלה שתיארנו לעיל, שיש לכל תהיליך. הטבלה השנייה היא ה-*Open files table* שמנוהלת על ידי מערכת הפעלה ומכליה את כל הקבצים שפותחו כרגע בכל המערכת. כל שורה בטבלה הזאת מייצגת קובץ פתוח של תהיליך מסוים (יכולות להיות כמה שורות לאותו קובץ אם תהיליכים שוניםפתחו אותו במקביל), והטבלה שומרת את ה-*offset* הנוכחי של כל קובץ. הטבלה השלישית היא ה-*node table* – *i*, שם נשמר את המיקום של הקובץ על הדיסק, תאריך גישה, גודל הקובץ, ומספר התהיליכים שפתחו את הקובץ. כנסוים עם הקובץ, ונשתמש ב-*(system call) close()*.

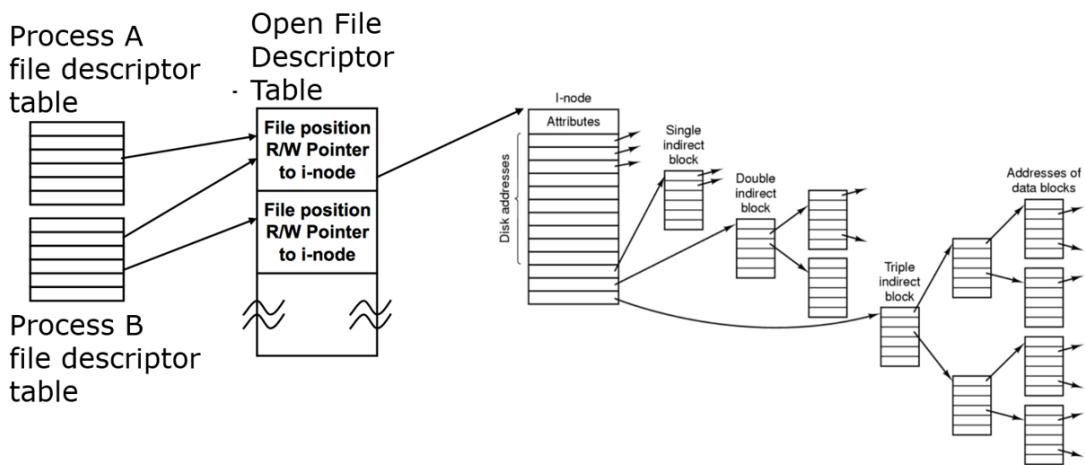
הסביר הנ"ל מתרגם את המימוש של לינוקס, במימושים שונים ניתן לאחד את הטבלאות לטבלה אחת.



לסיכום, כנתבקש לפתיחת קובץ:

1. ניגש ל-*File Path* שקיבלנו.
2. נעתק את ה-*inode* של הקובץ הרלוונטי לטבלה ה-*inode* שנמצאת בזיכרון הראשי.
3. נוסיף שורה ב-*open files table*.
4. נוסיף שורה ב-*file descriptor table*.
5. נזכיר למשתמש את מספר השורה ב-*file descriptor table*.

ונכל לדמות את הטבלאות והתהליכי באופן הבא:



מימוש תיקיות (ספריות) – Implementing Directories

games	attributes
mail	attributes
news	attributes
work	attributes

2 גישות למימוש תיקיות:

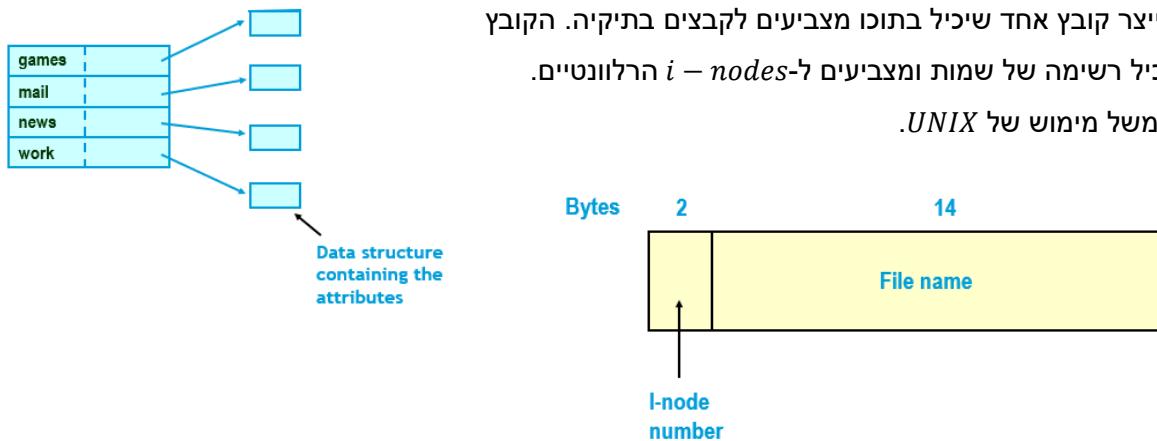
1. נוצר קובץ אחד שיכיל בתוכו את כל המידע על כל הקבצים בתיקיה.

(מיקומים בדיסק, הרשות וכולל..)

למשל מימוש של DOS – MS, פחות נפוץ כיום.

2. נוצר קובץ אחד שיכיל בתוכו מצביעים לקבצים בתיקיה. הקובץ יכול לשמש של שמות ומצביעים *ל-nodes* – *i* הRELONETIM.

למשל מימוש של UNIX.



דוגמיה:

במערכת הקבצים הבאה נרצה לגשת לכתובת `usr/ast/mbox` ולקראא את הבית המילוון.

Root directory	I-node 6 is for /usr	Block 132 is /usr directory	I-node 26 is for /usr/ast	Block 406 is /usr/ast directory
1 .		6 •		26 •
1 ..		1 ..		6 ..
4 bin	Mode size times	19 dick	Mode size times	64 grants
7 dev	132	30 erik	406	92 books
14 lib		51 jim		60 mbox
9 etc		26 ast		81 minix
6 usr		45 bal		17 src
8 tmp				
Looking up usr yields i-node 6				
I-node 6 says that /usr is in block 132				
/usr/ast is i-node 26				
I-node 26 says that /usr/ast is in block 406				
/usr/ast/mbox is i-node 60				

הערה: ניתן לראות באירור שהטהר הראשוני בטבלה הוא המיקום עצמו, מסומן בנקודה,

הטהר השני בטבלה הוא *the root directory* – מסומן בשתי נקודות.

המיקום של *root directory* ידוע, נתחיל ממש את החיפוש:

- נחפש ב-*root directory* את *usr* ונגלה שהוא נמצא ב-*node* – *i* מספר 6.
- נייבא את 6 – *i* – *node* מהדיסק ונגלה שהתוכן של "usr/" נמצא בבלוק 132.
- ניגש לבlok 132, ושם נחפש את "ast". נמצא אותו ב-*node* – *i* מספר 26.
- ב-*node* 26 – *i* נגלה שהתוכן של "usr/ast" נמצא בבלוק 406.
- נחפש בבלוק 406 את *mbox*, ונגלה שהוא נמצא ב-*node* 60 – *i*.
- מכיוון ש-*mbox* הוא קובץ, נרצה להגיע לבית המילוון ב-*node* – *i* שלו.
- נזכר שגודל ה-*node* – *i* הוא 128B, כל בלוק בגודל 4KB, וכל מצביע בגודל 4B.
- קיבל שיש לנו כ- 1000 מצביעים שנמצאים ב-single indirect.
- עד כה היו לנו 5 גישות לזכרון. כמו כן, הגישה ה 6 תהיה קריית ה-*node* – *i* – *data* השביעית – קריית המצביע, והשמינית קריית *data* שם נמצא את הבית המילוון.

מתוך הרצאה 12 שקופה 16

Disk Scheduling

כשתהlixir רוצה לבצע קרייה/כתיבה לדיסק, הוא מועותר על ה-*CPU* ו מעביר את האחריות למערכת הפעלה. מערכת הפעלה שומרת תור של קרייאות לדיסק שעדין לא בוצעו והוא בוחרת איזה פקודה לבצע להלאה. בחירת הפקודה הבאה מבוצעת לפי אחת מהגישות הבאות:

:FIFO

גישה פשוטה למימוש אך גורמת ל-*seek time* גבוהה, כלומר הזרוע שכותבת על הדיסק תצטרך לשנות כיוון הרבה פעמים. מערכת הפעלה חכמה ולכן כתוב בקשות就近ות אחת לשניה ביחד (mbeginet מקום על הדיסק), ובכך למזער את ה-*seek time*.

(Shortest Seek Time First (SSTF))

לפי גישה זו, נבחר את הבקשה הקרובה ביותר למיקום הזרוע על הדיסק (ולומר הבקשה הקרובה ביותר לבקשת האחרונה שבוצעה) ואחתה נבצע. מימוש זה מקטין את ה-*seek time* אבל עלול לגרום להרעה עבור בקשות מסוימות.

:Scan ("elevator algorithm")

גישה זו דומה ל-SSTF למעט כיון התזוזה של הזרוע בדיסק. הזרוע תזוז רק לכיוון אחד ואם בדרך היא הגיעו למידע שרלוונטי לאחת הבקשות, היא תבחר בבקשתה זו. (בדומה למעלית שנמצאת בקומה 2 ורוצה לעלות לקומה 8. המעלית תעוצר בדרך עבור נוסעים בקומות הביניים אבל לא תרד לקומה 1) אחרי שנגיעו לסופי הדיסק נבחר מקום חדש לפי אחת מהגישות הבאות:

- בקצת הדיסק נשנה את הכיוון וначיל מחדש.
- בקצת הדיסק נחפש את הבקשה הרחוקה ביותר וначיל מחדש.

Scheduling

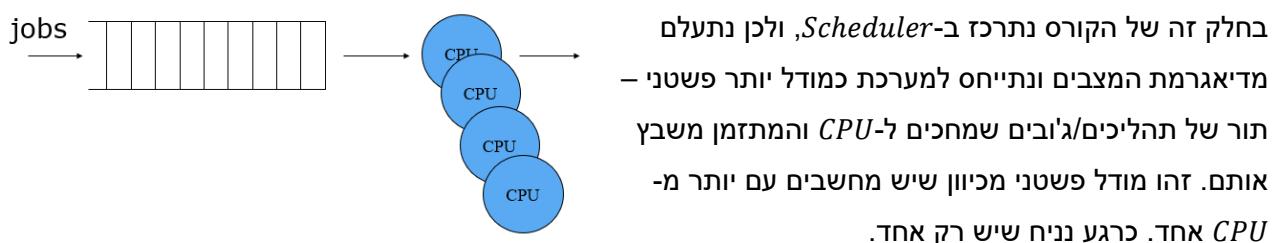
איך מערכת הפעלה בוחרת איזה תהליך או תחביב יירוץ על ה-CPU ברגע נתון? ולכמה זמן? ה-scheduler (מתזמן) הוא כל של מערכת הפעלה שמחלית מי רץ, את מי לעצור וכו'. ראיינו שככל תהליך יכול להיות במצב *running*, *waiting* או *ready*. ברוב המקרים, המתזמן יכול לעצור תהליך רץ ולהעביר אותו ל-*ready*. (כלומר שינוי מערכת הפעלה יзыва ולא התחליר עצמו). מתזמנים מסוג זה נקראים *Preemptive Scheduling*. הם יכולים לשנות מצבים מ-*ready* ל-*running*, ומן-*running* ל-*ready*, אבל הם לא אחראים על כניסה או יציאה ממצב *waiting*.
- *Suspended Ready*, *Trashing* on page 97
- *Suspended Ready* ל-*ready*, *Suspended blocked* ל-*waiting*. המתזמן אחראי גם על מעבר מ-*ready* ל-*blocked*.
את ביצוע הפקידים של מערכת הפעלה נוכל לפרק ל-2, מי שמחלית החלטות (*scheduler*), וממי אחראי על הביצוע (*dispatcher*).
.

:CPU Scheduler

מחלית איזה תהליך יירוץ על ה-CPU (ולפעמים גם לכמה זמן).
- *Short term scheduling*
○ בחירת תהליך מתוך כל התהליכים שנמצאים ב-*ready* והעברתו למעבד (המעבר עצמו מתבצע על ידי ה-*dispatcher*).
○ החלטה על הפסקת ריצה של תהליך שנמצא כרגע במעבד והעברתו ל-*ready* (נקרא גם *preemption*)
- *Mid term scheduler* – אחראי על החלפת מצב *ready* ל-*suspended ready* ווגם על העברה במצב *suspended blocked* למצב *blocked* (וכך ביעם להשפייע על התהליכים שיכולים לרוץ בזמן מסוים ולשפר ביצועים)
תהליכי *scheduler* מתעסק איתם נקראים ג'וביים.

:Dispatcher

המודול שאחראי על ביצוע החלטות של ה-*CPU scheduler*.
אחראי לבצע *context switch* (שינוי המצב ל *user mode* וכו').



⁵ לקריאה נוספת נספתחת: <https://www.tutorialspoint.com/short-term-vs-medium-term-vs-long-term-scheduling>

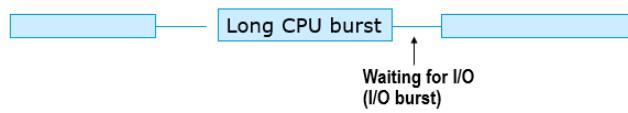
:Job/Process Characterization

נבחין בין שני סוגיים של ג'ובים:

1. *CPU Bound* – ג'ובים שיבזבזו יותר זמן במעבד (למשל עבור חישוב מדעי, חישוב מתמטי וכו').

במקרה זה יהיה לנו *long CPU burst*.

– הזמן שעובר בין שתי פעולות *I/O* כאשר ה-*CPU*עובד.



2. *I/O Bound* – ג'ובים שיבזבזו יותר זמן על פעולות *I/O* (ולא על חישובים).

במקרה זה יהיה *long I/O burst*, *short CPU burst*, כאשר הג'וב מבצע פעולות *I/O*.

– הזמן שעובר בין שתי פעולות *CPU*, כאשר הג'וב מבצע פעולות *I/O*.



:Scheduler

בכל פעם שיש לנו משאב עם יותר משתמש אחד,

נצרך *scheduler* שיחילט מי מקבל את המשאב הראשון. עד כה ראיינו:

- במערכות הפעלה - *CPU scheduler* (לפעמים גם *memory scheduler*, *admission scheduler* ואלא

(admission scheduler

.Printer scheduler -

.Packet Schedulers in a web – server -

.disk scheduler - מהליכת איזה בקשות לשרת קודם מתוכן הדיסק.

באופן כללי משתמשים באלגוריתם זה בכל מקום (אפילו בחימם הפרטיים כמו תור לקופה בסופר).

:מדדיהם של המתזמן (מדדיהם עיקריים)

- $\frac{(\# completed jobs)}{time unit}$ – **Throughput**

אם המערכת יציבה, מספר זה יהיה שווה ל- $\frac{(\# started jobs)}{time unit}$

(כי לאורק זמן מסווג *throughput* הוא קבוע).

- **Turnaround time** – כמה זמן ג'וב מחייב בתוך התור ומסיים את המשימה (זמן המתנה + זמן ריצה). נשים לב שלמתזמן לא תמיד יש שליטה על זמן הריצה של הג'וב.

מדדים משנהים:

- **Response Time** – הזמן שעובר עד שהמשתמש מקבל את התגובה בפעם הראשונה, כלומר הזמן עד שה-*CPU burst* הראשון נגמר.
- **Waiting time** – סה"כ הזמן שగ'וב מחייב ל-*CPU*.
- **CPU Utilization** – אחוז הזמן שבו ה-*CPU* עסוק בדברים "מעוילים" כמו להריצ' דברים (לא נחשב מועל).
- **Fairness** – ניסיון לשמר על הוגנות כלשה' בין הג'וביים השונים.

סתירות בין המטרות:

נשים לב שה-*throughput* האופטימלי ותקבלו כאשר נס'ים ג'וביים מוקדם ככל האפשר כדי להוריד את התקופה, ואילו על מנת לקבל זמן תגובה אופטימלי נרצהلتזמן את ה-*CPU burst* הראשון של כל ג'וב מוקדם ככל האפשר. ככל שנתזמן את ה-*CPU burst* הראשון של יותר ג'וביים, ה-*IRD*, כיוון שנ לצורך לבצע יותר *context switch*. כמו כן, על מנת לסייע ג'וביים מהר ככל האפשר נוצר שכמה שפחות ג'וביים ירצו במקביל – מה שיקטן את זמן התגובה.

מה המתזמן יכול/לא יכול לעשות?

מזכיר כי יש שני סוגים של מתזמים:

- **offline scheduling** – מתzman ש"ודע את העtid", יודיע איזה ג'וביים הולכים להגעה. (למשל מערכות מחזוריות שבן ניתן להסיק מהמחזור הראשון).
- **online scheduling** – מתzman שמקבל ג'וביים אחד אחרי השני מבלי לדעת איזה עשויים להגעה. האם המתzman יודיע את גודל הג'וביים? גודל המשאים שג'וב צרי? וכו'. האם המתzman *preemptive* או *non-preemptive*? (כלומר האם המתzman יכול לעזר ג'וב רץ) לחוב הוא יהיה *preemptive* כיון שלא נרצה שג'וב עם *CPU burst* גדול ירוץ ללא הפסקה ו"יתקע" את כל הג'וביים האחרים.

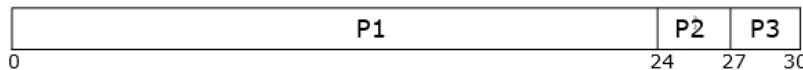
מימושים של Schedulers

First Come First Serve (FCFS)

Job	CPU Burst
P1	24
P2	3
P3	3

בrnehtn מתזמן המקבל את הג'ובים $P1, P2, P3$ אחד בנהנה שאין תקורה על context switch, ובנהנה שלכל ג'וב יש אחד בלבד, מתזמן מסוג זה יכנס את הג'ובים לפי הסדר באופן הבא:

Job	CPU Burst	Turnaround time	Waiting time
P1	24	24	0
P2	3	27	24
P3	3	30	27



$$\text{זמן המתנה ממוצע: } \frac{0+24+27}{3} = 17$$

$$\cdot \frac{3}{30} = 0.1 : throughput$$

- FCFS יש את אפקט השירה - *convoy effect*. ג'ובים קטנים יצטרכו לחכות בגל ג'וב אחד גדול. לו היינו מתזמנים קודם את הג'ובים הקטנים ורק אז את הג'וב הגדל, זמן המתנה היה יורד משמעותית.

পর্টোন: (SJF)

מתזמן מסוג זה מעריכם מסדר ההגעה של הג'ובים ומתזמן אותם לפי הגודל - הג'וב הקצר ביותר ראשון. עבור אותה דוגמה, המתזמן יעבד באופן הבא:

Job	CPU Burst	Turnaround time	Waiting time
P1	24	30	6
P2	3	3	0
P3	3	6	3



$$\text{זמן המתנה ממוצע: } \frac{6+0+3}{3} = 3$$

$$\cdot \frac{3}{30} = 0.1 : throughput$$

נבחן כי throughput לא השתנה כיון ששסימנו את אותם הג'ובים באותו זמן, אבל זמן המתנה הממוצע ירד מ-17 ל-3.

הערות:

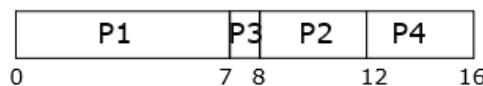
- SJF עובד offline (הוא יודע תמיד מי הג'וב הכי קצר).
- SJF הוא אלגוריתם שיוצר זמן המתנה ממוצע אופטימי.
- (לא קיים אלגוריתם טוב יותר מבחינת זמן המתנה ממוצע).
- כמו כן, גם preemption לא יעזור בקיצור זמן המתנה (כי הוא אופטימי בכל מקרה).

מתי Preemption יכול לעזר?

נסתכל בדוגמה הבאה:

Job	Arrival time	CPU Burst
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

בהתינוק הג'ובים $P1, P2, P3, P4$ כאשר כל אחד מהם מגיע בזמן אחר (כפי שניתן לראות בטבלה) בזמן 0 הג'וב היחיד שהמתזמן קיבל הוא $P1$, וכך הוא שירוץ. $P1$ יירוץ במשך 7 שניות, ובזמן זה נקבל את הג'ובים $P2, P3, P4$. *לא preemption המתזמן לא יכול לעזור את $P1$ ולהריץ את האחרים*, שכן הריצה תרואה באופן הבא (נשים לב שאחרי ש- $P1$ הסתיימ, נבחר את התהיליך עם הזמן הקצר ביותר מבין $P2, P3, P4$).

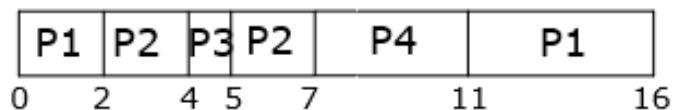


$$\text{זמן ממוצע: } \frac{0+6+3+7}{4} = 4$$

:Preemptive SJF (or SRT)

אם המתזמן *preemptive*, כשמגיע ג'וב חדש עם CPU burst יותר קטן ממנו שנשאר לג'וב הנוכחי, נעצור את הג'וב הנוכחי, ונריץ את הג'וב החדש. עבור אותה הדוגמה, נוכל לתאר את החלטות המתזמן באופן הבא:

Time	Scheduler Decision
0.0	Run P1
2.0	P1 has 5, P2 has 4. Preempt. Run P2.
4.0	P2 has 2, P3 has 1. Preempt. Run P3.
5.0	P3 finishes. Shortest remaining time P2. Run P2
7.0	P2 finishes. Shortest remaining time P4. Run P4.
11.0	Run P1.
16.0	P1 finishes.



$$\text{זמן המתנה ממוצע: } \frac{9+1+0+2}{4} = 3$$

הערה: נשים לב שבמודל זה אנחנו מעריכים מהזמן שלוקח להחליף ג'וביים רצים, יתכן שעלות ההחלפה תהפוך את האלגוריתם *SRT* לפחות משתלם. גם כאן האלגוריתם goede לשפר את זמן המתנה הממוצע והוא לא מתייחס למדדי איכות אחרים. כמו כן, האלגוריתם לא מקיים הוגנות ויכול להיות שיינוי ג'וביים שיורעבו.

תרגול 11

Scheduling

בתרגיל 2 מימושנו את אלגוריתם Round Robin, אבל בפועל יש הרבה אלגוריתמים לניהול ה-CPU:

<i>First come first serve (FCFS)</i>	-
<i>Shortest time first (SJF)</i>	-
<i>Shortest remaining time first (SRTF)</i>	-
<i>Priority scheduling (PS)</i>	-
<i>Round robin (RR)</i>	-
<i>Multi – level queue</i>	-
<i>Multi – level feedback queue</i>	-

נרצה כמובן את האלגוריתם ה"כי טוב", זהה שיופיע בקריטריונים הבאים:

- נרצה לנצל את ה-CPU utilization (max) – נרצה למקסם את שיעור הזמן שבו ישוור (כי הוא תמיד זמין).
- נרצה למקסם את מספר התהליכיים ששווים לרוץ בכל יחידת זמן.
- נרצה למזער את כמות הזמן שתהילך מהכה ל-ready queue Waiting time (min).
- נרצה למזער את כמות הזמן הכללית להרצת תhilic מסויימים. Turnaround time (min)

אין אלגוריתם אחד "הכי טוב", כל אלגוריתם טוב עבור מצבים מסוימים, ולכן על החסרונות / יתרונות של כל אחד.

First – Come, First – Served (FCFS) Scheduling

לפי אלגוריתם זה, התהיליך הראשון שביקש גישה ל-*CPU* יהיה הראשון לקבל אותה.

- האלגוריתם פשוט ביותר ל-*CPU – Scheduling*

- ממומש בעזרת תור *FIFO* (הוספה לסוף התור, הסרת מראש התור)

- הקוד פשוט לכתיבת והבנה

- נסתכל על התהיליכים הבאים:

Process	Burst Time
P_1	10
P_2	1
P_3	1

נניח שהם הגיעו לפי הסדר הבא: P_1, P_2, P_3 , אז ההרצתה של האלגוריתם הנ"ל תיראה כך:



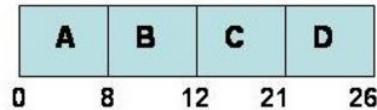
זמן המתנה ל- P_1 הוא 0 שניות, ל- P_2 $10 + cs$ ו- P_3 $11 + 2 cs$ שניות.

זמן המתנה הממוצע הינו $.7 + cs$.

- עבור התהיליכים הבאים:

Process	Burst Time
A	8
B	4
C	9
D	5

ההרצתה שליהם תיראה כך:



ונוכל לבדוק את הקритריונים באופן הבא:

Metric	FCFS
CPU Utilization	$26/(26 + 3cs)$
Turn around time	$((8) + (12 + cs) + (21 + 2cs) + (26 + 3cs))/4 = 16.75 + 1.5cs$
Waiting	$((0) + (8 + cs) + (12 + 2cs) + (21 + 3cs))/4 = 10.25 + 1.5cs$
Throughput	$4/(26 + 3cs)$

- אלגוריתם זה הוא אלגוריתם הוגן. (כי הראשון שהגיע הוא הראשון שקיבל שירות)

- זמן המתנה הממוצע לפחות פעמיים ממש ארוך.

הסיבה לכך היא שתהיליכים קצרים צריכים לחכות שתהיליכים ארוכים יסיימו קודם.

כלומר האלגוריתם בעל ביצועים טובים אם השונות בין התהיליכים היא קטנה.

- אלגוריתם זה הוא *preemptive – non* – כלומר ברגע שתהיליך קיבל את ה-*CPU*,

ה-*CPU* ישרת רק אותו עד שהוא יסימן / ישחרר את ה-*CPU* מרצונו.

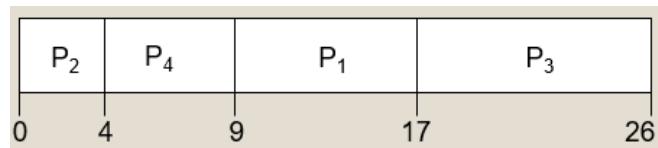
Shortest – Job – First Scheduling

כשה-*CPU* פנו, התהילה קצר ביותר יהיה הבא בתור להיכנס אליו.

- נקרא גם (*SJN*)
- נשתמש ב-*FCFS* (האלגוריתם קודם) במקרה של שווין.
- גם אלגוריתם זה הוא *non-preemptive*.
- אלגוריתם אופטימלי – ממוצע הזמן שההיליך צריך לתוכו הוא הקטן ביותר האפשרי.
- בעיות אפשריות:
 - לעיתים לא ידוע מה זמן הריצה שייקח לכל תהליך.
 - לא אלגוריתם הוגן, יכול לגרום להרעה.
- נסתכל על התהליכיים הבאים:

Process	Burst Time
P_1	8
P_2	4
P_3	9
P_4	5

נניח שclock הגינו ביחד, וכך ההריצה שלהם לפי האלגוריתם הנ"ל נראה כך:



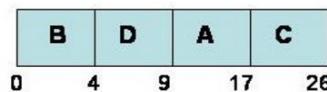
זמן ההמתנה ל- P_2 הוא 0 שניות, ל- P_4 4 שניות, ל- P_1 9 שניות ול- P_3 17 שניות.

זמן ההמתנה הממוצע הינו $\frac{0+4+9+17}{4} = 7.25$. (לעומת $\frac{0+4+9+17}{4} = 10.25$ עם *FCFS*)

- עברו התהליכים הבאים:

Process	Burst Time
A	8
B	4
C	9
D	5

ההריצה שלהם תיראה כך:



ונוכל לבדוק את הקритריונים באופן הבא:

Metric	FCFS
Utilization	$26/(26 + 3CS)$
Turn around time	$(4 + 9 + CS + 17 + 2CS + 26 + 3CS)/4 = 14 + 1.5cs$
Waiting	$(0 + 4 + CS + 9 + 2CS + 17 + 3CS)/4 = 7.5 + 1.5cs$
Throughput	$4/(26 + 3CS)$

Shortest Remaining Time First (SRTF)

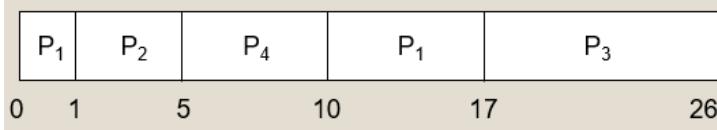
Shortest Remaining Time רק

- גרסה פרימיטיבית של SJF. אם תחילר מגע עם CPU burst באורק קצר יותר מהזמן שנותר לתחילר הנוכחי לרווח, נקדם אותו. תחיליכים קצרים מנהלים בצורה זריזה.
- אותן בעיות כמו SJF.

נסתכל על התהליכים הבאים:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

הפעם יש לנו גם זמני הגעה, ולכן ההערכתם לפי האלגוריתם הנ"ל נראה כך:



$$\text{זמן המתנה הממוצע הינו } \frac{(10-1)+(1-1)+(17-2)+(5-3)}{4} = 6.5$$

Priority Scheduling

באלגוריתם זה, ה-CPU מוקצה לתחילר עם העדיפויות הגבוהה ביותר.

- לכל תחילר יש קידימות
- מספר מותruk טווח קבוע, למשל 0 עד 7
- מספרים קטנים מסומנים קידימות גבוהה
- תחיליכים עם קידימות שווה ייבחרו לפי FCFS.
- SJF הוא מקרה פרטי של האלגוריתם זהה, כשהקידימות היא לפי זמני ריצה נמוכים
- נסתכל על התהליכים הבאים:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

הפעם יש לנו קידימות, ולכן ההערכתם לפי האלגוריתם הנ"ל נראה כך:



$$\text{זמן המתנה הממוצע הינו } \frac{6+0+16+18+1}{5} = 8.2$$

- קידימות של תחילר יכולה להיבחר באופן פנימי, למשל לפי כמות זיכרון נדרש וכו' או חיצונית לפי חשיבות וכו'.
- אלגוריתם זה יכול להיות Non – Preemptive ויכול להיות Preemptive – הגדרת עדיפות לתחילר שחיכה הרבה זמן.
- יכול לגרום להרעה. הפתרון יכול להיות aging – הגדלת עדיפות לתחילר שחיכה הרבה זמן.

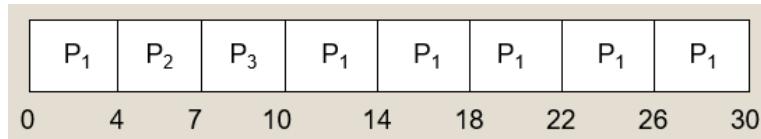
Round – Robin Scheduling

עבור אלגוריתם זה, נגיד ייחידת זמן קצרה בשם *time quantum*. (בדרכן כלל 100 – 10 מילישניות) ה-*queue ready* ממומש *CPU*-*FIFO*, והאלגוריתם הנ"ל יתייחס אליו כרשימה מעגלית. הוא יעבור על ה-*ready queue* ויקצה את ה-*CPU* לכל תהליך ל-*quantum* 1 זמן לכל היותר. עבור כל תהליך יתבצע 2 מקרים:

- ה-*CPU burst* נמשך יותר מ-*quantum* 1 (התהליך שחרר את ה-*CPU* מרצונו)
- ה-*CPU burst* ארוך יותר מ-*quantum* 1 (*context switch* יזום)
- אלגוריתם זה הוא *preemptive*.
- נסתכל על התהליכי הבאים:

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

אם ה-*quantum* הוא 4 מילישניות, ההערכתה של האלגוריתם הנ"ל נראהthus כך:



$$\text{זמן המתנה הממוצע הינו } \frac{6+4+7}{3} = 5.66.$$

אם *quantum* = 20 אז הריצה תיראה כך:

<u>Process</u>	<u>Burst Time</u>
P_1	53
P_2	8
P_3	68
P_4	24

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3	
0	20	28	48	68	88	108	112	125	145	153

זמן המתנה של כל תהליך הם:

$$P1: (68 - 20) + (112 - 88) = 72$$

$$P2: (20 - 0) = 20$$

$$P3: (28 - 0) + (88 - 48) + (125 - 108) = 85$$

$$P4: (48 - 0) + (108 - 68) = 88$$

זמן הריצה הם: $P1: 125, P2: 28, P3: 153, P4: 112$

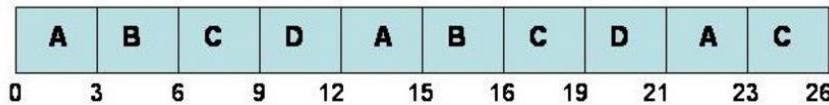
$$\text{זמן המתנה הממוצע הינו: } \frac{72+20+85+88}{4} = 66.25$$

$$\text{זמן הריצה הממוצע הינו: } \frac{125+28+153+112}{4} = 104.5$$

- נראה דוגמה נוספת. אם $quantum = 3$, עבור התהיליכים הבאים:

Process	Burst Time
A	8
B	4
C	9
D	5

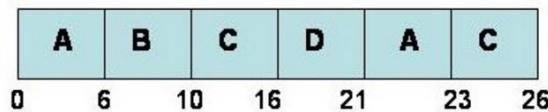
ההרצה של האלגוריתם תיראה כך:



ונוכל לחשב את הביצועים באופן הבא:

Metric	RR
Utilization	$26/(26 + 9CS)$
Turn around time	$(23 + 16 + 26 + 21 + 29cs)/4 = 21.5$ ignoring CS
Waiting	$(15 + 12 + 17 + 16 + 29cs)/4 = 15$ ignoring CS
Throughput	$4/(26 + 9CS)$

אם $quantum = 6$ אז ה嚮ה תיראה כך:



FCFS vc RR

בנהנזה ש-*context switch* לא לוקח זמן בכלל, האם *RR* תמיד טוב יותר מ-*FCFS*.
נניח שיש לנו 10 עבודות, כלן מתחילה באותו הזמן וכל אחת דורשת 100 שניות *CPU*.

Job #	FCFS CT	RR CT
1	100	991
2	200	992
...
9	900	999
10	1000	1000

אם ה-*quantum* של *RR* הוא 1 שניות, אז *FCFS* ו-*RR* יסימנו באותו הזמן,

אבל זמן התגובה הממוצע של *RR* גרווע בהרבה מ-*FCFS*.

RR גרווע כשלשנות בין העבודות הוא קטן (כלון באותו אורך למשל), אבל הוא טוב במציאות.

בהתוואה הנ"ל הנחנו שה-*context switch* לא לוקח זמן, אבל בפועל הוא לוקח זמן ובנוסף צריך לשתף את מצב ה-*CPU* בין כל העבודות, וכך *RR* לוקח יותר זמן אפילו עבור *context switch* אףו!

פירוט נספָּה:

	P_2 [8]	P_4 [24]	P_1 [53]	P_3 [68]		
	0	8	32	85	153	
Wait Time	Best FCFS	32	0	85	8	$31\frac{1}{4}$
	$Q = 1$	84	22	85	57	62
	$Q = 5$	82	20	85	58	$61\frac{1}{4}$
	$Q = 8$	80	8	85	56	$57\frac{1}{4}$
	$Q = 10$	82	10	85	68	$61\frac{1}{4}$
	$Q = 20$	72	20	85	88	$66\frac{1}{4}$
	Worst FCFS	68	145	0	121	$83\frac{1}{2}$
	Best FCFS	85	8	153	32	$69\frac{1}{2}$
Completion Time	$Q = 1$	137	30	153	81	$100\frac{1}{2}$
	$Q = 5$	135	28	153	82	$99\frac{1}{2}$
	$Q = 8$	133	16	153	80	$95\frac{1}{2}$
	$Q = 10$	135	18	153	92	$99\frac{1}{2}$
	$Q = 20$	125	28	153	112	$104\frac{1}{2}$
	Worst FCFS	121	153	68	145	$121\frac{3}{4}$

הכוונה שהעובדות הגיעו לפני הסדר מהעובדות הקצרה ביותר לארכאה ביותר. ניתן לראות שזמן הריצה והמתנה הממוצעים של *RR* נמצאים בערך זה המקורה הפוך. *Worst FCFS* בין *Best FCFS* לבין *Worst FCFS*.

Multilevel Queue Scheduling

רעיון האלגוריתם:

- נחלק את ה-*ready queue* למספר קבוצות נפרדות (למשל *Interactive, Batch, Student* וכו')
 - לכל קבוצה יהיה אלגוריתם *Scheduling* משללה
 - נבצע *Scheduling* גם בין הקבוצות עצמן כדי לבחור מבין כל העבודות.
- אם ה-*Scheduling* בין הקבוצות עצמן הוא יעדיף
 - קבוצה מסוימת לעומת קבוצות אחרות) אז ייתכן מצב של הרעבה.
 - אפשרות אחרת היא לבצע *time – slice* בין התורים (כלומר להחליף ביניהם לפי זמן)

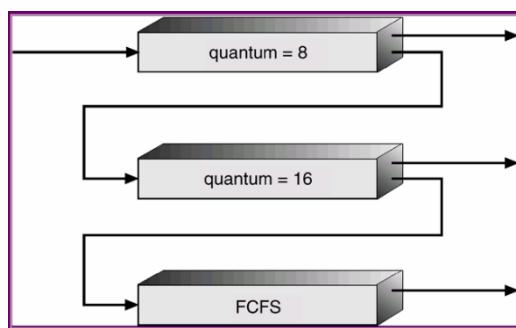
Multilevel Feedback Queue

רעיון האלגוריתם:

- דומה לאלגוריתם הקודם אך כאן בכל פעם שתהיליך מגע לסוף התור, נרייך פונקציה שתחזיר לנו *feedback* עדכני על התהיליך, ובהתאם נבחר לאיזה תור לשירותו. ככלمر תהיליך יכול לעבור בין תורים.
- למשל אם תהיליך מסוים משתמש ב-*CPU* יותר מדי זמן, נעביר אותו לתור עם עדיפות נמוכה יותר. לחילופין, אם תהיליך ממתין יותר מדי זמן ל-*CPU*, נעביר אותו לתור עם עדיפות גבוהה. (בצורה זו אנחנו מונעים הרעה)
- האלגוריתם כולל נקבע לפי הפרמטרים הבאים:
 - מספר התורים
 - אלגוריתם *the-g-Scheduling* שיש לכל תור
 - שיטת עזר שמחשבת לאיזה תור להכניס תהיליך שימושי בשירות
 - שיטת עזר שבודקת אם לשנמר את העדיפויות של התהיליך
 - שיטת עזר שבודקת אם לשדרג את העדיפויות של התהיליך

נראה דוגמה:

- עבור 3 תורים: Q_1, Q_0, Q_2 , שכולם בוחרים תהיליכים מתוכם בעורת *FCFS*
- נגידר שתהיליך מסוים מ- Q_0 יקבל הזדמנות לסיים תור = *quantum* 8 אם הוא לא סיים תור 8 יחידות זמן, נעביר אותו ל- Q_1 .
- לאחר שיגע תורו שוב, יש לו הזדמנות נוספת לסיים תור 16 יחידות זמן.
- אם גם זה לא הספיק לו, התהיליך יעבור ל- Q_2 שם יש *FCFS* ג'il ללא הגבלת זמן.



בדוגמה הזאת אנחנו נותנים לתהיליכים מהירים הזדמנות לסיים מהר, ובכך אנחנו מנסים להתקדם לאלגוריתם האופטימלי *Shortest – Job – First Scheduling* גם אם זמני הריצה לא ידועים מראש.

עד עכשיו ערכנו על אלגוריתמים של *CPU Scheduling*
נעבור לדבר אחד על *Parallel Systems Scheduling*

Parallel Systems Scheduling

במחשי עלי וחוות שרתים יש המון שרתים שMRIIZIM המון משימות במקביל.
מרכז Scheduler שיבצע את הסyncron בין המשימות לשרתים.

מחשי עלי: (ויקיפדיה) תחילת התיכון הגדרה זו למחשי בעלי מעבד יחיד מהיר במינוח, לאחר מכן נכללו בה מחשי שמהירותם מושגת באמצעות מעבדים רבים הפועלים במקביל לביצועה של משימה אחת, וכעת מרכיבים מחשי-על מאשכולות של מחשי מרובי מעבדים.

- מחשי עלי מספקים מהירות גבוהה מאוד לחישובים (HPC)
- לכל מחשב יש מערכת הפעלה משלהו
- המחשיים מחוברים ביניהם ברשת מהירה
- המחשיים מריצים המון משימות במקביל
- כל משימה מכילה את מספר המעבדים/מחשיים הנדרש עבורה
- כל משימה ממליצה על CPU תשמש בו עד שהיא מוגדרת עליי מרצונה



ה-Scheduler של מחשי עלי:

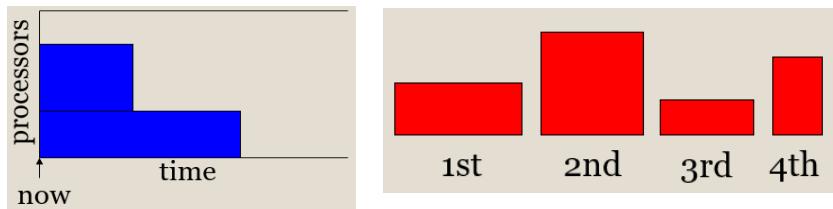
- מטרת Scheduler היא לזרז איזה מעבדים יריצו כל משימה.
- למשל FCFS, SJF, Backfilling ועוד.
- בשוקופיות 33-34 במצגת של תרגול 11 יש דוגמת הרצה עם Backfilling-FCFS.

The Easy Scheduler

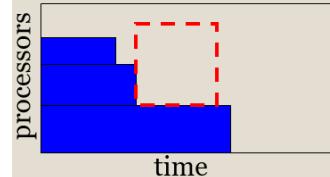
- אלגוריתם פשוט אך נפוץ מאוד
- אלגוריתם הוגן (יחסית)
- משתמש ב-Backfilling לפי זמני הגעה של העבודות

Easy Data Structures

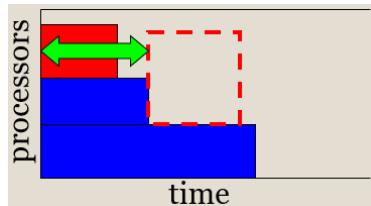
- נצרך לשמר רשימה של שימושות שרצות כרגע,
- כולל מספר המעבדים שהם תופסים ומתי זמן ה批示 המשוער שלהם. (בכחול)
- נצרך גם רשימה של שימושות שemmתיינות בתור (mmoינת לפי זמן הגעה),
- כולל מספר המעבדים שהם צריכים, מה זמן הricsה המשוער שלהם. (באדום)



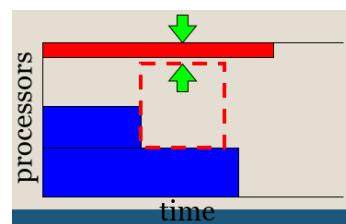
- ה-*Scheduler* יחלק שימושות לprocessors פנויים לפי *FCFS*
- ברגע שיגיע למשימה שאין מספיק מעבדים להריצ' אותה, הוא ישמור לה מקום,
- וימשייר לשbez משימות קטנות יותר כדי לנצל את כל המעבדים שנשארו. (זה *Backfilling*)



- משימה שתיבחר על ידי *Backfilling* היא משימה שעומדת לפחות אחד מהתנאים הבאים:
- המשימה צריכה לרוץ לפני המשימה הגדולה שהמירה לה מקום.



- המשימה יכולה לרוץ בעזרת מעבדים אחרים, שהמשימה ששרינו לא תצטרך.



- בשקופיות 39 – 37 יש אילוסטרציה שלמה של התהילה.

Runtime Estimates

- כמשמעותם מראים משימות חדשות, הם מספקים:
- את מספר המעבדים הנדרשים למשימה
- זמן ריצה משוער של המשימה
- זמני ריצה משוערים נדרשים כדי לנברא מתי מעבד מסוים יתפנה למשימות חדשות,
- או לטובת *Backfilling* (כדי לוודא שהמשימה תסתיים לפני המשימה ששרינו)

תרגול 12

תרגול זה יהיה תרגול שלמה מושגאים שלא העמכו בהם מספיק במהלך השנה.

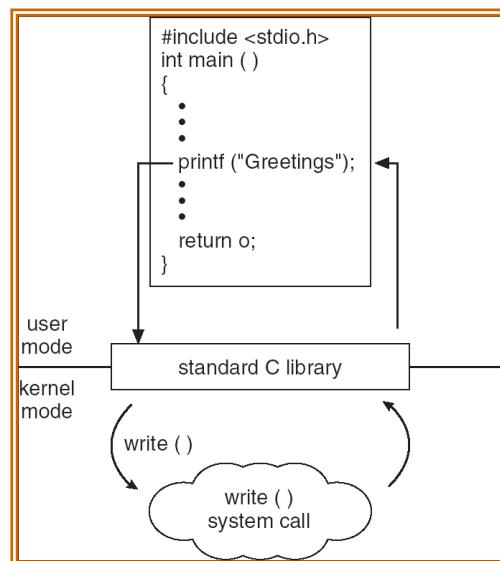
System Calls

דיברנו על [open on page 11](#) פונקציות כמו () שבעמודים קודמים התייחסנו אליהן כפונקציות מערכת (system calls), הן בפועל פונקציות עוטפות לפונקציות המערכת האמיתיות. לכל פונקציה מערכת יש מספר שימושי אליה. בקריאה לפונקציה העוטפת (למשל () open), המימוש הפנימי של הפונקציה כתוב במקום מסוים בזיכרון את המספר שימושי לפונקציית המערכת הרלוונטית, ותבצע trap (נזכר ש- *trap* זה *interrupt*) כשהשליטה תעבור למערכת הפעלה, היא תקרה מהזיכרון את המספר שהfonקציה שמרה, וכך היא תדע איזה handler המשתמש רוצה להריץ. (דוגמה שלנו – פтиחת קובץ) מערכת הפעלה תחסם (במידת הצורך) את התהיליך, וכשיגיעו תוצאות (למשל ה- *fd* של הקובץ הרצוי), היא תרשום את הפלט באזור מסוים בזכרון. כשהשליטה ב-CP-U תחזור לתהיליך, הפונקציה העוטפת תדע למשוך ממש את התוצאות המתאימות.

למערכת הפעלה 2 תפקדים בתהיליך הנ"ל. מצד אחד היא מספקת שירות Kernel, ומצד שני היא מספקת ספרייה Higher Language Interface שדרוכה לאפשר להשתמש בפונקציות המערכת. הפונקציות בספרייה זו מתבצעות ב-*user mode*, ניתן לקרוא להן כמו לכל פונקציה רגילה והן אלו שמבצעות את ה- *trap* כפי שהסביר לעיל.

דוגמה – קראיה לפונקציה printf:

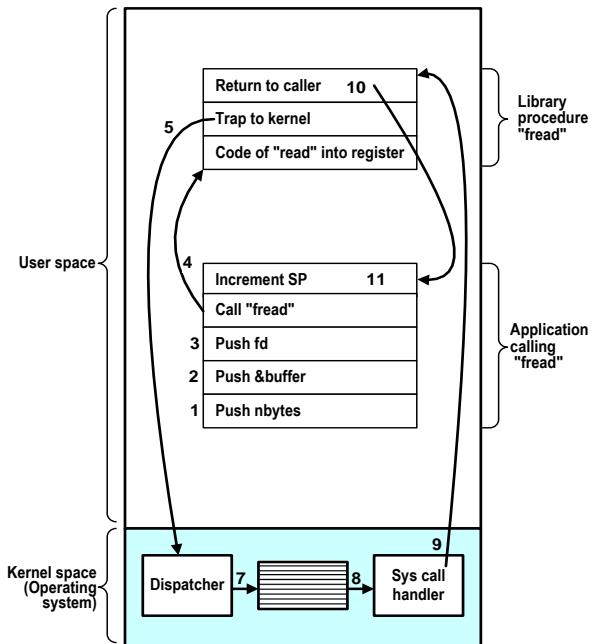
בקראיה לפונקציה printf (מתוך הספרייה הסטנדרטית של C), הפונקציה תבצע קראיה ל- *write*, שהיא פונקציית מערכת שמבצעת את פעולה ההדפסה בפועל (כתיבה לקובץ *stdout*). בסיום הריצה, התהיליך יקבל דיווח על הצלחה או כשלון בהתאם.



דוגמה נוספת – קרייה לפונקציה (`:fread(fd, buffer, nbytes)`)

נקוב בתמונה משMAIL לפי המספרים:

- ב-3 הצעדים הראשונים נדחוף את הארגומנטים למחסנית.
- נקרא לפונקציה (`fread`), ונתען את הקוד של `read` לריגיסטר.
- הפונקציה העוטפת תשליח `trap` למערכת הפעלה.
- מערכת הפעלה תנסה את המצביע *kernel mode* ל-*handler*.
- ותשתמש ב-*dispatcher* על מנת לחלץ מהרגיסטר את הקוד שצריך לבצע.
- מערכת הפעלה תריד את ה-*handler* של פונקציית המערכת הרצiosa, ותרשם את התוצאות במקום מסויים בזכירון.
- המצביע יחזיר ל-*user mode*.
- הפונקציה העוטפת תקרה מהזיכרון את התוצאות ותחזיר אותן לפונקציה הקוראת (הfonקציה של האפליקציה).



סוגי ה-API הנפוצים ביותר: API standards

אם ה-API יהיה אחד בין מערכות הפעלה, יוכל להריץ תוכניות שונות במערכות הפעלה שונות בקלות. בפועל, ישנו מספר סוגים של API:

- **API – POSIX** – פונקציות של רוב מערכות הפעלה מבוססות UNIX (לינוקס, Mac OS X).
- **API – Win32/Win64** – פונקציות של Windows.
- **API – Java API** – עבור Java virtual machine (JVM) (לא רלוונטי לקורס שלנו).

הערה: הפונקציות הן פונקציות מערכת (*system calls*) שמערכת הפעלה מספקת למשתמש.

סוגים של פונקציות מערכת:

- *system calls* של ניהול תהליכיים ותקשורת בין תהליכים (IPC).
- *system calls* של ניהול זיכרון (malloc, free).
- *system calls* של גישה וניהול קבצים.
- *system calls* של התקנים (devices).

דבר על הסוגים השונים במהלך התרג oligo.

מאפיינים של תהליכיים:

- לתהליך יש "אבא" יחיד (התהליך שיצר אותו), אבל יכולם להיות לו הרבה ילדים (תהליכים שהוא יצר).
- תהליך שנוצר יורש את הסביבה של "באב" שלו, כולל כשהוא נוצר הוא שכפול של האבא, ורק אחר כך הוא משנה את מה שצריך כדי להתאים את עצמו למצב החדש.
- כל תהליך שנוצר מתחילה עם 3 קבצים פתוחים: *stdin, stdout, stderr*.
- לכל תהליך יש מספר ייחודי הנקרא (*pid*) *process id* (*pid*).

הפונקציה (*pid t getpid(void)*):

getpid היא פונקציה שיזכרת תהליך חדש. התהליך החדש יהיה שכפול כמעט מדויק של האבא,

מלבד מספר מאפיינים שהיו שונים (כמו למשל *pid*). הקוד של התהליך החדש ושל אבא שלו יהיה

אותו קוד, ככלומר שני התהליכים יבצעו את השורה שבאה אחרי הקראיה ל-*(fork)*.

אפשר לדעת האם אנחנו בתהליך האבא או הבן באמצעות ערך ההחזרה של *(fork)*.

- ערך החזרה 0 – התהליך שנוצר מ-*(fork)* רץ.

- ערך החזרה גדול מ-0 – האבא רץ, ערך ההחזרה הוא *the pid* של התהליך החדש (הבן).

הפונקציה (*void exit(int status*):

exit היא פונקציה שמסיימת ריצה של תהליך. הפונקציה דואגת לעשות *flush* לכל הקבצים הפתוחים

(כליומר אם היה *buffer* לאחד הקבצים, היא תכתוב את תוכן שלו במקום המתאים),

וגם לסגור אותם אחר כך. אם הוקצתה זיכרון, היא תשחרר אותו. ה-*status* (קוד יציאה) יהיה זמין לתהליך

הבא (למשל אם במהלך הריצה ביצענו את הפקודה *exit(1)*, נראה בטרמינל שהתהליך הסתיים עם

קוד יציאה 1). הפונקציה (*exit()* סוגרת את התהליך כולו ו开会 אין משמעות לערך החזרה. (היא *void*)

הפונקציות ⁶:*wait, waitpid*

*pid = wait(int *stat_loc); pid = waitpid(pid_t pid, int *stat_loc, int options)*

הfonקציות הנ"ל ממתינות (מרדיימות את התהילך שקרה להן) עד שאחד הילדים של התהילך יסימם את הריצה שלו, ומחייבות את הסטטוס שהילד סימן אליו (סטטוס = ערך החזרה בסוף הריצה).

הfonקציה ()*wait* מקבלת מצביע ל-*int* (*שם ישמר הסטטוס*), ומחייבת שילד כלשהו יסימן.

הfonקציה ()*waitpid* מבצעת את אותה פעולה אבל עבור ילד ספציפי,

היא מקבלת כקלט *pid* כלשהו ומחייבת שהילד הספציפי זהה יסימן את הריצה שלו.

במצב הדיפולטיבי, ()*waitpid* מחייבת שהטהילך עם ה-*pid* הנוכחי יסימן, אבל אפשר לשנות בעזרה הארגומנט השלישי *options* שהוא תחכה עד שהוא אחר. (למשל לחכות עד שהילד ישלח סיגנל מסוים)

הfonקציות ⁷:*execl, execle, execlp, execve, execvp*

משפחחה של פונקציות שמאפשרות לשנות את הייעוד של התהילך מסוים. אמינוו שתהילך בן נוצר שכפוף של האבא, אבל מה אם נרצה שהבן יירץ קוד אחר? בשביל זה נוכל להשתמש בfonקציות הנ"ל.

כל הפונקציות מבצעות את אותו רעיון, אך הן מקבלות ארגומנטים שונים. למשל הפונקציה

```
int execl(const char *path, const char *arg0, ...);
```

מקבלת נתיב במחשב לתוכנית שאפשר להריץ אותה (*path* (*)), וARGINטמים נוספים.

כל השורות בקוד שבאותו אחרי *execl* לא יבוצעו, והתוכנית שנמצאת בנטייה הנוכחי היא זו שתורוץ.

במקרה של הצלחה, הפונקציה לא צריכה להחזיר כלום (כי היא מחליף את כל התוכנית),

ואם יש ערך החזרה (המשמעותו עם הקוד המקורי) סימן שהיאאה בהרצת הפונקציה. דוגמת הריצה:

```
execl( "/bin/ls", "ls", " -l ", NULL);
```

דוגמאות: ניצור תהילך חדש באמצעות ()*fork*, נודא שאין שגיאות, ואז נבדוק האם אנחנו בתהילך האבא או בתהילך הבן. אם אנחנו באבא, נחכה שהבן יסימן (באמצעות ()*waitpid*) (באמצעות ()*fork*).

נשים לב שבסוף יש *return* שרק האבא יכול להגיד אליו.

```
int main(void)
{
    pid_t pid;
    pid = fork();
    if (pid == -1) {
        /* When fork() returns -1, an error happened.
        */
        perror("fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        /* When fork() returns 0, we are in the child process.
        */
        printf("Hello from the child process!\n");
        _exit(EXIT_SUCCESS); /* exit() is unreliable here, so _exit must be used */
    } else {
        /* When fork() returns a positive number, we are in the parent process
        * and the return value is the PID of the newly created child process.
        */
        int status;
        (void)waitpid(pid, &status, 0);
    }
    return EXIT_SUCCESS;
}
```

⁶ לקריאה נוספת <http://man7.org/linux/man-pages/man2/wait.2.html> :*wait, waitpid*

⁷ לקריאה נוספת <https://linux.die.net/man/3/execlp> :*exec*

דוגמה נוספת:

בדומה לדוגמה הקודמת, נבדוק את ערך ההחזרה של `fork()` ונפעל בהתאם.

- אם ערך ההחזרה הוא 0 (כלומר אנחנו בתהילר הבן)

- נבדק את `my_pid, parent_pid` בהתאם

- נבצע מספר הדפסות

- נעשה `sleep 1-3` שניות. (כדי לוודא שהאב יסתים לפני הילד - עניין טכני)

- כשתגעורר, נשתמש ב-`execl` על מנת לבצע קוד חדש.

- אם ה-`execl` הצלח, לא נגיע לפיקודות אחרות.

- אם הייתה שגיאה ב-`execl`, נבצע הדפסה ונצא.

- אם ערך ההחזרה גדול מ-0 (אנו ב-`else` האחרון, בתהילר האב)

- נבצע מספר הדפסות ופעולות נוספות נוספות

- נשתמש ב-`(-) wait` כדי להמתין לבן עד שישים (יש לנו אחד לפחות רק לו).

- ככלנו לעשויות גם (`wait(NULL)` כי אין לנו צורך בערך ההחזרה).

```
pid_t my_pid=getpid(), parent_pid=getppid() , child_pid;
int status;

printf("\n Parent: my pid is %d\n\n", my_pid);
printf("Parent: my parent's pid is %d\n\n", parent_pid);
if((child_pid = fork()) < 0 ){ perror("fork failure"); exit(1);}
if(child_pid == 0){
    printf("\nChild: I am a new-born process!\n\n");
    my_pid = getpid();    parent_pid = getppid();
    printf("Child: my pid is: %d\n\n", my_pid);
    printf("Child: my parent's pid is: %d\n\n", parent_pid);
    sleep(3);
    execl("/bin/date", "date", 0, 0);
    perror("execl() failure!\n\n");
    _exit(1);
} else{
    printf("\nParent: I created a child process.\n\n");
    printf("Parent: my child's pid is: %d\n\n", child_pid);
    system("ps -acefl | grep ercal");  printf("\n \n");
    wait(&status); /* can use wait(NULL) since exit status
                     from child is not used. */
    printf("\n Parent: my child is dead. I gonna leave. \n ");
}
return 0;
```

שאלה ממבחן:

3. (6 נק') נניח כי תהlixir עם מזהה 1 (pid=1) מרים את הקוד הבא:

```
int a=fork();
int b=fork();
printf("a: %d, b: %d\n", a, b);
```

כמו כן הינו יייחד פוק של התהlixirים שנוצרים לאחר מכן גדלים ב- 1 (כלומר ה- pid של התהlixir הבא שיופיע יהיה 2 ועוד).
רשמו פלט אפשרי אחד של התוכנית.

פתרונות:

נציג חלק מהפתרונות האפשריים:

ראשית, אחרי ה-*fork()* הראשון, או שאנו בABA ו-*a = 2*, או שאנו בBN ו-*a = 0*.

- נניח שאנו בABA (*a = 2*), האבא יבצע את ה-*fork()* השני,

וגם CAN או SHM עם האבא ו-*a = b = 3* או שאנו BN ו-*a = 0*.

כלומר קיבלנו את ההדפסות האפשריות הבאות "a: 2, b: 3", "a: 2, b: 0", "a: 0, b: 3".

- נניח שאנו BN (*a = 0*), הבן יבצע את ה-*fork()* השני,

ואז או SHM עם האבא ו-*a = b = 4* או שאנו BN ו-*a = 0*.

כלומר קיבלנו את ההדפסות האפשריות הבאות "a: 0, b: 4", "a: 0, b: 0", "a: 4, b: 0".

יש פולטים אפשריים נוספים. בהקלטה של עדן מתאריך 10.6.19, יש הסבר מפורט בדקה 00:44.

כמו כן, מכיוון שהסדר לא משנה יש 4 אפשרויות לסידור ההדפסות,

ובכל שלב ניתן לבחור האם אנחנוABA או BN. לכן סה"כ $4 \cdot 2 = 8$ אפשרויות.

Memory Management calls

הפונקציה: *void *malloc (size_t size)*

malloc (כמו גם *calloc* ו-*realloc*) היא פונקציה להקצת זיכרון. הפונקציה מקצת זיכרון פנוי עבור אובייקט שהגודל שלו הוא *size*, ומחזירה מצביע למקום שהוקצתה בזיכרון.

הפונקציה: *void free(void *ptr)*

free היא פונקציה לשחרור זיכרון שהוקצתה על ידי *malloc*, *realloc*, *calloc* או *realloc*. הפונקציה תשחרר את הזיכרון ש-*ptr* מצביע עליו. אם *ptr* הוא לא ערך תקין (לא ערך ההחזרה של אחת מהפונקציות הנ"ל או שהזיכרון כבר שוחרר) ההתנהגות לא מוגדרת.

File Access Calls

- למדנו על מערכת הקבצים. ניבור כעת על פונקציות מערכת לניהול קבצים.
- אובייקט שניין לכתוב אליו / או לקרוא ממנו. לקובץ יש מספר תכונות ביןיהן סוג קבצים:
- קובץ רגיל.
 - *symbolic link* – מצביע לקובץ אחר כמו קישור דרך *Windows*.
 - תיקיה.
 - .random access – כתבים כל פעםתו אחד, כמו עכבר, מקלדת וכו', אין *Character device*
 - *Block Device* – כתבים בבלוקים שלמים, כגון *Hard Disk*, דיסק וכו'.
 - *Pipes* – תקשורת בין תהליכיים.
 - .*FIFO special file* – דרך נוספת תקשורת בין תהליכיים שונים.
 - .*sockets* –
 - .*Random number generators* –

כדי לגשת לקובץ, נדרש לפתוח אותו עם הפונקציה `(open()` שמחזירה `(file descriptor`

:fd = open(const char *path, int flag, ...)

(`open()` יוצרת את החיבור בין הקובץ *-fd* שלו (ערך החזרה).

path - שם הקובץ שאנו נפתח (או הכתובת שלו אם הוא לא באותה תיקיה).

flag - מצין את ההוראות:

- קרייה בלבד, כתיבה בלבד, קרייה וכתיבה.
- יצירה, הוספה, *exclusive*.
- *O_Direct* – כתיבה ישירה לקובץ (בלי לעبور דרך *Cache*).
- *O_Sync* – הכתיבה תתבצע באופן מיידי.

:err = close(int fd)

הפונקציה `(close()` משחררת את *-fd*, כלומר הופכת אותו להיות זמין להקצאה עבור קבצים חדשים. רק אם כל *-fd* המקיים לקובץ פתוח כלשהו נסגרו (כי מספר תוכניות יכולות לפתוח את אותו קובץ), הפונקציה תשחרר את הזיכרון הקשור לקובץ.

:b read = read(int fd, void *buf, int nbytes)

הפונקציה `(read()` קוראת *nbytes* בתים מהקובץ שמקורו *-fd* הנתון, אל תוך *buf*.

ערך החזרה הוא מספר אי שלילי שיציין את מספר הבטים שניקראו בפועל.

:b written = write(int fd, void *buf, int nbytes)

הפונקציה `(write()` כתבת *nbytes* בתים מהබאför *buf* אל תוך הקובץ שמקורו *-fd* הנתון.

ערך החזרה הוא מספר אי שלילי ויציין את מספר הבטים שנכתבו בפועל.

הפונקציה (`:where = lseek(int fd, off_t offset, int whence)`)

הפונקציה () `lseek` מאפשרת להזיז את ה-`file offset` (מה שמצוין את המיקום הנוכחי בקובץ - הסמן). () `lseek` תפעל באופן הבא:

- אם הוא `SEEK_SET`, `offset` של הקובץ צריך להיות הערך של הארגומנט `whence`.
- אם הוא `SEEK_CUR`, `offset` של הקובץ יהיה `offset` בתים מוסף הקובץ.
- אם `whence` הוא `SEEK_END`, `offset` של הקובץ יהיה `offset` מרגע הגודל של הקובץ ועדו.

הפונקציה מחזירה את המיקום החדש בקובץ. כדי לדעת איפה ה-`offset` הנוכחי בקובץ, משתמש ב:

`where = lseek(fd, 0, SEEK_CUR)`

הפונקציות `dup`, `dup2`:

פונקציות שנועדו לשכפל קובץ מסוים.

```
fd_new = dup(int fd);
int dup2(int oldfd, int newfd);
```

() `dup` מקבלת `fd` ומחזירה `fd` חדש כך שניהם מצביעים לאותו הקובץ.
() `dup2` מקבלת `fd` ישן ו-`fd` חדש, וגורמת לשניהם להצביע לאותו קובץ. אם ה-`fd` החדש מצביע לקובץ אחר, () `dup2` תסגור את הקובץ זהה, ואז תעתייך את ה-`fd` הישן ל-`fd` החדש. ערך ההחזרה יהיה ה-`fd` החדש (זה שדרסנו) במקרה של הצלחה, או 1 – במקרה של שגיאה.

הפונקציה `stat`:

מאפשרת לקבל מידע על קובץ מסוים.

```
err = stat (const char *path, struct stat *buf);
```

() `stat` מקבלת מצביע לסריאט `buf`, ואליו היא כתבתת את המידע של הקובץ.
`path` - שם הקובץ שאנו חיפש (או הכתובת שלו אם הוא לא באוטה תיוקה). הקובץ צריך להיות סגור. (כלומר אף אחד לא משתמש בו)
`struct stat` מכיל מספר ערכים חשובים כמו: מיקום, גודל, בעליים, הרשות, חוותות זמן ועוד..

הפונקציה `chmod()`:

ראינו הסבר מפורט על הפונקציה. [on page 129](#)

הפונקציה `pipe`

עד כה למדנו שתקשורת בין תהליכיים יכולה להתבצע בעזרת סיגנלים, זיכרון משותף, קבצים וסוקטים. כתע נראה דרך נוספת - `pipes`. הפונקציה `pipe()` יוצרת ערוץ תקשורת בין תהליכיים:

```
err = pipe(int fd[2]);
```

(`pipe` מקבלת מערך בגודל 2 ומכניסה לתוכו `fd` שאלוי כתבו ב-[1]`fd` ו-[0]`fd` שמננו יקראו ב-[0]`fd`.
הכתיבה ל-[0]`fd` תהיה לסופם הקובץ, מדיניות של *First In First Out*

דוגמה:

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main() {
    int pipefd2[2];
    pipe(pipefd2);
    if (fork() == 0) {
        dup2(pipefd2[1], STDOUT_FILENO);
        close(pipefd2[0]);
        close(pipefd2[1]);
        execl("/bin/ls", "ls", NULL);
        exit(EXIT_FAILURE);
    }
    if (fork() == 0) {
        dup2(pipefd2[0], STDIN_FILENO);
        close(pipefd2[0]);
        close(pipefd2[1]);
        execl("/usr/bin/file", "file", "-f-", NULL);
        exit(EXIT_FAILURE);
    }
    close(pipefd2[0]);
    close(pipefd2[1]);
    wait(NULL);
    wait(NULL);
    return 0;
}
```

בדוגמה זו אנחנו יוצרים ערוץ תקשורת באמצעות `pipe` ואז יוצרים תהליך חדש באמצעות `fork()`.

(נשים לב שבכל אחד מה-`if`'ים יש קריאה לפונקציה `fork`, סה"כ 2 קריאות בתוכנית)

אם אנחנו בתהליך הבן אז:

- בפקודה `dup2` אנחנו מחליפים את `STDOUT` ב-[1]`pipefd2`.
- כלומר הילד הולך להיות זה שקורא מערוץ התקשרות.
- נסגור את [1]`pipefd2`, `pipefd2[0]` (בפועל זה לא יסגור את הקבצים, קובץ יסגר רק אם כל ה-`fds` שלו סגורים, וכך הקובץ שהופנה ל-`STDOUT` עדין פתוח).
- נריץ את `"ls /usr/bin"` והפלט של הריצה יכנס ל-[0]`pipefd2` (ה-`STDIN` החדש).
- נמשיך ואז נקרא ל-`fork()` שוב (ב-`if` השני) – אם אנחנו בין אז:
- בפקודה `dup2` אנחנו מחליפים את `STDOUT` ב-[0]`pipefd2`,
- כלומר הילד הולך להיות זה שכותב אל ערוץ התקשרות.
- נסגור את [0]`pipefd2`, `pipefd2[1]`.
- נריץ פקודה חדשה, אבל הקלט יגיע מתוך [1]`pipefd2` (ה-`STDIN` החדש).

בפקודה `wait()` אנחנו מבצעים פעמיים בסוף, אנחנו מחכים ל-2 התהליכים החדשים שייצרנו ומואדים שהם יסיימו את הריצה שלהם לפני האבא.

File & Directory Management Calls

:mkdir()

(*mkdir*) יוצרת תיקייה חדשה ב-*path* הנוכחי. הרשות גישה לתיקיה יאותחלו עם *.mode*.

```
err = mkdir(const char *path, mode_t mode);
```

:rmdir()

(*rmdir*) מקבלת *path* לתיקיה, ומוחקת אותה. התיקיה תמחק רק אם היא ריקה.

```
err = rmdir(const char *path);
```

:chdir()

(*chdir*) מקבלת *path* ומשנה את ה-*working directory* הנוכחי של התחליר שקרה לה ל-*path*.
ואנו נקודת ההתחלה לחיפוש *relative pathnames* *working directory*

Hard Links

Hard links (וגם [בעמוד 126](#) ו-[page 118](#)) דיברנו על מערכת הקבצים ועל *metadata* – קובץ מכיל *data* ו-*metadata*. הוא לא מכיר את ה-*path* שלו, ושם הקובץ הוא לא חלק מה-*inode*.
לכל קובץ יכולים להיות מספר מקומות (אם יש לו קישורי דרך במקומות שונים).
כל קובץ מכיל *ref_count* – מספר המיקומים שמכילים קישור אליו.

:link()

יוצרת קישור חדש בין קובץ לתיקיה ומגדילה את *ref_count* ב-1.
*err = int link(const char *oldpath, const char *newpath);*

:unlink()

*err = unlink(const char *path)*

- הפונקציה מקבלת *path* לקובץ, מוחקת אותו מהתיקיה (מוחקת את הקישור בין ה-*inode* לתיקיה), ומקטינה את ה-*ref_count* שלו ב-1.
- אם *ref_count == 0*, ואין אף תהליר שמשתמש בקובץ כרגע, הקובץ ימחק מהזיכרון.
- אם *0 == ref_count* אבל תהליר אחד או יותר משתמשים בקובץ באותו רגע, הלינק ימחק מיד, אבל הקובץ עצמו ימחק רק כאשר כל התחליכים יסימנו להשתמש בו.

Device Management Calls

מערכות הפעלה בסיסיות כמו לינוקס מתייחסות לתקנים כאלו "קבצים מיוחדים".
גישה לתקן מתבצעת על ידי פתיחת הקובץ המווחד וגישה אליו דרך ה-*fd*.
קבצים מיוחדים לרוב יהיו בתיקייה */dev*.

הפונקציה (*ioctl*):

(*ioctl*) מאפשרת לבצע פעולות על התקנים.

```
int ioctl(int fd, int request, ... /* arg */);
```

fd – מייצג את התקן, *request* – מה שנרצה שהתקן יעשה.
(לא נרחב מעבר)

פונקציות נוספות

הפונקציה (*kill*) – שולחת סיגナル לתהיליך או קבוצה של תהליכים.

```
err = kill (pid_t pid, int sig);
```

pid – התהיליך שנרצה לשЛОוח לו סיגナル, *sig* – הסיגナル.
הפונקציה משתמשת לתקשורת בין תהליכים.

התהיליך שקרה ל-*kill* צריך להיות בעל הרשות לשילוח סיגנלים.

הפונקציה (*signal*) – ניתן לקרוא [בעמוד 27](#) על הפונקציה.

הפונקציה (*select*) – ניתן לקרוא [בעמוד 146](#) ו-[בעמוד 144](#) על הפונקציה.

דוגמה:

```
fd_set rfds;
struct timeval tv;
int retval;

/* Watch stdin (fd 0) to see when it has input. */
FD_ZERO(&rfds);
FD_SET(0, &rfds);
/* Wait up to five seconds. */
tv.tv_sec = 5;
tv.tv_usec = 0;
retval = select(1, &rfds, NULL, NULL, &tv);
/* Don't rely on the value of tv now! */
if (retval == -1)
    perror("select()");
else if (retval)
    printf("Data is available now.\n");
    /* FD_ISSET(0, &rfds) will be true. */
else
    printf("No data within five seconds.\n");
```

POSIX	Win32	Description
fork	CreateProcess	Create a new process
wait	WaitForSingleObject	The parent process may wait for the child to finish
execve	--	CreateProcess = fork + execve
exit	ExitProcess	Terminate process
open	CreateFile	Create a new file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from an open file
write	WriteFile	Write data into an open file
lseek	SetFilePointer	Move read/write offset in a file (file pointer)
stat	GetFileAttributesExt	Get information on a file
mkdir	CreateDirectory	Create a file directory
rmdir	RemoveDirectory	Remove a file directory
link	--	Win32 does not support "links" in the file system
unlink	DeleteFile	Delete an existing file
chdir	SetCurrentDirectory	Change working directory

Schedulers

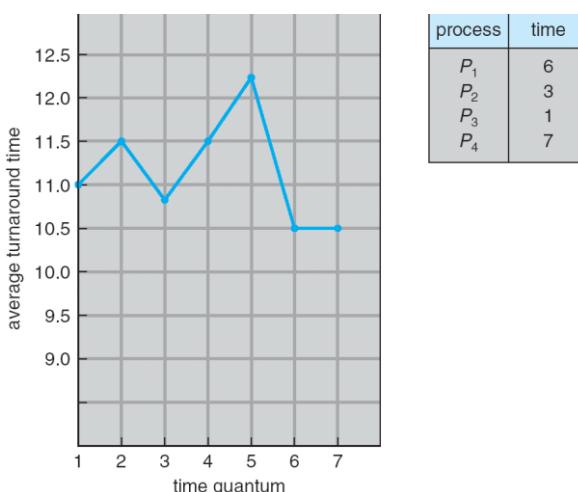
עד כה רأינו את ה-Schedulers הבאים:

- *FIFO/FCFS* – משרת עבודה לפי הסדר בו הם הגיעו.
- *Non-preemptive SJF* – כשעבודה הסתיימה, האלגוריתם בוחר את העבודה הקצרה ביותר.
- *Preemptive SJF (or SRT)* – כשעבודה הסתיימה או שעבודה חדשה הגיעה, העבודה הקצרה ביותר תיבחר (לפי הזמן שנשאר להם לרווח). אלגוריתם לא הוגן, יכול לגרום להרעה.
- *Interactive System* - עבור תהליכיים שזמן הריצה שלהם לא ידוע (כמו זמן הוא צריך קלטים מהמשתמש) נחשב את גודל העבודה לפי ה-*CPU burst size* שלו (כמה זמן הוא צריך לרווח עד שהוא יוותר עליו מרצונו). כדי לחשב את ה-*CPU burst size* הבא, נוכל לנחש לפי ממוצע מתעדכן של תהליכיים קודמים (*Moving Average*)

Round – Robin (RR) Scheduling

ראינו שאלגוריתם הוגן שפותר את בעיית ההרעה וכיוון רוב המערכת משתמשות בגרסה כלשוי שלו. הוא מקצה את ה-*CPU* בצורה הוגנת לכל העבודות ויש לו זמן המתנה ממוצע נמוך לעומת אלגוריתמים אחרים (עבור עבודות עם שונות גדולה של אורך העבודות עבור n עבודות, ייחידת זמן q , זמן המתנה המקסימלי לעובדה הוא $q \cdot (1 - n)$).

- אם q גדול, העבודות יסימנו לפני שיגמר להם ה-*quantum* ואז האלגוריתם עובד כמו *FCFS*.
- בנוסף מספר *Context Switches* קטן, ולכן היעילות גבוהה.
- אם q קטן, יהיו הרבה *Context Switches* והתקורה תהיה גדולה מדי. נרצה ש- q יהיה גדול מה-*burst time* של רוב העבודות, כדי שרוב העבודות יסימנו בריצה אחת של המעבד.



בגרף משמאל אפשר לראות את היחס בין גודל ה-*quantum* לבין הזמן הממוצע שלקח לתהליך לסיים. אפשר לראות שהgraf משתנה בקייניות, ואין קורלציה בין הפרמטרים.

הчисוב נעשה באופן הבא:

1. P1 P2 P3 P4 P1 P2 P4 P1 P2 P4 P1 P4 P1 P4 P4 (P1:15, P2:9, P3: 3, P4: 17. AVG:11)
2. P1 P1 P2 P2 P3 P4 P4 P1 P1 P2 P4 P4 P1 P1 P4 P4 P4 (P1: 14, P2:10, P3: 5, P4: 17. AVG: 11.5)
3. P1 P1 P1 P2 P2 P3 P4 P4 P4 P1 P1 P1 P4 P4 P4 P4 (P1: 13, P2: 6, P3: 7, P4: 17. AVG: 10.75)
4. P1 P1 P1 P1 P2 P2 P3 P4 P4 P4 P4 P1 P1 P4 P4 P4 (P1: 14, P2, 7, P3: 8, P4: 17. AVG: 11.5)
5. P1 P1 P1 P1 P1 P2 P2 P3 P4 P4 P4 P4 P1 P4 P4 (P1: 15, P2: 8, P3: 9, P4: 17. AVG: 12.25)
6. P1 P1 P1 P1 P1 P2 P2 P2 P3 P4 P4 P4 P4 P4 P4 P4 (P1: 6, P2: 9, P3: 10, P4: 17. AVG: 10.5)

מימוש

כפי שכבר אמרנו, תפקיד ה-*Scheduler* לבחור את התהיליך הבא שישתמש ב-*CPU* ל-*quantum* זמן, והטהיליך זהה ישתשמש ב-*CPU* עד שמיشهו יחסום אותו או עד שהוא יקרה לפונקציית מערכת (System Call) ובעצם יותר על ה-*CPU* מרצוינו.

בעיה: יש רק *CPU* אחד, ואם התהיליך הזה תופס את ה-*CPU*, איך האכיפה של ה-*quantum* מבוצעת? (מי עוצר את התהיליך כשנגמרתו לו הקווואנטה?)

פתרון: בשביל להתמודד עם הבעיה הנ"ל, נסיף תמייה חומרתית לאלגוריתם *RR* שתבצע פסיקות שעון בכל *quantum* זמן ותחזיר את מערכת הפעלה ל-*CPU*.

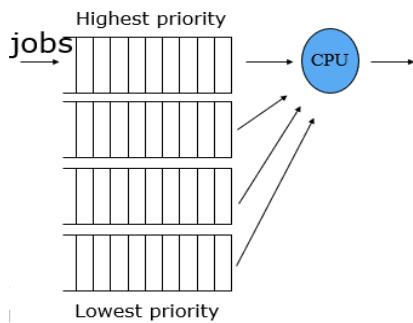
זו הדרך היחידה של מערכת הפעלה לקבל שליטה מחדש על ה-*CPU*,

וברגע שהוא קיבלה שליטה חוזרת היא תבצע את ה-*Context Switch* הבא.

הערה: גם אם יש רק תהיליך אחד שרצ באוטו רגע על המחשב, פסיקת השעון תבצע, ה-*Scheduler* יחזיר ל-*CPU* וזה הוא יבחר מחדש בתהיליך הבודד.

Priority Scheduling

- לכל תהיליך יש מספר שלם שמייצג את העדיפויות שלו (*Priority*)
- ה-*CPU* יוקצה לתהיליך עם העדיפויות הגבוהה ביותר.
- התהיליכים נמצאים בתורים לפי העדיפויות שלהם, כל תור יכול לעבוד עם *Scheduler* אחר (למשל *Non – Preemptive FCFS* או *Preemptive FCFS*) כל תור יכול להיות עם *Scheduler* אחר (*SJF*).



אם יכנסו הרבה תהיליכים עם עדיפויות גבוהה אחד אחריו השני, יהיה מצב של הרעה עבור תהיליכים עם עדיפויות נמוכה. 2 פתרונות אפשריים:

- *Time Slice* – כל תור מקבל נתחם (לא שווה!) מהזמן הכלול להרצת התהיליכים שלו.
- למשל בתמונה הנ"ל, התור הראשון של התהיליכים החשובים קיבל 60% מהזמן הכלול, השני קיבל 25% מהזמן, השלישי 10% והטור של התהיליכים הכי פחות חשובים קיבל 5%.

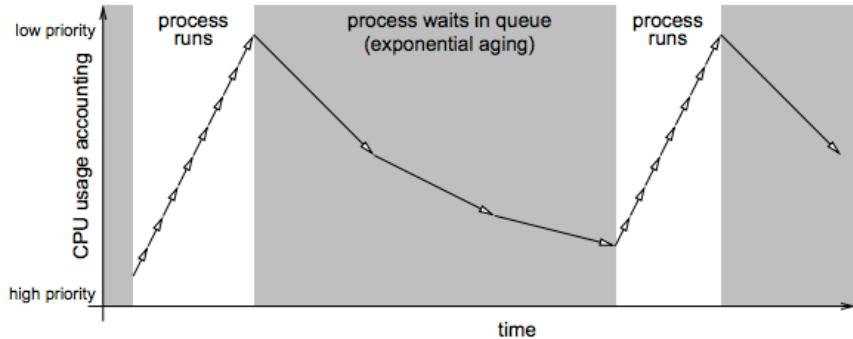
בצורה זו מובטח שככל תהיליך יגיע מתיישה לprocessor.

Aging – ככל שתהיליך נמצא בתור יותר זמן, העדיפות שלו תגדיל (ולבסוף הוא יהיה הראשון) שהרעה: יש גם גרסה אחרת של *Aging* שפועלת הפוך – תהיליכים חדשים מקבלים עדיפויות גבוהה, וכך הם משתמשים ב-*CPU* העדיפויות שלהם יורדת. כלומר תהיליכים גדולים לאט יורדים בעדיפות לטובות תהיליכים חדשים וכך יש התקומות. (פתרון זה לא פותר הרעה!)

בגרסאות ישנות של *UNIX* היה שימוש ב-*Aging* לפי החישוב הבא:

$$\text{Priority} = \text{CPU_Usage} + \text{Base}$$

 כאשר base זו העדיפות הגבוהה ביותר
 (כל ש-*Priority* קטן יותר, כך העדיפות גבוהה יותר)
 CPU_Usage גדל ב-1 בכל שנייה שהתהליך רץ במעבד, ומוליך פי 2 בכל שנייה שלא.



Multilevel Feedback Queue (General Case)
 , *Multilevel Feedback Queue* הוא מקרה פרטי של *Priority Scheduling*
 אלגוריתם כללי, המשמש ביותר מטור אחד ומאפשר לתהליכיים לעבור בין תוריהם.
 ההבדל בין האלגוריתמים נקבע לפי הפרמטרים הבאים:

- מספר התורים
- האלגוריתמים שנבחרו לנוהל כל תור.
- מתי לשדרג/לשנמן תהליך לטור אחר.
- לאיזה טור תהליך מסוים נכנס כשהוא צריך לקבל שירות.

Multiple – Processor Scheduling
 ה-*Schedulers* שלמנו הניתו שלמחשב יש *CPU* בודד, אך היום יש מחשבים עם יותר מעבד אחד,
 ומערכות עם יותר מ-*Core* אחד וה-*Schedulers* שלהם מורכבים יותר בהתאם. לא נרחב.

Real Time Scheduling
 במערכות *Real – Time*, שבהן חייב לסיים משימה עד זמן מסוים,
 יש *Schedulers* מיוחדים שיודעים להתמודד עם זמני סיום.
 – *Hard Real Time Systems* – מערכות בהן מובטח שימוש קרייטית تستטיים בזמן הסיום הרצוי.
 – *Soft Real Time Systems* – מערכות בהן משימות עם זמן סיום קרוב יותר יקבלו עדיפות גבוהה יותר
 (אך לא מובטח שהן אכן יסימנו בזמן הרצוי)

יש *Schedulers* נוספים כמו *Lottery scheduling*, *Guaranteed scheduling* שאין צורך להרחיב
 עליהם, חשוב רק להבין שיש **מונע סוגים**.

איך נערך Scheduler ?

Schedulers – לפי מודל זה, ההשוואה בין *Deterministic modeling (Analytic evaluation)*

שונים מתבצעת בעזרת פרמטרים ידועים (מספר התהיליכים, *quantum size* וכו')

.*Waiting Time*, *Turnaround Time*, *Response Time*, *Throughput*

ובעזרת מדדים כמו *Scheduler* – לא תמיד הפרמטרים ידועים, ולכן נספתח למודל *Queuing models*

שלמה של ניהול תורם.

– לפעמים המתמטיקה של ניהול התורים מסובכת מדי, ובמקרה זה אפשר לבצע

סימולציות על שירותי ענקיים, אך מדובר בהערכתה לא מדוקט.

Scheduler – פרמטר נוסף להערכת *Scheduler* הוא סיבוכיות המימוש. נרצה שה-

יה פשטוט יחסית למימוש, ולא נבצע חישובים מסובכים מדי כל *Context Switch*.

I/O (Input/Output)

נרכיב כעת על התקני קלט/פלט:

- התקנים שבעזרתם המחשב יכול לתקשר עם המשתמש. יש התקנים

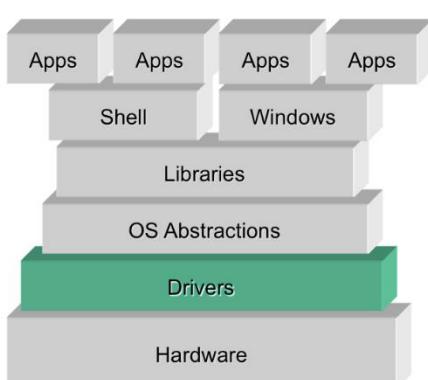
שהם רק התקני קלט כמו מקלחת ועכבר, יש התקנים שמקבלים רק פלט כמו מדפסת ומסר, ויש התקנים

שהם גם קלט וגם פלט כמו כרטיס רשת.

– התקני אחסון, מרובם אפשר גם לקרוא וגם לכתוב, אבל יש גם כאלה שאפשר רק

לקראן מהם (למשל *CD ROM* אם לא צורבים אותו).

כל התקן הוא שונה. חלקם רק התקני קלט, חלקם רק התקני פלט, חלקם שניהם, חלקם עובדים לפי בלוקים וחלק לפי אוטוות, מהירותם שונות ועוד.. מערכת הפעלה צריכה לאפשר משך נוח לשימוש בכל סוג התקנים.



בעמוד 13 רأינו שככל מערכת מחולקת לשכבות שונות. השכבה

התחתונה היא שכבת החומרה שמכילה בין היתר את-*CPU*

ואת-*bus* שתפקידו לקשר בין כל התקנים ל-*CPU*.

מעל שכבת החומרה יש את שכבת הדרייברים שתפקידם לקשר

בין החומרה למערכת הפעלה, ואשר השכבות מייצגות את

מערכת הפעלה והAPPLICATIONS השונות. לאורק הקורס הרחובנו

בעיקר על השכבות הגבוהות, כעת נרכיב על דרייברים.

תפקיך מערכות הפעלה בהקשר של O/I

- לאפשר תצוגה לוגית (אבסטרקטית) של התקנים.
- להס梯יר פרטיים שלא רלוונטיים למשתמש כמו משק ההתקן.
- להסתיר את אופן ניהול השגיאות של התקנים שונים.
- לבצע פעולות במעבד "במקביל" זהה SHA-0/I ייעוד (כפי שהרחבנו לכל אורך הקורס)
- לאפשר למספר תהליכיים לוחוק באותו התקן.
- להגן על תהליכי מסויים מתהליך אחר (למשל למנוע כתיבה בו בזמןית לדיסק)
- לנצל סדר גישות לתקנים שיכולים לעבוד בקשה אחת בו בזמןית (כמו מדפסות)

Device Controllers

- להתקני O/I יש מחשב קטן, שנקרא Controller או Adapter, שיודע לבצע פעולות ספציפיות הקשורות להתקן. הוא נמצא בהתקן עצמו וכאן הוא יכול לבצע אותן במקביל ל-CPU.
- הממשק בין ה-Controller להתקן הוא *Controller-level* – *low*. למשל עבור הדיסק, ה-*Controller* צריך להכיר את התקן עד לרמת הциינדרים והסקטוריים.

Drivers

- כל התקן O/I צריך דרייבר – קוד שמכיל מידע ספציפי על התקן ומאפשר לשלוט בו.
- הקוד נכתב לרוב על ידי מי שיצר את התקן, והוא רץ בקרנל.
- יש לדרייבר API קבוע כמו *Open, Close, Write, Read, Seek, Flush* וכו'.
- כשהדרייבר מקבל פקודה כמו *get/put char* הוא מתקשר עם ה-*Controller* שיטפל בבקשתו.
- התקשרות בין הדרייברים (ה-*CPU*) ל-*Controllers* מתבצעת דרך ה-*bus*.
- ה-*CPU* יכול להריץ פקודות O/I מיוחדות, או לחילופין להריץ פקודות רגילות של כתיבה וקריאה מהזיכרון (...*lw, sw*...), שהמיקום אליו הוא יכתוב ב-RAM ממופה לרגיסטרים של ה-*Controller*.
- מבלי שהוא מודע לזה.

Polling vs Interrupts

2 שאלות עיקריות שנותרו:

- מי מעביר את המידע בין ההתקנים לזיכרון הראשי?

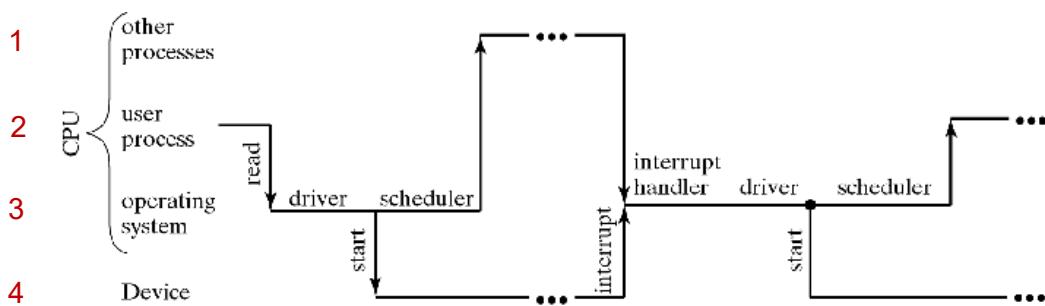
- איך הדרייבר יודע מתי ההתקן מוכן? (מתי לקרוא ממנו מידע)

在这 2 之間 הגישות שונות שפותחות את השאלות הנ"ל.

לפי CPU, ה-*Polling* אחראי להעביר כל תו מה-*Controller Buffer* וAliases. Controller יש ביט *busy* שמוסמן אם הוא מבצע פעולה כלה, או שאפשר לקרוא/לכתוב ממנו. ה-*CPU* בודק באופן תדיר (לולאת *wait busy*) אם הביט דלוק או לא, וכך הוא יודע אם ההתקן מוכן.

היום לא משתמשים ב-*Polling*, והגישה הנפוצה יותר היא באמצעות פסיקות (*Interrupts*).
לפי גישה זו, ה-*CPU* עדין אחראי להעביר מידע מה-*Controller Buffer* וAliases, אבל הדרך שלו לדעת אם ההתקן מוכן היא באמצעות פסיקות שעון. כשההתקן מסיים הוא שלוח פסיקה ל-*bus*, Interrupt Handler יודע לעזרו את ה-*CPU* שיטפל בפסיקה.

סיכום התהיליך:



הشرطוט הנ"ל מתאר פעולה קרייה מהתקן *0/I* לאחר קרייה ל-*read* מתוך תהליך כלשהו.

הشرطוט מכיל 4 שלדות: תהליכיים כלליים, התהליך שקורא ל-*read*, מערכת הפעלה והתקן.

- הشرطוט מתחילה מהשורה השנייה (משמאל לימין), כשה-*user process* מבצע קרייה ל-*read*.
- מ裏יצה *read* מ裏יצה *System Call* שגורמת ל-*trap* ומעבירה את ה-*CPU* למערכת הפעלה.
- מערכת הפעלה מ裏יצה את ה-*driver*
- הדרייבר מפעיל את ההתקן ומחזיר את ה-*CPU* ל-*CPU* שיבחר *Process Scheduler* שירות.
- אמרנו שה-*Controller* של ההתקן רץ במקביל ל-*CPU*, لكن שלדות 1 ו-4 מופיעות במקביל.
- כשההתקן יסיים, הוא ישלח פסיקה ל-*bus*, ה-*Interrupt Handler* יחזיר את מערכת הפעלה ל-*CPU* ומערכת הפעלה תריץ את הדרייבר.
- הדרייבר יקרא את המידע מה-*buffer*, יכתוב אותו לזכרון ויחזיר את המושכות ל-*Scheduler*.
- ה-*Scheduler* יבחר את התהליך הבא שירוץ וכן הלאה.

DMA

בכל מערכות המחשב בימינו יש DMA (שגם הוא עובד בעזרת *DMA Controller*), ומערכות כאלה ה-*CPU* לא יהיה זה שעביר את המידע אלא רק זה שMRIIZ את הפקודות (הפקודות של ה-*Driver* Controller גם את המיקומים אליוים צריך לכתוב בזיכרון הראשי). ה-*CPU* יהיה זה שייגש למיקומים האלה בפועל, ובינתיים ה-*CPU* יהיה פניו למשימות אחרות. כשה-*Controller* יסימן הוא ישלח פסיקה כדי לעדכן את ה-*CPU*.

DMA Operations

דוגמה להעברת מידע מהדיסק לזיכרון, בעזרת DMA:

1. ה-*Driver* מקבל הוראה להעביר מידע מהדיסק ל-*buffer* בכתובת X
2. ה-*Driver* אומר ל-*Disk Controller* להעביר C בתים מהדיסק ל-*buffer* בכתובת X
3. ה-*Disk Controller* מתחילה את ההעברה בעזרת DMA
4. ה-*Disk Controller* שלוח כל בית ל-*Disk Controller*
5. ה-*Disk Controller* מעביר בתים ל-*buffer* בכתובת X , מגדיל את הכתובת בזיכרון אליו הוא כותב, ומקטין את C עד $C = 0$
6. כ- $C = 0$, ה-*Disk Controller* שלוח פסיקה ל-*CPU* כדי לעדכן שההעברה הושלמה.

Cycle Stealing

ה-*DMA Controller* "מתחרה" עם ה-*CPU* על גישה לזיכרון. אם ה-*CPU* מנסה לגשת לזיכרון אבל באותו רגע הזיכרון משרת את ה-*DMA Controller*, ה-*CPU* יצטרך לחכות לתו. בפועל זו לא בעיה כי ה-*CPU* ניגש רוב הזמן ל-*Cache* ולרגיסטרים נוספים, ולא לזיכרון עצמו.

Hierarchical Model of the I/O System

מערכות הפעלה מודרניות מחלקות את הדריברים ל-3 סוגים:

- כללה שעובדים עם בלוקים (כמו זיכרון ו디יסק)
 - כללה שעובדים עם סטרימים (קוראים/cotברים בית אחורי בית למשל מקלדת)
 - כללה שעובדים עם פקודות (כמו ה-*Network Controller*)
- דריברים מסוימים משתמשים בפונקציות דומות (למשל קריית בלוק, כתיבת בלוק וכו') ולכן הפעלה ממשות את הדברים המשותפים האלה וחוסכות חלק מהעבודה למי שכותב את הדריברים.

תרגול 13

תרגול זה הוא תרגול חוזר על כל החומר מתחילה הסטטוס.
מומלץ לראות את הlecture, נסיף כאן רק דברים חשובים.

Interrupts

בpage 14 ו-בעמ' 17 הרחמנו על פסיקות וראינו כי יש שני סוגי:

- *External interrupts* – נגרמות על ידי המתקן חיצוני בזמן רנדומלי במהלך ביצוע התוכנית.
- *Internal interrupts (Exceptions)* – נגרמות כשההמעבד מזהה שגיאה בזמן הרצת פקודה, למשל חילוקה באפס, *Segmentation Fault*, *Privileged Instruction*, פקודה שלא קיימת, או *trap*. ניתן לחלק את ה-*internal interrupts* לשתי קבוצות:
 1. **Aborts**: מתרחשות במהלך ריצה רגילה של התוכנית בגלל שגיאה (חילוקה באפס, *Page Fault* וכו')
 2. **Traps**: מתרחשות במהלך ריצה רגילה של התוכנית אבל לא בגלל שגיאה.

Page fault – דוגמה

- תוכנית מבקשת מידע שלא נמצא כרגע בזכרון, ולכן יוצרת *exception*.
- בעקבות ה-*exception*, מערכת הפעלה טעונה את המידע מהדיסק ל-*Main Memory*.
- התוכנית מקבלת את המידע שהוא מביל לדעת שהיא שגיאת, וממשיכה הלאה.

Internal Interrupts – התמודדות עם

- ה-*CPU* מטפל ב-*Internal Interrupt* באופן דומה ל-*External Interrupt*.
פרט לכך שב-*External Interrupt* יהיה שלב נוסף של יצירת הפסיקה על ידי ה-*CPU* (כי המעבד מקבל את הפסיקה מהמתקן חיצוני)
 1. שמירת המצב הנוכחי ב-*PCB* (ערכי הרגיסטרים וכו').
 2. העברת השליטה למערכת הפעלה וטיפול בבקשתו.
 3. שייחזור המערכת למצב הקודם מה-*PCB*.
 4. החזרת השליטה לתהילך.

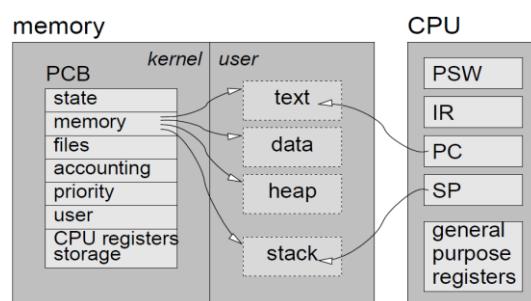
Signals

דיברנו על סיגנלים וראינו כי סיגナル הוא נוטפיקציה (התראה) הנוצרת על ידי מערכת הפעלה ונשלחת לתהיליך מסוים כדי להתריע על אירועים חשובים. במצב זה התהיליך יעצור את מה שהוא עונה (כਮון אחרי שישים את ההוראה הנוכחית) ויטפל בסיגナル מיידית. הטיפול יכול להיות על ידי דיפולטיבי שמערכת הפעלה נתנה לו, או שניתן להגדיר את ה-*handler* בלבד. בנגדוד לפסיקה, הסיגナル נוצר על ידי מערכת הפעלה ונשלוח לתהיליך ואילו פסיקה מתקבלת ומופעלת על ידי מערכת הפעלה. *External Interrupts* נוצר ממחומרה, *Internal Interrupts* נוצר מהתוכנה. ב-UNIX, לכל סיגナル יש שם ומספר. ניתן להשתמש בפקודה *man kill* כדי לראות את סוגי השונים.

:PCB

דיברנו והרחבנו על ה- *PCB* – טבלה בזיכרון שמתארת את המצב של כל תהיליך.

- ...running, ready, blocked - state -
- זיכרון - מצבים לטבלאות דפים. (נזכר ש-*heap, stack, data*-*data* אלו מושגים שנכונים רק לזכרון הווירטואלי, והם לא קיימים במאם). -
- טבלת הדפים הפתוחים של כל תהיליך. files -
- סטטיסטיות הקשורות לתהיליך. accounting -
- העדיפות של תהיליך. priority -
- רגיסטרים של ה- *CPU* הקשורים לתהיליך, כמו *PC, SP* וכו.. -



שאלות מבחנים משנים קודמות

יכיזד ניתן למשתמש מתזמן ב-User Space ? (כלומר לכתוב תוכנית שהיא תתזמן לתהליכים אחרים)

פתרונות:

It can be implemented in user level by using the signals SIGSTOP and SIGCONT. SIGSTOP suspends a process (and this signal cannot be blocked). Once a process is suspended the OS will quickly swap out the memory of this process since it is not used. Once the process is send SIGCONT the swapped out memory will be brought back to main memory.

A simple user level process that check the status of processes every couple of seconds and decides which processes should be suspended or resumed.

ה. במערכת קבצים מסורתית מבוססת inodes, כמה גישות לדיסק נדרשות לכל הפחות כדי לקרוא את הבית ה-70978 בקובץ `/home/moishe/mail/?` הניתן שגודל בЛОק הוא 1024 בתים, inode מכיל 6 מצביעים ישירים לבLOCKים ועוד 3 מצביעים עקיפים (*direct, double indirect, triple indirect*), כל מדריך מאוכסן בבלוק יחיד, ובתחלת הפעולה נמצא בזכרון רק ה-inode של שורש מערכת הקבצים. נמקו. (5 נק')

פתרונות⁸:

סה"כ 8 גישות לקריאת בלוק, בשרשראת ההצבעות הבאה:
מדריך של השורש / ,

inode של home , מדריך של home

inode של moishe , מדריך של moishe

inode של mail

בלוק מצביע single indirect של mail

בלוק מידע של הקובץ .

טעות נפוצה: לכל ספרייה, כולל השורש, יש מדריך (רשימה של תוכן הספרייה, מאוכסנת באופן דומה לתוכן קובץ), וגם inode (מצביעים לרשימה הנ"ל, הרשות וכד'). הם יושבים במקומות שונים בדיסק.

⁸ קישור למבחן: <http://cs.tau.ac.il/~tromer/courses/os11-12/2012-a-exam-solutions.pdf>

ב. להלן פתרון לביעית הקטע הクリיטי לשני תהליכיים (0 שקול ל TRUE ו-1 שקול ל FALSE):

```
At process i ∈ {0,1}
shared boolean flag[2] = {false};
shared int turn = 0;
do
{
    turn = 1-i;
    flag[i] = false;
    while !(flag[1-i] && turn==1-i);
    critical section
    flag[i] = true;
    remainder section
} while(true);
```

- א. האם המימוש עשוי לגרום לדעוק? deadlock? נמק.
- ב. האם יתכן שייתרחש deadlock אם נוריד את השורה; flag[i]=false וначתול את flag[2] להיות ?true
- ג. נניח שמבצעים את השינויים מהסעיף הקודם. האם תוכנת המונעת ההדדיות מתקיימת? הוכיח תשובה.

סעיף א':

המימוש עלול לגרום לדעוק deadlock. למשל: אם תהליך א' ינסה להיכנס ויגיע לדעוק critical condition ושם יהיה context switch לתהליך ב' שגם יגיע עד לדעוק critical condition, אף אחד מהם לא יוכל להיכנס כי [i-1] flag של שניהם הוא false ולפיכך הביטוי ב- critical condition של שניהם הוא true ולפיכך הם יחכו. נשים לב שאפשר יהיה להיכנס לקטע קוד הクリיטי רק אם flag[i]==true אבל זה יכול להיות רק אם מישחו הגיעו לדעוק deadlock. ולפיכך תמיד יהיה exit code.

סעיף ב':

לא. כי [i-1] flag תמיד יהיה true ומאחר מכן turn יהיה 0 או 1 נדע שלא יתכן deadlock. למעשה change turn משנה את המשמעות שלו ל- "האם צריך לחכות?". כלומר אם turn=i זה אומר ש- i צריך לחכות.

סעיף ג':

לא. נראה שהה מונעת ההדדיות מתקיימת. מרגע תהליך 0 – מצב 1 = turn ו- אז נכנס ל- CS. מרגע תהליך 1, מצב 0=turn ו- אז נכנס גם הוא ל- CS.

2. הבדל אחד בין תהליך לבין kernel thread הוא
- עם ריבוי תהליכיים ניתן לנצל ריבוי מעבדים, אבל עם kernel threads לא
 - עם kernel threads מונעים את הבעיה של חסימה כאשר אחד מבצע פעולה O/I, אבל הבעיה קיימת כמשתמשים בריבוי תהליכיים
 - תהליכיים זוקרים לתיווך של מערכת הפעלה כדי לתקשר, ו-kernel threads לא
 - kernel threads מרים רק קוד של מערכת הפעלה, ואילו תהליכיים יכולים להריץ קוד משתמש

פתרונות:

סעיף ג.

9. ב-UNIX המידע אודות הבלוקים המכילים את התוכן של קובץ שמור ב-node בארגון דמי עץ. הסיבה לכך שהעץ "מעוטר" במקומות להיות עץ מלא היא
- הרצון למנוע מצב בו השורש הופך לצוואר בקבוק
 - הרצון למנוע את זבוז הזמן והשתח הכרוך בשימוש בבלוקים בלתי ישירים עבור קבצים קטנים
 - הרצון לנצל עד תום את השטח המוקצה ל-node
 - חוסר החשיבות של איזון העץ לאור בכך שאין מדובר בחיפוש אחר איבר שרירותי

פתרונות:

סעיף ב.

1. במערכת נתונה ישנו 3 תהליכיים A,B,C ושלושה משאבים T,S,R. כמפורט בטבלה הבאה:

תהליך A	תהליך B	תהליך C	תהליכיים
R	מבקש S	מבקש T	סדר בקשות ושהוריים
S	מבקש T	מבקש R	לתהליכיים (מלמעלה למטה)
R	משחרר S	משחרר T	
S	משחרר T	משחרר R	

נתונה כאן רק סדרת בקשות ושהוריים של משאבים עבור כל אחד מהתהליכיים. איזה אפשר לטעון?

- ל手続きים אלו אין זימנון היוצר קיפאון כי יש רק 3 תהליכיים ו-3 משאבים.
- ל手続きים אלו אין זימנון היוצר קיפאון כי כל צמד מביניהם מבקש רק משאב אחד מסוית.
- ל手続きים אלו יש זימנון היוצר קיפאון ואין זימנון ללא קיפאון.
- ל手続きים אלו יש זימנון היוצר קיפאון ויש זימנון ללא קיפאון.
- בשאלה אין נתונים מספקים כדי לבחור בתשובה מפורטת.
- אף תשובה מפורטת לא צריכה להיות מסומנת.

פתרונות:

- סעיפים א', ב' – לא נכונים. למשל: אם C מקבל את T, B מקבל את S ו- A מקבל את R. אך C ממתין ל-R, B ממתין ל- T ו- A ממתין ל- S. במצב זה יש דלולוק.
- סעיף ד' נכון ולאחר בדוגמא מעל מקבלים דלולוק. כמו כן, יש תזמון שבו אין דלולוק (אם C מסיים לרוץ, אחרי B מתחילה ומסיים לרוץ ואחריו A מתחילה ומסיים לרוץ).

- 5.21 Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections at any point in time. As soon as N connections are made, the server will not accept another incoming connection until an existing connection is released. Explain how semaphores can be used by a server to limit the number of concurrent connections.

פתרונות:

סביר להניח שיש לו לאת while so שבה עושים accept. הרעיון הוא לעשות בסוף החלק של ה- so קרייה ל- semaphore שמאתחל ל- N וכך לא הגיע לשלב של ה- accept עד שלא עברו את ה- semaphore

- (4 נק) ii. כדי לאפשר תקשורת, שרת מבצע את קריאות המערכת `socket`, `bind`, ו-`listen`. אילו מהמצבים הבאים יגרמו לכישלון לפחות אחת מהקריאות האלה? (סמן את כל הנכונים)
- (א) ה-port המבוקש כבר נתפס על ידי תהליך אחר
 - (ב) המחשב שMRIIZ את הלקוח נפל
 - (ג) ביצוע פקודה ה-`bind` לפני פקודה ה-`listen`
 - (ד) הגענו לגבול של מספר ה- file descriptors המותרים לנו

פתרונות:

סעיפים א,ד.

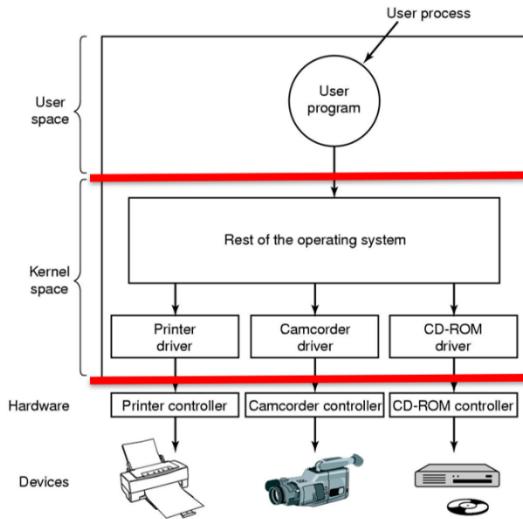
- (4 נק) iii. מנגד, קחו ציריך לבצע את קריאות המערכת `socket` ו-`connect`. אילו מהמצבים הבאים יגרמו לכישלון לפחות אחת מהקריאות האלה? (סמן את כל הנכונים)
- (א) ה-port המבוקש כבר נתפס על ידי תהליך אחר
 - (ב) המחשב שMRIIZ את השרות נפל
 - (ג) אף תהליך לא רשם עם ה- port המבוקש במחשב היעד
 - (ד) הגענו לגבול של מספר ה- file descriptors המותרים לנו

פתרונות:

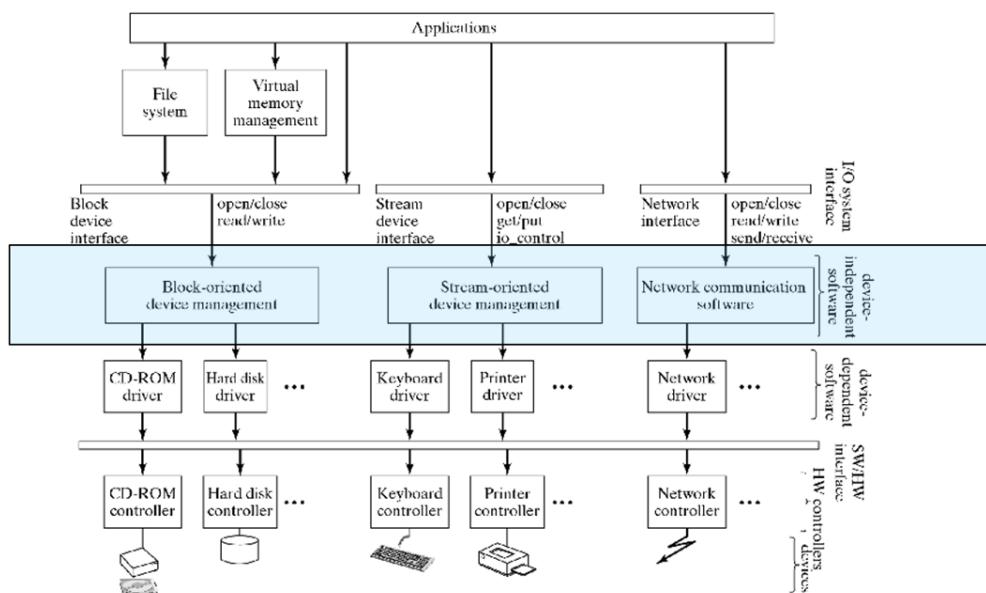
סעיפים ב,ג,ד.

נקודות אחרונות לגבי התקני I/O

במפגשה 13 הרחיבנו על נושא ה-I/O אבל לא הספקנו את כל השקופיות, נשלים אותן כעת. השרטוט הבא מתאר באופן מופשט את התקשרות בין המשתמש, למערכת הפעלה, להתקנים:



בנוסף, בסוף המפגש הזכרנו שמערכת הפעלה מסוגת דרייברים ל-3 קבוצות (כאלה שעובדים עם בלוקים/סטרימרים/רשתות), ולכן מרכזת הפעלה מימושת את הדברים המשותפים האלה וחוסכת חלק מהעבודה למי שכותב את הדרייברים. בشرطוט הבא אפשר לראות את ההיררכיה:



Device Independent Techniques

ממשק שמערכת הפעלה מספקת כדי לתת מענה למגוון שלם של התקנים דומים לצרכים לעשوت פעולות דומות ולן אין טעם למשם אותם בכל דרייבר בנפרד. (כפי שכבר הזכרנו)

דוגמאות לשיטות שיכלוטו להיכל במכשירים של מערכת הפעלה:

- **Buffering/Caching** – כל התקנים יכולים להשתמש בbabfer ו-*Cache* שמערכת

הפעלה מספקת (במקום למשח מחדש בדרייבר).

- **התמודדות עם שגיאות** – נבחין בין סוגי השגיאות:

○ *Persistent SW error* – בעיה שנפתחה על ידי התקינה מחדש.

○ *Transient SW/HW errors* – פועלה בעיתית שצריך לחזור עליה שוב, למשל:

- כתיבה/קריאה מהדיסק

- תיקון הקוד (קורה לפעמים שאמורים לקבל 0, ומתקבלים 1)

- *Persistent HW errors* – יותר מידי מידע מאוכסן.

ניתן לטפל בשגיאות הנ"ל באמצעות דומה בכל התקנים, ולן אין צורך שכל דרייבר יגדיר התמודדות עם שגיאות בעצמו.

- **Device scheduling/Sharing** – אלגוריתמי תזמון, כמו למשל אלגוריתם שמקוצר את

ה-*seek time* של הדיסק, או אלגוריתם לבחירת תחילה' שצריך להשנות אם יש יותר מידי תהליכיים. אלגוריתמים אלה יכולים להיות רלוונטיים לכל התקנים מאותו סוג.

וירטואלייזציה

Virtualize – מונicode: וירטואלייזציה היא הפעלת תוכנה באופן בלתי תלוי בתשתיות המחשב, באמצעות דימוי של תשתיות אמיתיות. יכולר להפוך משהו מודומה (שאיפילו לא יודע שהוא מודומה) לארה מאכוטית של המחשב עצמו, ושיתפרק אליו הוא אמיתי ויבצע פעולות ממשיות.

Virtualization היא היצירה של גרסה מודומה וירטואלית במקום ממשית של דברים כמו מערכת הפעלה, שירות, התקן לאחסון זיכרון, או משאבים של רשתות תקשורת. וירטואלייזציה מתמודדת עם הרחבה או החלפה של אינטראפייס קיימ שיקקה את ההתנהגות של מערכת אחרת.

דוגמאות:

- תהליך שרך במחשב "חושב" שהוא רץ בלבד, הוא חשב שכל ה-*CPU* שלו, ומערכת הפעלה נותנת לו את התשתיות בשבייל זה.
- ניתן לחלק דיסק כך他会 *partitions*. *partition* זה חלוקה לוגית של הדיסק על מנת ליצור שני דיסקים נפרדים, כל *partition* חושב שהוא כל הדיסק, ובפועל יש זרוע אחת וקורה מגנט אחד הפעולים בשני החלקים.
- זיכרון וירטואלי – וירטואלייזציה ל-*RAM* (כפי שמדובר בעבר).
- (*Virtual private network* (*VPN*) – אפשר (בין היתר) לדמות שהמחשב שלנו גולש מרשת אחרת (למשל אריה"ב).

מכונה וירטואלית – *VM*:

חומרת המחשב התקדמה מאוד בשנים האחרונות, ומחשבים בימינו הם בעלי כוח חישובי חזק מאוד. ברוב המקרים אנחנו לא ננצל את הכוח החישובי של המחשב וזה בזבוז (צריך לנצל את ה-*CPU* כמה שיותר!). מכונה וירטואלית היא פתרון לביעיה הנ"ל שפותח בשנות ה-60 ע"י *IBM* לחברות גדולות כמו בנקים, אבל רק בסוף שנות ה-90 התרחב לקהיל הרחב. מכונה וירטואלית היא וירטואלייזציה של מחשב *VMware* שהנגישה את הרעיון הזה לקהיל הרחב. מכונה וירטואלית היא וירטואלייזציה של מחשב ותפקידה להרייך מכונה (ווירטואלית) קטנה יותר בתחום המכונה האמיתי.

נסתכל למשל על חברת שספקת שירותי לקוחות. בלי הפתרון הנ"ל, החברה צריכה לרכוש שרת אמיתי לכל לקוח. לעומת זאת, אם נרייך מספר מכונות וירטואליות על שירות מסוים, שככל אחת תשמש לקוחות אחר, יוכל לחסוך בCAPEX של מכונות אמיתיות ולתת פתרון למספר לקוחות בעלויות של שירות אחד בלבד. (מספר המכונות הווירטואליות שנוכל להרים על המכונה תלוי בחומרה ובכמות השימוש של לקוחות)

הцитוט הבא של עובדי IBM מתאר ומודגיר את הרעיון של מכונה וירטואלית:

"*A VM is an efficient, isolated duplicated of a real machine*" [Popek&Goldberg, 1974]

ונתח את מילوت המפתח בציוט:

- **Duplicate** – מכונה וירטואלית מתנהגת כמוו היא מכונה אמיתית. תוכנית שרצה במכונה

הוירטואלית לא תהיה מודעת לכך שהיא וירטואלית, היא תחשוף שהיא רצה "על הברזל".

ומשתמשת בחומרה אמיתית, למרות שהיא משתמשת בחומרה מדומה

(בפועל, ניתן לנחש על פי הביצועים שהתוכנית רצה בסביבה וירטואלית ולא אמיתית).

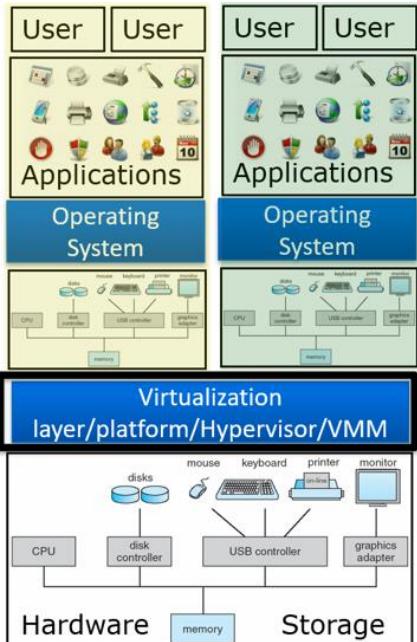
- **Isolated** – אם יש כמה מכונות וירטואליות שרצות על אותו שרת, כל מכונה תחשוף שהיא

היחידה. אף מכונה לא תדע על קיומה של מכונה אחרת. במידה ו-2 מכונות רצות לתקשר

ביןיהן, הן יפתחו ערז תקשורת בעזרת סוקטים כמו כל 2 מכונות מרוחקות שמתקשרות ביניהן.

נקודה זו קרייטית מאוד מבחינת אבטחה, מיידע מכונה אחת לא יכול לצלוג למכונה אחרת.

- **Efficient** – ריצה שהיעילות של המכונה המדומה תהיה דומה ליעילות של מכונה אמיתית.



באיור שמאל, השכבה התחתונה היא החומרה האמיתית של המחשב. מעליה קיימת שכבה נוספת, ה-*hypervisor*, שאחראית לדמות את החומרה האמיתית. מעל שכבה זו יש מספר מכונות וירטואליות שכל אחת מהן רצה כמכונה עצמאית עם חומרה וירטואלית, אבל בפועל הן כולן חולקות בחומרה האמיתית.

שכבה החומרה הוירטואלית כוללת *CPU*, זיכרון ו-*I/O*. אם ריצה למשל להשתמש במקלדת במכונה הוירטואלית, נעביר את הבקשה מהחומרה הוירטואלית **ל-hypervisor**, והוא ימפה את הבקשה לחומרה האמיתית. כמו כן, שכבה ה-*hypervisor* מאפשרת תמייה במספר מכונות וירטואליות, מספקת בידוד חזק ומנהל את המשאים הפיזיים.

למה VM זה נושא חשוב?

מכונה וירטואלית היא אחת הטכנולוגיות החשובות עבור הענן. *Infrastructure as a service* הוא ביטוי נפוץ בתחום שמתאר מתן תשתיות כשירות, או במילים אחרות מאפשר להקים סביבות העבודה לפיהו - מפרט המחשב, מקום, שעות ספציפיות ביום וכו'. נראה מס' פתרונות חשובים ש-VM מאפשר:

Server Consolidation

- מכונות וירטואליות חוסכות במספר השירותים האמיטיים ומונצל אותן כמו שיוטר.
- מכונות וירטואליות מבודדות אחת מהשנייה ולכן מאפשרות להקים מספר מערכות על אותן שרתות תוך שמירה על אבטחה מלאה.
- וירטואלייזציה חוסכת בחומרה ובאנרגייה – חברות שירותים גדולות כמו אמזון וגוגל משלימות **הmeno** שימוש כל חדש על השירותים שלהם, וכל שרת מסויף לעליות. בנוסף, ככל שיש יותר שירותים, המתבצעים בו הם נמצאים מתחם יותר, וכך נדרש לתחזק מערכות קירור עצומות שלוקחות עד שימוש. מכונות וירטואליות חוסכות הון לחברות שירותים גדולות.

Disaster Recovery

- אם קרה משהו לשרת (נשרף, נפל וכו') לא ניתן יהיה לשחזר אותו (אם אין לו גיבוי לשרת אחר וגם אז לא יהיה מהיר). לעומת זאת מכונה וירטואלית קל יותר לגבות ולהחזיר לשרת אחר במקרה הצורך, נרחב בהמשך.

High Availability

- מהסיבה הנ"ל, למכוון וירטואליות יש יותר זמינות ממכוון רגילים. למשל אם *Gmail* או *Whatsapp* מפסיקים לעבוד, תוך זמן קצר מאוד הם חוזרים לעבוד כי הם "דילגו" לשרת אחר.

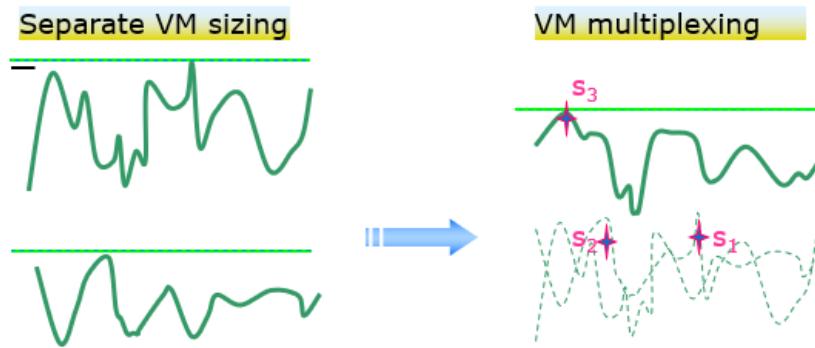
Testing and Deployment

- הרבה פעמים נרצה לבדוק דברים בסביבה קטנה ולא בחומרה האמיתית שיכולה להתקלקל, להיתקע או לדרש פעולה חדשה של המחשב. אם במהלך הבדיקה נפגע בחומרה הוירטואלית, לא יקרה כלום והמחשב עצמו לא יתקע.

Desktop Consolidation

- *Legacy* – ניתן להריץ במכונה וירטואלית מערכת הפעלה ישנה, ודרך להריץ תוכנות ישנות או משחקי ישנים (כאלו שלא יכולים לרוץ על המחשב המקורי ויעבדו רק בסביבות ישנות).
- פיתוח תוכנה

:VM workload multiplexing

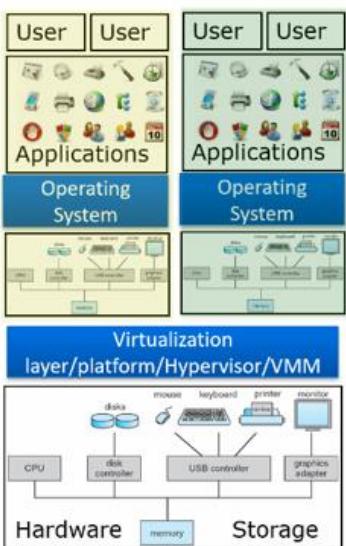


We expect $s_3 < s_1 + s_2$. Benefit of multiplexing !

באזור הנ"ל מצד שמאל יש לנו 2 גרפים שמתארים את עומס העבודה של 2 מכונות וירטואליות. מימין יש גרף של מכונה וירטואלית שהריצה את הפעולות של 2 המכונות משמאלי. אם S_1 הוא עומס העבודה המקורי של המכונה הווירטואלית הראשונה, ו- S_2 המקורי של השנייה, אם $S_3 \leq S_1 + S_2$ (המקסימום של המכונה הווירטואלית שמריצה את שניהם הוא לכל היוטר הסכום) אז בפועל, בגלל שכל מכונה ביצעה פעולות שונות בזמןים שונים, S_3 לרוב יהיה קטן הרבה יותר מהסכום שלהם, וככה אפשר להשתמש בשרת קצת יותר חזק במקום 2 שרתיים שלא מנוצלים עד הסוף.

:VM Encapsulation

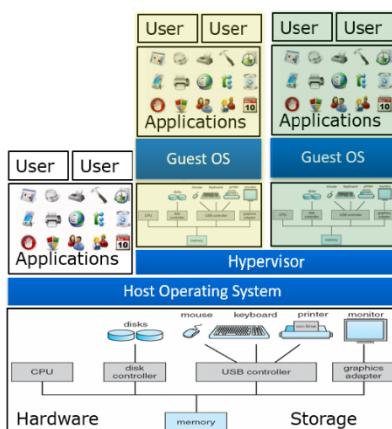
כל מכונה וירטואלית היא בפועל תוכנה (אין לכלי VM באנט *CPUs*, דרייברים וכו'), אך ניתן לשמור כל מכונה כזו כקובץ (באופן דומה לשימרת המצב עבור תהליך שמערכת ההפעלה עוזרת את הריצה שלו). לכל VM יש את המידע של ה-OS, אפליקציות, *data*, זכרון ומצלבים. ניתן להעביר את כל הקובץ למקום אחר ולהריץ אותו שם, או להריץ מספר עותקים שלו. כדי לקבל את הקובץ זהה נקבע *Snapshot* – שימירת המצב הנוכחי של ה-VM בקובץ. אם המחשב נפל, או שנרצה לשכפל את העותק של ה-VM יוכל לעשות זאת בקלות על ידי העברת הקובץ לשרת אחר. לפועלן העברת משרת אחד לאחר קוראים *Migration*. ככלומר, *VM Encapsulation* מוגדר גם *Rapid System Provisioning*, *migration*, וגם הקצאת מערכות דומות במהירות (*Rapid System Provisioning*) ומשימה. למשל אם יש עליה בדרישת עבור משימה מסוימת ניתן להקצתה עוד מכונות וירטואליות דומות למשימה.



סוג 1: *Hypervisors*

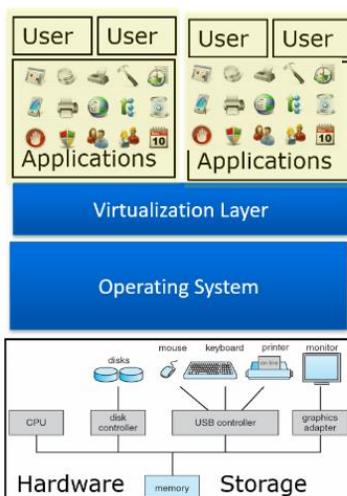
Type 1 Hypervisor / Native / Bare – metal .1

Hypervisor Type 1 פועל באופן הבא: מעל החומרה האמיתית יש שכבה של וירטואלייזציה, ומעליה נרץ מכונות וירטואליות הכוללות חומרה וירטואלית. מעל החומרה הוירטואלית של כל מכונה וירטואלית יש *Guest OS* – מערכת הפעלה אחרת. מעל מערכת הפעלה יש *Host OS* – ממשק הפעלה אחר. מעל *Host OS* יש אפליקציות ומעליה המשמש הירטואליות.



Type 2 Hypervisor / Hosted .2

Hypervisor Type 2 פועל באופן הבא: נרץ מכונות וירטואליות בתוך מערכת הפעלה ראשית (*Host OS*). מערכת הפעלה הראשית יכולה להריץ אפליקציות מהמשמש (צד שמאל באירור), ובנוסף קיימת שכבה *Hypervisor* שמעליה אפשר להריץ מכונות וירטואליות עם מערכות הפעלה וירטואליות כמו בסוג 1.



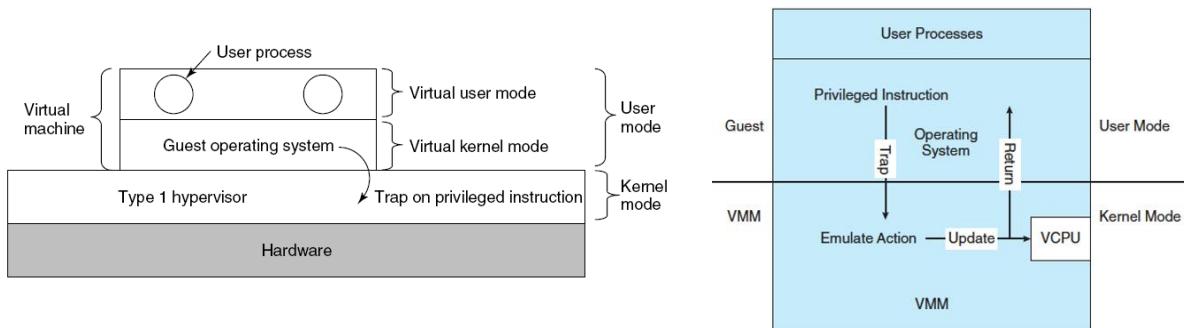
OS – level Virtualiz/ Container/ Docker .3

לפי גישה זו, יש שכבה אחת של חומרה אמיתית, מעליה מערכת הפעלה ומעל יש שכבה של וירטואלייזציה שמריצה אפליקציות בסביבות עבודה שונות. כל אחת מהאפליקציות עשויה וירטואלייזיה למערכת הפעלה. למכונה וירטואלית מסווג זה נקרא קונטינר, והוא רץ מעל מערכת הפעלה, מאוד דומה לתהליכים שרצים במקביל מעל מערכת הפעלה. ההבדל הוא שככל קונטינר הוא אוסף של תהליכים ולא תהליך יחיד, והוא גם נותן דברים נוספים כמו קונפיגורציות, התקנות של תוכנות (התקנות של קונטינר אחד לא משפיעות על קונטינרים אחרים) ועוד ..

בעיה אפשרית עם מכונות וירטואליות:

כפי שלמדנו, מערכות הפעלה משתמשות ב-*Kernel Mode/User Mode* כדי להגן על עצמן. *Privileged Instructions*, לא נוכל לבצע *User Mode*, אם מכונה וירטואלית היא תהילך שרך ב-*User Mode*, פסיקות וקריאות לפונקציות מערכת דרך ה-*Guest OS* כמו למשל לפתיחת קובץ. (ה-*Guest OS* מבחינתו שלוח ל-*CPU* פקודה לפתיחת קובץ אבל הבית ב-*CPU* יציין שמדובר ב-*User Mode* ולכן יזרוק שגיאה). יש לבעה זהז מספר פתרונות, וכיום מעבדים חדשים שתומכים בוירטואלייזציה מספקים הוראות חדשות המאפשרות ל-*VM* להתמודד עם *traps*. נראה מספר פתרונות אחרים לבעה.

Trap and Emulate – פתרון נפוץ בעיקר ל-*Type 1 Hypervisor Type*, זה שרך על החומרה האמיתית "Bare Metal" (בלי מערכת הפעלה שמתווכת באמצעות). לפי פתרון זה, המכונה הוירטואלית מריםה *Guest OS* - מערכת הפעלה שהיא ב-*Kernel Mode* (נקרא *Kernel Mode*) (ה-*Guest OS* מריםה הפעלה שחושבת שהיא ב-*User Mode*, מסמך שהיא ב-*User Mode* ולכן תזרק שגיאה אם היא מנסה לבצע פקודה *Privileged Instructions* (שהוא עצמו עובד ב-*Kernel Mode*) וזה הוא יבצע ע"י ה-*CPU*). את השגיאה יתפօן ה-*Hypervisor* (שהוא עצמו עובד ב-*Kernel Mode*) ויטופל על ידי ה-*Guest OS* ולא על ידי ה-*Hypervisor*.

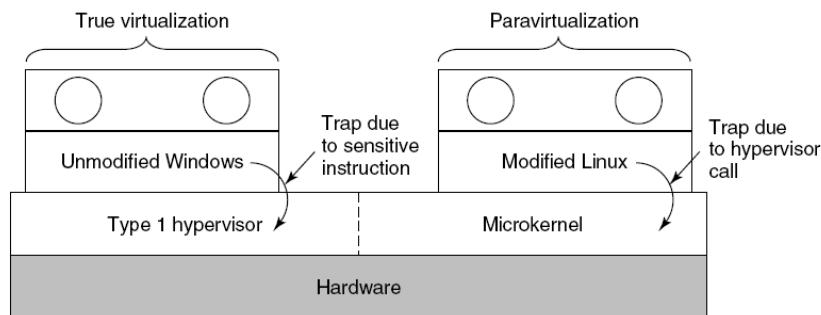


Binary Translation – לפי פתרון זה, אם אנחנו ב-*Virtual User Mode* הפקודות (הビנאריות) ירוצו בלי בעיה על ה-*CPU* הפיזי. אם אנחנו ב-*Virtual Kernel Mode*, ה-*Hypervisor* יבדוק מתיור אוסף של פקודות בינהיות, אם יש פקודות מיוחדות. (פקודה מיוחדת היא פקודה שיש לה משמעות שונה אם היא רצתה ב-*User Mode* מאשר ב-*Kernel Mode* כמו למשל מיקומי זיכרון שציריך לתרגם וכו') אם שום פקודה לא מיוחדת, נריץ אותה. אם יש פקודות מיוחדות, יש 2 אופציות:
 - נחליף אותן עם *traps*, נפעיל אותן, ואז נפעיל לפי גישת ה-*Trap and Emulate*.
 (כלומר זה שילוב של 2 הגישות)

- נתרגם את הפקודות הבינהיות לסט חדש של פקודות בינהיות נכונות ונריץ אותן.
- כלומר נבצע את ה-*Emulate* כבר בשלב זה, עם הפקודות הבינהיות.

עבור 2 *Hypervisor Type*, *Binary Translation* עם *Hypervisor Type 2* רץ בתוך מערכת הפעלה. הוא קורא את הקוד בבלוקים, ומփש בבלוקים *Privileged Instructions*. אם הוא מצא כאלו, הוא מתרגם אותם (*Binary Rewriting* או *Binary Translation*). לאחר מכן הוא מעלה את הבלוק ל-*Cache* ומריץ אותו. בצורה זו, כל הבלוקים שעברו שינוי יהיו ב-*Cache*, ובפעם הבאה שהוא יתקל בה הוא לא יצטרך לתרגם אותם שוב. כלומר מהירות תיגע רק בהתחלה.

- הפתרון היעיל ביותר. אמרנו שנוכל לזהות מתי מדובר בסביבה וירטואלית (בגלל הביצועים), אך נוכל לשנות את ה-SI-*Guest OS* כך שכל הקрайות ל-*Hypercalls* ישתנו ל-*Hypervisor Calls*. בדומה זו נפחית את מספר ה-*traps*. הרבה יותר קל (ויעיל) לשנות את ה-*Source Code* (רשום בשפה עילית כלשהי) שיתאים להצעה הנ"ל, מאשר לתרגם הוראות בינהיות להוראות אחרות. הבעיה בכך היא שמספר ה-*Guest OS* במיוחד לכל *Host OS*, ככלומר המכונה הירטואלית כן תהיה מודעת לזה שהיא וירטואלית. מסיבה זו נctrar גם לוודא לפניה כל *Migration* (מעבר לשרת אחר), שהשרת הזה מרים *Host OS* דומה לזה שהוא עליון עכשו. זה מוסיף המון פיתוח ופוגע במידולריות.

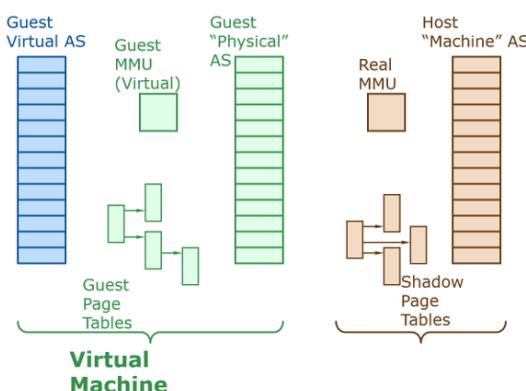


Hardware Assistance

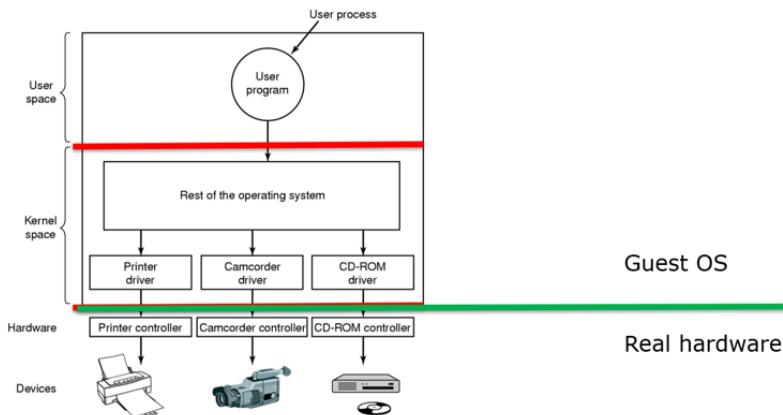
כפי שהזכרנו, היום בדרך כלל יש תמיכה של החומרה עצמה שמאפשרת וירטואלייזציה במעבד. ש-*Privileged Mode* חדש לווירטואלייזציה, הוראות חדשות (*vmrun*, *vmexit*), ומבנה נתונים חדש: *VM control block*. בדומה זו זה המעבד יודע שהוא מרים מכונה וירטואלית.

2 נקודות נוספות שצריך לטפל בהן בהקשר של מכונות וירטואליות

Virtualizing Virtual Memory – למcona וירטואלית שמריצה *Guest OS* יש מרחב כתובות מסוימת בזיכרון (היא לא משתמשת בכל הזיכרון הפיזי כמובן). אבל בנוסף, היא צריכה להקצות מרחב כתובות לתהליכים שרצים אצלה, וכן לשכבה נוספת נספotta של כתובות. בכל פעם שתהילך שרך ב-*OS* ניגש לכתובות בזיכרון, יש (*VMMU*) *Virtual MMU* שמתרגם את הכתובת הירטואלית שלו לכתובת שהוא חושב שהיא הכתובת האמיתית, אז ה-*MMU* הרגיל מתרגם את הכתובת הזאת לכתובת האמיתית שנמצאת בזיכרון הפיזי של המחשב. (ופועלות נוספות שלא נרchieb עליון בקורס זה)

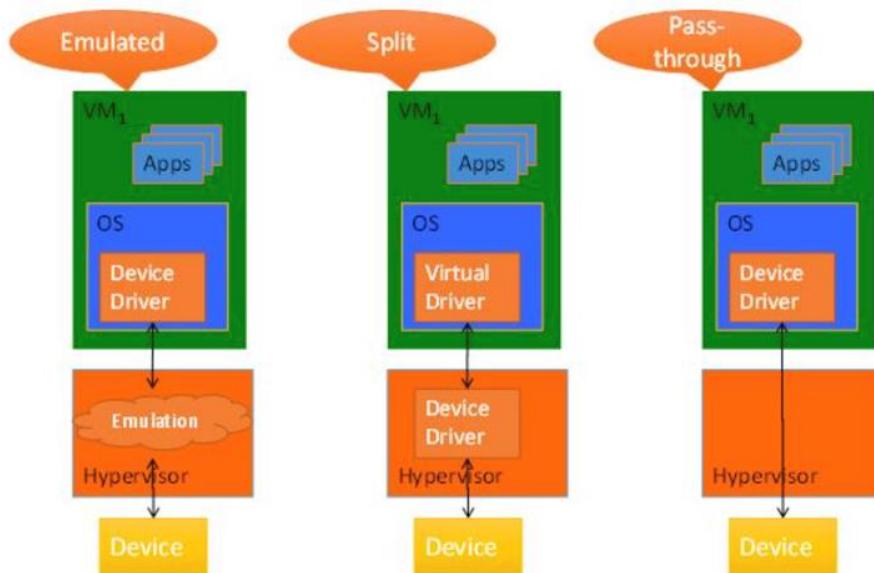


I/O Virtualization - בעולם הווירטואלי, כדי שהתקנים חיצוניים יתקשרו עם הקרנל, נצטרך להויסף *Hypervisor* ממשק כלשהו שייר לשבבת ה-*I/O Virtualization*.

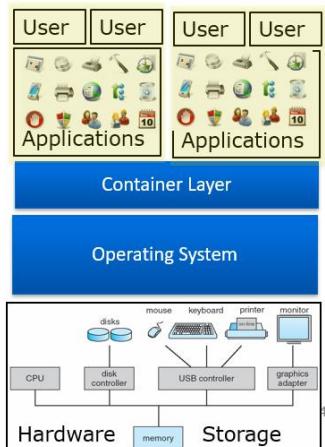


3 שיטות מרכזיות לאפשר זאת:

- **Emulated** (אמולציה) – סוג של *Trap and Emulate*, הדריבר מתוך מערכת ההפעלה מפעיל פקודה, ה-*Hypervisor* מקבל אותה ועשה אמולציה.
- **Split** – סוג של *Paravirtualization* בرمת הדריבר, הדריבר יודע שהוא וירטואלי ולא אמיתי, והוא מתקשר בעדרת ממשק פשוט עם ה-*Hypervisor* האמתי. בדומה זאת דרייברים וירטואליים לא צריכים למשש שוב את הלוגיקה של הדריברים האמיתיים.
- **Pass Through** – לפי גישה זו, ה-*OS* מקבל גישה ישירה לתקני ה-*I/O* (בכך שהדריבר מותקן ישירות על המכונה הווירטואלית והוא לא עובר דרך ה-*Hypervisor*). משתמשים בגישה זו בעיקר ברשותות תקשורת, איפה שעוברות כמויות גדולות של מידע ושיבוט לביצועים.



Containers



קונטינרים הם שלב בין תהליכיים לבין מכונות וירטואליות. למשל, תהליכיים חולקים ב-*File System* וקונטינרים לא. לדוגמה, בדרך כלל יש: אפליקציות, תלויות (*Dependencies*), ספריות, קבצים ביןאריים וקבץ קונפיגורציה. קונטינרים מבצעים וירטואלייזציה למערכת הפעלה ולא לחומרה. לכל קונטינר יש את ה-*File System* שלו, זיכרון משלה וקבצים של מערכת הפעלה שمدמים אליו הוא היחיד שרש על מערכת הפעלה. ניתן להבהיר קונטינר משרת אחד לאחר, אבל רק אם הם משתמשים באותה מערכת הפעלה.

:Containers vs Virtual Machines

- מכונות וירטואליות מכילות מערכת הפעלה מלאה (כלומר את כל הקוד, למשל כל LINQ'ס).
- ברוב המקרים זה לא נחוץ ומספיק היה להשתמש בסביבה הקיימת. בקונטינר נשתמש בכל מה שאנו יודעים ממערכת הפעלה הקיימת, אין שכבה של *Hypervisor*, ולכן יש פחות הוראות *indirection*, וכמו כן, זה משפר את זמן הריצעה.
- מכיוון ש-*VM* עובד עם מערכת הפעלה נפרדת, יש יותר בידוד מאשר בקונטינרים.
- *VM* יכול לרוץ על כל שרת ואילו קונטינר חייב לרוץ רק על המערכת הפעלה שהוא גבנה אליה.
- *VM* יכול להריץ מערכות הפעלה שונות, ואילו קונטינר יכול להריץ רק את ה-*Host OS*.
- קונטינרים הינם וירטואלייזציה של מערכת הפעלה, ואילו *VM* הוא וירטואלייזציה של החומרה.

יתרונות:

- קונטינרים הרבה יותר קטנים (שוקלים פחות).
- בקונטינר יש הרבה פחות *Resource Intensive*.
- הרבה יותר קל ליצור קונטינר (מעט ברמת יצירה של תהליך), לוקח כמה שניות ואףלו פחות.
- קונטינרים מאפשרים להקנות משאבי לתהליכיים בקצבות, ולהריץ אפליקציות בסביבות שונות.
- באופן כללי קונטינרים יותר מהירים. הפיתוח, ניצול המשאים, בדיקה והפרישה של האפליקציות והשירותים יותר מהירים. כל אלו חוסכים זמן וכסף.

חסרונות:

- **אבטחה.** קונטינרים מספקים פחות בידוד, ולכן פחות אבטחה למי שמשתמש בהם.
- ***VM* חולקים רק את ה-*Hypervisor*,** ולכן יש פחות סיכוי לפגיעה אחד בשני.
- **פחות פונקציונליות.** כל הקונטינרים באותו השרת מרכיבים את אותה מערכת הפעלה ולא ניתן להעביר אותם למערכת הפעלה אחרת.

תרגול 14

תרגול 14 כולל שאלות ופתרונות מבחנים של שנים קודמות – ספיילים ! אז כדאי לעبور על המציג/התרגול המוקלט אחרי שמתנסים בשאלות האלה לבד.