

**נהל אתחול (Boot loader)** - היא תוכנה המופעלת כחלק מתהליך האתחול של מחשב וטוענת את מערכת ההפעלה. התוכנה שמורה במיקום מוגדר בזיכרון ההתקן והיא פשוטה וקטנה בהרבה ממערכת ההפעלה, ולכן ניתן לטעון אותה בקלות רבה יותר. ישנם שני מנהלי אתחול הפועלים זה לאחר זה. מנהל האתחול הראשון צרוב כקושחה בזיכרון כדוגמת ROM, והוא פונה לזיכרון אחסון (כגון מחיצה בדיסק הקשיח, החסן נייד) בו קיים מנהל אתחול תוכנית נוסף הממשיך את התהליך.

**זיכרון גישה אקראית (RAM)** - ראשי תיבות של **Random Access Memory** הוא שם כללי למספר רב של סוגי זיכרון מחשב, המתאפיינים כולם ביכולת המעבד לגשת ישירות לכל תא בזיכרון לפי כתובתו, לכתוב בו ולקרוא ממנו. ההתייחסות הנפוצה לזיכרון מחשב היא למעשה ההתייחסות לזיכרון הגישה האקראית הראשי שלו. סוג הזיכרון הנמצא בשימוש נפוץ ביותר בימינו הוא DRAM. זיכרון מסוג זה הוא זיכרון נדיף, (volatile) כלומר מאבד את תוכנו עם ניתוק הזיכרון ממקור האנרגיה שלו. בנוסף, DRAM דורש רענון (refresh) מספר פעמים בשנייה כדי לשמור את תוכנו.

**זיכרון לקריאה בלבד ROM** - ראשי תיבות של **Read-only Memory**, הוא אמצעי לאחסון נתונים, דמוי זיכרון מחשב, המכיל נתונים הנכתבים בו פעם אחת, ונקראים ממנו פעמים רבות. שבבים אלה אינם ניתנים לכתובה חוזרת (בצורה פשוטה), ומכאן השם קריאה-בלבד. בניגוד לRAM, תוכן ה-ROM נשמר גם לאחר ניתוק מקור מתח. הגישה ל-ROM פשוטה יותר ומהירה בהרבה מאשר גישה לאמצעי אחסון מגנטיים ואופטיים כגון דיסק קשיח או תקליטור. בעבר שימש ה-ROM לאחסון מערכת ההפעלה כולה במחשבים אישיים. בהמשך נותרו ל-ROM מספר תפקידים עיקריים:

- תוכנת אתחול (Boot): תוכנה שמטרתה לבצע את הפעולות הראשונות בזמן הדלקת המחשב, כולל בדיקת תקינות רכיבי המחשב וטעינת מערכת ההפעלה לזיכרון.
- BIOS - אוסף רוטינות אשר תומכות בפעולות הקלט והפלט הבסיסיות של המחשב.
- צריבה של תוכנה קבועה על רכיבים המיועדים לביצוע פעולות מוגדרות מראש, כגון במעגל משולב תלוי יישום.

כיום, יותר ויותר פעולות במחשב שמולאו בעבר על ידי שבבי ROM (כולל תוכנת ה-BIOS) מבוצעים על ידי זיכרון הבזק, שיתרונו בכך שהוא מחיק (לא כמו ROM) ואינו תלוי בחיבור למקור מתח כמו זיכרון נדיף כך שניתן לעדכן את התוכנה המאוחסנת בו לגרסה חדשה יותר ללא צורך בהחלפתו כפי שיש לעשות כאשר משתמשים ב-ROM.

**מעבד (CPU)** - או בשמו המלא **יחידת עיבוד מרכזית (CPU - Central Processing Unit)** הוא רכיב חומרה במחשב המבצע את הפקודות המאוחסנות בזיכרון המחשב.

בהתאם לארכיטקטורת פון נוימן הפקודות אותן מקבל המעבד מאפשרות לו קריאת מידע מהזיכרון או מהתקנים שונים, ביצוע פעולות חשבוניות ולוגיות על מידע זה וכתובת תוצאות החישוב בחזרה לזיכרון או לחלופין שליחתו להתקנים חיצוניים. הפעולות הלוגיות מאפשרות בקרת זרימה וחזרה על פקודות ככל שנדרש. הפקודות הן בסיסיות ביותר ובנויות, כל אחת, מרצף קצר של ביטים. רצף זה קרוי שפת מכונה. כל דגם של מעבד מתאפיין בסט פקודות משלו. בהתאם לתזת צ'רץ'-טיורינג דלות שפת המכונה אינה מהווה מגבלה בביצוע תוכנית מחשב כלשהי וההבדל בין מעבדים שונים יבוא לידי ביטוי רק בביצועים.

**אוגרים או רגיסטרים (Registers)** - כל אוגר הוא יחידה אחת של זיכרון פנימי מהיר ביותר הנמצא לרוב בתוך ה-CPU של מחשב אשר מאפשר אחסון ערכים, בדרך כלל זמנית, עבור פעולות בסיסיות שונות מסט הפקודות של המעבד (חיבור, חיסור, והשוואה). יש מעבדים בהם האוגרים הם ייעודיים, כלומר פעולות מסוימות מוגבלות לאוגר או אוגרים מסוימים, ולעומתם מעבדים אחרים (בדרך כלל מסוג RISC) בהם לכל האוגרים פונקציונליות זהה. מספר האוגרים במחשב נמוך יחסית ולכל היותר יגיע לעשרות מעטות.

**זיכרון מטמון (Cache)** - זיכרון מהיר במיוחד, המותקן על המעבד. יש מעבדים בהם זיכרון המטמון מוגבל לפקודות בלבד, אחרים בהם המטמון משותף לפקודות ונתונים, ואחרים בהם יש מטמונות נפרדים לקוד ולנתונים. נהוג לסווג את זיכרון המטמון הבנוי על שבב המעבד לשלושה סוגים. יש מעבדים ללא מטמון, או עם חלק אך לא כל סוגי המטמון המתוארים:

- **L0** - המטמון המהיר ביותר, אך גם המוגבל ביותר בגדלו. בדרך כלל פועל בקצב השעון של המעבד עצמו.
- **L1** - מטמון פחות מהיר וגדול יותר מ-L0

- **L2 - מטמון מהיר פחות וגדול יותר מ-L1** בשבבים מרובי מעבדים, לפעמים מטמון L2 משותף למספר מעבדים.

**מערכת הפעלה -** היא תוכנה המנהלת את משאבי החומרה והתוכנה במחשב. בנוסף, מערכת ההפעלה מספקת את התשתית הנחוצה להרצה של יישומי ההפעלה המתבצעת עם הדלקת המחשב, הקרויה אתחול. מערכת ההפעלה היא רכיב חיוני בכל מחשב.

מערכת ההפעלה מספקת שלושה ממשקים:

1. ממשק משתמש (User Interface)
2. ממשק עבור החומרה על ידי מנהלי התקנים
3. ממשק תכנות היישומים (API).

ניתן למנות שלושה תפקידים עיקריים של מערכת ההפעלה:

1. הקצאת משאבי החומרה
2. תזמון פעולות רכיבי החומרה ומרכיבי התוכנה.
3. העמדת תשתית משותפת ומסגרת מאורגנת של ממשק ושירותים למשתמש ולחבילות התוכנה.

### – Extended Machine

המכונה מספקת:

- יציבות: לא מתרסקת.
- ניידות: יכולה להריץ קוד ביותר מסוג אחד של מכונות.
- אמינות: תמיד מגיבה באותו אופן.
- בטיחות: לא עושה משהו מסוכן.
- התנהגות טובה: פועלת בסביבה נאותה.

### Resource Manager – תומך במכשירים רבים בו זמנית

למשל: מקלדת, עכבר, מדפסת, רמקולים, מיקרופון.

משתף משאבים בין משתמשים ותוכניות באופן הוגן: כל תוכנית זוכה לשינוי להפעלה, בבטחה: מגן מפני שחיתות, ביעילות: שימוש במשאבים הזמינים כדי לספק את השירות הטוב ביותר. מסוגל להקצות משאבים למשתמשים כגון דיסקים, זיכרון, ממשקי רשת, טיימרים, מסופים / צגים, מדפסות לייזר וכו'.

**פסיקה (Interrupt) -** היא אות המגיע למעבד, מצד רכיב חומרה או תוכנה ומורה להפסיק תהליך מסוים בכדי לעבד תהליך אחר הדורש טיפול חשוב יותר (לאחר מכן המחשב יחזור לעבד את התהליך שהופסק). בעת קבלת הפסיקה משהה המחשב את ביצועה הסדרתי של התוכנית, כדי להפעיל שגרת טיפול בפסיקה. לאחר הטיפול בפסיקה, ממשיך המחשב בביצוע הסדרתי של התוכנית. הדבר דומה לאדם המבצע מלאכה כלשהי, ומפסיק כדי לענות לשיחת טלפון, ולאחר סיום השיחה, ממשיך במלאכתו מהנקודה שהופסקה. פסיקות תוכנה נמצאות בשימוש נרחב במחשבים הפועלים בריבוי משימות ומהוות הפסקת פעולת תוכנית מחשב שלא דרך בקרת זרימה.

**Interrupt handler -** מטפל בפסיקה, הידוע גם בשם ISR, הוא גוש קוד מיוחד המשויך למצב הפרעה ספציפי. Interrupt handlers יזומים על ידי הפרעות חומרה, הוראות להפסקת תוכנה או חריגות תוכנה ומשמשות ליישום device drivers או מעברים בין מצבי פעולה מוגנים, כגון system call.

**קריאת מערכת (system call) -** היא בקשה שמבצעת תוכנת מחשב מליבת מערכת ההפעלה (kernel) כדי לבצע פעולה שהיא אינה יכולה לבצע בעצמה. קריאות מערכת הן האחראיות על החיבור שבין המשתמש לליבת המערכת, ובכך מאבזרת את המשתמש ונותנת לו שימוש מרבי בפונקציונליות שהיא מציעה. הדבר כולל בין היתר יכולת קבלת גישה לרכיבי חומרה (למשל קריאת קובץ מהדיסק הקשיח), ליצירת תהליך חדש, להעברת מידע בין תהליכים ועוד.

**מנהל התקן (driver) -** תוכנית מחשב המאפשרת לתוכנית מחשב אחרת, לתקשר עם חומרה כלשהי או עם תוכנה אחרת הפועלת בפורמט שונה באמצעות מימוש הממשק שלה ומתן API לעבודה מולו.

**ממשק תכנות יישומים (API)** – ראשי תיבות Application Programming Interface, הוא ערכה של ספריות קוד, פקודות, פונקציות ופרוצדורות מן המוכן, בהן יכולים המתכנתים לעשות שימוש פשוט, בלי להידרש לכתוב אותן בעצמם כדי שיוכלו להשתמש במידע של היישום שממנו הם רוצים להשתמש לטובת היישום שלהם.

**מרחב משתמש (user space)** - הוא חלק בזיכרון המחשב המוקצה על ידי מערכות הפעלה עבור תוכנות בשימוש המשתמש, וזאת להבדיל ממרחב הליבה המשמש רק את ליבת מערכת ההפעלה ומנהלי התקנים. ברוב מערכות ההפעלה זיכרון זה עשוי להיות מועבר לזיכרון משני כאשר תם הזיכרון הפיזי הפנוי. בדרך כלל כל תוכנית שרצה מקבלת מרחב זיכרון וירטואלי משלה בתוך מרחב המשתמש ואין לה גישה לזיכרון של תוכניות אחרות, מסיבות של יציבות ואבטחה.

**ליבה (Kernel)** - היא הרכיב המרכזי של מרבית מערכות ההפעלה. זהו הגשר שבין תוכניות המחשב לבין עיבוד הנתונים עצמו שמבוצע ברמת החומרה. אחד התפקידים העיקריים של הליבה הוא ניהול משאבי המערכת (התקשורת שבין רכיבי החומרה והתוכנה). בדרך כלל, בתור הרכיב הבסיסי של מערכת הפעלה, הליבה יכולה לספק את שכבת האבסטרקציה ברמה הנמוכה ביותר עבור המשאבים (בייחוד עבור מעבדים והתקני קלט-פלט) שהתוכניות צריכות לשלוט עליהם על מנת לבצע את תפקידן.

בדרך כלל הליבה הופכת שירותים כאלה לזמינים עבור תהליכים של תוכניות באמצעות מנגנוני תקשורת בין תהליכים (inter-process communication) וקריאות מערכת. המשימות של מערכות הפעלה מבוצעות באופן שונה על ידי ליבות שונות, בהתאם לעיצוב ולמימוש שלהן. בעוד שליבות מונוליתיות (Monolithic Kernels) מריצות את כל הקוד של מערכת ההפעלה באותו מרחב כתובות על מנת לשפר את ביצועי המערכת, מיקרו-ליבות (Microkernels) מריצות את מרבית שירותי מערכת ההפעלה במרחב המשתמש כשרתים, במטרה לשפר את האמינות והתחזוקתיות של מערכת ההפעלה. קיים מגוון של אפשרויות בין שתי הדוגמאות הקיצוניות האלה. ליבת מערכת ההפעלה היא התוכנית היחידה אשר מוכנה לריצה בכל זמן שהוא. בנוסף, הליבה היא התוכנה היחידה שיכולה לבצע אוסף פקודות בצורה אטומית (ללא שום הפרעה או הפסקה מגורם כלשהו). כל מערכת הפעלה חייבת ליבה כדי לפעול, אך הליבה אינה בהכרח ייחודית למערכת ההפעלה - מערכות הפעלה שונות יכולות להשתמש בליבה זהה. לדוגמה, על ליבת לינוקס ניתן להריץ הפצת לינוקס שולחנית אך גם אפשר להריץ את מערכת ההפעלה אנדרואיד למכשירים ניידים.

**תהליכון (Thread)** - נקרא גם חוט, מושג במדעי המחשב המשמש במערכות הפעלה כדי לתאר הקשר ריצה במרחב כתובות. מערכות הפעלה מודרניות מאפשרות לנהל במסגרת ריצה של תהליך (Process) מספר תהליכונים הרצים במקביל במרחב כתובות אחד. במערכות אלו כל תהליך חדש מתחיל את ביצועו באמצעות 'תהליכון ראשי' אשר עשוי בהמשך ליצור תהליכונים נוספים. מנגנון הריצה באמצעות תהליכונים מאפשר לספק למשתמש במערכת ההפעלה מהירות תגובה ורציפות פעולה כאשר התהליך (Process) מבצע כמה משימות במקביל. למה צריך threads? תהליך אחד יהיה איטי מדי ושני תהליכים לא יכולים לגשת לאותו מקום בזיכרון.

יתרונות:

- שיתוף קבצים פתוחים, מבני נתונים, משתנים גלובליים, תהליכי ילדים (child processes) וכו'.
- תהליכונים עמיתים (peer threads) יכולים לתקשר ללא שימוש ב system calls.
- תהליכונים מהירים פי 10-100 ליצירה (create) / סיום (terminate) / החלפה (switch) מאשר תהליכים.

חסרונות:

- אבטחה ויציבות: קבצים פתוחים, מבני נתונים, משתנים גלובליים, תהליכי ילדים וכו' משותפים בין כל התהליכונים הנמצאים תחת אותו process.
- איתות (signal) לתהליכון מסוים משפיע על כל התהליכונים ב process זה.

**תהליך (Process)** - מופע של תוכנית מחשב שמופעל על ידי מערכת מחשב שיש לה היכולת להפעיל מספר תהליכים בו זמנית. תוכנית מחשב היא בעצמה רק אוסף פקודות, בעוד שתהליך הוא ההפעלה של אותן פקודות. כך למשל, הפעלה של מספר מופעים של אותה תוכנה יגרום לעיתים קרובות למספר תהליכים של התוכנה להיפתח בו זמנית. כדי לאפשר למשתמש להפעיל מספר תהליכים במקביל, מתבצעת על ידי סדרן התוכניות (Scheduler) חלוקת זמן מעבד בין התהליכים.

בסיום ריצתו, התהליך מחזיר ערך יציאה (Exit status) שמאפשר לתהליך האב שיצר אותו לקבל מידע על סיום מוצלח של התהליך או לחלופין ערך שגיהא המעיד על סוג הכישלון שמנע מהתהליך להסתיים בהצלחה. כל תהליך מכיל מרחב זיכרון וירטואלי, תהליכון אחד לפחות, page table (מפת זכרונות - שומר את כל הקצאות הזיכרון), מחסנית, ערימה, data של התוכנה שהפעילה את התהליך, File/Socket, Signals, משתנים גלובליים, הרשאות, id ועוד.

תהליך שסיים את פעולתו אך עדיין קיים בטבלת התהליכים כדי שיתאפשר לתהליך האב לקרוא את ערך היציאה שלו מכונה תהליך זומבי והתהליך האב אוסף אותו בעזרת פקודת wait(). בהתחלה התהליך "נוצר" (create) על ידי טעינתו מהתקן אחסון משני (כונן דיסק קשיח) לזיכרון הראשי. בכל רגע נתון התהליך נמצא באחד משלושת המצבים הבאים:

1. Running - פועל (למעשה משתמש במעבד באותו רגע).
2. Ready - מוכן (ניתן להפעיל; הופסק זמנית כדי להפעיל תהליך אחר).
3. Blocked - חסום (לא מצליח לרוץ עד שמתרחש אירוע חיצוני כלשהו).

ולבסוף ברגע שהתהליך מסיים את הביצוע, או שמסתיים על ידי מערכת ההפעלה, אין בו עוד צורך. התהליך מוסר באופן מיידי או מועבר למצב "הסתיים" (terminated). כשהוא מוסר, הוא רק ממתין להסרתו מהזיכרון הראשי.

| Process                             | Thread  |
|-------------------------------------|---|
| Data ייחודי                         | Data משותף  |
| Code ייחודי                         | Code משותף  |
| Open I/O ייחודי                     | Open I/O משותף  |
| טבלת signals ייחודית                | טבלת signals משותף  |
| מחסנית ייחודית עבור כל thread שלו   | מחסנית ייחודית בהשאלה מהמחסנית של process*                    |
| PC ייחודי                           | PC ייחודי   |
| Register ייחודי                     | Register ייחודי   |
| מצב/סטטוס ייחודי                    | מצב/סטטוס ייחודי  |
| Context switch כבד                  | Context switch קל   |
| ערימה                               | TLS/TSS (thread safe segment & thread safe storage)           |
| משתנה גלובלי (environment variable) | CPU – affinity mask "להדביק" את thread לליבה/סט ליבות מסויים. |
| Command line                        | Thread priority   |
| Page table                          |   |
| TLB                                 |   |

\*Signal handlers חייב להיות משותף בין כל הthreadים בprocess, עם זאת לכל thread יש mask של blocking/pending signals.

לסיכום: thread הוא לריצה, ו process הוא מעטפת משאבים שכוללת גם יכולות ריצה בעזרת אחד או יותר threads.

## שירות מקבילי (Multiprogramming) –

1. משפר את ניצול המעבד – computation, communication(I/O) & overlapping.
2. שירות סודו-מקבילי (במקרה של מעבד בודד).
3. שירות מקבילי אמיתי (במקרה של מעבד רב ליבות).
4. משפר משימות אינטראקטיביות.

## ניהול זיכרון (Memory management) - הפונקציה האחראית על ניהול הזיכרון הראשי של המחשב.

פונקציית ניהול הזיכרון עוקבת אחר הסטטוס של כל מיקום זיכרון, שהוקצה או חינמי. הוא קובע כיצד מוקצה זיכרון בין התהליכים המתחרים, מחליט מי מקבל זיכרון, מתי הם מקבלים אותו וכמה הם רשאים.

כאשר מוקצה זיכרון הוא קובע אילו מיקומי זיכרון יוקצו. הוא עוקב אחרי הזיכרון שהשתחרר או שאינו מוקצה ומעדכן את הסטטוס שלו.

זה שונה מapplication memory management (ניהול זיכרון יישומים) שזה איך process מנהל את הזיכרון שהוקצה לו על ידי מערכת ההפעלה.

**החלפת הקשר (context switch) -** מעבר בין הרצת שני תהליכים באמצעות המעבד. באמצעות החלפת הקשר, מספר תהליכים יכולים לחלוק את אותו מעבד. החלפת הקשר שומרת את מצב האוגרים במעבד בזיכרון המחשב, ולאחר מכן מכניסה למעבד את נתוני ריצת תהליך אחר, לאחר מכן, מחזירה את נתוני הריצה של התהליך הקודם אל המעבד וממשיכה את ריצתו מאותה הנקודה בה הפסיק. החלפת הקשר היא כלי חיוני במחשבים הפועלים בריבוי.

משימות הרצות על מעבד יחיד. תהליך החלפת ההקשר נחשב לתהליך בזבזני מבחינת משאבי מערכת ועל כן מערכות הפעלה מנסות לבצע אופטימיזציה בשימוש בהן.

קיימים שלושה תרחישים בהם יש צורך בהחלפת הקשר:

- במחשב הפועל בריבוי משימות, קיים מתזמן (scheduler) הקובע את סדר הרצת התוכניות ומקצה "זמן מעבד" לכל תהליך. כש"זמן מעבד" של תהליך אחד מסתיים, מופעלת פסיקה (interrupt) שתגרור החלפת הקשר.
- ארכיטקטורות מסוימות (למשל ארכיטקטורת x86 של אינטל) הן מונעות-פסיקה. משמעות הדבר היא שבמידה והמעבד צריך למשל לבצע קריאה מהדיסק, הוא ישלח את בקשת הקריאה ויעבור לבצע פעילות אחרת במקום להמתין. עם סיום הקריאה מהדיסק תופעל פסיקה שתגרור החלפת הקשר. הפסיקה מהדיסק תטופל על ידי interrupt handler.
- במעבר בין מצב משתמש (user mode) למצב ליבה (kernel mode), תלוי במערכת ההפעלה, תיתכן החלפת הקשר.

שלבי העבודה של context switch:

1. אמור למעבד להפסיק להפעיל את המשימה וסמן את המשימה כנעצרה.
2. שמור registers: PC(program counter), SP(stack pointer), FP(frame pointer) נוספים של המשימה.
3. טען רישומי registers: PC, SP, FP מהמשימה המתקרבת.
4. אופציונלי: בצע flash ל TLB ב MMU(memory management unit) ו/או עדכן מטמונים(caches).
5. עדכן את המשימה החדשה כפועלת
6. התחל לבצע את התהליך החדש.

הערה: בthreads במרחב הגרעין (kernel) scheduler של המערכת הפעלה אחראי על context switch, ובthreads במרחב user scheduler user ממש או ספריה כלשהי אחראית על context switch. בLinux כאשר המשתמש פותח thread מערכת ההפעלה דואגת עבורו לthread ברמת הגרעין.

### **User lever threads – מיושם בספריות ברמת המשתמש.**

החלפת תהליכונים אינה צריכה לתקשר עם מערכת ההפעלה או לגרום להפרעות. היישום של המשתמש מתזמן את זמן המעבד של התהליך לתהליכונים הפנימיים שלו. משתמש רק במעבד יחיד, שכן מערכת ההפעלה לא תקצה מספר מעבדים לתהליך אחד. הגרעין אינו יודע דבר על threads ברמת המשתמש ומנהל אותם כאילו היו תהליך בעל thread בודד. (בניהול ע"י התהליך, התהליכונים שקופים למערכת ההפעלה ו-system call מאחד התהליכונים יקפיא את כל התהליך מכיוון שלמערכת ההפעלה אין היכרות עימם אלא רק עם התהליך עצמו והיא מקפיאה אותו, עבודה רק על מעבד יחיד כיוון שהמעבד רואה יישות אחת, ולא הגיוני שיקצה לה שני מעבדים).

#### יתרונות:

- תאימות: ניתן להטמיע במערכת הפעלה שאינה תומכת בthreads.
- ייצוג פשוט: כל thread מיוצג בפשטות על ידי PC, registers, מחסנית וblock בקרה קטן, הכל מאוחסן במרחב הכתובות של תהליך המשתמש.
- ניהול פשוט ומהיר: יצירת thread, סנכרון ומעבר בין threads יכולים להיעשות ללא התערבות של הגרעין, ולכן הם זולים יותר ומהירים פי 100 בהשוואה לשרשורים ברמת הגרעין.

#### חסרונות:

- חוסר תיאום בין thread לבין ליבת מערכת ההפעלה. לכן, התהליך בכללותו מקבל פרוסה חד פעמית של זכרון ללא קשר לשאלה האם לתהליך יש thread אחד או 1000 threads.
- קריאת מערכת (system call) באחד מהthreads גורמת למערכת ההפעלה לחסום את כל התהליך, גם אם נותרו תהליכונים ניתנים להפעלה בתהליכים אלה.
- חוסר היכולת של הגרעין להבחין בין תהליכונים ברמת המשתמש מקשה על תכנון מתזמן (scheduler) בין תהליכונים מאותו התהליך.

### **Kernel lever threads – כל התהליכונים גלויים לליבה.**

הגרעין מנהל ומתזמן את התהליכונים. קיימות קריאות מערכת ליצירה וניהול תהליכונים.

#### יתרונות:

- הליבה יכולה לתזמן בצורה חכמה בין תהליכים עם מספר תהליכונים שונה.
- תהליכונים ברמת הליבה טובים במיוחד עבור יישומים החוסמים (blocks) לעתים קרובות.
- במעבד מרובה ליבות, כמה מעבדים יכולים להריץ בו זמנית תהליכונים שונים מאותו התהליך.

- יצירה, ניהול והחלפת תהליכים הם הרבה יותר יקרים ואיטיים מאשר ברמת המשתמש.

### Fork – קריאת מערכת שבה תהליך יוצר עותק של עצמו.

Fork() היא השיטה העיקרית ליצירת תהליכים במערכות הפעלה דומות ליוניקס.

מזלג () יוצר תהליך חדש על ידי שכפול process שקרא לפונקציה.

התהליך החדש מכונה תהליך הילד, תהליך ההתקשרות מכונה תהליך האב.

תהליך הילד ותהליך ההורה מתנהלים במרחבי זיכרון נפרדים.

בזמן מזלג () שני חללי הזיכרון מכילים את אותו התוכן.

כתיבה לזכרון או מיפוי קבצים המבוצע על ידי אחד התהליכים אינם משפיעים על השני.

תהליך הילד הוא כפילות מדויקת של תהליך ההורה למעט הנקודות הבאות:

- לילד יש מזהה תהליך ייחודי משלו (id), וה PID אינו תואם לאף id של קבוצת תהליכים קיימת.
- מזהה התהליך של ההורה של הילד זהה לזהות התהליך של ההורה.
- הילד לא יורש את מנעולי הזיכרון (memory lock) של ההורה.
- ניצולי משאבי התהליך (Process resource utilizations) ומוני זמן המעבד (CPU time counters) מאותחלים לאפס אצל הילד.
- קבוצת ה Signals של הילד ריקה.
- הילד לא יורש התאמות סמפורות (semaphore adjustments) מההורה שלו.

ערך החזרה: בהצלחה, ה- PID של תהליך הילד מוחזר אצל ההורה, ו- 0 מוחזר אצל הילד. בכישלון, -1 מוחזר אצל ההורה, לא נוצר תהליך של ילד, ו- errno מוגדר לציין את השגיאה.

הערה: לפעמים לא נצטרך שהבן יקבל עותק של כל תכולת האב, במקרה כזה נוכל להשתמש בCOW.

שאלה: האם fork משכפל רק את threadn שקרא לו או את כל threads בprocess?

תשובה: תלוי במערכת הפעלה – UNIX ימים רבים תומכים בשני סוגי fork, בLinux רק threadn שקרא לפונקציה משוכפל ולעיתים זה גורם לבעיות.

### Wait – קריאת מערכת שממתינה לשינוי בתהליך בן.

קריאות מערכת הללו משמשות להמתין לשינויים במצב אצל ילד ולקבלת מידע על הילד שמצבו השתנה.

שינוי במצב נחשב להיות: הילד הופסק (terminated) הילד נעצר על ידי איתות (stop signal) או שהילד התחדש באות (cont signal). במקרה של ילד שהופסק, ביצוע wait מאפשר למערכת לשחרר את המשאבים הקשורים לילד; אם לא מתבצע wait, הילד המופסק נשאר במצב "זומבי".

**Exec** – משפחת הפונקציות exec () מחליפה את תמונת התהליך (process image) הנוכחית בתמונת תהליך חדשה.

```
int execl(const char *path, char *const argv[]);
```

**path:** should point to the path of the file being executed.

**argv[]:** is a null terminated array of character pointers.

### (COW) Copy-on-write – היא אסטרטגיית אופטימיזציה, אשר נעזרים בה בתכנות.

Copy-on-write נובעת מההבנה שמספר משימות שונות יכולות להשתמש בעותקים זהים של מידע, כלומר אין צורך לבצע עותק של המידע לכל תהליך.

במקום זה, ניתן להצביע על אותו המשאב מכל התהליכים הדורשים עותק זה.

כאשר יש מספר תהליכים המשתמשים באותו המשאב מקבלים ניצול טוב יותר של משאבים. כאשר עותק מקומי

עובר שינוי, פרדיגמת COW לא מתחייבת כי המשאב המשותף לא השתנה בינתיים על ידי תוכנית אחרת. לכן שיטה זו נוחה מאוד אם רק העדכון האחרון חשוב.

דוגמה ל COW במערכת Linux: כאשר קריאת המערכת fork מופעלת, עותק של כל הדפים התואמים לתהליך

האב נוצרים ונטענים לאזור זיכרון נפרד על ידי מערכת ההפעלה לטובת התהליך הבן. פעולה זו אינה נדרשת

במקרים מסוימים. למשל, במקרה שנבצע דרך התהליך הבן את קריאת המערכת execv או בזמן יציאה לאחר זמן קצר מרגע ה fork.

כאשר תהליך הבן צריך להריץ פקודה לטובת תהליך האב, אין צורך להעתיק את דפי תהליך האב, זאת מפני ש-

execv מחליף את מרחב הכתובות של התהליך.

במקרים כאלו, טכניקת COW באה לידי ביטוי. בעזרת טכניקה זו, כאשר מתרחש ה-fork, דפי תהליך האב לא

מועתיקים לתהליך הבן. במקום זאת, הדפים משותפים בין תהליך האב ותהליך הבן. בכל פעם שתהליך (אב או בן)

משנה דף, נוצר עותק נפרד של הדף הספציפי אשר שונה לתהליך הרלוונטי. תהליך זה ישתמש בדף החדש במקום

הדף המשותף בכל הפעולות העתידיות. התהליך האחר (זה אשר לא שינה את הדף) ממשיך להשתמש בעותק המקורי של הדף (אשר לא משותף יותר). זוהי בדיוק טכניקת COW מפני שהדף מועתק בזמן שחלק מהתהליכים כותבים אליו.

**תהליך יתום (orphan process)** - תהליך יתום הוא תהליך מחשב שתהליך האב שלו הסתיים או הופסק (terminated), אם כי הוא ממשיך לפעול בעצמו. תהליכים שכאלו עוברים תהליך reparenting כאשר היתומים "מאומצים" על ידי תהליך המערכת (PID1).

**Errno** - משתנה מערכת, אשר נקבע על ידי קריאות מערכת במקרה של שגיאה. בדרך כלל מציין מה השתבש, הקיום או השגיאה מסומנים על ידי ערך ההחזרה של הפונקציה, בדרך כלל 1-1. errno הוא thread מקומי thread-safe, כלומר הגדרתו בthread אחד אינה משפיעה על ערכו בשום thread אחר. חשוב להקפיד על קוד הגנתי ולהשתמש ב-errno לעתים קרובות (לבדוק שהערך המוחזר מהפונקציה הוא לא errno ורק אז להמשיך).

**File descriptors** - במערכות הפעלה של מחשבי יוניקס, מתאר קבצים (FD) הוא identifier ייחודי עבור קובץ או משאב קלט / פלט אחר, כגון socket או pipe. מתאר קבצים הוא מספר שלם המציין את אינדקס הערך בטבלת מתאר הקבצים. FD בדרך כלל יש ערכים שלמים שאינם שליליים, כאשר הערכים השליליים שמורים לציון "אין ערך" או תנאי שגיאה. FD הם חלק מממשק ה-API של POSIX. טבלת מתאר קבצים מוחזקת על ידי כל תהליך ומכילה פרטים על כל הקבצים הפתוחים. כל תהליך של יוניקס (למעט אולי daemons) צריך לכלול שלושה מתארים של קבצי POSIX סטנדרטיים, המתאימים לשלושת הזרמים הסטנדרטיים: ערך 0 עבור stdin, ערך 1 עבור stdout, ערך 2 עבור stderr.

## – Threads on POSIX: pthread

API של POSIX, בקימפול מוסיפים pthread-lp (בספריה הזו כל הפונקציות של הטרדים ממומשות).

```
int pthread_create (pthread_t* thread, pthread_attr_t* attr, void* (*start_func)(void*), void* arg)
```

יוצרת תהליכון חדש המתבצע במקביל לתהליכון שקרא לפונקציה.

בהצלחה, identifier של התהליכון החדש שנוצר נשמר במיקום שהארגומנט של התהליכון מצביע עליו, ומחזיר 0. בשגיאה, קוד שגיאה שונה מאפס מוחזר.

הארגומנט attr מאפשר להחיל תכונות על התהליכון החדש (למשל מנותק-detached, מדיניות תזמון-scheduling policy). יכול להיות NULL (ברירת מחדל).

Start\_func הוא מצביע על הפונקציה שהתהליכון יתחיל לבצע. הפונקציה מקבלת ארגומנט בודד מסוג void\* ומחזירה void\*.

arg הוא הפרמטר שהstart\_func מקבל.

```
pthread_t pthread_self ()
```

הפונקציה מחזירה את identifier של התהליכון.

```
int pthread_join (pthread_t th, void** thread_return)
```

משהה את ביצוע התהליכון הקורא עד לסיום התהליכון הנקרא.

בהצלחה, ערך ההחזרה של th נשמר במיקום שמצביע על ידי thread\_return, ו-0 מוחזר. בשגיאה, קוד שגיאה שונה מ-0 מוחזר.

לכל היותר תהליכון אחד יכול לחכות לסיומו של תהליכון נתון. קריאה ל-pthread\_join מתהליכון th שבו תהליכון אחר כבר ממתין לסיום מחזירה שגיאה.

th הוא המזהה של התהליכון שצריך לחכות לו.

thread\_return הוא מצביע לערך המוחזר של התהליכון th (יכול להיות NULL).

**void pthread\_exit (void\* ret\_val)**

מסיים (terminate) את ביצוע תהליכון הקריאה. לא מסיים את כל התהליך אם קוראים לו מהפונקציה הראשית.

אם ret\_val שונה מnull, ret\_val נשמר, וערכו ניתן לתהליכון שביצע join לתהליכון הזה. כלומר, זה יכתב לפרמטר thread\_return בקריאה pthread\_join.

**תקשורת בין תהליכים (inter-process communication או IPC) - אוסף של שיטות ומנגנונים להעברת נתונים בין תהליכונים בתוך אותו תהליך או בין תהליכים שונים.**

התקשורת יכולה להתבצע בין תהליכים שרצים על גבי אותו המחשב, או על גבי מחשבים שונים המחוברים ברשת. ניתן לחלק את השיטות לתקשורת בין תהליכים לקטגוריות הבאות:

- העברת מסרים

- סנכרון

- זיכרון משותף

- הפעלת פרוצדורות מרחוק

השיטה הנבחרת לתקשורת בין תהליכים תלויה ברוחב הפס, בזמן ההשהיה בתקשורת בין התהליכונים, ובסוג הנתונים המועברים.

הסיבות ליצירת תשתית המאפשרת שיתוף פעולה בין תהליכים: שיתוף מידע, זירוז תהליכים, מודולריות, נוחות, הפרדת הרשאות.

לכל process יש בגרעין PCB (process control block) שבו כל האינפורמציה על הprocess, ולכל thread ב process יש בגרעין TCB (thread control block).

נניח ש process עושה עבירה, ניגש לזיכרון שלא הוקצה או מחלק ב0, במקרה כזה על ה CPU נוצר exception condition והוא מתנהג כמו interrupt הוא היה צפוי ויש ל handler.

לדוגמא - אם זה חלוקה ב0 יש handler שנקרא floating point exception handler והוא יטפל בזה.

ה CPU לא יכול להמשיך ולכן הוא צריך להוריד את הprocess מה CPU ולהגיד למי שכתב את הקוד שיש באג.

לכן kernel אומר ל process שיש בעיה והוא מוריד אותו. הוא עושה זאת על ידי שליחת signal ל process.

הוא יגרום לכך שבprocess תהיה התנהגות דיפולטיבית כמו לרשום core dump (סטטוס מתי זה קרה, למה זה קרה. זה יעזור אחר כך למתכנת לפתוח את הקובץ ולנתח את הבעיה) ואחר כך יוריד את הprocess.

**איתות (Signal) -** איתותים הם צורה מוגבלת של תקשורת בין תהליכים (IPC), המשמשת בדרך כלל במערכות הפעלה, Unix דמוי יוניקס ומערכות הפעלה אחרות התואמות POSIX.

אות היא התראה אסינכרונית שנשלחת לתהליך או לתהליכון ספציפי באותו תהליך כדי להודיע לו על אירוע.

כאשר נשלח אות, מערכת ההפעלה קוטעת את זרימת הביצוע הרגילה של תהליך היעד כדי להעביר את האות.

ניתן להפסיק את הביצוע במהלך כל פקודה שאינה אטומית.

אם התהליך רשם בעבר מטפל באותות (signal handler), השגרה הזו מבוצעת (מה שהוגדר עבור הsignal

בתהליך). אחרת, מטפל האותות המוגדר כברירת מחדל מבוצע.

Embedded programs עשויות למצוא את האותות שימושיים לתקשורת בין-תהליכים, מכיוון שאותות בולטים ביעילותם האלגוריתמית.

איתותים דומים להפרעות, ההבדל הוא שהפרעות מתווכות על ידי המעבד ומטופלות על ידי הליבה ואילו האותות מתווכים על ידי הליבה (אולי באמצעות קריאות מערכת) ומטופלות על ידי תהליכים.

הליבה עשויה להעביר הפרעה כאות לתהליך שגרם לה (דוגמאות אופייניות הן SIGSEGV, SIGBUS, SIGILL ו-SIGFPE).

קיימים אותו סינכרוניים (חלוקה ב0) ואותות א-סינכרוניים (לפי שעון).

דוגמאות לsignals:

- SIGSEGV – הפרת SEGmentation.

- SIGFPE - Floating point error, למשל חלוקה ב-0.

- SIGILL - פעולה לא חוקית.

- SIGINT - Interrupt, למשל על ידי משתמש באמצעות הקשה על C + ctrl.

כברירת מחדל גורם לסיום התהליך.



- SIGABRT – סיום (termination) לא תקין, למשל על ידי לחיצה על Ctrl + Q.
  - SIGTSTP – השעיית (Suspension) תהליך, למשל על ידי לחיצה על Ctrl + Z.
  - SIGCONT – גורם לתהליך מושעה לחידוש הביצוע.
- הערה: שלושת הsignals הראשונים הם סינכרוניים מכיוון שהם יכולים להגיע רק כתגובה לפעולה שבוצעה.

Signal מעובד לאחר שתהליך חוזר Interrupted או ממל system call ולפני שהאות מגיע הוא עובר בקוד של המשתמש.

מכאן יש 3 אפשרויות:

- התנהגות דיפולטיבית - אין התייחסות בקוד ולכן הsignal ממשיך בהתנהגות ברירת המחדל שלו (לדוגמה: exit, core, stop, ignore, continue).
- התעלמות - Ignore (SIG\_IGN), התהליך לא יתבצע בלי שום שרידים.
- טיפול בsignals – handling, פונקציה של המשתמש שמטפלת בsignals מסויימים (לדוגמה לפני ביצוע SIGABRT תהיה הדפסה מסויימת למסך).

### טבלת signals –

לכל תהליך יש טבלת איתותים וכל אות מוצג כערך בטבלה, האם להתעלם ממנו? ובמידה ולא מתעלמים, איזה פעולה לבצע?

### טיפול באותות (Signal handler) –

לתהליך יש מספר אפשרויות, להתעלם מאות ולהשתמש בפעולת ברירת המחדל של האות או שיהיה לו פונקציית טיפול באותות הנקראת כאשר האות המסויים התקבל לתהליך זה. הערה: פונקציית טיפול באותות היא לכל תהליך לכל אות. במקרה כזה אנו אומרים כי התהליך תופס את האות.

### איך מטפלים בsignals?

Signal handler זו פונקציה שכותבים בתוך process. התכנית רצה (normal program flow), מתקבל signal, הcontext switch מעביר את התהליך למרחב הגרעין ומבצע שם את הטיפול באות (לעיתים לגרעין יש handler שגובר על handler שהמשתמש כתב) אם הוחלט שהprocess ממשיך אז הcontext switch מחזיר את התהליך למרחב המשתמש, הוא מבצע את handler שכתב עבור האות, וחוזרים למרחב הגרעין בעזרת sigreturn() הוא מאחסן את כל הקונטקסט של התהליך ומחזיר אותו למרחב המשתמש להמשיך ריצת הקוד. סך הכל מתבצע context switch ארבע פעמים. Signal מיוחדים: SIGKILL & SIGSTOP, שני signals שלא ניתן לתפוס (kill הורג את התהליך וstop עוצר אותו, ניתן להמשיך אותו בעזרת SIGCONT).

הערה: בביצוע fork() כל הסטטוסים של signals מועתקים אך לאחר ביצוע execvp() נעשה reset וכל signals חוזרים להתנהגות הדיפולטיבית שלהם.

מטפלי האותות מבוצעים במצב משתמש (user mode), ולכן הם עשויים להיות מונעים על ידי תהליכון אחר, בדיוק כמו כל תהליכון ברמת המשתמש. מטפל האותות עשוי לבצע קריאת מערכת (system call). עם זאת, קיימת רשימה של פונקציות בטוחות של אות לא סינכרוני (Async-signal-safe functions) כלומר פונקציות שאינן interrupted לאחר קריאה ממטפל האותות. תהליך יכול להגדיר כי לאחר קבלת אות ספציפי, במקום לבצע את פעולת ברירת המחדל, מטפל האותות שלו ייקרא. זה נעשה באמצעות קריאות המערכת sigaction() & signal(). מומלץ להיות עקביים: השתמש תמיד באותה פונקציה מתוך השניים. Signal() פשוטה יותר ובאופן היסטורי, נעשה בה שימוש נרחב יותר, עם זאת, sigaction() גמישה ויציבה יותר.

```
sighandler_t signal (int signum, sighandler_t handler);
```

מתקין מטפל אותות חדש עבור האות עם מספר signum.

מטפל האותות מוגדר לhandler שעשוי להיות: פונקציה שצוינה על ידי המשתמש, SIG\_IGN (התעלם מהאות) או SIG\_DFL (השתמש בפעולות ברירת המחדל של האות).

Signal() הוא one-shot, צריך לקרוא לו שוב לאחר כל אות שנתפס.

```
int sigaction (int signum, const struct sigaction *act, struct sigaction *oldact);
```

Signum – מספר האות

Act – מצביע למבנה שמכיל מידע רב כולל מצביע לפונקציית מטפל האותות החדשה.

Oldact – אם הוא שונה מnull מטפל האותות הישן יישמר בתוכו

```
int sigprocmask (int how, const sigset_t *set, sigset_t *oldset);
```

משנה את רשימת האותות החסומים (blocks signal) כעת.

SIG\_SETMASK – קבוצת האותות החסומים מוגדרת לקבוצת הארגומנטים set.

SIG\_BLOCK – מערך האותות החסומים הוא האיחוד בין set והסט הנוכחי.

SIG\_UNBLOCK – האותות בset מוסרים מהמערכת הנוכחית של האותות החסומים.

זה חוקי לנסות לבטל חסימה של אות שאינו חסום.

```
int sigprocmask (int how, const sigset_t *set, sigset_t *oldset);
```

Oldest – אם הוא שונה מnull, oldset יחזיק את הערך הקודם של signal mask.

sigset\_t – מבנה נתונים בסיסי המאחסן אותות באמצעות מערך ביטים, אחד לכל סוג אות. יש לאתחל ולערוך את המבנה באמצעות פונקציות כגון sigemptyset() & sigfillset().

הערה: signal handler לא "רואה" את כל המשתנים שך process.

שליחת סיגנלים - ניתן לשלוח סיגנל דרך המקלדת, דרך CMD ובאמצעות system call. שליחת אותות דרך המקלדת:

- Ctrl-C שולח את SIGINT (signal-interrupt), כברירת מחדל, הדבר גורם לסיום התהליך.
  - Ctrl-Q שולח את SIGABRT, גורם לסיום לא תקין (abort).
  - Ctrl-Z שולח את SIGTSTP, כברירת מחדל, הדבר גורם לתהליך להשעות הביצוע (suspend).
- שליחת אותות דרך command line:
- raise(sig) - שלח את התהליכון שרץ כעת.
  - <PID> <signal> - kill - שולח את האות שצוין ל PID שצוין (אם לא צוין שום אות, נשלח את ה- TERM).
  - <PNAME> <signal> - killall - יכול לשמש לשליחת אותות מרובים לתהליכים המריצים פקודות ספציפיות, או בבעלות משתמש מוגדר, או בעלי גיל מסוים ועוד.
  - <PID> fg - ממשיכה בביצוע תהליך מושעה על ידי שליחת אות SIGCONT, זה יגרום לתהליך שחודש לפעול בחזית.

הערה: לא כל התהליכים יכולים לשלוח אותות לכל שאר התהליכים. רק הגרעין ומשתמש העל (superuser) יכולים לשלוח אותות לכל התהליכים. תהליכים רגילים יכולים לשלוח אותות רק לתהליכים שבבעלות אותו משתמש.

### קבוצת תהליכים (process-group) –

קבוצת תהליכים היא אוסף של תהליכים הקשורים זה לזה. לכל תהליך יש מזהה (PID) ומזהה קבוצתי (PGID). לכל התהליכים בקבוצת תהליכים מוקצה אותו מזהה קבוצת תהליכים (PGID). ניתן לשלוח את התהליך יחיד או לקבוצה. משמש את המעטפת לשליטה במשימות שונות המבוצעות על ידה. פונקציות שימושיות:

- getpid() - מחזיר את ה- PID של התהליך.
- getpgid() - מחזיר את ה- PGID של התהליך.

- `setpgrp()` - מגדיר את ה-PGID של התהליך הזה להיות שווה ל-PID שלו.
- `setpgrp(int pid1, int pid2)` - מגדיר את PGID של pid1 של התהליך להיות שווה ל-PGID של pid2.

## – Pipe

צינור הוא חיבור בין שני תהליכים, כך שהפלט הסטנדרטי מתהליך אחד הופך לקלט הסטנדרטי של התהליך השני. במערכת ההפעלה UNIX, צינורות שימושיים לתקשורת בין תהליכים קשורים (inter-process communication). צינור הוא תקשורת חד כיוונית בלבד, כלומר נוכל להשתמש בצינור כך שתהליך אחד יכתוב לצינור, והתהליך השני ייקרא מהצינור.

הוא פותח צינור, שהוא אזור של זיכרון ראשי (main memory that) המטופל כ"קובץ וירטואלי". הצינור יכול לשמש את התהליך היוצר (creating process), כמו גם את כל תהליכי הילד שלו, לקריאה וכתובה. תהליך אחד יכול לכתוב ל"קובץ הווירטואלי" הזה ותהליך קשור אחר יכול לקרוא ממנו. אם תהליך מנסה לקרוא לפני שמהו נכתב לצינור, התהליך מושעה (suspended) עד שנכתב משהו. קריאת מערכת pipe מוצאת את שתי המיקומים הזמינים הראשונים בטבלת הקבצים הפתוחה של התהליך ומקצה אותם לקצות הקריאה והכתיבה של הצינור.

```
int pipe (int fds[2]);
```

Parameters:

fd[0] will be the fd(file descriptor) for the read end of pipe.

fd[1] will be the fd for the write end of pipe.

Returns : 0 on Success, -1 on error.

צינורות עובדים בFIFO, מתנהגים כמו תור.

גודל הקריאה והכתיבה לא צריך להיות זהה כאן, אנו יכולים לכתוב 512 בתים בכל פעם, אך לקרוא רק בית אחד בכל פעם דרך הצינור.

כאשר אנו משתמשים בfork() בכל תהליך שהוא, file descriptors נותרים פתוחים על פני תהליך הילד וגם על תהליך האב. אם אנו קוראים לfork() לאחר יצירת צינור, אז ההורה והילד יכולים לתקשר באמצעות הצינור.

**מתזמן (scheduler)** - פעולה של הקצאת משאבים לביצוע משימות. המשאבים עשויים להיות מעבדים, קישורי רשת או כרטיסי הרחבה. המשימות עשויות להיות threads, processes או זרימת נתונים. פעילות התזמון מתבצעת על ידי תהליך הנקרא מתזמן. מתזמנים מתוכננים להשאיר את כל משאבי המחשב עסוקים (כמו באיזון עומסים), לאפשר למשתמשים מרובים לשתף משאבי מערכת ביעילות, ולהשיג איכות שירות היעד. המתזמן הוא מהותי לחישוב עצמו, והוא חלק מהותי ממודל הביצוע של מערכת מחשב, הייעוד של מתזמן הוא לבצע ריבוי משימות במחשב עם יחידת עיבוד מרכזית אחת (CPU). לטיפול בinterrupt יש את העדיפות הגבוהה ביותר (הוא נמצא high-halves). ובbottom-halves המתזמן עובד ומתזמן משימות שלתהליכים ותהליכונים בגרעין ובמרחב המשתמש.

וגי OS:

**Batch** – פעולות שאין להם עדיפות גבוהה עבודות חישוב שלוקחות זמן (backup לדוגמה) ולא דורשת אינטרקציה עם משתמש או עם מערכת I/O. משימות עיקריות: היעילות נמדדת במספר jobs שהושלמו בפרק זמן נתון.

**Interactive computing** - מחשוב אינטראקטיבי מתייחס לתוכנה שמקבלת קלט מהמשתמש תוך כדי הריצה. תוכנה אינטראקטיבית כוללת תוכניות נפוצות, כגון word processors או יישומי גיליונות אלקטרוניים. לשם השוואה, תוכניות שאינן אינטראקטיביות פועלות ללא התערבות המשתמש. מחשוב אינטראקטיבי מתמקד באינטראקציה בזמן אמת ("דיאלוג") בין המחשב למפעיל, והטכנולוגיות המאפשרות להם.

אם התגובה של מערכת המחשב מורכבת מספיק, נאמר שהמערכת מנהלת אינטראקציה חברתית; מערכות מסוימות מנסות להשיג זאת באמצעות יישום ממשקים חברתיים.

משימות עיקריות:

תגובה מהירה יותר כדי לספק את הדרישות של משתמשים שזקוקים לturnaround מהיר בעת ניפוי באגים בתוכנית שלהם ולאפשר לכל משתמש לתקשר ישירות עם מערכות הפעלה.

## – Real-Time

לא מעניין אותם ניצול של ה-CPU אלא רק עמידה ב-deadlines. משימות עיקריות: משומש בסביבות קריטיות בזמן, אמינות היא המפתח וזמן תגובת המערכת חייב לעמוד במועד האחרון (deadline).

### מטרות המתזמן:

בכל המערכות:

- Fairness – לתת לכל process שיתוף הוגן ב-CPU.
- policy enforcement – לראות שהמדיניות המוצהרת מתבצעת.
- Balance – לשמור על כל החלקים שבמערכת עסוקים.

במערכות Batch:

- Throughput (תפוקה) – למקסם את העבודות לפי שעה.
- Turnaround time – למנן את הזמן בין הגשת העבודה לסיום העבודה.
- CPU utilization – לשמור על ה-CPU עסוק כל הזמן.

במערכות Interactive:

- Response time – לענות לבקשות מהר (כדי לענות על הציפיות של המשתמש נעלה את העדיפות של המשימה).

- Proportionality – לענות על הציפיות של המשתמש.

במערכות Real-time:

- Meeting-Deadlines – להימנע מלאבד data, עמידה בזמנים.
- Predictability – תהיה צפוי וידוע כל הזמן.

### גישות המתזמן:

למה שנרצה להחליף ל context אחר?

- מתזמן Non-preemptive – החלפת משימות יכולה להתבצע רק בשירותי מערכת שהוגדרו במפורש, למשל עד שהמשימה לא ירדה מה-CPU כי היא סיימה או שהחליטה בעצמה לעצור או שinterrupt עצר אותה המתזמן לא מחליף משימה.
- מתזמן preemptive – משימה עשויה להיות מתוזמנת לפעולה במועד מאוחר יותר (למשל, היא יכולה להיות מתוזמנת על ידי המתזמן עם הגעת משימה "חשובה יותר").

### אלגוריתמים של מתזמן:

#### **כל הקודם זוכה (First come First served – FCFS):**

- Non-preemptive.
- הוגן (Fair) באופן זמן ההמתנה.
- טוב עבור מערכות Batch (convoy effect – אפקט שיירה).
- לא יעיל עבור מערכות Interactive (I/O).

#### **עבודה קצרה קודם (Shortest Job First – SJF):**

- Non-preemptive.
- ממנן את זמן הturnaround.
- משך העבודה צריך להיות יודע מראש.
- כל המשרות צריכות להיות זמינות בהתחלה, כדי שיהיה יעיל.
- חסרון: יכול לגרום להרעבה (starvation) כיוון שמשימה ארוכה לא תקבל CPU לעולם.

#### **הזמן הנותר הקצר ביותר ראשון (Shortest Remaining Time First – SRTF):**

- 2 אפשרויות: preemptive & non-preemptive.
- ממנן את זמן הturnaround וגם את waiting time.
- טוב עבור Interactive jobs (I/O).
- צריך לדעת את זמן העבודה שנותר.
- חסרון: יכול לגרום להרעבה (starvation) כיוון שמשימה ארוכה לא תקבל CPU לעולם.

#### **יחס התגובה הגבוה ביותר הבא (Highest Response Ratio Next – HRRN):**

- Non-preemptive.
- מנסה למנוע את החיסרון של ה-job הקצר תחילה על ידי התחשבות בזמן המתנה של ה-job
- $$Priority = \frac{waiting\ time + estimated\ run\ time}{estimated\ run\ time} = 1 + \frac{waiting\ time}{estimated\ run\ time}$$

#### **Round Robin**

- Preemptive
- הוגן באופן של חלוקת משאבים בין עבודות.
- פרוסות זמן רב לעומת פרוסות זמן קצר (קוונטי).
- לכל העבודות יש את אותה עדיפות.
- הערה: אם ניתן קוונטים גדולים מדי לכל עבודה האלגוריתם יראה כמו FCFS, לעומת זאת אם ניתן קוונטים קטנים מדי נבזז המון זמן על context switch.

#### מתזמן לפי עדיפות (Priority scheduling):

- Preemptive or Non-Preemptive
- עדיפות קבועה.
- עדיפות לפי סוג (I/O אל מול CPU).
- עדיפות דינמית (1/f, all up, running down).

#### תזמון מתקדם (Advanced scheduling):

- תור עדיפות (קבוע)
- תור עדיפות (קוונטי כפול)
- תזמון מובטח (1/#process)
- תזמון lottery.
- הוגנות בעלים (Owner Fairness).

#### תזמון מובטח (Guaranteed scheduling):

- ה CPU עובד ומחשב כל הזמן, לכל תהליך לפי היחס בין זמן המעבד המוקצה וכמות זמן המעבד שהתהליך זכאי לו.
- בוחר את התהליך עם היחס הנמוך ביותר.
- הוגן – מבטיח 1/n מזמן המעבד לכל משימה (עבור n משימות).

#### תזמון עם מספר תורים (Multi-Level Queue Scheduling):

- מחלק את את ready queue למספר תורים.
- כל תור מפעיל מנגנון תזמון ואלגוריתם משלו.
- תהליך מקבוצת עדיפות נמוכה יותר עשוי לפעול רק אם אין תהליך בעדיפות גבוהה יותר.
- מונע הרעבה לפי גיל: תהליך "ישן", שהמתין זמן רב, מקודם לתור בעל עדיפות גבוהה יותר.
- תהליך שנהנה מזמן המעבד מועבר לתור בעל עדיפות נמוכה יותר.
- תורים ברמה נמוכה הם בעדיפות נמוכה יותר, אך יש להם קוונטים ארוכים יותר.
- המדיניות המדויקת של קידום, הורדה והמדיניות של התור עשויה להשתנות.

קריטריונים למדידה (לפי הקריטריון נחליט באיזה אלגוריתם של המתזמן להשתמש):

1. Throughput – כמות התהליכים שהסתיימו בכל יחידת זמן (time unit).
2. Efficiency: CPU utilization – אחוז הזמן שה CPU עסוק (computation & communication).
3. Turnaround time – זמן ממוצע משליחת העבודה ועד להשלמתה.
4. Waiting time – סכום כל האינטרוולים שבהם התהליך היה ב ready-queue.
5. Response time – זמן התגובה, הזמן מאז ששלחו את העבודה ועד לזמן שהעבודה מקבלת CPU בפעם הראשונה.
6. Fairness – תהליכים דומים צריכים לקבל שירות דומה.

### תרגילים טובים במצגת T4\_ThreadsSched 8

#### Real-Time OS –

לא מעניין אותם ניצול של ה CPU אלא רק עמידה ב deadlines. משימות עיקריות: משומש בסביבות קריטיות בזמן, אמינות היא המפתח וזמן תגובת המערכת חייב לעמוד במועד האחרון (deadline).

#### סוגי TR-OS לפי ההשלכות של אי עמידה בזמנים:

- Safety-Critical System - מערכת קריטית לבטיחות היא סוג של RT-OS עם השלכות קטסטרופליות.
- Hard RT System - מערכת RT קשה מבטיחה עמידה במועדים עבור כל משימות ה- RT. אין ערך של תוצאה לאחר המועד האחרון.
- Soft RT System - מערכת RT רכה מספקת עדיפות למשימות RT על פני משימות שאינן RT. עומד ברוב deadlines. יש עדיין ערך של תוצאה גם לאחר ה deadline.

הערה: Embedded System != RT-OS

## תכונות וטיפול בזכרון:

פחות פיצ'רים מאשר במערכת ההפעלה של שולחן העבודה / השרת:

חלק ממערכות ה-RT הן למטרות חד פעמיות, כלומר מיתוג או ניתוב מנות (packets), מכון טיל, ממשק GPS וחישוב נתיב קצר ביותר. אין ממשק משתמש ויכולות חומרה מוגבלות. כתובת זיכרון:

טיפול אמיתי בעבודה עם כתובות פיזיות (Real Addressing working with physical addresses) - נדיר מאוד בימינו.

כתובת רילוקיישן - הוספת ערך relocation register לתרגום. יישום VM מלא.

## דרישות מימוש:

- גרעין Preemptive.
- Priority-based preemptive scheduler.
- Low latency – הגעה מהירה ל-CPU בלי להיתקע איפשהו.
- Minimized jitter (maximized predictability) – צפי לזמן שלוקחת משימה (לדוגמה בין 9-11 שניות).
- Event Latency <- הזמן שעובר מהרגע שעבודה מגיעה ועד לביצוע הפעולה (הזמן שעובר מהרגע שבלם הרכב מופעל ועד שהמכונית נעצרת לחלוטין).
- Interrupt Latency <- זמן ממועד ההפרעה (interrupt) ועד תחילת פונקציית ה-ISR (interrupt handler).
- Dispatch Latency of Scheduler <- הזמן הדרוש למתזמן לעצור משימה אחת ולהתחיל משימה אחרת.

הערה: RT אינו הוגן (אין התחשבות ב-Fairness)!

המטרה העיקרית של מתזמן RT היא לעמוד במועדים לכל משימות ה-RT המתוזמנות.

ניצול מקסימלי של המעבד (maximum CPU-utilization), התפוקה הטובה ביותר (best throughput), תפנית ממוצעת מינימלית (minimum average turnaround), זמני תגובה והמתנה (response and waiting times) כולם לא רלוונטיים עבור RT.

אם יש לך deadline להקצאת OS היום, אין זמן ואין עניין להוגנות. אתה מבצע את המשימה הדחופה ביותר בכדי לעמוד בdeadline. הוגנות אף פעם לא עוזרת לך לעמוד בdeadline.

## מדיניות תזמון RT:

איזו אלגוריתם תזמון מבטיח עמידה בכל מועדי הזמן עבור מערך משימות נתון? כיצד להוכיח שאלגוריתם תזמון נתון יעבוד בעומס עבודה מסוים? מה הפירוש של המונח "אלגוריתם אופטימלי" לתזמון RT?

## תזמון עבודה – הגדרות:

arrival time – a, הזמן שבו משימה מוכנה לביצוע.

absolute deadline – d, המועד האחרון לסיום המשימה.

s/f – הזמן שבו המשימה התחילה והסתיימה.

worst case execution time – C, זמן חישוב או זמן ביצוע הגרוע ביותר, משך הזמן הדרוש למעבד להשלמת העבודה ללא הפרעות.

response time – R, משך הזמן מאז ההגעה ועד לסיום העבודה: (f-a).

relative deadline – D, משך הזמן מאז ההגעה ועד המועד המוחלט: (d-a).

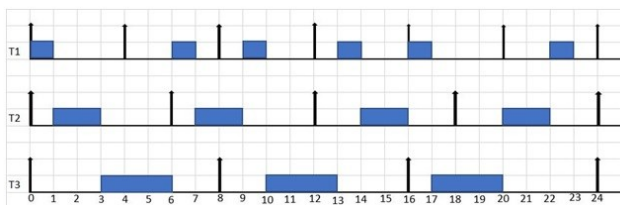
מתי נפספס את deadline? אם  $R > D$  או  $f > d$ .

ניצול כולל של כל משימות ה-RT (Total Utilization of all RT-Tasks)  $U = \sum (\frac{C_i}{P_i})$  כאשר P היא תקופת

ההפעלה ובדרך כלל שווה למועד deadline D.

אם  $U > \# \text{ of CPU}$  המתזמן אינו יכול לעמוד בכל הdeadlines - העומס גבוה מדי.

משימת RT מוגדרת כך:  $Task(C, D, P)$ .



### אלגוריתם תזמון - Earliest Deadline First (EDF)

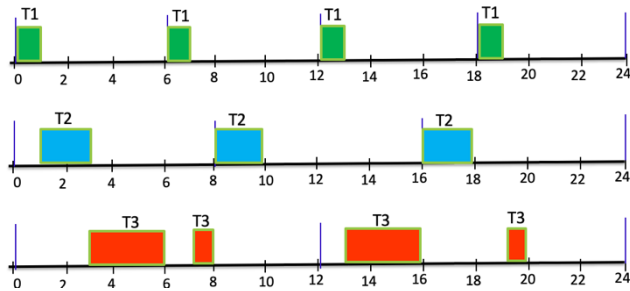
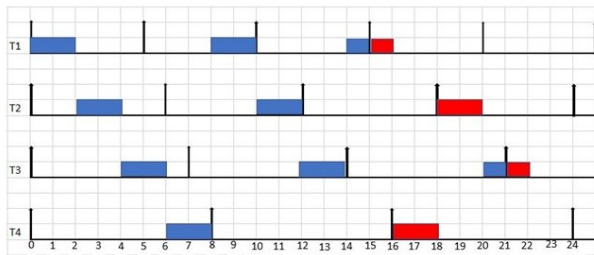
T1 (1,4,4), T2 (2,6,6), T3 (3,8,8)

$$U = \frac{1}{4} + \frac{2}{6} + \frac{3}{8} = 0.958 \rightarrow \text{feasible}$$

Transient Overload and Domino Effect

T1 (2,5,5), T2 (2,6,6), T3 (2,7,7), T4 (2,8,8)

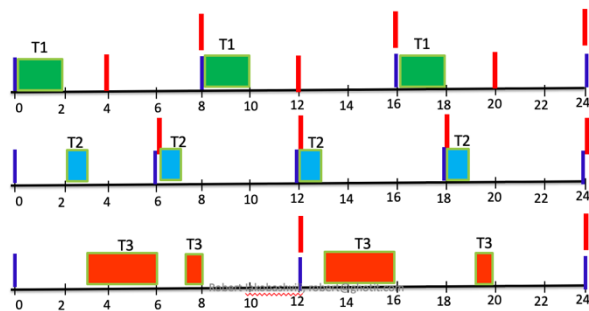
$$U = \frac{2}{5} + \frac{2}{6} + \frac{2}{7} + \frac{2}{8} = 1.269 \rightarrow \text{not feasible}$$



### אלגוריתם תזמון - קצב מונוטוני (RM)

למשימה עם תקופה קטנה יותר יש עדיפות גבוהה יותר.

T1 = (1,6,6), T2 = (2,8,8), T3(4,12,12)



### אלגוריתם תזמון - Deadline-Monotonic (DM)

למשימה עם תאריך יעד יחסי קטן יותר יש עדיפות גבוהה יותר.

T1 = (2,8,4), T2 = (1,6,6), T3(4,12,12)

### Linux & Real-Time

לינוקס הוא לא מערכת RT בגלל שהאינטראקציה עם החומרה נעשית על ידי kernel, והאפליקציות רצות באזור המשתמש, הגרעין לא יכול להעביר משימת kernel לטובת משימת user וגם ה latency הם גבוהים. האם ניתן לשנות ולהפוך את לינוקס ל RT?

| RM   | EDF   |
|--|---|
| Low overhead of scheduling: $O(1)$ with priority sorting in advance                                  | High overhead of scheduling: $O(\log n)$ with AVL tree  |
| For static-priority  | For dynamic priority. Optimal   |
| The exact schedulability test is complex, but boundary test is simple                                | Schedulability test is easy ( $D == T$ )  |
| Least upper bound of $U$ : 0.693   | Least upper bound of $U$ : 1.0 ( $D == T$ )   |
| In general, requires more preemption.  | In general, requires less preemption.   |
| Practice: easy to implement.   | Practice: Complex to implement due to dynamic priorities, but there are known industry designs (Linux).                   |
| Rather stable. Even if some lower priority tasks fail to meet the deadlines, others still can do it. | Not stable. If a task fails to meet its deadline, the system may fail due to domino effect. Admission control is desired. |

באמצע שנות ה-90 הגיע מפתח והציע עמוד שהיה מחוץ ללינוקס.  
מה הוא עשה? הוא הוריד את הסטטוס של משימות הkernel, ז"א הkernel יכול להיות preempted, ובנוסף הוא הוריד משמעותית את latency ועוד.

### Linux Scheduling

- SCHED\_OTHER / SCHED\_NORMAL - מדיניות רגילה של חלוקת זמן RR.
- SCHED\_BATCH - RR כאשר ההנחה היא שהמשימות אינן אינטראקטיביות ומחויבות למעבד עם פרוסת ברירת מחדל של 1.5 שניות. מדיניות ידידותית למטמון (cache).
- SCHED\_IDLE - RR עם פרוסה גבוהה יותר הניתנת למשימות בעדיפות נמוכה.
- SCHED\_FIFO - FIFO ללא חיתוך זמן.
- SCHED\_RR - פרוסת זמן RR עם preemption.
- SCHED\_DEADLINE - מדיניות חדשה שמועברת על ידי מתזמן ה-EDF תזמון שרת רוחב פס קבוע (CBS) שעוזר ליציבות ה-EDF ומניעת אפקט דומינו.

**CPU bound** - תהליך שיש לו צורך רב בcomputing.  
**I/O bound** - תהליך עם צורך נמוך בCPU, באופן כללי לתהליכים אלה יש אינטראקציה של קלט / פלט.

### Shared Object (Critical Sections)

איך נדאג שרק תהליכון אחד יכנס לקטע קוד קריטי ואף תהליכון אחר לא יצליח להיכנס עד שהתהליכון שבפנים יסיים?

1. פעולה אטומית – לא מתאים עבור המקרה הכללי.
2. Spinning waiting - כמו לולאת while שתחסום את התהליכון עד שהתהליכון השני ישחרר אותו. חסרון: לוקח המון CPU יתרון: שהפעולה שמתבצעת ב CS ודטרמיננטית ואז לא שווה להכניס את התהליך לשינה ולהעיר אז אם הקטע קוד קצר וידוע מראש אז הוא עדיף.
3. Sleeping – להרדים את התהליכון ולהעיר אותו כשהוא יוכל להיכנס ל CS. חסרון: יקח זמן מהרגע שנשלח פקודה להעיר את התהליכון ועד שהוא יכנס לקטע הקוד. יתרון: צורך מעט מאוד CPU.

### תנאים עבור פתרון טוב:

- Mutual Exclusion - אין שני תהליכים בקטע הקריטי (CS) בו זמנית.
- Deadlock Freedom - אם תהליכון אחד או יותר מנסים להיכנס ל CS, בסופו של דבר אחד ייכנס אליו.
- Starvation Freedom - תהליך שמנסה להיכנס ל- CS יצליח בסופו של דבר להיכנס אליו.
- Logic Solution - הפתרון לא יכול להיות תלוי במהירות המערכת או hardware.

### פתרון 0 לפתרון CS – ביטל כל interrupt:

חסרונות: מסוכן מאוד ולא מתאים לסביבה של multiprocessing.

### פתרון 1 חילופים קפדניים (strict alternation):

```
While(TRUE) {  
    while (turn != process); /* Busy waiting */  
    enter_Critical_Section();  
    turn = ~(process);  
    exit_Critical_Section ();  
}
```

int turn = 0;  
/\*process 0 & process 1

חסרונות: תהליכים חייבים להיכנס ל CS לסירוגין – מפר את תנאי "Deadlock Freedom".  
המתנה עמוסה (busy waiting) - בזבז משאבי ה CPU.



## פתרון 2 (Peterson's algorithm) sleep & wakeup

הרעיון – כל עוד אני מעביר את התור לprocess השני והוא מעוניין (interested), אני מחכה.

`N=2;`

```
int interested[N]; /* all initially 0 (FALSE) */

interested[0]=interested[1]=FALSE;

int turn; /* Should be named NOT MY TURN */

void enter_region(int process){ /* who is entering 0 or 1 ? */
    int other = 1-process; /* opposite of process */
    interested[process] = TRUE; /* signal that you're interested */
    turn = process; /* set flag - NOT my turn */
    while (turn == process &&
           interested[other] == TRUE); /* null statement */
}

void leave_region(int process){ /* who is leaving 0 or 1 ? */
    interested[process] = FALSE; /* departure from critical region */
}
```

Process 1

```
int interested[2];
interested[0]=interested[1]=FALSE
int turn;

void enter_region(int process){
    int other = 1;
    interested[0] = TRUE;
    turn = 0;
    while (turn == 0 &&
           interested[1] == TRUE);
}

void leave_region(int process){
    interested[0] = FALSE;
}
```

Process 2

```
int interested[2];
interested[0]=interested[1]=FALSE
int turn;

void enter_region(int process){
    int other = 0;
    interested[1] = TRUE;
    turn = 1;
    while (turn == 1 &&
           interested[0] == TRUE);
}

void leave_region(int process){
    interested[1] = FALSE;
}
```

שאלה 1: מה יקרה אם נחליף את הסדר של השורות שמסומנות בכתום?  
סימולציה:

P1 does turn:=1;  
P0 does turn:=0;  
P0 does interested[0]:=true;  
P0 does while(turn == 0 && interested [1]);  
P0 enters the CS.  
P1 does interested [1]:=true;  
P1 does while(turn == 1 && interested [0]);  
P1 enters the CS.

תשובה: נקבל Mutual exclusion violation – שני התהליכים יהיו במקביל בתוך CS.

שאלה 2: מה יקרה במידה ונשנה את התנאי בלולאה להיות - `while (turn != process && interested[other] == TRUE)` –  
תשובה: משמעות השינוי היא שנסתכל על משתנה turn לא בתור ויתור, אלא בתור השתלטות.  
גם כאן יתקבל Mutual exclusion violation כיוון שיכול להיווצר מצב שבו שני תהליכים יכנסו במקביל ל-CS.  
טענה: אלגוריתם Peterson לא יעבוד במקרה של CPU 2 סוגי סנכרון בין CPU:

- Cache coherent system – סנכרון מהיר של cache ונעשה באופן אוטומטי, כלומר שני CPU רואים בקוד את אותו הדבר והturn שלהם שניהם זהה – סנכרון כזה הוא מאוד יקר.

- Non-cache coherent system – במידה ושינינו turn של אחד מה CPU ייקח מעט זמן עד שזה יתעדכן אצל ה CPU השני, ולכן לא ניתן להסתמך על העובדה ששניהם רואים בדיוק אותו דבר בכל רגע נתון. סנכרון שכזה יותר נפוץ במחשבים, במערכת עם סנכרון כזה אלגוריתם Peterson לא יעבוד כיוון שהנתונים לא יתעדכנו אצל שני התהליכים ולכן נצטרך ליצור סנכרון בין שני processes לאחר כל שורת קוד. \ מה הפתרון? נעטוף את CS במנעול – נלמד בהמשך.

### – Test-and-Set Lock Spinning Algorithms

הרעיון: יישום חלקים בחומרה שמסוגלים לבצע memory barrier (חסימת זיכרון), סנכרון בין ה chach ו lock על משתנים. למעשה הפעולה מתבצעת בצורה אטומית ולכן כל CPU יים מתעדכנים בשינויים שנעשו בפעולה האטומית דוגמת הרצה: תהליך 1 נכנס ללולאה ומריץ את Test-and-Set(0), הפונקציה מחזירה לו 0 ולכן הוא נכנס CS ובמקביל ה value של שאר התהליכים הוא 1 ולכן CS נעול מבחינתם והם לא יכולים להיכנס (גם הערך של value נעול כיוון שהוא יחזור להיות 0 רק אחרי שהוא ייצא מה CS). הערה: לאלגוריתם אין הבטחה ל Starvation Freedom.

Assembly

```

mutex:=0

spin_lock :

    TSL     REG, mutex      # prev value is stored in REG and mutex:=1
    CMP     REG, 0
    JNE     waiting        # jump to continue waiting
    RET     # locked successfully; enter the CS

waiting:
    # waiting for unlocking

    CALL    thread_yield   # yield the CPU; run on the next scheduling
    JMP     spin_lock      # return to spin_lock on the next scheduling

spin_unlock :
    MOV     mutex, 0
    RET
        
```

Each thread, does:

1. **spin\_lock();**
2. Critical Section
3. **spin\_unlock();**

### Pseudo-Code

Init the value:=0

Each thread, does:

1. **Wait-for** test-and-set(value) = 0
2. The Critical Section
3. value:=0

**Test-and-set(value)**

*do atomically after full RW memory barrier*

prev:=value

value:=1

return prev # return and synchronize all CPUs

### :Sleeping synchronization algorithm

Multiprocessing זקוק לכלים לניהול משאבים משותפים. כל הפתרונות שראינו עד כה, היו בזבזנים (spinning algorithm) ויקרים. פתרון: תמיכה מה hardware – פקודות אטומיות, זול מהיר ופשוט.

Binary Semaphores: מודל זה בנוי מ 2 פעולות אטומיות – Up(1) & Down(0). פעולות אטומיות מתבצעות עם full write/read, memory barrier לפני ואחרי הפעולה.

```

init(S) {
    S=1;
}
down(S) {
    if (S==0) { //or: while (s==0); What's the difference?
        block process (or put him in sleeping mode);
    }
    S=0;
}
up(S) {
    S=1;
    if (there are blocked processes) {
        wake one (or few) up;
    }
}
        
```

1. הדגל שמסמל על מצב החלק הקריטי מאותחל ל-1. כאשר הוא 1 אז החלק הקריטי הפנוי.
2. כניסה לאזור הקריטי מתבצעת ע"י down(s). אם s=1 תשנה אותו ל-0 (מסמן שהאזור הקריטי תפוס) ותיכנס לאזור הקריטי. אם s=0 סימן שהאזור הקריטי תפוס כרגע והתהליך שקרא ל down יישלח ל"תור שינה", שם הוא במצב שינה, במצב זה הוא לא ניתן לתזמון ולא צורך CPU. 3. כאשר התהליך שנמצא באזור הקריטי מסיים, הוא מבצע up(s). בפעולה זו, הדגל חוזר להיות 1 (אזור קריטי פנוי) ואז נעיר לפחות תהליך אחד מ"תור השינה" אל ה-ready queue.

**Counting Semaphores:** דומה ל Binary רק כשכאן או מתחלים את s להיות מספר התהליכים שאנו מאשרים שירוצו במקביל באזור הקריטי.

```
init(S) {
    S=N; // N=number of simultaneously allowed processes in CS
}
down(S) {
    if (S==0) {
        block process;
    }
    S--;
}
up(S) {
    S++;
    if (there are blocked processes) {
        wake one up;
    }
}
```

1. אנחנו מתחלים את הדגל שלנו להיות שווה למספר התהליכים שאנחנו מאשרים שירוצו במקביל בתוך האזור הקריטי.
  2. אותו דבר כמו הקודם, אלא שכאן במקום לשנות את הדגל ל 0 אנחנו מקטינים אותו ב 1.
  3. אותו דבר כמו הקודם, אלא שכאן במקום לשנות את הדגל ל 1 אנחנו מגדילים אותו ב 1.
- הערה: semaphore לא מבטיח Starvation Freedom ולא כל המימושים מונעים busy waiting.

**Negative-Valued Semaphores:** במבט ראשון מאוד דומה ל counting השוני הוא שב Down/Up קודם כל

```
init(S) {
    S=N; // N=number of simultaneously allowed processes in CS
}
down(S) {
    S--;
    if (S<0) {
        block process;
    }
}
up(S) {
    S++;
    if (S<=0){
        wake one up;
    }
}
```

1. אנחנו מתחלים את הדגל שלנו להיות שווה למספר התהליכים שאנחנו מאשרים שירוצו במקביל בתוך האזור הקריטי.
2. בהפעלה של down() אנחנו קודם כל מקטינים ורק לאחר מכן מכניסים את thread ל sleeping queue, היתרון שלו על גבי counting הוא שאנחנו יודעים כמה תהליכים נמצאים במצב שינה (S- תהליכונים).
3. על פי אותה לוגיקה, בביצוע up() קודם נגדיל את S ורק לאחר מכן נעיר תהליך כלשהו.

הערה: מההגיון של negative-value/counting מספר threads שיכולים לעבור דרך הקטע הקריטי מבלי ללכת לישון יהיו  $N + \# up() \text{ command that has been executed}$ .



**דוגמה לשימוש (בינארי):** נניח ויש לנו שני תהליכים על אותה תכנית. בתכנית קיימים שני קטעי קוד כאשר אנחנו רוצים שקטע קוד B ירוץ תמיד רק אחרי שקטע קוד A כבר התבצע. במקרה כזה אנחנו נועלים את קטע B בעזרת down() ומשחררים בעזרת up(). קטע הקוד A יהיה פתוח, אבל בסופו אנחנו נבצע up() על מנת לשנות את הדגל.

**דוגמה לשימוש בעייתי:** נניח ויש לנו קוד ובתוכו יושב איזשהו קטע קריטי. השימוש ב semaphore מאלץ אותנו לשים down() לפני ו up() אחרי. נניח ובמהלך פעילות התכנית קיים איזשהו thread שנכנס לקטע קריטי. מכיוון ש semaphore הוא משותף לכולם, יכול להיות מצב ש thread אחר יבצע up() מתוך פונקציה אחרת ואז יוכל להיכנס לפונקציה שננעלה ע"י ה thread הראשון. מצב כזה יכול לקרות כאשר יש לנו כמה קטעי קוד קריטיים בתכנית. הדרך למנוע זאת היא להשתמש ב mutex (אימפלמנטציה של Binary Semaphores). Mutex משתמש ב lock() וב unlock(), כאשר בתוך lock() יש קריאה ל down() בתוספת מזהה ייחודי של thread שביצע את הנעילה וכך כאשר נקרא ל unlock() הוא יבדוק האם זה אותו thread שנעל ורק אם כן הוא יבצע up(). אפשר לחשוב על mutex כיוש של Binary Semaphores, הוא מכיל את אותם תכונות בתוספת התכונה החשובה Thread ownership.

תכונות של Mutex:

1. Thread ownership – רק מי שנעל יכול לשחרר.
2. מטפל בבעיית Reentry – מאפשר רקורסיה לתוך CS. אם אותו הליכון קורא ברקורסיה לפונקציה הוא לא יקרא שוב ל lock(), אלא יגדיל את counter שסופר כמה פעמים הוא נכנס לפונקציה ברקורסיה ובכל חזרה מהרקורסיה counter יקטן וכשיגיע ל 0 הוא יבצע lock().
3. רק thread שאתחל (בצוע init) את mutex יכול להרוס אותו (ביצוע destroy).
4. תומך בהורשה של עדיפויות (priority) ויכול למנוע החלפות של עדיפויות.

שאלה 1 (counting semaphore): ננסה לייצר counting semaphore באמצעות binary semaphore ( $s_b$ ) ואינטגר  $s_{int}$ , בכל הניסיונות  $s_{int}=2$ .

```
down (Sint) {
    down (Sb) ;
    Sint--;
    if (Sint > 0)
        up (Sb) ;
}
```

```
up (Sint) {
    Sint++;
    up (Sb) ;
}
```

Suppose initially  $N=2 \rightarrow s_{int}=2, s_b=1$

Process 0:

$s_b=0; s_{int}=1; s_b=1$ ; Enter CS

$\rightarrow$  Context switch

Process 1:

Down:  $s_b=0; s_{int}=0; s_b=1$ ; Enter CS; Exit CS

Up:  $s_{int} == 0$  (for doing  $s_{int}++$ , which is NOT atomic).

$\rightarrow$  Context switch

Process 0: Exit CS

Up:  $s_{int} == 0$  (for doing  $s_{int}++$ , which is NOT atomic).  $s_{int} = s_{int} + 1 = 1$ .

(...)

$\rightarrow$  Context switch

Process 1:  $s_{int} = s_{int} + 1 = 1$

BANG:  $s_{int}$  is 1, while it should have been 2  $\rightarrow$  one more process should have been enabled access to the CS.

## ניסיון 1

ניתן לראות בדוגמת ההרצה שקיים מצב שבו שני תהליכים נכנסו במקביל ל-CS. הבעיה כאן היא בפונקציית ה- $down()$ . כאשר שני תהליכים נשלחים ל-CS, התהליך השני יוריד את ה-counter ל-0 וינעל לעצמו את האפשרות לבצע  $up()$ , למרות שהוא כן צריך להחזיק באופציה הזאת. הדרך הנאיבית לטפל בבעיה היא לשנות את התנאי ב- $down$  להיות  $= >$  במקום  $>$ . אבל במקרה כזה שלושה תהליכים עלולים להיכנס ל-CS במקביל.

## ניסיון 2

```
down (Sint) {
    down (Sb) ;
    Sint--;
    if (Sint > 0)
        up (Sb) ;
}
```

```
up (Sint){
    down(Sb);
    Sint++;
    up(Sb);
}
```

Suppose initially  $N=2 \rightarrow s_{int}=2, s_b=1$

Proc 0:

$s_b=0; s_{int}=1; s_b=1$ ; Enter CS

$\rightarrow$  Context switch

Proc 1:

$s_b=0; s_{int}=0; s_b=0$ ; Enter CS

As  $s_b == 0$ , no process can arrive to  $s_{int}++$ , so its value (and the value of  $s_b$  as well) is reset forever!

בניסיון הזה שינינו את פונקציית ה- $up()$  והוספנו לה  $down(s_b)$ .

גם כאן מתעוררת בעיה דומה, ברגע שנגיע לתהליך השני הוא יוריד את ה-counter ל-0 ואז לא יהיה ניתן להיכנס ל- $up()$  כדי להעלות את ה-counter.

גם כאן ניתן לשנות את התנאי ב- $down$  להיות  $= >$  במקום  $>$ .

אבל שוב היינו מקבלים מצב שבו שלושה תהליכים עלולים להיכנס ל-CS במקביל. מתוך 2 הניסיונות הנ"ל אנו מסיקים את המסקנה הבאה:

כדי לייצר counting semaphore  
Binary semaphore  
אנו זקוקים לשני Binary semaphore.

## ניסיון 3

```
down (S) {
    down (S1) ;
    down (S2) ;
    Sint--;
    if (Sint > 0)
        up (S2) ;
    up (S1) ;
}
```

```
up (S) {
    down (S1) ;
    Sint++;
    up (S2) ;
    up (S1) ;
}
```

Process 0:

$S1=S2=0$ ,

$S=1$

$S1=S2=1$ ,

(CS)

$\rightarrow$  Context switch

Process 1:

$S1=S2=0$

$S=0$

$S1=1$

$\rightarrow$  Context switch

Process 2:

$S1=0$

Cannot proceed / enter the CS

$\rightarrow$  Context switch

Process 0: finish CS. Call Up:

$S1=0 \rightarrow$  cannot release the lock!

$s_{int} - \text{counter}$ .

$S1$  - binary semaphore שמגן על ה-counter  
 $S2$  - binary semaphore שמגן על ה-CS.

כאשר נריץ 3 תהליכים במקביל התהליך הראשון מתחיל וכל הבינאריים משתנים ל-0 ושהוא נכנס ל-CS הם חוזרים ל-1. ה-counter יורד ל-1, התהליך השני מתחיל, כל הבינאריים משתנים ל-0 וכשהוא נכנס ל-CS,  $S1$  חוזר ל-1 אבל  $S2$  נשאר 0 (כי הגדרנו את  $S2$  לשמור על ה-CS). התהליך השלישי יתחיל, ינסה לבצע  $down()$  ונתקע (כמו שצריך). התהליך השלישי ביצע  $down(s1)$  ונתקע ב- $down(s2)$  לאחר שהתהליך הראשון מסתיים הוא מבצע  $up()$  ובשורה הראשונה הוא צריך לעשות  $down(s1)$  אבל התהליך ה-3 כבר הוריד אותו והגענו למצב שהנעילה לא משתחררת.

## הפתרון הנכון (solution of Barz)

```
down(S) {  
    down(S2);  
    down(S1);  
    Sint--;  
    if (Sint > 0)  
        up(S2);  
    up(S1);  
}
```

```
up(S) {  
    down(S1);  
    Sint++;  
    if (Sint == 1)  
        up(S2);  
    up(S1);  
}
```

## בעיית Producer-Consumer -

הבעיה: בהינתן מעבד בעל N ליבות, מספר גדול של עבודות בלתי תלויות שיש לבצע ויצרני עבודות הנקראים Producers. נרצה לדעת מהו המודל האפקטיבי ביותר (מהיר ומנצל חומרה בצורה נכונה) שניתן לעבד בעזרתו את כל העבודות.

פתרון גרוע - יצירת thread לכל עבודה (אסון ארכיטקטוני):

- אי אפשר ליצור מספר אינסופי של תהליכים כיוון שקיים limit כלשהו.
- לכל תהליכון מוקצה זיכרון וזוהי הקצאה מבזבזת לחלוטין כי רוב הזיכרון לא ימומש.
- נניח שיש לנו מליון תהליכים במעבד עם 16 ליבות, scheduler יצטרך לעבוד מאוד קשה כדי לתת לכולם זמן ב-CPU ובנוסף המון זמן יתבזבז על context switch.
- פתרון לא טוב - יצירת thread בודד שיטפל בכל העבודות:
- thread יעבוד רק על ליבה אחת במקום לנצל את כל N הליבות שיש במעבד.
- פוגע בעיקרון multiprocessing.
- פתרון טוב - שימוש ב-Threadpool:

- ניצר מספר אופטימלי של threads ע"י החישוב הבא:  $k \geq 1$ ,  $\# \text{ of threads} = N * k$ , כאשר N זה מספר הליבות ו-k הוא פרמטר התלוי באחוז העבודות הדורשות CPU.
- עבור בעיות שהן דורשות 100% CPU מספיק לנו N תהליכים כדי לנצל את כל המעבדים, עבור בעיות שהן 50% CPU ו-50% I/O נצטרך 2N תהליכים כדי לנצל את כל המעבדים.
- כעת נכנס לעומק בעיית ה-Producer-Consumer - ראשית נגביל אסור ליצור מבנה נתונים ללא מגבלה של כמות אובייקטים ולכן נגיל את מבנה הנתונים ל-M מקומות, נשים לב ל-3 בעיות:
  1. כאשר ה-Producer ירצה להכניס עבודות הוא יצטרך לוודא שמבנה הנתונים לא מלא (Full).
  2. כאשר ה-Consumer ירצה לעבד עבודות הוא יצטרך לבדוק שמבנה הנתונים לא ריק (Empty).
  3. כאשר ניגשים למבנה הנתונים, נרצה שהכניסה תהיה בטוחה ושרק תהליכון אחד יוכל לבצע שינוי (הכנסה/הוצאה) במבני הנתונים בכל זמן נתון.
- דוגמת קוד של Producer & Consumer:

```
#define      N      100                                /* Buffer size */
Mutex       UseQ = 1;                                /* access control to CS */
semaphore   empty = N;                                /* counts empty buffer slots */
semaphore   full = 0;                                 /* counts full buffer slots */
void producer(void) {
    int      item;
    while(1) {
        produce_item(&item);                          /* generate something... */
        down(&empty);                                  /* decrement count of empty */
        down(&UseQ);                                    /* enter critical section */
        enter_item(item);                              /* insert into buffer */
        up(&UseQ);                                       /* leave critical section */
        up(&full);                                       /* increment count of full slots */
    }
}
```

```
void consumer(void) {
    int      item;

    while(1) {
        down(&full);                                    /* decrement count of full */
        down(&UseQ);                                    /* enter critical section */
        consume_item(&item);                            /* take item from buffer */
        up(&UseQ);                                       /* leave critical section */
        up(&empty);                                     /* update count of empty */
    }
}
```

## הסבר הקוד:

ראשית נדגיש שאנו עובדים עם שני Threadpools, אחד של Producer ואחד של Consumer. בנוסף הקוד כתוב בשפת C ויש להתייחס אליו כאל פסודו-קוד ואין להתעכב על האלמנטים של C. לצורך הדוגמה נשתמש במבנה נתונים מסוג תור.

שלבי העבודה של Producer כאשר הוא רוצה להכניס עבודה חדשה לתור:

1. יצירת העבודה באמצעות הפונקציה `produce_item(&item)`.
2. הורדת ה `counter` של מספר המקומות הריקים בתור ע"י הפונקציה `down(&empty)` שמקבלת את ה `empty semaphore`, פונקציה זו מורידה 1 מה `empty semaphore`, במידה `empty` כבר על 0, זה אומר שאין מספיק מקומות פנויים בתור ולכן ה `Producer` הספציפי הזה ימתין עד שיהיה מקום בתור להכנסת עבודות חדשות.
3. לאחר שה `Producer` העלה את ה `empty` ב 1 הוא ינסה לקבל גישה לתור באמצעות `Mutex UseQ` ע"י הפונקציה `down(&UseQ)`, במידה והצליח לקבל גישה לתור הוא עובר לשלב הבא, אחרת הוא ימתין.
4. כעת ה `Producer` יכניס את העבודה לתור ע"י `enter_item(item)` ויפתח את השימוש בתור ליצרנים/צרכנים אחרים באמצעות `up(&UseQ)`.
5. לאחר שהכניס את העבודה לתור ה `Producer` יעלה את מספר המקומות המלאים בתור באמצעות הפונקציה `up(&full)`.

שלבי העבודה של Consumer כאשר הוא רוצה להוציא עבודה מהתור:

1. הורדת ה `counter` של מספר מספר המקומות המלאים בתור ע"י הפונקציה `down(&full)` שמורידה 1 מה `full semaphore`, במידה `full` כבר על 0, זאת אומרת שאין עבודות בתור ולכן ה `Consumer` הספציפי הזה ימתין שם עד שעבודה חדשה תיכנס לתור.
2. לאחר שה `consumer` הוריד את ה `full` ב 1, הוא ינסה לקבל גישה לתור באמצעות ה `Mutex UseQ` ע"י הפונקציה `down(&UseQ)`, במידה והצליח לקבל גישה הוא עובר לשלב הבא, אחרת הוא ממתיין.
3. כעת ה `Consumer` יוציא את העבודה מהתור ע"י `consume_item(item)` ויפתח את השימוש בתור ליצרנים/צרכנים אחרים באמצעות `up(&UseQ)`.
4. לאחר שהוציא את העבודה מהתור ה `Consumer` יעלה את מספר המקומות הריקים בתור באמצעות הפונקציה `up(&empty)`.

מספר דגשים:

1. ה `Producer` מנסה להוריד את ה `counter` של `empty`, כשיגיע ל 0 אין מקום לעבודות נוספות ולכן ה `Producers` יילכו לישון.
2. ה `Consumer` מנסה להוריד את ה `counter` של `full`, כשיגיע ל 0 אין עבודות בתור ולכן ה `Consumers` יילכו לישון.

שאלה 1: מה יקרה אם נבצע החלפה כמתואר

בתמונה בין שורות הקוד?

תשובה: ההכנסה לתור לא תהיה מאובטחת כי יכולים

להיות עד 100 תהליכונים ( `Producers & Consumers`) בקטע הקוד הזה וכולם מנסים לשנות

את התור במקביל ללא הבטחה שרק אחד יקבל גישה

לתור בכל פעם.

ה `Consumer` יכול להוציא את העבודה לפני שהוא

הוכנס לתא בכלל וזה אומר לקבל ערך שגוי.

```
void producer(void) {
    int item;
    while(1){
        produce_item(&item); /* generate something... */
        down(&empty);        /* decrement count of empty */
        down(&UseQ);          /* enter critical section */
        up(&UseQ);            /* leave critical section */
        enter_item(item);     /* insert into buffer */
        up(&full);            /* increment count of full slots */
    }
}
```

```
void producer(void) {
    int item;
    while(1){
        produce_item(&item); /* generate something... */
        down(&empty);        /* decrement count of empty */
        down(&UseQ);          /* enter critical section */
        enter_item(item);     /* insert into buffer */
        up(&full);            /* increment count of full slots */
        up(&UseQ);            /* leave critical section */
    }
}
```

שאלה 2: מה יקרה אם נבצע החלפה כמתואר

בתמונה בין שורות הקוד?

תשובה: כאן לא תהיה בעיה אבל עדיף שלא כיוון

שאין צורך להגן על `full` באמצעות ה `mutex` של התור.

```

void consumer(void) {
    int item;

    while(TRUE) {
        down(&UseQ);           /* enter critical section */
        down(&full);           /* decrement count of full */
        remove_item(&item);    /* take item from buffer */
        up(&UseQ);              /* leave critical section */
        up(&empty);             /* update count of empty */
        consume_item(item);    /* do something... */
    }
}

```

**שאלה 2:** מה יקרה אם נבצע החלפה כמתואר בתמונה בין שורות הקוד?  
**תשובה:** ה Consumer יקבל גישה לתור וינעל אותו אך כיוון שלא נעשה בדיקה שקיימות בכלל עבודות בתור יכול ליווצר מצב שהוא נכנס לתור ואין עבודות פנויות ולכן ה Consumer יילך לישון וכך הגענו למצב של Deadlock – אף אחד לא יכול לגשת ל CS.

### – Deadlock

4 תנאים הכרחיים ליצירת deadlock:

משאב – semaphore, mutex, printer, etc.

- Mutual exclusion – המשאב משומש ע"י תהליך אחד בכל זמן נתון.
- Hold and wait – תהליך יכול לבקש משאב תוך החזקת משאב אחר.
- No preemption – רק תהליך יכול לשחרר את המשאב שאותו הוא מחזיק (תהליך אחר לא יכול להעניף את התהליך מהמשאב).
- Circular wait – שני תהליכים או יותר הממתינים למשאבים המוחזקים ע"י תהליכים אחרים (לדוגמה, תהליך A מחכה למשאב שתפוס ע"י תהליך B, ותהליך B מחכה למשאב שתפוס ע"י משאב A).

נשים לב שכדי שיווצר deadlock חייב שכל ארבעת התנאים יתקיימו במקביל, לכן נשאלת השאלה מה ניתן לתקן בכל אחד מהתנאים כדי למנוע את יצירת ה deadlock.

- Mutual exclusion – לא תמיד ניתן למנוע, לדוגמה אם תהליך מסוים תפס mutex, רק הוא ישתמש בו כי זה "האופי" של mutex (היעדרות עלולה לגרום לאובדן שליטה במשאבים ולהפרות מזיקות).
- Hold and wait – לא תמיד ניתן למנוע, לצורך ביצוע עבודות מסוימות צריך להצטייד ב A, B, C ולא רק במשאב אחד (היעדרות יכולה לגרום לתוכניות לפעול לזמנים ארוכים יותר עם זמן המתנה רב יותר).
- No preemption – בדומה ל Mutual exclusion, קיימים משאבים שזה "האופי" שלהם (היעדרות עלולה לגרום לאובדן נתונים חשובים שלא גובו).
- Circular wait – כמעט בלתי אפשרי ליישום, ואם כן, לא ניתן להרחבה.

### דוגמה מטפורית ל deadlock:

איתי וליאב, כל אחד בנפרדת צריכים לבצע עבודה שחייב בה גם פטיש וגם את חפירה, איתי תפס את הפטיש וליאב תפס את החפירה.

נעבור על התנאים לבדוק אם יש כאן deadlock.

- יש כאן mutual-exclusion - כי רק איתי משתמש בפטיש ורק ליאב משתמש באת החפירה.
- Hold and Wait - כי איתי לקח את הפטיש ומחכה גם לאת החפירה ובאותו אופן ליאב לקח את האת ומחכה לפטיש.
- no preemption - בניח שאין כאן בוס שיכול לקחת להם את הכלי עבודה ולכן התנאי הזה גם מתקיים. רק איתי יכול לשחרר את הפטיש ורק ליאב יכול לשחרר את האת.
- Circular wait – איתי מחכה למשאב המוחזק ע"י ליאב וליאב מחכה למשאב המוחזק ע"י איתי (Circular dependency). ולכן נוצר כאן deadlock.

איך הדוגמה תראה בקוד?

#### Itai's Code:

Take p  
 Hold-And-Wait  
 Take m

#### Liaiv's Code:

Take m  
 Hold-And-Wait  
 Take p



1. נאחד את המשאבים למקטע אטומי, נשתמש בmutex.

|  |  |
|--|--|
| <p><b>Itai's Code:</b></p> <p>Mutex.lock<br/>Take p<br/>Take m<br/>Make the job<br/>Mutex.unlock</p> | <p><b>Liav's Code:</b></p> <p>Mutex.lock<br/>Take m<br/>Take p<br/>Make the job<br/>Mutex.unlock</p> |
|--|--|

2. תפיסה באותו הסדר, שניהם ינסו לגשת קודם לפטיש ורק אחרי שהשיגו את הפטיש הם ינסו לתפוס את האת.

|  |  |
|--|--|
| <p><b>Itai's Code:</b></p> <p>Take m (or wait)<br/>Take p<br/>Make the job</p> | <p><b>Liav's Code:</b></p> <p>Take m (or wait)<br/>Take p<br/>Make the job</p> |
|--|--|

כאן לא ייוצר deadlock כי הראשון שייקח את המ יקח גם את p. בנוסף נשים לב שגם אם איתי יקח ראשון את m והscheduler יכניס בנקודה הזו את ליאב, ליאב יתקע במ כי m כבר תפוס, ואז scheduler יחזור בחזרה לאיתי שייקח את גם את p ויתחיל לבצע את העבודה. בפתרון זה שברנו את הcircular dependency.

כדי למנוע מצב deadline צריך לוודא שלפחות אחד מ-4 התנאים אינו מתקיים.  
נקצה משאבים רק אחרי שנוודא שהם "בטוחים" (safe), כלומר המשאבים לא יכולים להוביל למצב של deadlock.  
נמצא גרף מעגלי של תהליכים ומשאבים.  
נסה לעשות recover ע": להרוג את התהליך או לקחת ממנו את המשאב.

### שאלה 1:

יהיו  $R_1, R_2, \dots, R_n$  משאבים במערכת.  
ונניח שהמשאבים ייחודיים – קיים מופע יחיד לכל משאב.  
הוכיחו שאם ניתן לתהליכים לבקש משאבים רק על פי הזמנתם קיים מצב שבו נקבל deadlock.  
תהליך P יכול לבקש משאב  $R_b$  בזמן שהוא מחזיק משאב  $R_a$  רק במידה  $b > a$ .

### הוכחה:

נניח שמערכת כוללת תהליכים  $P_1, P_2$  ומשאבים  $R_1, R_2$ .  
נניח כי  $P_1$  מחזיק במשאב  $R_1$  ו  $P_2$  מחזיק במשאב  $R_2$ .  
עכשיו  $P_1$  מבקש את  $R_2$  ועכשיו הוא ממתין ל  $P_2$  שישחרר אותו.  
כדי שיהיה deadlock במערכת,  $P_2$  צריך לבקש את  $R_1$  אבל הדבר מנוגד להנחה כי ניתן לבקש משאבים רק בסדר עולה.  
תנאי 4 (Circular wait) נמנע.

נניח שהמערכת שלנו נוטה למצבי deadlock ושהתהליכים שלה הם  $P_1, P_2, \dots, P_n$  ונדרוש שתנאי 4 (שהכרחי למצב deadlock) קורה.

נגדיר מצב  $P_i - R_k \rightarrow P_j$  המציין שהתהליך  $P_i$  מבקש משאב  $R_k$  המוחזק על ידי תהליך  $P_j$ .  
כדי שיתרחש deadlock תת קבוצה של  $P_{i_1}, P_{i_2}, \dots, P_{i_m}$  של  $\{P_1, P_2, \dots, P_n\}$  העומדת בתנאי הבא בהכרח קיימת:

$$P_{i_1} - R_{j_1} \rightarrow P_{i_2} - R_{j_2} \rightarrow \dots \rightarrow R_{j_{m-1}} \rightarrow P_{i_m} - R_{j_m} \rightarrow P_{i_1}$$

לכל תהליך  $P_{i_s}$  שבו  $s \neq 1$ ,  $P_{i_s}$  מחזיק משאב  $R_{j_{s-1}}$  ומבקש את משאב  $R_{j_s}$ .

ולכן  $J_{s-1} < J_s$  לכל  $s$  כיוון שהמשאבים ממוספרים ומתבקשים בסדר עולה.

לכן נקבל ש  $J_{s1} < J_2 < \dots < J_m$ .

המצב של deadlock אנו מסיקים ש  $J_m < J_1$  ומצד שני  $J_1 < J_m$  ולכן מתקיים circular wait, אך מכיוון שזה בלתי אפשרי, circular wait לא תתרחש, והמערכת היא deadlock-free.

## The Banker's Algorithm

אלגוריתם למניעת deadlock.

למערכת יש שלושה מצבים –

1. בטוח (Safe) – לא deadlock, יש סדר תזמונים כלשהו שבו כל תהליך יכול לפעול עד לסיומו, גם אם לפתע כולם מבקשים מיד את מספר המשאבים המרבי שלהם.
2. לא בטוח (Unsafe) – לא deadlock אבל יכול להיות שיהיה בהמשך.
3. Deadlock.

משאבים:

Vectors:

- E מספר המשאבים בקיימים מכל סוג.
- P מספר המשאבים מכל סוג שנמצאים תחת בעלות של תהליכים.
- A מספר המשאבים זמינים מכל סוג.
- Matrices – (שורות-תהליכים, עמודות-משאבים):
- C המטריצה הנוכחית שהוקצתה.
- R המטריצה המבוקשת.

האלגוריתם:

1. חפש שורה במטריצה R שכל צרכי המשאב בהם קטנים או שווים ל A והמשך לשלב 2.
- אם אין שורה כזו, עבור לשלב 3.
2. נניח שתהליך בשורה שנבחרה מסיים (זה אפשרי).
- סמן את התהליך כterminated והוסף את כל המשאבים שלו לזמין A.
3. חזור על שלבים 1 ו-2 עד שכל התהליכים יסומנו כterminated, מה שאומר שהמערכת בטוחה (Safe), או עד שמתרחש deadlock, מה שאומר שהמערכת לא בטוחה (Unsafe).

## שאלה 2:

A=

| R <sub>1</sub> | R <sub>2</sub> | R <sub>3</sub> | R <sub>4</sub> |
|----------------|----------------|----------------|----------------|
| 2              | 1              | 0              | 0              |

נסתכל על תמונת המצב של מערכת בעלת חמישה processes וארבעה resources. האם המערכת נמצאת בdeadlock? האם המערכת עלולה להגיע למצב של deadlock?

| Process        | current allocation |                |                |                | max demand     |                |                |                | still needs    |                |                |                |
|----------------|--------------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
|                | R <sub>1</sub>     | R <sub>2</sub> | R <sub>3</sub> | R <sub>4</sub> | R <sub>1</sub> | R <sub>2</sub> | R <sub>3</sub> | R <sub>4</sub> | R <sub>1</sub> | R <sub>2</sub> | R <sub>3</sub> | R <sub>4</sub> |
| P <sub>1</sub> | 0                  | 0              | 1              | 2              | 0              | 0              | 1              | 2              | 0              | 0              | 0              | 0              |
| P <sub>2</sub> | 2                  | 0              | 0              | 0              | 2              | 7              | 5              | 0              | 0              | 7              | 5              | 0              |
| P <sub>3</sub> | 0                  | 0              | 3              | 4              | 6              | 6              | 5              | 6              | 6              | 6              | 2              | 2              |
| P <sub>4</sub> | 2                  | 3              | 5              | 4              | 4              | 3              | 5              | 6              | 2              | 0              | 0              | 2              |
| P <sub>5</sub> | 0                  | 3              | 3              | 2              | 0              | 6              | 5              | 2              | 0              | 3              | 2              | 0              |

כמות המשאבים הזמינים שאף אחד לא תפס

מטריצה שמציגה את מספר המשאבים שיש לכל תהליך לפי סוגים

מטריצה שמציגה עבור כל תהליך את מספר המשאבים שהוא צריך לפי סוגים

מטריצת request: כמה משאבים התהליך צריך כדי להשלים את העבודה

נבנה את request matrix באמצעות חיסור התא המתאים ב max demand לבין current allocation. נעבור על שלבי האלגוריתם:

1. בשלב 1 נבחר את תהליך 1 כי רק לו יש את כל המשאבים.
2. נעבור לשלב 2, נניח שתהליך 1 סיים, נסמן אותו כterminated ונעביר את כל המשאבים שלו ל A.
3. כעת נראה שתהליך 4 יכול מסופק.
4. נבחר אותו, נסמן אותו כterminated, נשחרר את המשאבים ונעדכן את A.
5. עתה ניתן לבחור את תהליך 5 וכן הלאה, בסוף נגלה שכל התהליכים במצב terminated ולכן המערכת במצב Safe.

A=

| R <sub>1</sub> | R <sub>2</sub> | R <sub>3</sub> | R <sub>4</sub> |
|----------------|----------------|----------------|----------------|
| 2              | 1              | 1              | 2              |

A=

| R <sub>1</sub> | R <sub>2</sub> | R <sub>3</sub> | R <sub>4</sub> |
|----------------|----------------|----------------|----------------|
| 4              | 4              | 6              | 6              |

### שאלה 3:

אם בקשה ל (0, 0, 1, 0) מגיעה מ-  $P_3$ , האם ניתן לאשר את הבקשה באופן מיידי בבטחה? באיזו מצב (deadlock, safe, unsafe) נעניק מיד את כל הבקשה למערכת? אילו תהליכים, אם בכלל, עלולים להיות deadlock אם כל הבקשה הזו תוענק באופן מיידי?

תשובה:

בואו נראה מה עלול לקרות אם הבקשה של  $P_3$  תיעשה מיד.

נניח ש- $P_3$  דרש את  $R_2$ .

A=

| $R_1$ | $R_2$ | $R_3$ | $R_4$ |
|-------|-------|-------|-------|
| 2     | 0     | 0     | 0     |

| Process | current allocation |       |       |       | max demand |       |       |       | still needs |       |       |       |
|---------|--------------------|-------|-------|-------|------------|-------|-------|-------|-------------|-------|-------|-------|
|         | $R_1$              | $R_2$ | $R_3$ | $R_4$ | $R_1$      | $R_2$ | $R_3$ | $R_4$ | $R_1$       | $R_2$ | $R_3$ | $R_4$ |
| $P_1$   | 0                  | 0     | 1     | 2     | 0          | 0     | 1     | 2     | 0           | 0     | 0     | 0     |
| $P_2$   | 2                  | 0     | 0     | 0     | 2          | 7     | 5     | 0     | 0           | 7     | 5     | 0     |
| $P_3$   | 0                  | 0     | 3     | 4     | 6          | 6     | 5     | 6     | 6           | 5     | 2     | 2     |
| $P_4$   | 2                  | 3     | 5     | 4     | 4          | 3     | 5     | 6     | 2           | 0     | 0     | 2     |
| $P_5$   | 0                  | 3     | 3     | 2     | 0          | 6     | 5     | 2     | 0           | 3     | 2     | 0     |

שינינו את A ל (2, 0, 0, 0) ואת השורה של  $P_3$  still need ל (2, 5, 2, 2).

עכשיו  $P_1, P_4, P_5$  יכולים להסתיים (terminate).

ווקטור A יתעדכן ל (8, 9, 6, 4), כלומר לא ניתן לספק את "הצרכים" של  $P_2$  וגם לא את של  $P_3$ .

לכן, לא בטוח להיעתר לבקשת  $P_3$ .

תהליכים  $P_2$  ו- $P_3$  עשויים להיכנס לdeadlock.

A=

| $R_1$ | $R_2$ | $R_3$ | $R_4$ |
|-------|-------|-------|-------|
| 2     | 0     | 0     | 0     |

A=

| $R_1$ | $R_2$ | $R_3$ | $R_4$ |
|-------|-------|-------|-------|
| 2     | 0     | 1     | 2     |

A=

| $R_1$ | $R_2$ | $R_3$ | $R_4$ |
|-------|-------|-------|-------|
| 4     | 3     | 6     | 6     |

A=

| $R_1$ | $R_2$ | $R_3$ | $R_4$ |
|-------|-------|-------|-------|
| 4     | 6     | 9     | 8     |

| Process | current allocation |       |       |       | still needs |       |       |       |
|---------|--------------------|-------|-------|-------|-------------|-------|-------|-------|
|         | $R_1$              | $R_2$ | $R_3$ | $R_4$ | $R_1$       | $R_2$ | $R_3$ | $R_4$ |
| $P_1$   | 0                  | 0     | 1     | 2     | 0           | 0     | 0     | 0     |
| $P_2$   | 2                  | 0     | 0     | 0     | 0           | 7     | 5     | 0     |
| $P_3$   | 0                  | 0     | 3     | 4     | 6           | 5     | 2     | 2     |
| $P_4$   | 2                  | 3     | 5     | 4     | 2           | 0     | 0     | 2     |
| $P_5$   | 0                  | 3     | 3     | 2     | 0           | 3     | 2     | 0     |

### – Unbounded Priority Inversion

נניח ואנחנו נמצאים במערכת real-time ויש לנו 3 משימות עם priority שונה (low, medium & high).

High וmedium היו עסוקים (נניח שהמתנו ל I/O) ובינתיים low התחיל לעבוד והגיע ותפס את mutex,

ונכנס לCS. בינתיים high סיים את ההמתנה ל I/O והוא רוצה להתחיל לעבוד באותו CS אך mutex נעול ע"י low.

לכן high נכנס לתור המתנה (sleeping queue) של mutex. בזמן זה medium סיים ורצה להיכנס למשימה

אחרת ומכיוון שיש לו עדיפות גבוהה יותר משל low הCPU החליף ביניהם.

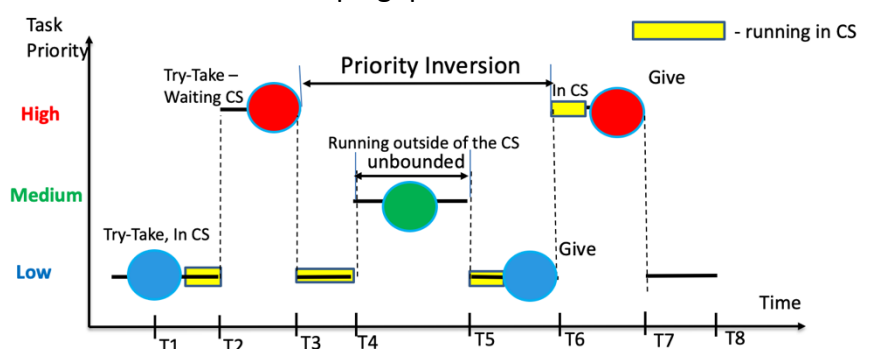
נשים לב שהpriority task הוא unbounded כיוון שכל עוד medium לא יסיים את עבודתו low והmedium יתקעו

ורק לאחר שהmedium יסיים את עבודתו, low יכנס ויסיים את עבודתו ורק אז high יתעורר וייכנס לCS.

שורה תחתונה – priority inversion זהו מצב שבו תהליך עם עדיפות נמוכה נמצא בCPU בעוד שתהליך אחר בעל

עדיפות גבוהה יותר נמצא בsleeping queue.

- T1 – LP-Task locks the shared resource (SHR)
- T2 – HP-Tasks is ready. Context switch to HP-Task
- T3 – HP-Task wants to take the SHR, but waits since it's taken by LP-Task. Therefore, LPH continues execution till T4.
- T4 – MP-Task is ready and preempts LP-Task since it doesn't require SHR.
- T5 – MP-Task yields or completes. The duration between T4 and T5 is not known and cannot be predicted = Unbounded
- T6 – LP-Task releases the SHR.  
Finally, HP-Task can take SHR and proceed.  
HP-Task completes SHR and releases it.
- T7 – HP-Task yields the CPU or completes. Any task, i.e. LH-Task can take the CPU.
- T3 – T6 – **Priority Inversion**
- T4 – T5 – **Unbounded run of MP-Task**



איך נמנע priority inversion?  
 באמצעות פרוטוקול שנקרא priority inheritance protocol שניתן ליישם רק על mutex כיוון שרק למשאב זה יש thread ownership.  
 איך הפרוטוקול עובד?  
 ברגע שtask מגיע לתור של mutex, במידה והעדיפות שלו גבוהה מהעדיפות של התהליך שנעל את mutex, מערכת ההפעלה מעלה את העדיפות של התהליך שנעל את mutex עד לרמה של התהליך שמחכה בתור (boost priority) וברגע שהוא יסיים ויצא מהCS מערכת ההפעלה תחזיר לו את העדיפות הקודמת שלו.  
 אם נתייחס לדוגמה הקודמת, בזמן שההhigh נכנס לתור מערכת ההפעלה היתה מבצעת boost priority לlow ואז medium כלל לא היה מחליף אותו וכך ההhigh היה נכנס לפני low.

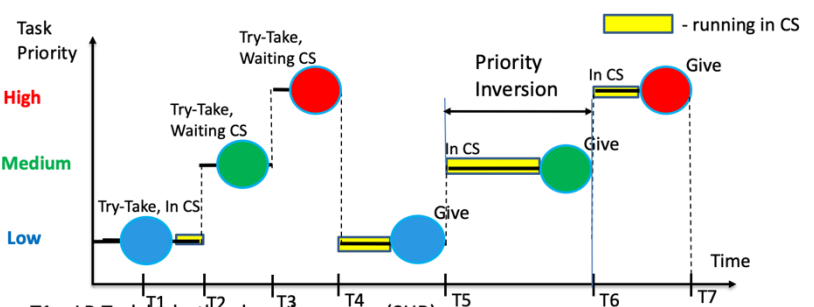
דגשים:

- לא ניתן להשתמש בפרוטוקול זה על semaphore.
- פרוטוקול זה אינו deadlock free.

### – Bounded Priority Inversion

נניח ואנחנו נמצאים במערכת real-time ויש לנו 3 משימות עם priority שונה (low, medium & high).  
 High וmedium היו עסוקים (נניח שהמתינו לI/O) ובינתיים low התחיל לעבוד והגיע ותפס את mutex, ונכנס לCS. בינתיים medium סיים את ההמתנה ולכן context switch מכניס אותו (יש לי עדיפות גבוהה יותר) וכשהוא הוא רוצה להתחיל לעבוד באותו CS הוא מגלה שהmutex נעול ע"י low ולכן medium נכנס לתור המתנה (sleeping queue) של mutex. בזמן זה ההhigh סיים את ההמתנה העדיפות שלו גבוהה משל medium ולכן context switch מכניס אותו וכשהוא רוצה להיכנס לCS הוא גם מגלה שהוא נעול ולכן גם הוא ממתיין. נשים לב שהmedium והhigh נמצאים בFIFO queue.  
 context switch מחזיר את low עד שהוא מסיים לעבוד בCS, medium מקבל זמן בCPU עד שהוא מסיים ולאחר מכן ההhigh.  
 priority inversion מסוג זה הוא bounded כיוון שההhigh ממתיין זמן מוגבל (לכל מי שלפניו תור) ולעומת זאת unbounded תהליכים נוספים יכולים להגיע ולהיכנס לפניו ולכן זמן ההמתנה שלו אינו מוגבל.

T1 – LP-Task locks the shared resource (SHR)  
 T2 – MP-Task is ready. Context switch to MP-Task  
 T3 – MP-Task wants to take the SHR, but waits since it's taken by LP-Task.  
 T4 – Both MP and HP-Task are waiting in FIFO queue of SHR, but it's taken by LP-Task.  
 T5 – LP-Task yields or completes CS and releases SHR.  
 T5 – Due to FIFO ordering, MP-Task takes the SHR and runs while HP-Task is waiting.  
 T5 – T6 – **Priority Inversion.**  
 The priority inversion is **bounded** since the duration of usage SHR by MP-Task could be predicted or estimated.  
 T6 – MP-Task yields or completes CS and releases SHR.  
 T6 – HP-Task takes SHR and runs  
 T7 – HP-Task yields or completes CS and releases SHR.



### – Priority Inheritance Protocol

כאשר משימה בעדיפות גבוהה יותר מבקשת את אותו משאב, עדיפות הביצוע של התהליך שמחזיק את המשאב מוגברת (boosting) לרמת העדיפות של המשימה המבקשת.  
 המשימה חוזרת לעדיפות הקודמת שלה כשהיא משחררת את המשאב.  
 הערה: פרוטוקול זה אינו deadlock-free.

## – Ceiling Priority Protocol (CPP)

ישנן שתי גרסאות לפרוטוקול:

- Original Ceiling Priority Protocol (OCP)
- Immediate Ceiling Priority Protocol (ICPP)

פרוטוקול זה מונע deadlock ו Unbounded Priority Inversion (מצב של bounded Priority Inversion אפשרי).

הגדרות הפרוטוקול:

A.  $Ceil(s)$  – "התקרה" של semaphore זה העדיפות של task עם העדיפות הגבוהה ביותר שמשתמשת

semaphore.

B. Task(i) עם עדיפות מסוימת יכולה לנעול semaphore (lock) רק עם העדיפות של task גבוהה מה  $ceil(s)$  של כל semaphore שנועלים כרגע ע"י task אחרים.

C. אם B לא נכון, task תהיה נעולה על ידי semaphore אחר  $S^*$  (semaphore עם ה  $ceil$  הגבוה ביותר מכל semaphore שנועלים כרגע ע"י משימות אחרות).

D. אם task(i) נעולה (blocked) כרגע ע"י  $S^*$ , task שמחזיקה את  $S^*$  יורשת את העדיפות של task(i).

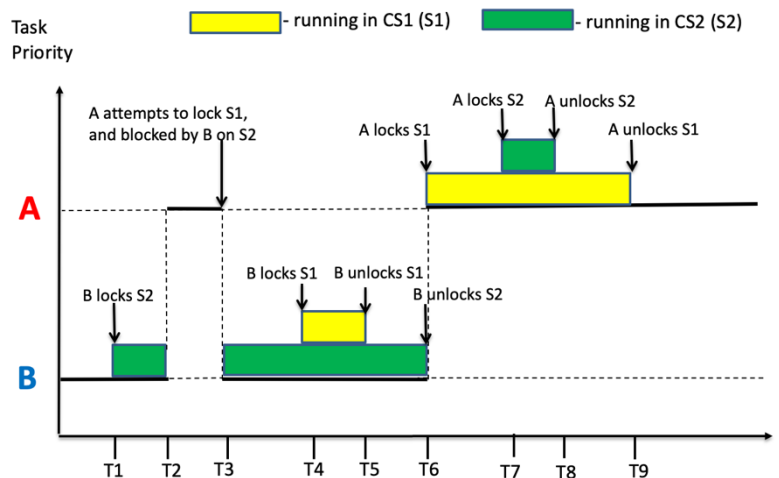
## דוגמה:

| Task Name | Time | Priority | Action  | Sem Ceiling                      |
|-----------|------|----------|---|----------------------------------|
| A         | 50   | 3        | lock (S1)<br>lock (S2)<br>...<br>unlock (S1)<br>unlock (S2) | $ceil(S1) = 3$<br>$ceil(S2) = 3$ |
| B         | 500  | 2        | lock (S2)<br>lock (S1)<br>...<br>unlock (S1)<br>unlock (S2) |                                  |

נתונות 2 משימות A ו B, בעלי זמני ביצוע שונים ועדיפויות שונות ולכל אחת יש משימה אחרת.

כאשר ה  $ceil$  של כל אחד משני semaphore שקיימים במשימות מוגדרים להיות העדיפות של A כיוון שזו task עם העדיפות הגבוהה ביותר שמשתמשת בכל אחד מה semaphore.

- T1 – "B" tries to lock S2. It succeeds since no lock is held by another task.
- T2 – "A" starts to run and preempts "B"
- T3 – "A" tries to lock S1. It fails since "A's" priority (3) is not strictly higher than the ceiling of S2 (3) held by "B"
- "A" is blocked by "B" ... on s2 and the priority of "B" is raised to 3.
- T4 – "B" tries to lock s1. It succeeds since there are no locks held by any other tasks. The priority inversion is **bounded**.
- T5 – "B" unlocks S1
- T6 – "B" unlocks S2
- The priority of "B" is lowered to its assigned priority (2)
- T6 – "A" preempts "B", attempts to lock S1 and succeeds
- T7 – "A" tries to lock S2. Succeeds
- T8 – "A" unlocks S2.
- T9 – "A" unlocks S1.



הסבר: B מנסה לנעול את  $S_2$  ומצליח כיוון שאף משימה אחרת לא מחזיקה אותו.

לאחר מכן משימה A נכנסת והצטמצם עובר אליה כיוון שהעדיפות שלה גבוהה יותר, היא מנסה לנעול את  $S_1$  אבל לא מצליחה כיוון שהעדיפות של A היא 3 והיא לא גדולה ממש מה  $ceil$  של כל semaphore שנועלים כרגע ע"י task אחרים (במקרה זה  $S_2$  נעולה ע"י B וה  $ceil(S_2)=3$ ). ולכן A ננעלת ע"י B והעדיפות של B עולה ל3.

הצטמצם עובר ל B שממשיך לעבוד ולאחר מכן מנסה לנעול את  $S_1$  ומצליח כיוון שאף משימה אחרת לא מחזיקה אותו. נשים לב שזמן העבודה של  $S_1$  בתוך  $S_1$  זה בדיוק ה bounded priority inversion.

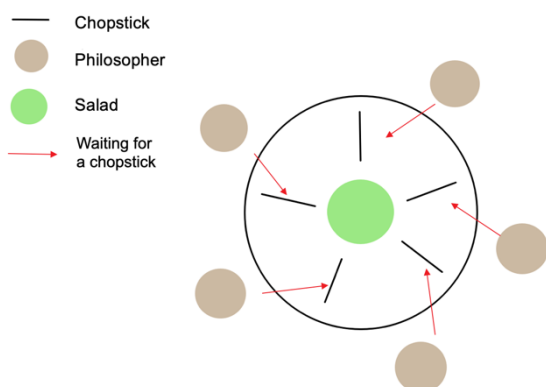
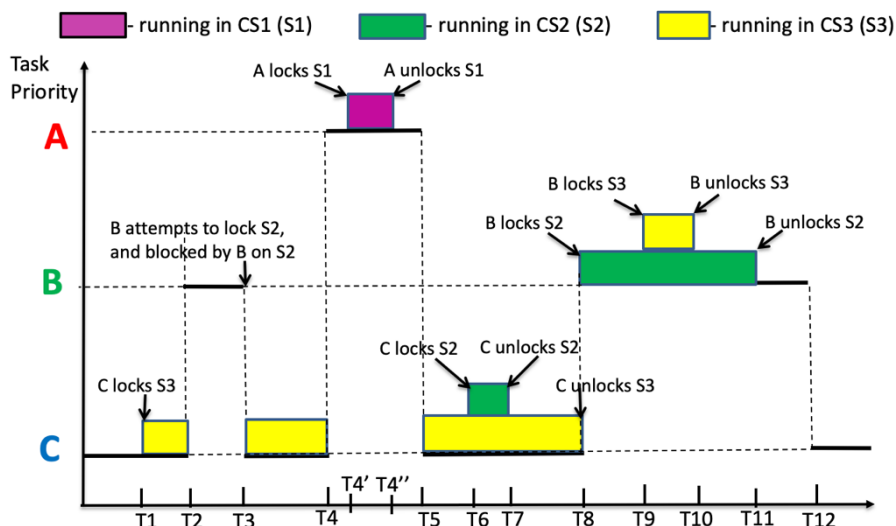
B משחרר את הנעילה על  $S_1$  ולאחר מכן גם על  $S_2$  והעדיפות של B מתעדכנת בחזרה ל2.

כעת A מקבל זמן CPU לבצע את העבודה שלו (נועל את  $S_1$  כיוון שאף אחד לא מחזיק אותו וכלל לגבי  $S_2$  משחרר אותם ומסיים).

| Task Name | Time | Priority | Action   | Sem Ceiling   |
|-----------|------|----------|--|---|
| A         | 50   | 3        | lock (S1) ...<br>unlock (S1) ...                                     | $\text{ceil}(S1) - 3$<br>$\text{ceil}(S2) - 2$<br>$\text{ceil}(S3) - 2$ |
| B         | 500  | 2        | lock (S2) ...<br>lock (S3) ...<br>unlock (S3) ...<br>unlock (S2) ... |   |
| C         | 3000 | 1        | lock (S3) ...<br>lock (S2) ...<br>unlock (S2) ...<br>unlock (S3) ... |   |

נתונות 3 משימות A, B, C בעלי זמני ביצוע שונים ועדיפויות שונות ולכל אחת יש משימה אחרת. כאשר הceil של כל אחד משלושת semaphores שקיימים במשימות מוגדרים להיות העדיפות הגבוהה ביותר מבין כל המשימות שמשתמשות בהן.

- T1 – "C" tries to lock S3. It succeeds since no lock is held by another task.
- T2 – "B" starts to run and preempts "C"
- T3 – "B" tries to lock S2 and fails because the priority of "B" is not strictly higher than the ceiling of S3 held by C.
  - "B" blocks on S3 blocked by C.
  - "C" inherits the priority of "B".
- T4 – "A" preempts "C".
- T4' – "A" tries to lock S1 and succeeds since the priority of "A" is higher than the ceiling of S3.
- T4'' – "A" unlocks S1
- T5 – "A" completes.
- T5 – C resumes,
- T6 – C tries to lock S2 and succeeds (it is C itself that holds S3 and not "other tasks", and there are no "other tasks" holding semaphores).
- T7 – "C" unlocks S2
- T8 – "C" unlocks S3 and the priority of "C" is lowered to its assigned priority (1)
- T8 – "B" preempts "C", tries to lock S2 and succeeds.
- T9 – "B" locks S3
- T10 – "B" unlocks S3
- T11 – "B" unlocks S2
- T11 – "B" completes and "C" is resumed.



## בעיית הפילוסופים הסועדים – Dining Philosophers

N פילוסופים יושבים סביב שולחן עגול וכולם רוצים לאכול. כדי להתחיל לאכול הם צריכים להחזיק 2 צ'ופסטיקים (forks), כאשר אין מספיק לכולם. באיור ניתן לראות שאם כל אחד מהפילוסופים יקח את אתה fork שמימנו קודם נגיע למצב של deadlock כיוון שכל הפילוסופים נמצאים במצב של Hold & Wait.

פתרון Textbook:

נגדיר mutex מרכזי semaphore עבור כל אחד מהפילוסופים.

כאשר פילוסוף ירצה לאכול הוא ינעל את mutex וינסה לקחת 2 מזלגות, במידה ויצליח הוא יתחיל לאכול וישחרר את mutex, במידה ולא היו לו מזלגות לקחת הוא נכנס למצב המתנה וברגע שהמזלגות משני הצדדים שלו יתפנו יעירו אותו והוא יתחיל לאכול.

```

#define N 5 /* number of philosophers */
#define LEFT (i+N-1)%N /* number of i's left neighbor */
#define RIGHT (i+1)%N /* number of i's right neighbor */
#define THINKING 0 /* philosopher is thinking */
#define HUNGRY 1 /* philosopher is trying to get forks */
#define EATING 2 /* philosopher is eating */

typedef int semaphore; /* semaphores are a special kind of int */
int state[N]; /* array to keep track of everyone's state */
semaphore mutex = 1; /* mutual exclusion for critical regions */
semaphore s[N]; /* one semaphore per philosopher */

void philosopher(int i) /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {
        think(); /* repeat forever */
        take_forks(i); /* philosopher is thinking */
        eat(); /* acquire two forks or block */
        put_forks(i); /* yum-yum, spaghetti */
        /* put both forks back on table */
    }
}
    
```

ניתן לראות בקוד שאנו מגדירים N סועדים, סועד ימני ושמאלי של כל אחד מהסועדים ו3 מצבים שבהם סועד יכול להיות (חושב – ממתין, רעב – ינסה לקחת מזלגות ואוכל). מגדירים מערך של N semaphore (אחד לכל סועד), mutex מרכזי ומערך של סועדים שמחזיק את המצב שבו נמצא כל סועד.

הפונקציה philosopher() מוגדרת כך: תחשוב (זמן המתנה מסוים), תנסה לקחת מזלגות, תאכל ותחזיר את המזלגות.



```

void take_forks(int i)
{
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}

void put_forks(i)
{
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}

void test(i) /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}

```

הפונקציה **take\_forks()** מוגדרת כך:  
 תנעל את הmutex תעדכן שאני במצב רעב,  
 תבדוק האם אני יכול לקחת מזלגות (**test()**),  
 תשחרר את הmutex ותבצע את **down()** את  
 semaphore שלי.

מתי semaphore שלי ננעל? הפונקציה  
**test()** בודקת שאני אכן במצב רעב  
 ושהמזלגות מימני ומשמאלי פנויים (אם  
 הסועדים מימני ומשמאלי לא במצב "אוכל"  
 אז המזלגות פנויים), במידה והם פנויים  
 המצב שלי מתעדכן ל"אוכל" ואני מבצע **up()**  
 semaphore שלי.

במידה והמזלגות לא פנויים לא יתבצע **up()**  
 semaphore וכתוצאה מכך הסועד יתקע  
 ב**down()**.

בפונקציה **put\_forks()** תחילה נועלים את

mutex, מעדכנים את המצב ל"חושב" כיוון שסיימתי לאכול, ושולחים לפונקציה **test()** את השכן הימני והשמאלי  
 שלנו – פה טמון הרעיון העיקרי של הקוד.

כיוון שהסועד שחרר את המזלגות שלו במידה ואחד השכנים שלו "תקוע" וממתין למזלגות הוא "יעיר" אותו כיוון  
 שהוא יעדכן אותו למצב "אוכל" ויבצע לו **up()** ולכן הוא יוכל לבצע **down()** בפונקציה **take\_forks()** ולהשתחרר.  
 בסוף הפונקציה **put\_forks()** נשחרר את mutex.

היתרון בפתרון זה הוא שהוא מונע starvation כיוון שתמיד מישהו יעיר אותך כשיתפנו המזלגות.  
 החסרון בפתרון זה הוא שבמצב של המון פילוסופים יהיו לנו המון התנגשויות והמון פילוסופים שיתקעו בגלל mutex  
 המרכזי (בזבזני מאוד).

### פתרון LR:

כאן לא נשתמש בmutex מרכזי ולכן פתרון זה הוא פחות בזבזני.

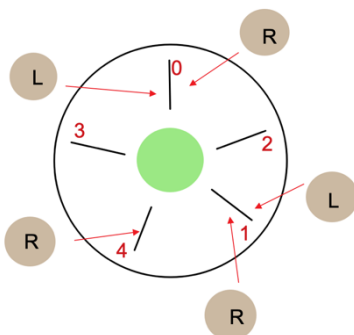
נגדיר שני סוגים של פילוסופים, ימניים ושמאליים.

סועד ימני ינסה לקחת קודם את המזלג שממימנו והשמאלי ינסה לקחת תחילה את  
 המזלג שמשמאלו.

נקצה לפילוסופים את היד (שמאלי או ימני) אחד אחרי השני לחילופין (אחד שמאלי,  
 אחד ימני וכן הלאה).

נמספר מזלג שרירותי להיות 0, ונמספר את השאר כמתואר בתמונה.

וכך כל פילוסוף לוקח מזלג בסדר עולה (או יורד).



אלגוריתם זה הוא גם starvation-free כיוון שאם פילוסוף A ממתין למשאב המוחזק ע"י פילוסוף B, אז כאשר B  
 ישחרר את המשאב הוא לא יוכל לתפוס אותו שוב לפני שA יקבל אותו.

ומכיוון שהאלגוריתם הוא deadlock-free, פילוסופים ימשיכו לקחת ולשחרר משאבים, בסופו של דבר B ישחרר את  
 המשאב A ויכול להשיג אותו.

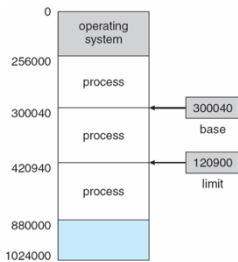
בנוסף אנו לא צריכים להניח שהמזלגות הם "הוגנים" (fair) כיוון שבכל זמן נתון רק פילוסוף אחד מחכה למשאב.

## זיכרון (memory) –

בעבר מתכנתים היו כוכבים תוכנה לכתובות פיזיות (physical addresses) וברגע שניסו להריץ את התוכנה במחשב אחר הכתובות זיכרון שם היו שונות ולכן התוכנה לא תצליח לרוץ כיוון שכתובות פיזיות תלויות בגודל הזיכרון (RAM size), סוג הזיכרון (RAM device) ובמעבד (CPU).

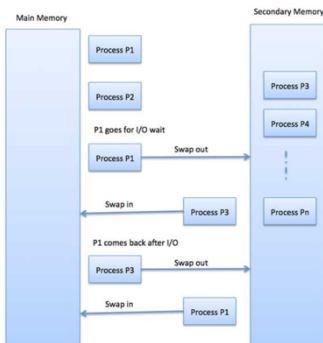
### פתרון 1: pre-process Base & Limit

צמד register בסיס ומגביל שמגדירים את שטח הכתובות עבור התהליך חסרונות:



1. נצטרך לדעת מראש את כמות הזכרון שהprocess צריך.
2. לנהל את כל processes במקביל בזכרון – לא ניתן להכיל את כל התהליכים בזיכרון.
3. המעבד צריך לבדוק כל ניסיון של התהליך לגשת לזיכרון כדי לוודא שהוא עומד במגבלה שלו.

### פתרון 2: swapping

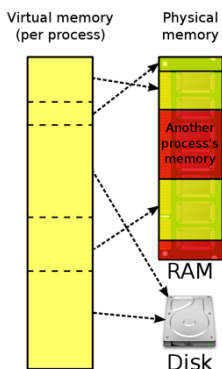


מנגנון שבו ניתן להחליף (להעביר) תהליך באופן זמני מהזיכרון הראשי לאחסון משני (דיסק) ולהפוך את הזיכרון לזמין לתהליכים אחרים. בשלב מאוחר יותר, המערכת מחליפה את התהליך מהאחסון המשני לזיכרון הראשי. אמנם הביצועים מושפעים בדרך כלל מתהליך ההחלפה אך הוא מסייע בהפעלת מספר רב של תהליכים גדולים במקביל וזו הסיבה שההחלפה ידועה גם כטכניקה לדחיסת זיכרון.

חסרונות:

1. הזמן שלוקח תהליך ההחלפה כולל את הזמן שלוקח להעביר את התהליך כולו לדיסק משני ואז להעתיק את התהליך חזרה לזיכרון - פעולה כבדה לתהליך שיש לו שימוש גדול בזיכרון.
2. לא ניתן לדעת מראש כמה זיכרון התהליך צריך.
3. במהלך הזמן, ההחלפה יוצרת "חורים" של כמויות קטנות של זיכרון זמין במפת הזיכרון.

### פתרון 3: Paging and Virtual Memory



לבסוף כדי לפתור את הבעיה הומצא מנגנון Hardware Independent Memory, המנגנון ממיר את הכתובות הפיזיות שבזיכרון לכתובות וירטואליות (addressing) וכך המתכנת עובד מול כתובות וירטואליות ולא פיזיות. כאשר:

$$\text{Virtual address} = \text{Physical address} + \text{Normalization offset}$$

היתרון המשמעותי הוא שנוכל לתמוך במרחב כתובות שאינו תלוי בזיכרון הפיזי (במחשב של 32 סיביות יש  $2^{32}$  כתובות במרחב הכתובות הווירטואלי).

### Page:

כל תהליך מקבל מהזכרון הראשי גוש זיכרון וירטואלי באורך קבוע, המתואר על ידי ערך יחיד בטבלת העמודים (page table). זוהי יחידת הנתונים הקטנה ביותר לניהול זיכרון במערכת הפעלה של זיכרון וירטואלי. באופן דומה, מסגרת עמודים (page frame) היא גוש הזיכרון הפיזי הצמוד ביותר באורך קבוע, אליו ממופה דף זיכרון על ידי מערכת ההפעלה.

גודל כל עמוד נקבע בדרך כלל על ידי ארכיטקטורת המעבד. באופן מסורתי, לדפים במערכת היה גודל אחיד, כגון 4,096 בתים (4KB).

מערכת עמודים בגדלים קטנים משתמשת ביותר עמודים, ודורשת טבלת עמודים שתופסת מקום רב יותר. לדוגמה, אם  $2^{32}$  כתובות וירטואליות ממופות לעמודים של 4KB ( $2^{12}$  bits), מספר העמודים הווירטואליים הוא  $2^{32} / 2^{12} = 2^{20}$ . עם זאת, אם גודל עמוד גדל ל-32KB ( $2^{15}$  bits) נדרש רק ל- $2^{17}$  עמודים.

### :paging

מערכת ההפעלה של המחשב, באמצעות שילוב של חומרה ותוכנה, ממפה את כתובות הזיכרון המשמשות תוכנית, הנקראות כתובות וירטואליות, לכתובות פיזיות בזיכרון המחשב. אחסון ראשי, כפי שנראה על ידי תהליך או משימה, מופיע כמרחב כתובות רציף או כאוסף של פלחים רציפים. מערכת ההפעלה מנהלת רווחי כתובות וירטואליים והקצאת זיכרון אמיתי (פיסי) לזיכרון וירטואלי.



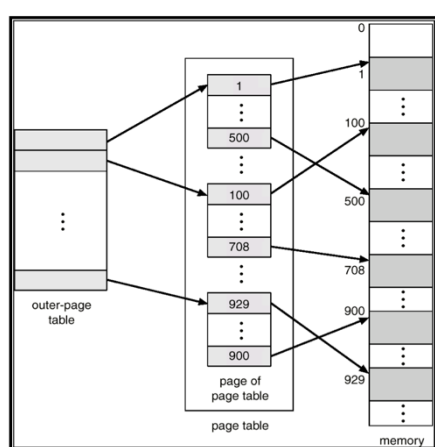
## Page Table:

טבלת עמודים היא מבנה נתונים המשמש מערכת זיכרון וירטואלית לאחסון המיפוי בין כתובות וירטואליות לכתובות פיזיות. טבלת העמודים היא מרכיב מרכזי בתרגום כתובות וירטואליות אשר הכרחי לטובת גישה לנתונים בזיכרון הראשי.

כיצד נוכל להתמודד עם טבלאות גדולות מדי? ישנם מספר תוכנות ומרכיבים שצריכים להתקיים מבלי קשר לגודל הטבלה. פתרון קיצוני אחד הוא שכל טבלאות המיפוי יהיו בחומרה, זה לא אפשרי מכיוון שאמנם הגישה מאוד מהירה אבל זה יקר מאוד, במיוחד עבור טבלאות גדולות, פתרון קיצוני אחר הוא להחזיק הכל בזיכרון הראשי, זה יהיה מנגנון עם המון מצביעים registers וזה גם יקר מאוד כי אז אנחנו מכפילים כל reference לזיכרון. ולכן נבצע paging על page table עצמו.

שיקולים לגבי טבלת העמודים:

- יכולה להיות גדולה מאוד – מליון דפים עבור 32bit בגודל 4KB.
- חייבת להיות מאוד מהירה.
- כדי להימנע מלשמור טבלאות דפים שלמות בזיכרון - נהפוך אותן לרב-שכבתיות, ונימנע מלהפנות לזיכרון הראשי בכל הוראה על ידי שמירה במטמון (cache).



Two Level Paging

במקום להחזיק מערך רציף עם המון כניסות (entrees) של כל הכתובות הוירטואליות נחזיק outer-page table שיצביע למערכים של page table וכך לא כל החלקים שבטבלה החיצונית יהיו מאוכלסים אלא רק החלקים שבתפוסה. כתובת לוגית במכונת 32bit עם 4K גודל עמוד מחולקת ל: מספר העמוד המורכב מ-20bit, offset המורכב מ-12bit. ומכיוון שטבלת העמודים מדורגת (טבלה חיצונית ופנימית) גם מספר העמוד מחולק לאינדקס  $p_1$  של 10bit עבור הטבלה החיצונית ואינדקס  $p_2$  של 10bit עבור הטבלה הפנימית.

Two-level paging עוזרת כיוון שרוב התהליכים אינם צריכים את כל שטח הזיכרון הוירטואלי שהוקצה להם.

למשל תהליך במכונת 32bit משתמש ב-4MB עבור מחסנית, 4MB עבור ערימה ו-4MB עבור code segment (1000 דפים בגודל 4KB עבור כל אחד).

סה"כ 12MB בשימוש מתוך 4GB – רק 3 טבלאות דפים פנימיות עבור כל אחד מהם וטבלה אחת חיצונית נצרכים במקרה זה (ייתכן ולמשל code segment לא יופיע באופן רציף בזיכרון ואז נצטרך יותר טבלאות פנימיות).

## טבלאות דפים הפוכות Inverted page tables –

Page table רגיל לא פרקטי עבור מכונות 64bit:

$$4K \text{ page size} / 2^{52} \text{ pages} \times 8 \text{ bytes} \rightarrow 30M \text{ GB page tables!}$$

טבלת דפים הפוכה מאוכסנת לפי physical page frame ולא לפי virtual pages:

$$1 \text{ GB of RAM} \& 4K \text{ page size} / 256K \text{ entries} \rightarrow 2 \text{ MB table}$$

טבלת עמודים הפוכה אחת משמשת לכל התהליכים הנמצאים כעת בזיכרון.

כל כניסה (entry) מאחסנת אילו תהליכים / עמודים וירטואליים ממפים אליו.

משתמשים בhash table כדי למנוע חיפוש לינארי לכל דף וירטואלי

בנוסף לhash table, רישומי TLB משמשים לאחסון ערכי טבלת עמודים שהשתמשו בהם לאחרונה (cache).

הערה: אם הטבלה משותפת לכל התהליכים, ייתכן מצב ששני תהליכים ממפים בטבלה לאותו קטע קוד (רצים על אותה תוכנית למשל), כדי שכל תהליך מיפוי יידע מי מבצע את השאילתה הזאת, אנחנו נגיע לטבלה עם שני משתנים: מזהה תהליך (pid) וכתובת וירטואלית (virtual address).

- 64-bit computer
- Size of physical memory: 4GB
- Size of page: 4KB
- **How many pages are possible?**
- Each entry in the page table contains
  - Location of the required page frame in the physical memory (if presents)
  - 6-bits ctrl info (Valid, dirty, referenced etc).
- **What's the size of the page table?**

פתרון:

# of Page table entries:  $2^{(64 - 12)} = 2^{52}$ # of pages in the memory:  $4\text{GB} / 4\text{KB} = 1\text{M} \rightarrow$  addr. Of page in mem. Requires 20bits.Total size:  $(2^{52}) * 26$ 

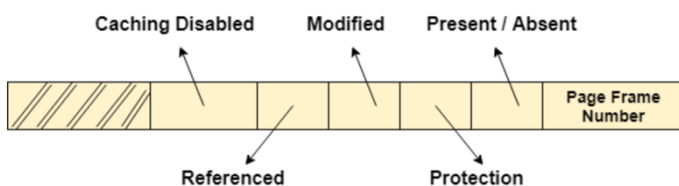
Assuming that only 1TB can code valid addresses:

We've "only"  $2^{(40-12)}$  optional pages.Still have  $26 * 2^{(40-12)} = 26 * 2^{28} \approx 32 * 2^{28} = 2^{33} = 8\text{Gbit} = 1\text{GB}$ .

Same order of magnitude as the main mem itself!

**Page Table Entry – כוללת**

- Page frame number (physical address)
- Present/absent (valid) bit
- Dirty (modified) bit
- Referenced (accessed) bit
- Protection
- Caching disable/enable

**– Memory Management Unit (MMU)**

יחידת חומרת מחשב שיש בה את כל ההפניות (reference) לזיכרון והן מועברות בעצמה, ומבצעת בעיקר תרגום כתובות זיכרון וירטואלי לכתובות פיזיות.

**– Translation Lookaside Buffer (TLB)**

מאגר מבט לתרגום (TLB) הוא מטמון זיכרון המשמש להפחתת הזמן שלוקח לגשת למיקום זיכרון של המשתמש. זוהי חלק מיחידת ניהול הזיכרון (MMU), TLB מאחסן את התרגומים האחרונים של זיכרון וירטואלי לזיכרון פיזי למעשה אפשר לקרוא לו מטמון לתרגום כתובות.

תהליך הresolving:

1. תהליך/kernel צריך לתרגם כתובת וירטואלית.

2. השאילה מגיעה לMMU/TLB, במידה והוא קיים אצלו הוא מחזיר לתהליך את הכתובת המבוקשת.

3. במידה והכתובת לא קיימת בTLB:

3.1. אם הכתובת חוקית, כלומר קיימת בPage Table:

3.1.1. נבצע PT Walk נמצא את הכתובת ונעדכן את הTLB.

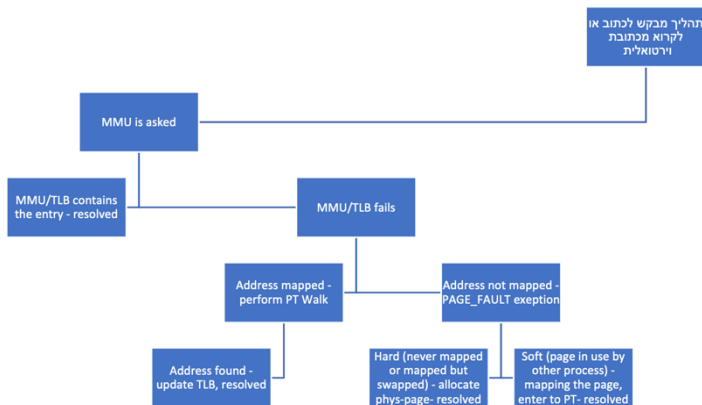
3.2. אם הכתובת כלל לא ממופה תיזרק שגיאת PAGE\_FAULT.

3.2.1. במידה וזו שגיאת Soft: זה אומר שהכתובת קיימת אך ממופה לתהליך אחר, נמפה את הדף עבור

התהליך הנוכחי ונעלה ב1 את הusage counter.

3.2.2. במידה וזו שגיאת Hard: זה אומר שהכתובת כלל לא קיימת, נבצע אלוקציה, נמפה את הדף ונכניס

אותו לpage table.



הערה חשובה: לפעמים נרצה ליצור עותק של TLB באופן לוקאלי (לאחר עדכון TLB), הסיבה שיוצרים עותק לוקאלי ברמת ה process היא שכאשר מתבצע context switch וה TLB מבצע flush ל cache הנוכחי וה process החדש שנכנס מחליף את ה TLB שב MMU ל TLB שלו וכך בעצם לא צריך למפות מחדש כתובות שהיו בשימוש.

#### שאלה:

- Consider a 32-b system
- Each process has its own page table
- Per-process page table size: 1 page
- Page Table Entry size: 1 Word (32-b)
- **A)** What should be the page size?
- **B)** How much data should be copied from the memory for initializing a process with 1KB mem?
- **C)** What's the maximal memory a process may consume before a page fault occurs?

#### פתרון:

A) Denote the page's size by x Bytes. The number of entries in x is  $x/4$ , because each entry takes 4 B.

We would like this single page to map all possible pages in the system.

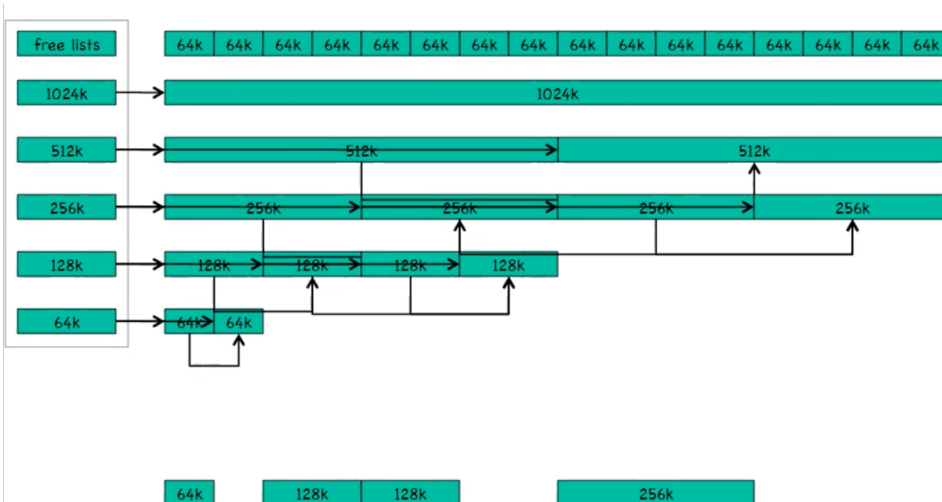
The number of pages in the system is  $2^{32}/x$ .

Therefore we have  $2^{32}/x = x/4 \rightarrow x = 2^{17}$ .

B) A process of 1KB requires 1 data page + 1 page table page  $\rightarrow 2x = 2^{18}$ B.

C) Assuming that upon starting the process only its single page is copied to the memory (no pre-fetching of additional pages), the process may consume only  $x=2^{17}$  B. Above it, a page fault occurs.

מה קורה כאשר קוראים ל malloc()? מתבצעת פנייה לספרייה שנקראת GLIBC (זו לא ספרייה ב kernel), במידה ויש לספרייה זיכרון היא תחזיר אותו, אחרת, היא מבקשת מהגרעין בצורה הבאה: היא קוראת לפונקציה mmap2() שהיא System Call והיא מחזירה את הגודל שהוקצה ביחידות של page. לאחר מכן מתבצעת לוגיקה של Knuth's Buddy Allocator כדי להקצות מתוך הדפים כתובות נצרכות. GLIBC מחזיר רק את כמות הכתובות שנדרשו ע"י malloc(), ושומר את היתר עבור הקריאות הבאות.



#### – Knuth's Buddy Allocator

טכניקת הקצאת זיכרון של Buddy היא אלגוריתם של הקצאת זיכרון המחלק את הזיכרון למחיצות כדי לנסות לספק בקשת זיכרון בצורה המתאימה ביותר. מערכת זו עושה שימוש בפיצול הזיכרון לחצאים כדי לנסות לתת חלק זיכרון בצורה הטובה ביותר.