

## סיכום מערכות הפעלה:

**נהל אתחול (Boot loader)** - היא תוכנה המופעלת כחלק מתהליך האתחול של מחשב וטוענת את מערכת הפעלה. התוכנה שמורה בזיכרון ההתקן והיא פשוטה וקטנה בהרבה מערכות הפעלה, וכן ניתן לטוען אותה בקלות רבה יותר. ישנו שמי מנהלי אתחול הפעלים זה לאחר זה. מנהל האתחול הראשון צרוב כקובץ בזיכרון ROM, והוא פונה לזכרון אחסון (כגון מחיצה בדיסק הקשיח, החסן נייד) בו קיים מנהל אתחול תוכנתי נוסף הממשיר את התהליך.

**זיכרון גישה אקראית (RAM)** -ראשי תיבות של Random Access Memory הוא שם כללי למספר רב של סוגים זיכרון מחשב, המתאפשרים ככל יכולת המעבד לגשת ישירות לכל תא בזכרון לפי כתובתו, לכתוב בו ולקרוא ממנו. ההתייחסות הנפוצה לזכרון מחשב היא למעשה התיחסות לזכרון הגישה האקראית הראשי שלו. סוג הזכרון הנמצא בשימוש נפוץ ביותר בימינו הוא DRAM. זיכרון מסווג זה הוא זיכרון נדיף, (volatile) כלומר מאבד את תוכנו עם ניתוק הזיכרון ממקור האנרגיה שלו. בנוסף זיכרון מסווג זה הוא זיכרון נדיף, (volatile) כלומר מאבד את תוכנו עם ניתוק הזיכרון ממקור האנרגיה שלו. בנוסף זיכרון מסווג זה הוא זיכרון נדיף, (volatile) כלומר מאבד את תוכנו עם ניתוק הזיכרון ממקור האנרגיה שלו. בנוסף זיכרון מסווג זה הוא זיכרון נדיף, (volatile) כלומר מאבד את תוכנו עם ניתוק הזיכרון ממקור האנרגיה שלו. בנוסף זיכרון מסווג זה הוא זיכרון נדיף, (volatile) כלומר מאבד את תוכנו עם ניתוק הזיכרון ממקור האנרגיה שלו. בנוסף זיכרון מסווג זה הוא זיכרון נדיף, (volatile) כלומר מאבד את תוכנו עם ניתוק הזיכרון ממקור האנרגיה שלו.

**זיכרון לקריאה בלבד ROM** – ראשי תיבות של Read-only Memory, הוא אמצעי לאחסן נתונים, דמוי זיכרון מחשב, המכיל נתונים הנכתבים בו פעם אחת, ונקראים ממנו פעמים רבות. שבבים אלה אינם ניתנים לכתיבה חוזרת (בצורה פשוטה), ומכאן השם קריאה-בלבד.

בניגוד לRAM, תוכן ROM נשמר גם לאחר ניתוק מקור מתח. הגישה לROM פשוטה יותר ומהירה בהרבה מאשר גישה לאמצעי אחסון מגנטיים ואופטיים כדוגמת דיסק קשיח או תקליטור.

בעבר שימש ROM לאחסן מערכת הפעלה כולה במחשבים אישיים. בהמשך נותרו לROM מספר תפקידים: עיקריים:

- תוכנת אתחול (Boot) תוכנה שמרתה לבצע את הפעולות הראשונות בזמן הדלקת המחשב, כולל בדיקת תקינות רכיבי המחשב וטעינת מערכת הפעלה לזכרון.
- BIOS - אוסף רוטינות אשר תומכות בפעולות הקלט והפלט הבסיסיות של המחשב.
- צריבה של תוכנה קבועה על רכיבים המיועדים לביצוע פעולות מוגדרות מראש, כגון מנגנון משולב תליין ישום.

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

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

בהתאם לארכיטקטורת פון גוימן הפקודות אותן מקבל המעבד מאפשרות לו קראת מידע מהזיכרון או מהתקנים השונים, ביצוע פעולות חישוביות ולוגיות על מידע זה וכטיבת תוצאות החישוב בזיכרון לזכרון או להלופין שליחתו להתקנים חיצוניים. הפעולות הלוגיות מאפשרות בקרת זרימה וחזרה על פקודות כל שנדרש. הפקודות הן בסיסיות ביולוגיות, כל אחת, מרצף קצר של ביטים. רצף זה קורי שפט מכונה. כל דגם של מעבד מתאפיין בסט פקודות משלו. בהתאם לתזות צ'רץ'-טירינג דלות שפט המכונה אינה מהויה מוגבלת בביבוצע תוכנית מחשב כלשהו וההבדל בין מעבדים שונים יבוא לידי ביטוי רק בביבוצים.

**אגרים או רגיסטרים (Registers)** – כל אוגר הוא יחידה אחת של זיכרון פנימי מהיר ביותר הנמצא לרוב בתוך הCPU 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 (בדרכם כלל מסוג RISC) בהם לכל האוגרים פונקציונליות זהה. מספר האוגרים במחשב נמוך יחסית וכל היותר יגיע לעשרות מעותות.

**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 מטמוןות נפרדיים לקוד ולנתונים. נהוג לסמן את זיכרון המטמון הבניי על שבב המעבד לשולשה סוגים. יש מעבדים ללא מטמון, או עם חלק אך לא כל סוג המטמון המתוארים:

- L0 - המטמון מהיר ביותר, אך גם המוגבל ביותר בגודלו. בדרך כלל פועל בקצב השעון של המעבד עצמו.
- L1 - מטמון פחות מהיר וגודלו יותר מ-L0

- L2 - מטמון מהיר פחות וגודל יותר מ-1. לשבבים רבים מעבדים, לעיתים מטמון L2 משותף למספר מעבדים.

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

מערכת הפעלה מספקת שלושה ממשקים:

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

ניתן למנות שלושה תפקידים עיקריים של מערכת הפעלה:

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

### – Extended Machine

המכונה מספקת:

- יציבות: לא מתرسקת.
- ניידות: יכולה להריץ קוד בויתר סוג אחד של מכונות.
- אמינות: תמיד מגיבה באופן אופן.
- בטיחות: לא עשו משהו מסוכן.
- התנהגות טובה: פועלת בסביבה נאותה.

### Resource Manager – תומך במכשירים רבים בו זמנית

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

משתף משאבי בין משתמשים ותוכניות באופן הוגן: כל תוכנית זוכה לשינוי להפעלה, בבטחה: בגין מפני שחריות, ביעילות: שימוש במשאים הזמינים כדי לספק את השירות הטוב ביותר.

מסוגל להקצות משאבי למשתמשים כגון דיסקים, זיכרון, ממשקי רשת, טימרים, מסופים / צגים, מדפסות לייזר וכו'

**פסקה(Interrupt)** - היא אות המגיע למעבד, מצד רכיב חומרה או תוכנה ומורה להפסיק תהליך מסוים בכך לעבד בתהליך אחר הדורש טיפול יוטר (לאחר מכן המחשב יחזיר לעבד את התהליך שהפסיק).

בעת קבלת הפסקה המשזה המחשב את ביצוע הסדרתי של התוכנית, כדי להפעיל שגרת טיפול בפסקה. לאחר הטיפול בפסקה, ממשיך המחשב ביצוע הסדרתי של התוכנית. הדבר דומה לאדם המבצע מלאכה כלשהי, ומפסיק כדי לענות לשיחת טלפון, ולאחר סיום השיחה, ממשיך במלאתו מהנקודה שהפסיקה.

פסקות תוכנה נמצאות בשימוש נרחב במחשבים הפעילים בריבוי מישימות ומהוות הפסקת פעולה תוכנית מחשב שלא דרךZRימה.

**Interrupt handler** - מטפל בפסקה, הידוע גם בשם ISR, הוא גוש קוד מיוחד המשויך למצב הפרעה ספציפי. Interrupt handlers יזמים על ידי הפרעות חומרה, הוראות להפסקת תוכנה או חריגות תוכנה ומשמשות לישום system call או מעברים בין מצבים פועלה מוגנים, כגון device drivers.

**קריאה מערכת (system call)** - היא בקשה שמבצעת תוכנת מחשב מליבת מערכת הפעלה (מהאלן) כדי לבצע פעולה שהיא יכולה לבצע עצמה. קריאות מערכת הן האחריות על החיבור שבין המaesystem ליבת המערכת, ובכך מאבזרת את המשתמש ונונ坦ת לו שימוש מרבי בפונקציונליות שהיא מציעה. הדבר כולל בין היתר יכולת לקבל גישה לרכיבי חומרה (למשל קריאת קובץ מהדיסק הקשיח), ליצור תהליך חדש, להעברת מידע בין תהליכיים ועוד.

**מנהל התקן (driver)** - תוכנית מחשב המאפשרת לתוכנית מחשב אחרת, לתקשר עם חומרה כלשהי או עם תוכנה אחרת הפעלת בפורמט שונה באמצעות ממשק שלה ומטען API לעבודה מולו.

**ממשק תכנות יישומים (API)** – ראשי תיבות Application Programming Interface, הוא ערכה של ספריות קוד, פקודות, פונקציות ופרצדרות מן המוקן, בהן יכולם המתכוונים לשות שימוש פשוט, בלי להידרש לכתוב אותן בעצמן כדי שיוכלו להשתמש במידע של היישום שממנו הם רוצים להשתמש לטובת היישום שלהם.

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

**llibה (Kernel)** – היא הרכיב המרכזי של מרבית מערכות הפעלה. זהו הגשר שבין תוכניות המחשב המקורי המשמש רק את ליבת המערכת עצמה שמצוע ברמת החומרה. אחד התפקידים העיקריים של הליבה הוא ניהול משאבי המערכת (התקשרות שבין רכיבי החומרה והתוכנה). בדרך כלל, בתוך הרכיב הבסיסי של מערכת הפעלה, הליבה יכולה לספק את שכבת האבטרנטזיה בrama הנומוכה ביותר עבור המשאבים (ביחוד עבור מעבדים והתקני קלט-פלט) שהתוכניות צריכות לשלוט עליהם על מנת לבצע את תפקידן.

בדרכן כל הליבה הופכת שירותים אלה לזרים עבור תהליכיים של תוכניות באמצעות מנגנוני תקשורת בין תהליכיים (process communication) וקריאות מערכת. בעוד המשימות של מערכת הפעלה מבוצעות באופן שונה על ידי ליבות שונות, בהתאם לעיצוב ולIMPLEMENTATION שליבות מונוליתיות (Monolithic Kernels) מ Ritchot את כל הקוד של מערכת הפעלה באותו מרחב כתובות על מנת לשפר את ביצוע המערכת, מיקרו-ליבות (Microkernels) מ Ritchot את מרבית השירותים מערכת הפעלה במרחב המשמש כשרתיים, במטרה לשפר את האמינות והתחזוקתיות של מערכת הפעלה. קיימן מגוון של אפשרויות בין שתי הדוגמאות הקיצניות האלה.

ליבת מערכת הפעלה היא התוכנית היחידה אשר מוכנה לrintה בכל זמן שהוא. בנוסף, הליבה היא התוכנה היחידה שיכולה לבצע אוסף פקדות בקרה אוטומטית (לא שום הפרעה או הפסקה מוגרים כלשהו). כל מערכת הפעלה חייבת ליבת כדי לפעול, אך הליבה אינה בהכרח ייחודית למערכת הפעלה - מערכות הפעלה שונות יכולות להשתמש בליבה זהה. לדוגמה, על ליבת לינוקס ניתן להריץ הפצת לנוקס שולחנית אך גם אפשר להריץ את מערכת הפעלה אנדרואיד למכשירים ניידים.

**תהליכיים (Thread)** – נקרא גם חוט, מושג במדעי המחשב המשמש במערכות הפעלה כדי לתאר הקשר ריצה במרחב כתובות.

מערכות הפעלה מודרניות מאפשרות לנצל במסגרת ריצה של תהליך (Process) מספר תהליכיים במקביל במרחב כתובות אחד. במערכות אלו כל תהליך חדש מתחילה את ביצועו באמצעות 'תהליך ראשי' אשר עשוי בהמשך ליזור תהליכיים נוספים. מגנון הריצה באמצעות תהליכיים מאפשר לספק למשתמש במערכת הפעלה מהירות גבוהה וריציפות פעולה כאשר התהליך (Process) מבצע כמה משימות במקביל.

למה צריך threads ?

תהליך אחד יהיה איטי מדי ושני תהליכיים לא יכולים לגשת לאותו מקום בזיכרון.

יתרונות:

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

חרוגונות:

- אבטחה ויציבות: קבצים פתוחים, מבני נתונים, משתנים גלובליים, תהליכי ילדים וכו' משותפים בין כל התהליכיים הנמצאים תחת אותו process.
- איתות(alarm) לתהליך מסוים משפיע על כל התהליכיים באותו process זה.

**תהליך (Process)** – מופע של תוכנית מחשב שמופעל על ידי מערכת מחשב שיש לה יכולת להפעיל מספר תהליכיים בו זמן.

תוכנית מחשב היא בעצם רק אוסף פקודות, בעוד שתהליך הוא הפעלה של אותן פקודות. כך למשל, הפעלה של מספר מופעים של אותה תוכנה יגרום לעיתים קרובות למספר תהליכיים של התוכנה להיפתח בו זמן. כדי לאפשר למשתמש להפעיל מספר תהליכיים במקביל, מתבצעת על ידי סדרן התוכניות (Scheduler) חלוקת זמן מעבד בין התהליכים.

בסיום ריצתו, התהיליך מוחזיר ערך יציאה (Exit status) שמאפשר לתהיליך האב שיצר אותו לקבל מידע על סיום מוצלח של התהיליך או לחלוון ערך שגיאה המעיד על סוג היכישלון שמנע מההתהיליך להסתיים בהצלחה. כל תהיליך מכיל מרחב זיכרון וירטואלי, תהיליכון אחד לפחות, table (מפת page) – שומר את כל הקצאות הזיכרון), מחסנית, ערימה, data של התוכנה שהפעילה את התהיליך, File/Socket, Signals, הרשות, גלובליים, ועוד.

תהליך שיש לו את פעלותו אף עדין קיים בטבלת התהיליכים כדי שייתאפשר לתהיליך האב לקרוא את ערך היציאה שלו מכונה תהיליך זומבי וההתהיליך האב אוסף אותו בעדרת פקודות (wait).

בהתחלת התהיליך "נוצר" (create) על ידי טעינתו מהתקן אחסון מסוים (כונן דיסק קשיח) לזכרון הראשי. בכל רגע נתנו התהיליך נמצא באחד משולשת המצבים הבאים:

1. Running - פועל (למעשה משתמש במעבד באותו רגע).

2. Ready - מוכן (ניתן להפעיל; הופסק זמן נט欢喜 כדי להפעיל התהיליך אחר).

3. Blocked - חסום (לא מסוגל לרוץ עד שמתרכש אירוע חיצוני כלשהו).

ולבסוף ברגע שההתהיליך מסיים את הביצוע, או שמסתois על ידי מערכת ההפעלה, אין בו עוד צורך. התהיליך מסור באופן מיידי או מעבר במצב "הסתיים" (terminated). כשהוא מסור, הוא רק ממתיhn להסרתו מהזיכרון הראשי.

Process	Thread
Data "יחודי"	Data משותף
Code "יחודי"	Code משותף
O/I "יחודי"	O/I משותף
טבלת signals "יחודית"	טבלת signals משותף
מחסנית ייחודית עבור כל threads של process*	מחסנית ייחודית בהשאלה מהמחסנית של thread
PC "יחודי"	PC "
Register "יחודי"	Register "
מצב/סטטוס "יחודי"	מצב/סטטוס "
Context switch כבד	Context switch קל
ערימה	(thread safe segment & thread safe storage) TLS/TSS
משתנה גלובלי (environment variable)	CPU "להדביק" את thread ללילה/סט ליבות מסוים.
Command line	Thread priority
Page table	
TLB	

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

לסייע: thread הוא לרכיב, וprocess הוא מעתפת משאביים שכוללת גם יכולות ריצה בעזרת אחד או יותר threads.

### שירות מקבילי (Multiprogramming)

1. מושפר 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. מושפר משימות אינטראקטיביות.

**ניהול זיכרון (Memory management)** – הפונקציה האחראית על ניהול הזיכרון הראשי של המחשב. פונקציית ניהול הזיכרון עוסקת אחר הסטטוס של כל מקום זיכרון, שהוקצה או חינמי.

הוא קובע כיצד מוקצת זיכרון בין התהיליכים המתחרים, מחליט מי מקבל זיכרון, متى הם מקבלים אותו וכמה הם רשאים.

כאשר מוקצת זיכרון הוא קובע אליו מיקומי זיכרון יוקזו. הוא עוקב אחרי הזיכרון שהשתחרר או שאינו מוקצת וمعدכן את הסטטוס שלו.

זה שונה מapplication memory management (ניהול זיכרון יישומים) שבו איך התהיליך מנהל את הזיכרון שהוקצת לו על ידי מערכת הפעלה.

**החלפת הקשר (context switch)** – מעבר בין הריצת שני תהיליכים באמצעות החלפת הקשר, מספר תהיליכים יכולים לחלק את אותו מעבד. החלפת הקשר שומרת את מצב האוגרים במעבד בזכרון המחשב, ולאחר מכן מכינסה למעבד את נתוני ריצת תהיליך אחר, לאחר מכן, מוחזירה את נתוני הריצת של התהיליך הקודם הקודם אל המעבד וממשיכה את ריצתו מאותה הנקודה בה הפסיק. החלפת הקשר היא כמעט במחשבים הפעילים בריבוי

משימות הרצות על מעבד יחיד. תהליך החלפת הקשר נדרש לתהילך בזבזני מבחינת משאבי מערכת ועל כן מערכות הפעלה מנסות לבצע אופטימיזציה בשימוש בהן.

קיימות שלושה תרחישים בהם יש צורך בהחלפת הקשר:

- במחשב הפועל בריבוי משימות, קיימtzman (scheduler) הקובע את סדר הרצת התוכניות ומקצה "זמן מעבד" לכל תהילך. כ"ז זמן מעבד" של תהילך אחד מסתיים, מופעלת פסיקה (interrupt) שתגרור החלפת הקשר.

• ארכיטקטורות מסוימות (למשל ארQUITECTURES 68x של אינטל) הן מוגנות-פסיקה. משמעות הדבר היא שבמדייה והמעבד צריך לבקש קריאה מהדיסק, הוא ישלח את בקשה לקריאה וייעבור לבצע פעולה 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) למצבי ליבה (kernel mode), תלוי במערכת הפעלה, תיתכן החלפת הקשר.

שלבי העבודה של context switch:

1. אמור למעבד להפסיק להפעיל את המשימה וסמן את המשימה כנעכירה.

2. שומר registers PC(program counter), SP(stack pointer), FP(frame pointer) נוספים של המשימה.

3. טען רישומי PC, SP, FP registers PC, SP, FP מהמשימה המתקרבת.

4. אופציונלי: בצע flash בTLB בunit (caches).

5. עדכן את המשימה החדשה כפועלת.

6. התחל לבצע את התהילך החדש.

**הערה:**threads במרחב הגרעין (kernel) scheduler של המערכת הפעלה אחראי על threads,context switch במרחב הusers scheduler שהמשמש מימוש או ספריה(Cluster) אחראית על context switch. ובאזוריםThreads במרחב הusers scheduler למשתמש פותח thread דואגת עבורו לthread ברמת הגרעין.

### User lever threads – מושם בספריות ברמת המשתמש.

החלפת תהילכנים אינה צריכה לתקשר עם מערכת הפעלה או לגרום להפרעות.

השימוש של המשתמש מזמן המעבד של התהילך לתהילכנים הפנימיים שלו.

משתמש רק במעבד יחיד, שכן מערכת הפעלה לא תקצה מספר מעבדים לתהילך אחד.

הגרעין אינו יודע דבר על threads ברמת המשתמש ומנהל אותם כמו תהילך thread בעל thread בודד.

(בניהול ע"י התהילך, התהילכנים שוקפים למערכת הפעלה ו-system call מאחד התהילכנים יקפיא את כל התהילך מכיוון שלמערכת הפעלה אין היקרות עימם אלא רק עם התהילך עצמו והוא מקפיאה אותו, עבודה רק על מעבד יחיד כיוון שהמעבד רואה ישות אחת, ולא הגיוני שיקפה לה שני מעבדים).

יתרונות:

• תאימות: ניתן להטמע במערכת הפעלה שאינה תומכתthreads.

• ייצוג פשוט: כל thread מיוצג בפשטות על ידי PC, registers, מחסנית ו-block בקרה קטן, הכל מאוחסן במרחב הכתובות של התהילך המשתמש.

• ניהול פשוט ומהיר: יצירת thread, סנכרון ומעבר בין threads יכולם להיעשות ללא התערבות של הגרעין, וכן הם זולים יותר ומהירים פי 100 בהשוואה לשרשורים ברמת הגרעין.

חסרונות:

• חוסר תיאום בין thread לבין מערכת הפעלה. לכן, התהילך בכללותו מקבל פרוסה חד פעמיות של זכרון ללא קשר לשאלה האם לתהילך יש thread אחד או threads 1000.

• קריית מערכתthreads (system call) באחד מה們הדים גורמת למערכת הפעלה לחסום את כל התהילך, גם אם נותרו תהילכנים ניתנים להפעלה בתהילכנים אלה.

• חוסר היכולת של הגרעין לבדוק בין תהילכנים ברמת המשתמש מקשה על תכנון מזמן (scheduler) בין תהילכנים מסוימים התהילך.

### Kernel lever threads – Kernel lever threads

כל התהילכנים גלויים ללביה.

הגרעין מנהל ומזמן את התהילכנים.

קיימות קריאות מערכת ליצירה וניהול תהילכנים.

יתרונות:

• הלביה יכולה לזמן בצורה חכמה בין תהילכנים עם מספר תהילכנים שונה.

• תהילכנים ברמת הלביה טוביים במיוחד עבור יישומים החסומים(blocks) לעתים קרובות.

• במעבד מרובה ליבות, כמה מעבדים יכולים להריץ בו זמני תהילכנים שונים מאותו התהילך.

- יצירה, ניהול והחלפת תהליכיונים הם הרבה יותר יקרים ואיטיים מאשר בرمת המשמש.

**Fork** – קריית מערכת שבה תהליך יוצר עותק של עצמו. (*Fork* היא השיטה העיקרית לייצור תהליכיים במערכות הפעלה דומות לIONIX).

מزلג () יוצר תהליך חדש על ידי שכפול הпроцесс שקרה לפונקציה.

התהליך החדש מכונה תהליך הילד, תהליך ההתקשרות מכונה תהליך האב. תהליך הילד ותהליך ההורה מתנהלים מרחבם זיכרון נפרדים.

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

כתיבת זיכרון או מיפוי קבצים המבוצע על ידי אחד התהליכיים אינם משפיעים על השני.

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

- ליד יש מזהה תהליך ייחודי משלו (id), והוא PID אינו תואם לאף זו של קבוצת תהליכיים קיימת.
- מזהה התהליך של ההורה של הילד זהה לזהות התהליך של ההורה.
- הילד לא יורש את מנעולי הזיכרון (memory lock) של ההורה.

(CPU time counters) (Process resource utilizations) ומוני זמן המעבד (Processor utilization) מאותחלים לפחות אצל הילד.

• קבוצת Signals של הילד ריקה.

• הילד לא יורש התאמות סטטוס (semaphore adjustments) מההוראה שלו.

ערך החזרה: במקרה, ה- PID של תהליך הילד מוחזר אצל ההוראה, ו- 0 מוחזר אצל ההוראה, לא נוצר תהליך של ילד, ו- errno מוגדר לציין את השגיאה.

הערה: לעיתים לא נצטרך שהבן יקבל עותק של כל תכולת האב, במקרה זה נוכל להשתמש בWCO.

שאלה: האם הfork משכפל רק את thread שקרה לו או את כל threads בprocess? תשובה: תלוי במערכת הפעלה – UNIXים רבים תומכים בשני סוגי הfork, באסן Linus רק הthead שקרה לפונקציה משוכפל ולעתים זה גורם לביעות.

**Wait** - קריית מערכת שמתינה לשינוי בתהליך בן.

קריאות מערכת הללו משמשות להמתין לשינויים במצב אצלILD ולקבלת מידע על הילד שמצוותו השתנה.

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

**Exec** - משפחת הפונקציות exec () מחליף את תמונהו של התהליך (process image) הנוכחי בתמונהו של התהליך החדש.

```
int execv(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 בתכונות.

Copy-on-write נובעת מההנחה שמספר שימושות שונות יכולות להשתמש באותו זיהום של מידע, כלומר אין צורך לבצע עותק של המידע לכל התהליך.

במוקם זה, ניתן להציג על אותו המשאב מכל התהליכיים הדורשים עותק זה.

כאשר יש מספר תהליכיים המשתמשים באותו המשאב מקבלים ניצול טוב יותר של משאבי. כאשר עותק מקומי עובר שינוי, פרדיגמת COW לא מתחייבת כי המשאב המשותף לא השתנה ביןתיים על ידי תוכנית אחרת. לכן שיטה זו נוחה מאוד אם רק העדכן האחרון חשוב.

דוגמא ל-COW-במערכת Unix: כאשר קריית המערכת fork מופעלת, עותק של כל הדפים המתואימים לתהליך

האב נוצרים ונטענים לאזור זיכרון נפרד על ידי מערכת הפעלה לטובת התהליך הבן. פעולה זו אינה נדרשת במקרה מסוימים. למשל, במקרה שבוצע דרך התהליך הבן את קריית המערכת exec או בזמן יציאה לאחר זמן קצר מרגע ה fork.

כאשר התהליך הבן צריך להריץ פקודה לטובת התהליך האב, אין צורך להעתיק את דפי התהליך האב, זאת מפני ש- exec מחליף את מרחב הכתובות של התהליך.

במקרים כאלו, טכניקת COW בא להידי ביטוי. בעדרת טכנית זו, כאשר מתרחש ה-fork, דפי התהליך האב לא מועתקים לתהליך הבן. במקרה זאת, הדפים משותפים בין התהליך האב ותהליך הבן. בכל פעם שתהליך (אב או בן) משנה דף, נוצר עותק נפרד של הדף הספציפי אשר שונה לתהליך הרלוונטי. התהליך זה השתמש בדף החדש במקומות

הדף המשותף בכל הפעולות העתידיות. התהיליך האخر (זה אשר לא שינה את הדף) ממשיר לשימוש בעותק המקורי של הדף (אשר לא משותף יותר). זהו בדיקת טכניקת COW מפני שהדף מועתק בזמן שחלק מהתהיליכים כתובים אליו.

**טהיליך יתום (orphan process)** - תהיליך יתום הוא תהיליך מוחש שטהיליך האב שלו הסתיים או הופסק(terminated), אם כי הוא ממשיר לפעול בעצמו. תהיליכים שכאלו עוברים תהיליך reparenting כאשר היתומים "מאומצים" על ידי תהיליך המערכת init (PID1).

**errno** - משתנה מערכת, אשר נקבע על ידי קריאות מערכת במקורה של שגיאה. בדרך כלל מציין מה השتبש, הקיימים או השגיאה מסומנים על ידי ערך ההחזרה של הפונקציה, בדרך כלל thread\_errno הוא thread-local-thread-safe, כלומר הגדרתו thread-safe אחד אינה משפיעה על ערכו בשום אחר. חשוב להקפיד על קוד הגנטוי ולהשתמש בו- schores לעתים קרובות (לבדוק שהערך המוחזר מהפונקציה הוא לא סורק אז להמשיר).

**File descriptors** - במערכות הפעלה של מחשבי יוניוקס, 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 לציין "אין ערך" או תנאי שגיאה. FD הם חלק ממשיק ה API - POSIX.

טבלת מתאר קבצים מוחזקת על ידי כל תהיליך ומכליה פרטים על כל הקבצים הפתוחים. כל תהיליך של יוניוקס (למעט אולי daemon) צריך לכלול שלושה מתארים של קבצי POSIX סטנדרטיים: המתאים לשולשת הזרמים הסטנדרטיים:

- ערך 0 עבור stdin, ערך 1 עבור stdout, ערך 2 עבור stderr.

### – Threads on POSIX: pthread API של POSIX, בקימפול מוסיפים pthread – (בספריה זו כל הפונקציות של הטרדים ממומשות).

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

יצירת תהיליכון חדש המתבצע במקביל לתהיליכון שקרה לפונקציה.

בהצלחה, identifier של התהיליכון החדש שנוצר נשמר במיקום שהארגון של התהיליכון מצביע עליו, ומהזיר 0. בשגיאה, קוד שגיאה שונה מואפס מהזיר.

הארגון של attr מאפשר להציג תכונות על התהיליכון החדש (למשל מנוטק-detached, מדיניות זמן-scheduling policy).

יכול להיות NULL (בירית מחדל).

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

arg הוא הפרמטר שהחומר start\_func מקבל.

**pthread\_t pthread\_self ()**

הפונקציה ממחזירה את identifier של התהיליכון.

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

משהה את ביצוע התהיליכון הקורא עד לסיום התהיליכון הנקרא.

בהצלחה, ערך ההחזרה של th נשמר במיקום שמצוין על ידי thread\_return, ו- 0 מוחזר. בשגיאה, קוד שגיאה שונה מ0 מוחזר.

לכל היוטר תהיליכון אחד יכול לחכות לסיומו של תהיליכון נתון. קריאה ל- join\_join מטהיליכון th שבו תהיליכון אחר כבר ממתן לסיום מחזירה שגיאה.

```
void pthread_exit (void* ret_val)
```

מס'ם (terminate) את ביצוע התהיליכון הקרייה. לא מס'ם את כל התהיליך אם קוראים לו מהפונקציה הראשית.

אם ret\_val שונה מnull, ret\_val נשמר, ורכזו ניתן לתהיליכון שביצע join לתהיליכון זהה. כמובן, זה יכתב לפרמטר thread\_return thread\_join pthread.

**תקשורות בין תהיליכים (inter-process communication או IPC)** - אוסף של שיטות ומנגנונים להעברת נתונים בין תהיליכונים בתוך אותו תהיליך או בין תהיליכים שונים.

התקשורות יכולה להתבצע בין תהיליכים שרוצים על גבי אותו המחשב, או על גבי מחשבים שונים הממחברים ברשות. ניתן לחלק את השיטות לתקשורות בין תהיליכים לקטגוריות הבאות:

- העברת מסרים
- סyncronization
- זיכרון משותף
- הפעלת פרוצדורות מרוחק

השיטה הנבחרת לתקשורות בין תהיליכים תלויות ברוחב הפס, בזמן ההשניה בתקשורות בין התהיליכונים, ובסוג הנתונים המועברים.

הסיבות לייצור תשתיות המאפשרות שיתוף פעולה בין תהיליכים: שיתוף מידע, זירוז תהיליכים, מודולריות, נוחות, הפרדת הרשאות.

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

נניח ש process עושה פעולה, ניגש לזכרון שלא הוקצה או מחלוקת ב-0, במקרה זה על ה CPU נוצר exception condition והוא מתנהג כמו interrupt הוא היה צפוי ויש לו handler.

לדוגמא- אם זה חלקה ב-0 יש שנראה handler שנקרא floating point exception handler והוא יטפל בהז.

הCPU לא יכול להמשיך ולכן תפסיק את הprocess מהCPU ולהגיד למי שכתב את הקוד שיש באג.

לכן kernel אומר לkernel שבסבב תhiba התנהגות דיפולטיבית כמו לרשום core dump (סטטוס מתי זה קרה, ומה זה קרה. זה יגרום לכך שבסבב תhiba התנהגות דיפולטיבית כמו לרשום core dump (סטטוס מתי זה קרה, ומה זה קרה. זה יעזר אחר כך למתקנת לפתח את הקובץ ולנתח את הבעה) ואחר כך יוריד את הprocess.

**אותות (Signal)** - איותים הם צורה מוגבלת של תקשורת בין תהיליכים(IPC) , המשמשת בדרך כלל במערכות הפעלה, אובייקטים ייחודיים ומערכות הפעלה אחרות התואמות POSIX.

אותן היא התראה אסינכרונית שנשלחת לתהיליך או לתהיליכון ספציפי באותו תהיליך כדי להודיע לו על אירוע. כאשר נשלח אותן, מערכת הפעלה קוטעת את זרימת הביצוע הרגילה של תהיליך היעד כדי להעביר את אותן.

ניתן להפסיק את הביצוע במהלך כל פקודה אחרת אוטומטית. אם התהיליך רשם בעבר מטפל באותות(signal handler), השגרה החזו מבוצעת(מה שהוגדר עבור signal handler). אחרת, מטפל אותן המוגדר כבירית מחדל מבוצע.

Embedded programs עושים שימוש במקרים מסוימים לתקשורות בין-תהיליכים, מכיוון שאותות בולטים ביעילותם האלגוריתמית.

איותים דומים להפרעות, ההבדל הוא שהפרעות מתווכות על ידי המעבד ומטופלות על ידי הליבה ואילו אותן מתווכים על ידי הליבה (אולי באמצעות קרייאות מערכת) ומטופלות על ידי תהיליכים.

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

קיימים אותן סינכרוניים (חלוקת ב-0) ואותות א-סינכרוניים (לפי שיעון).

דוגמאות לsignals:

SIGSEGV – הפרת SEGmentation •

SIGFPE – Floating point error •

SIGILL – פעולה לא חוקית. •

SIGINT – Interrupt •

כבירית מחדל גורם לסיום התהיליך. •

- SIGABRT – סיום (termination) לא תקין, למשל על ידי לחיצה על Q + Ctrl + .
- SIGTSTP - השעית (Suspension) לתהיליך, למשל על ידי לחיצה על Z + Ctrl + .
- SIGCONT - גורם לתהיליך מושעה לחידוש הביצוע.

הערה: שלושת ה`signals` הראשונים הם סינכרוניים מכיוון שהם יכולים להגיע רק כתגובה לפעולה שבוצעה.

Signal מעובד לאחר שתהיליך חוזר מ-`Interrupt` או `call system` ולפניהם שהאות מגיע הוא עבר בקוד של המשתמש.

כאן יש 3 אפשרויות:

- התנהגות דיפולטיבית - אין התייחסות בקוד ולכן `signal` ממשיך בהתקנות ברירת המחדל שלו.  
(לדוגמה: `exit, core, stop, ignore, continue`).
- התעלמות – `SIG_IGN`Ignore, התהיליך לא יבצע בלי שום שרידים.
- טיפול ב-`signal`, פונקציה של המשמש שמתפלת ב-`signals` מסוימים (לדוגמה לפני ביצוע `SIGABRT`).

### טבלת signals –

כל תהיליך יש טבלת איותיות וכל אות מוצג כערך בטבלה, האם להתעלם ממנה? ובמידה ולא מתעלמים, איזה פעולה לבצע?

### טיפול באותות (Signal handler)

לזהיליך יש מספר אפשרויות, להתעלם מאות ולהשתמש בפועלות ברירת המחדל של האות או שייהי לו פונקציית `signal`.

טיפול באותות הנקראת כאשר האותים המשווים התקבל לתהיליך זהה.

הערה: פונקציית טיפול באותות היא לכל תהיליך לכל אות.  
במקרה זה אנו אומרים כי התהיליך תופס את האות.

### אייר מטפלים ב-`signals`?

`process` או פונקציה שכותבים בתחום `process`.

התכנית רצה (normal program flow), מקבל signal, context switch, מחליף את התהיליך למרחב הגראן ומבצע שם את הטיפול באותות (לעתים לרגרען יש handler שגובר על `context switch` שהוא משתמש כתוב) אם הוחלט שההשוך מחליף את `context switch` מחזיר את התהיליך למרחב המשמש, הוא מבצע את `handler` שכותב עבור האות, וחוזרים למרחב הגראן בעזרת `sigreturn()` והוא מוחזיר את הקונטקט של התהיליך ומחזיר אותו למרחב המשמש להמשך ריצת הקוד. כך הכל מתבצע context switch ארבע פעמים.

signals מוחדים: SIGKILL & SIGSTOP, שני signals שלא ניתן לTrap (kill הורג את התהיליך וstop עוצר אותו, ניתן להמשיך אותו בעזרת SIGCONT).

הערה: ביצוע `fork()` כל הסטטוסים של ה-`signals` מועתקים אך לאחר ביצוע `execvp()` נעשה reset וכל ה-`signals` חוזרים להתקנות הדיפולטיבית שלהם.

מטפל האותות מבוצעים במצב משתמש (user mode), ולכן הם עשויים להיות מונעים על ידי תהליכון אחר, בדיקן כמו כל תהליכון ברמת המשמש.

מטפל האותות עשוי לבצע קריית מערכת (system call).

עם זאת, קיימת רשימה של פונקציות בטוחות של אות לא סינכרוני (Async-signal-safe functions) כמו `lomem` פונקציות שאין interrupted לאחר קבלת אות ספציפי, במקומם לבצע את פעולה ברירת המחדל, מטפל האותות שלו יקראה.

זה נעשה באמצעות קראות המערכת (`signal` & `sigaction`).  
זה מומלץ להיות עקיבים: השתמש תמיד באותה פונקציה מתוך השינויים.

`Signal` פשוטה יותר ובאופן היסטורי, נעשה בה שימוש נרחב יותר, עם זאת, (`sigaction`) גמישה ויציבה יותר.

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

מתיקן מטפל אותן חדש עבור האות עם מספר `signum`.

מטפל האותות מוגדר ל-`sighandler` שעשי להיות: פונקציה שמצוינה על ידי המשמש, `SIG_IGN` (התעלם מהאות) או `SIG_DFL` (השתמש בפעולות ברירת המחדל של האות).

`()` הוא one-shot, צריך לקרוא לו שוב לאחר כל אחת שנפתחה.

```
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 מולאנו מטפל האותות הישן ישמר בתוכו

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

משנה את רשיימת 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 set מוסרים מהמערכת הקיימת של האותות החסומים.

זה חוקי לנסוט לבטל חסימה של אות שאינו חסום.

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

Oldset – אם הוא שונה מולאנו יחזיק את הערך הקודם של המאיץ signal.

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

הערה: לא "רואה" את כל המשתנים שערך process.

שליחת סיגנלים – ניתן לשלוח סיגナル דרך המקלדת, דרך CMD ובעיצועם call system.

שליחת אותות דרך המקלדת:

- Ctrl-C שולח אות SIGINT (signal-interrupt), כבירת מחדר, הדבר גורם לסיום התהילה.
- Ctrl-Q שולח אות SIGABRT , גורם לשימוש לא תקין (abort).
- Ctrl-Z שולח אות SIGTSTP, כבירת מחדר, הדבר גורם לתהילך להשעות הביצוע (suspend).

שליחת אותות דרך הערך command line:

- - raise(sig) – שליח אות לתהילון שרך כעת.
- - kill <signal> <PID> – שליח את האות שמצוין ל PID שצוין (אם לא צוין שום אות, נשלח אות ה- TERM).
- - killall <signal> <PNAME> – יכול לשמש לשילוח אותות רבים לתהילכים המריצים פקודות ספציפיות או בבעליות משתמש מוגדר, או בעלי גיל מסוים ועוד.
- - fg <PID> – ממשיכה בביצוע תהילך מסוימת על ידי שליחת אות SIGCONT, זה יגרום לתהילך חדש לפעול בחזית.

הערה: לא כל התהילכים יכולים לשלוח אותות לכל שאר התהילכים.

רק הגרעין ומשתמש העל (superuser) יכולים לשלוח אותות לכל התהילכים. התהילכים רגילים יכולים לשלוח אותות רק לתהילכים שבבעליות אותו משתמש.

### קבוצת תהילכים (process-group) –

קבוצת תהילכים היא אוסף של תהילכים הקשורים זה לזה. כל תהיליך יש מזהה (PID) ומזהה קבוצתי (PGID) .

כל תהילכים בקבוצת תהילכים מוקצה אותו מזהה קבוצת תהילכים (PGID) .

ניתן לשלוח אות לתהילך יחיד או לקבוצה.

משמש את המעטפת לשיליטה במסימות שונות המבוצעות על ידה.

פונקציות שימושיות:

- int getpid() – מחזיר את ה- PID של התהילך.

- int getpgid() – החזיר את ה- PGID של התהילך.

- setpgrp() - מגדיר את ה- PGID של התהיליך זהה להיות שווה ל- PID שלו.
- setpgrp(int pid1, int pid2) - מגדיר את PGID pid1 של התהיליך להיות שווה ל- PGID של pid2.

### Pipe

צינור הוא חיבור בין שני תהיליכים, כך שהפלט הסטנדרטי מתהיליך אחד הופך לקלט הסטנדרטי של התהיליך השני. במערכות ההפולחן UN\*X, צינורות שימושיים לתקשורת בין תהיליכים קשורים (inter-process communication).

צינור הוא תקשורת חד כיוונית בלבד, כלומר יכול להשתמש בzinor כך שתהיליך אחד יכתוב לצינור, והטהיליך השני יקרא מהצינור.

הוא פותח צינור, שהוא איזור של זיכרון ראשי (main memory) המתוול כ"קובץ וירטואלי".  
הzinor יכול לשמש את הרטהיליך היוצר (creating process), כמו גם את כל תהיליכי הילד שלו, לקריאה וככיבתה.  
טהיליך אחד יכול לכתוב ל"קובץ הוירטואלי" זהה וטהיליך קשור אחר יכול לקרוא ממנו.  
אם הטהיליך מנסה לקרוא לפניו שימושו כתוב לצינור, הטהיליך מושעה (suspended) עד שנכתב משזה.  
קריאה ממערכת סוקון מוצאת את שתי המיקומים הראשונים בטבלת הקבצים הפתוחה של הטהיליך ומקרה  
אותם לנקודות הקריאה והככיבתה של הzinor.

`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 בFOIFO, מתנהגים כמו תור.

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

כאשר אנו משתמשים בfork() בכל תהיליך שהוא, file descriptors נוטרים פתוחים על פניו תהיליך הילד וגם על תהיליך האב. אם אנו קוראים לfork() לאחר יצירת zinor, אז ההורה והילד יכולים לתקשר באמצעות הzinor.

**מתזמן (scheduler)** - פעולה של הקצאת משאבי לביצוע משימות. המשאים עשויים להיות מעבדים, קישורי רשת או כרטיסי הרחבה. המשימות עשויה להיות threads, processes או זרימת נתונים.  
פעולות המתזמן מתבצעת על ידי הטהיליך הנקרא מתזמן. מתזמנים מתוכננים להשאיר את כל משאבי המחשב לעסוקים (כמו באיזון עומסים), לאפשר למשתמשים רבים לשף משאבי מערכת ביעילות, ולהציג אפשרות שירות היעד.  
המתזמן הוא מהותי לחישוב עצמו, והוא חלק מהותי ממודול הביצוע של מערכת מחשב, היעוד של מתזמן הוא לבצע ריבוי משימות במחשב עם יחידת עיבוד מרכזית אחת (CPU).

טיפול בתקני interrupt יש את העדיפות הגבוהה ביותר (הוא נמצא high-halves).

ובძאנט-bottom-half המתזמן עובד ומתזמן משימות שלטהיליכים וטהיליכונים בגרעין ובמרחב המשתמש.

### סוגי OS:

**Batch** – פעולות שאינן להם עדיפות גבוהה בעבודות חישוב שלוקחות זמן (Backup לדוגמה) ולא דורשת אינטראקציה עם משתמש או עם מערכת I/O.  
משימות עיקריות: היעילות נמדדת במספר הjobs שהושלמו בפרק זמן נתון.

**Interactive computing** – מחשב אינטראקטיבי מתייחס לתוכנה שמקבלת קלט מהמשתמש תוך כדי הרכיצה. תוכנה אינטראקטיבית כוללת תוכניות נפוצות, כגון word processors או יישומי גילוונות אלקטרוניים. לשם השוואה, תוכניות שאינן אינטראקטיביות פועלות ללא הטעבות המשתמש.

מחשב 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 מסות להציג זאת באמצעות יישום ממשקים חברותיים.

משימות עיקריות:  
תגובה מהירה יותר כדי לספק את הדרישות של משתמשים שזוקקים turnaround מהיר בעת ניפוי באגים בתוכנית שלהם ולאפשר לכל משתמש לתקשר ישירות עם מערכות הפעלה.

## – Real-Time

לא מעניין אוטם ניצול של הCPU אלא רק עמידה בdeadlines. משימות עיקריות: משומש בסביבות קריטיות בזמן, אמינותה היא המפתח וזמן תגובה המערכת חייב להיות בעמוד האחרון (deadline).

### מטרות המתזמן:

בכל המערכות:

- Fairness – לחת לכל process שיתוף הוגן בCPU.
- policy enforcement – לראות שה מדיניות המוצהרת מטבחת.
- Balance – לשמר על כל החלקים שבמערכת עסוקים.

במערכות :Batch

- Throughput (תפקה) – למקסם את העבודות לפי שעה.
- Turnaround time – למן את הזמן בין הגשת העבודה לסיום העבודה.
- CPU utilization – לשמר על הCPU עסוק כל הזמן.

במערכות :Interactive

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

במערכות :Proportionality

- – לענות על הציפיות של המשתמש.

במערכות :Real-time

- Meeting-Deadlines – להימנע מלאבד data, עמידה בזמן.

• Predictability – תהיה צפוי וידעו 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 מהCPU כי היא סיימה או שהחילתה עצמה לעצור או שruptured 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 (FCFS – First come First served)**

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

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

- Non-preemptive
- ממן את זמן turnaround.
- משך העבודה צריך להיות ידוע מראש.
- כל השירותות צריכים להיות זמינים בהתחלה, כדי שייהיה יעיל.

חסרון: יכול לגרום לרעבה (starvation) כיון שהיא ארוכה לא לקבל CPU לעולם.

#### :**זמן הנותר הקצר ביותר ראשוני (SRTF – Shortest Remaining Time First)**

- 2 אפשרויות: preemptive & non-preemptive
- ממן את זמן turnaround וגם את הזמן waiting time.
- טוב עבור jobs I/O (Interactive).
- צריך לדעת את זמן העבודה שנותר.
- חסרון: יכול לגרום לרעבה (starvation) כיון שהיא ארוכה לא לקבל CPU לעולם.

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

- Non-preemptive
- מנסה למנוע את החיסרונו של job הקצר תחילתו על ידי התחשבות בזמן המתנה של 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 (קונטי).
- כלל הבדיקות יש את אותה עדיפות.

הערה: אם ניתן קונטיים גדולים מדי לכל עבודה האלגוריתם יראה כמו FCFS, לעומת זאת אם ניתן קונטיים קטנים מדי נבדך הזמן זמן על context switch.

## מתzman לפי עדיפות (Priority scheduling) :

- Preemptive or Non-Preemptive •
- עדיפות קבועה. •

עדיפות לפי סוג (O/I) אל מול CPU. •  
(1/f, all up, running down). •

## תזמן מתקדם (Advanced scheduling) :

- טור עדיפות (קבוע) •
- טור עדיפות (קונטי כפול) •
- זמן מובטח (1/#process) •
- זמן lottery. •

הוגנות בעליים (Owner Fairness) .

## תזמן מובטח (Guaranteed scheduling) :

הCPU עובד ומחשב כל הזמן, לכל תהליך לפי היחס בין הזמן המעובד המקצה וכמות זמן המעובד שהתהליך זכאי לו.

- בוחר את התהליך עם היחס הנמוך ביותר.
- הוגן – מבטיח  $\frac{1}{n}$  זמן המעובד לכל משימה (עבור  $n$  משימות).

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

- מחלק את העשויות ready למספר תורים.
- כל טור מפעיל מנגןון תזמן ואלגוריתם משלה.
- התהליך מקבוצת עדיפות נמוכה יותר עשוי לפעול רק אם אין תהליך בעדיפות גבוהה יותר.
- מנוע הרעבה לפי גיל: התהליך "ישן", שהמתין זמן רב, מקודם לטור בעל עדיפות גבוהה יותר.
- התהליך שננהנה זמן המעובד מועבר לטור בעל עדיפות נמוכה יותר.
- תורים ברמה נמוכה הם בעדיפות נמוכה יותר, אך יש להם קונטיים ארוכים יותר.
- ה מדיניות המדיקת של קידום, הורדנה והמדיניות של התווך עשויה להשנות.

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

- Throughput – כמות התהליכים שהסתיימו בכל יחידת זמן (time unit).
- Efficiency: CPU utilization – אחוז הזמן שהCPU שעוק (computation & communication).
- Turnaround time – זמן ממוצע משילת העבודה ועד להשלמה.
- Waiting time – סכום כל האינטראולים שבינם התהליכי היה בשעון-ready.
- Response time – זמן התגובה, הזמן מאז שלחו את העבודה ועד לזמן שהעובדת מקבלת CPU בפעם הראשונה.
- Fairness – תהליכי דומים צריכים לקבל שירות דומה.

## תרגילים טובים במצגת 8

### – Real-Time OS

לא מעניין אותנו ניצול של הCPU אלא רק עמידה בזמני deadlines. משימות עיקריות: משמש בסביבות קritisיות בזמן, אמינות היא המפתח וזמן תגובה המערכת חייב לעמוד בזמן האחרון (deadline).

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

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

הערה: Embedded System != RT-OS

## תכונות וטיפול בזיכרון:

פচות פיצרים מאשר במערכת הפעלה של שולחן העבודה / השרת:  
 חלק ממערכאות ה RT - הן למטרות חד פעמיות, כמו מיתוג או ניתוב מנוגט (packets), מכון טיל, משק GPS וחישוב נתיב קצר ביוטר.  
 אין משק משתמש ויכולות חומרה מוגבלות.  
 כתובת זיכרון:

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

כתובת רילוק'ישן - הוספה ערך relocation register לתרגום.  
 יישום VM מלא.

## דרישות מימוש:

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

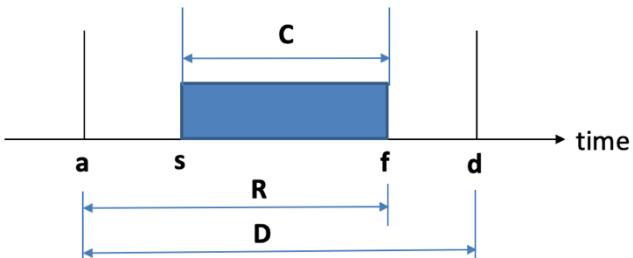
## הערה: RT אינו הוגן (אין התחשבות בfairness!).

המטרה העיקרית של מתזמן RT היא לעמוד במועדים לכל משימות ה- RT המתווזמות. ניצול מקסימלי של המעבד (best throughput), התפקוה הטובה ביותר ביותר (maximum CPU-utilization), זמני תגובה והתנה (response and waiting times) (minimum average turnaround), זמני תגובה והתנה (minimum average turnarounds).

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

## מדיניות זמן RT:

איזו אלגוריתם תזמין מבטיח עמידה בכל מועד הזמן עבור מערכ משימות נתון?  
 כיצד להוכיח שאלגוריתם תזמין נתון יעבוד בעומס עבודה מסוים?  
 מה הפירוש של המונח "אלגוריתם אופטימלי" לתזמין RT?



### תזמון עבודה – הגדרות:

a – arrival time, הזמן שבו משימה מוכנה לביצוע.

d - absolute deadline , המועד האחרון לשימוש המשימה.

f/s – הזמן שבו המשימה התחילה והסתיימה.

C - worst case execution time . הגראן ביותר, משך הזמן הדרוש למאבד להשלמת העבודה ללא הפרעות.

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

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

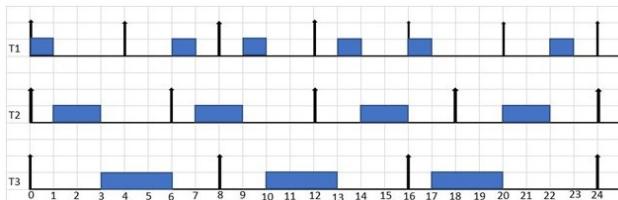
מתי נפססו את deadlines ? אם  $D > R$  או  $d < f$ .

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

הפעלה ובדרך כלל שווה למועד הdeadline.

אם  $# CPU > U$  המתזמן אינו יכול לעמוד בכל deadlines - העומס גבוה מדי.

משימת RT מוגדרת כך:  $.Task(C, D, P)$



### אלגוריתם תזמון - EDF

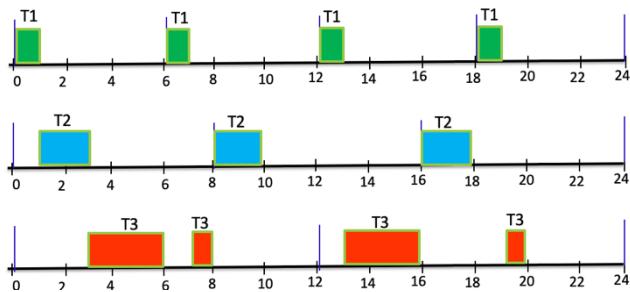
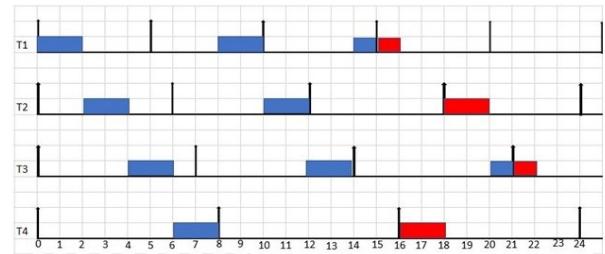
T1 (1,4,4), T2 (2,6,6), T3 (3,8,8)

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

Transient Overload and  
Domino Effect

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

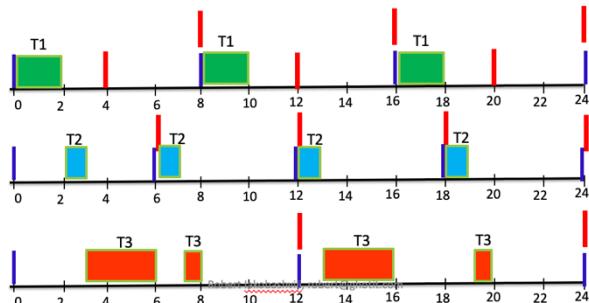
$$U = \frac{2}{5} + \frac{2}{6} + \frac{2}{7} + \frac{2}{8} = 1.269 \rightarrow \text{not feasible}$$



### 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 - (DM)

למשימה עם תאריך עד חסוי 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, והאפליקציהיות רצות באזורי המשתמש, הגרעין לא יכול להעיף משימת kernel לטובת משימת user וגם ה latency הם גבוהים.  
אם ניתן לשנות ולהפוך את לינוקס לRT?

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 הגיעו מפתח והציג עמוד שהיה מחוץ ל[min]. מה הוא עשה ? הוא הוריד את הסטטוס של משימות kernel , ז"א kernel יכול להיות preempted ובנוסף הוא הוריד משמעותית את latency וعود.

### :Linux Scheduling

- SCHED\_OTHER / SCHED\_NORMAL - מדיניות רגילה של חלוקת זמן RR.
- SCHED\_BATCH - RR כאשר ההנחה היא שהמשימות אינן אינטרקטיביות ומחייבות מעבד עם פרוסת ברירה מוחלט של 1.5 שניות. מדיניות ידידותית למטען (cache).
- SCHED\_IDLE - RR עם פרוסה גבוהה יותר הנינתה למשימות בעדיפות נמוכה.
- SCHED\_FIFO - FIFO ללא חיתוך זמן.
- SCHED\_RR - פרוסות זמן RR עם preemption.
- SCHED\_DEADLINE - מדיניות חדשה שמועברת על ידי מזמן ה- EDF תזמון שרת רוחב פס קבוע (CBS) שעזר ליציבות ה-EDF ומונעת אפקט דומינו.

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

### :Shared Object (Critical Sections)

אר נdag שرك תהיליכון אחד יכנס לקטע קוד קרייטי ואף תהיליכון אחר לא יוכל להיכנס עד שהטהיליכון שבפנים יסימן?

1. פעולה אוטומטית – לא מתאים עבור המקרה הכללי.
2. Spinning waiting – כמו לולאת while שתחסום את התהיליכון עד שהטהיליכון השני ישחרר אותו. חסרון: לוקח הזמן CPU יתרון : שה פעולה שמתבצעת ב- CS ודרטמיניסטית אז לא שווה להכניס את התהיליך לשינה ולהעיר אז אם הקטע קוד קצר וידיע מראש אז הוא עדיף.
3. Sleeping – להרדים את התהיליכון ולהעיר אותו כשהוא יכול להיכנס ל- CS. חסרון: יקח זמן מהרגע שנשלח פקודה להעיר את התהיליכון ועד שהוא יכנס לקטע הקוד. יתרון: צורך מעט מאוד CPU.

### :תנאים עבור פתרון טוב:

- Mutual Exclusion - אין שני תהליכיים בקטע הקרייטי (CS) בו זמנייה.
- Deadlock Freedom - אם תהיליכון אחד או יותר מנוטם להיכנס ל- CS, בסופו של דבר אחד יכנס אליו.
- Starvation Freedom - תהיליך שמנסה להיכנס ל- CS יצלה בסופו של דבר להיכנס אליו.
- Logic Solution - הפתרון לא יכול להיות תלוי בנסיבות המערכת או בhardware .

**פתרון 0 לפתרון CS – ביטול כל הdeadlock:** חסרונות: מסוכן מאוד ולא מתאים לסביבה של multiprogramming.

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

```
While(TRUE) {
    int turn = 0;
    /*process 0 & process 1
    ...
    */
    while (turn != process); /* Busy waiting */
    enter_Critical_Section();
    turn = ~process;
    exit_Critical_Section ();
}
```

חסרונות: תהליכיים חייבים להיכנס ל- CS לシリוגן – מפר את תנאי "Deadlock Freedom" . המתנה עומסה (busy waiting) – בזבוז משאבי ה- CPU.

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

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

$N=2$ ;

```

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

int turn;
/* all initially 0 (FALSE) */

void enter_region(int process){
    int other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while (turn == process &&
           interested[other] == TRUE);
    /* null statement */
}

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

```

```

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 1

```

/* who is leaving 0 or 1 ? */
/* departure from critical region */

```

```

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;
}

```

Process 2

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.

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

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

שאלה 2: מה יקרה במידה ונסנה את התנאי בלולאה להיות - (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 כיון שגם להיווצר מצב שבו שני תהליכי יכנסו במקביל ל-CS.

טענה: אלגוריתם Peterson לא יעבוד במקרה של 2 CPU

סוגי סנכרון בין CPUs:

- Cache coherent system – סנכרון מהיר של cachen וונעשה באופן אוטומטי, ככלומר שני הCPUs רואים בקוד אותו הדבר והturn של שניהם זהה – סנכרון זהה הוא מאוד יקר.

• Non-cache coherent system – במידה ושינינו turn של אחד מהCPUs יקח מעט זמן עד שזה יתעדכן אצל CPUs השני, ולכן לא ניתן להסתר מכך על העבודה שניהם רואים בדיקון אותו דבר בכל רגע נתון. סyncronon שכזה יותר נפוץ במחשבים, במערכות עם סyncronon כזה אלגוריתם Peterson לא יעבוד כיון שהנתונים לא יתעדכו אצל שני התהליכים ולכן נצטרך ליצור סyncronon בין שני processes לאחר כל שורת קוד. מה הפתרון? נעצר את CS במנועול – נלמד בהמשך.

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

הרענון: ישום חלקים בחומרה שמסוגלים לבצע memory barrier (חסימת זיכרון), סyncronon בין הchachimים ו-lock על משתנים. למעשה הפעולה מתבצעת בצורה אוטומטית ולכן כל CPUs מתחדדים בשינויים שנעשה בפועל האוטומטי דוגמת הריצה: תחיליך 1 נכנס ללולאה ומරיץ את (0) Test-and-Set, הפקחציה מחזירה לו 0 ולכן הוא נכנס ל-CS ובמקביל העשוי value של שאר התהליכים הוא 1 ולכן CS נועל מבחינותם והם לא יכולים להיכנס (גם הערך של value נועל כיון שהוא יחזור להיות 0 רק אחרי שהוא יצא מה-CS). הערה: לאלגוריתם אין הבטחה livelock.

```

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
waiting:
    RET          # locked successfully; enter the CS
    # 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

```

Assembly

### 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 זוקק לכליים לניהול משאבי משותפים. כל הਪתרונות שראינו עד כה, היו בזבזנים (spinning algorithm) ויקרים. פתרון: תמייהה מההארה – פקודות אוטומיות, זול מהיר ופשוט.

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

1. הדגל שמסמל על מצב החלק הקרייטי מאותחל ל-1.

כאשר הוא 1 אז החלק הקרייטי הפנוי.

2. כניסה לאזרור הקרייטי מתבצעת ע"י (s)down.

אם 1=s תנסה אותו ל-0 (מסמן שהאזרור הקרייטי תפוך ותיכנס לאזרור הקרייטי).

אם 0=s סימן שהאזרור הקרייטי תפוך כרגע והתהליך שקרה לחown ישלח ל"טור שינה", שם הוא במצב שינה, במצב זה הוא לא ניתן לתזמן ולא צורף CPU.

3. כאשר התחליך שנמצא באזרור הקרייטי מסיים, הוא מבצע (s)up.

בפועל זה, הדגל חוזר להיות 1 (אזרור קרייטי פנוי) וניר לפחות תחליך אחד מ"טור השינה" אל ה-ready queue.

```

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;
    }
}

```

## Counting Semaphores: דוגמה לBinary רק כשכאן אנו מתחלים את S להיות מספר התהליכים שאנו מאשרים

```

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;
    }
}

```

שירוצו במקביל באזור הקרייטי.

1. אנחנו מתחלים את הדגל שלנו להיות שווה למספר התהליכים שאנו מאשרים שירוצו במקביל בתוך האזור הקרייטי.

2. אותו דבר כמו הקודם, אלא שכן במקום לשנות את הדגל ל0 אנחנו מקטינים אותו ב1.

3. אותו דבר כמו הקודם, אלא שכן במקום לשנות את הדגל ל1 אנחנו מגדילים אותו ב1.

הערה: semaphore לא מבטיח freedombusy waiting. ולא כל המימושים מנועים

## Negative-Valued Semaphores: במבט ראשון מואוד דוגמה לcounting השוני הוא שבUp/Down 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;
    }
}

```

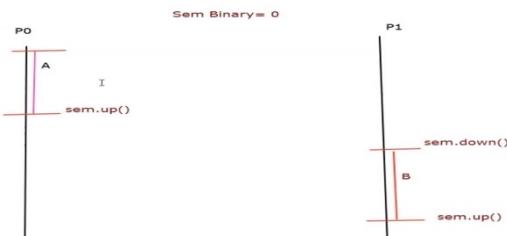
מקטינים/מגדילים את S ורק לאחר מכן מבצעים בדיקה.

1. אנחנו מתחלים את הדגל שלנו להיות שווה למספר התהליכים שאנו מאשרים שירוצו במקביל בתוך האזור הקרייטי.

2. בהפעלה של ()down אנחנו קודם כל מקטינים ורק לאחר מכן מבצעים את thread sleeping לsleeping, היתרון שלו על גבי counting הוא שאנו יודעים כמה תהליכי נמצאים במצב שונה (S-תהליכי).

3. על פי אותה לוגיקה, בביטוי ()up קודם נגדיל את S ורק לאחר מכן נערר תהליך כלשהו.

הערה: מההגיוון של negative-value/counting מספר threads שיכולים לעבור דרך הקטע הקרייטי מוביל ללקת לישון יהו (command that has been executed) ()up # + N.



דוגמה לשימוש (בינארי): נניח שיש לנו שני תהליכי על אותה תכנית. בתכנית קיימים שני קטעי קוד כאשר אנחנו רוצים שקטען קוד B יירוץ תמיד רק אחרי שקטען קוד A כבר הtbody. במקרה זה אנחנו נועלם 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 בעיתתי: נניח שיש לנו קוד ובתוכו ישב איזשהו קטע קרייטי. השימוש semaphore מאלץ אותנו לשימוש ()down לפני ()up אחריו. נניח ובמהלך פעילות התכנית קיימ איזשהו thread שנכנס לקטע קרייטי.

מכיון ש semaphore משותף לכלם, יכול להיות מצב ש thread אחר יבצע ()up מתוך פונקציה אחרת ואז יוכל להיכנס לפונקציה שנענלה ע"י thread הראשון. מצב זה יכול לקרות כאשר יש לנו כמה קטעי קוד קרייטיים בתכנית. הדריך למנוע זאת היא להשתמש mutex (אימפלמנטציה של Binary Semaphores).

Mutex משתמש ב()lock וב()unlock, כאשר בתוך ()lock יש קראיה ל()down בתוספת מזהה ייחודי של thread.

שביצע את הנעילה וכן כאשר נקרא ל()unlock הוא יבודוק האם זה אותו thread שנען ונקה אם כן הוא יבצע ()up. אפשר לחוש על mutex כירוש של Binary Semaphores, הוא מכיל את אותן תכונות בתוספת התוכנה החשובה Thread ownership.

## תכונות של Mutex: Thread ownership

1. Thread ownership – רק מי שנען יכול לשחרר.

2. מטפל בבעיית Reentrycy – מאפשר רקורסיה לתוך CS. אם אותו הליכון קורא ברקורסיה לפונקציה הוא לא יקרא שוב ל()lock שוב, אלא יגדיל את counter שסופר כמה פעמים הוא נכנס לפונקציה ברקורסיה ובכל

חזרה מהרקורסיה counter יקטן וכשיגיע ל0 הוא יבצע ()lock.

3. רק thread שאתחול (בצורך init) את mutex יכול להרשות אותו (בצורך destroy).

4. תומך בהורשה של עדיפויות (priority) ויכול למנוע החלפות של עדיפויות.

**שאלה 1** (binary semaphore) : ננסה ליצור counting semaphore באמצעות counter –  $S_{int}$  (בכל הניסיונות 2  $S_{int}$ ) .

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

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

## ניסויון 1

ניתן לראות בדוגמה ההרצה שקיים מצב שבו שני תהליכיים נכנסו במקביל ל- $CS$ .  
הבעיה כאן היא בפונקציית  $(down)$ .  
כאשר שני תהליכיים נשלחים ל- $CS$ , התהיליך השני יוריד את counter ל-0 וינעל לעצמו את האפשרות לבצע  $(up)$ , למקרה שהוא צריך להחזיק באופציה הזאת.  
הדרך הנאייה לטפל בעיה היא לשנות את התנאי בחושם להיות  $\geq$  במקומ>. אבל במקרה זה שלושה תהליכיים עלולים להיכנס ל- $CS$  במקביל.

Suppose initially  $N=2 \rightarrow Sint=2, Sb=1$   
Process 0:  
 $Sb=0; Sint=1; Sb=1$ ; Enter CS  
→ Context switch  
Process 1:  
Down:  $Sb=0; Sint=0; Sb=1$ ; Enter CS; Exit CS  
Up:  $Sint == 0$  (for doing  $Sint++$ , which is NOT atomic).  
→ Context switch  
Process 0: Exit CS  
Up:  $Sint == 0$  (for doing  $Sint++$ , which is NOT atomic).  $Sint = Sint+1 = 1$ .  
(...)  
→ Context switch  
Process 1:  $Sint = Sint+1 = 1$   
BANG:  $Sint$  is 1, while it should have been 2 → one more process should have been enabled access to the CS.

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

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

## ניסויון 2

בניסויון זהה שינוינו את פונקציית  $(up)$  והוספנו לה  $(Sb)$ .  
גם כאן מתעוררת בעיה דומה, ברגע שנגייע לתהיליך השני הוא יוריד את counter ל-0 ואז לא יהיה ניתן להיכנס ל- $(up)$  כדי להעלות את counter .

גם ניתן לשנות את התנאי בחושם להיות  $\geq$  במקומ>.

אבל שוב הינו מקבלים מצב שבו שלושה תהליכיים עלולים להיכנס ל- $CS$  במקביל.  
מתוך 2 הניסיונות הנ"ל אנו מסיקים את המסקנה הבאה:  
כדי ליצור counting semaphore אנו זקוקים לשני *Binary semaphores*.  
*Binary semaphore*

Suppose initially  $N=2 \rightarrow Sint=2, Sb=1$   
Proc 0:  
 $Sb=0; Sint=1; Sb=1$ ; Enter CS  
→ Context switch  
Proc 1:  
 $Sb=0; Sint=0; Sb=0$ ; Enter CS  
As  $Sb == 0$ , no process can arrive to  $Sint++$ , so its value (and the value of  $Sb$  as well) is reset forever!

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

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

## ניסויון 3

.counter –  $S_{int}$   
counter1 binary semaphore - S1  
counter2 binary semaphore - S2

כאשר נירץ 3 תהליכיים במקביל התהיליך הראשון מתחילה וכל הבינאריים משתנים ל-0 והוא נכנס ל- $CS$  המ חזירם ל-1.  $counter$  יורד ל-1, התהיליך השני מתחילה, כל הבינאריים משתנים ל-0 וכשהוא נכנס ל- $CS$ ,  $S1$  חוזר ל-1 אבל  $S2$  נשאר 0 (כי הגדרנו את  $S2$  לשמור על  $CS$ ). התהיליך השלישי יתחל, ינסה לבצע  $(down)$  ונתקע (כמו שציריך). התהיליך השלישי ביצע  $(down)(S1)$  ונתקע  $(S2)$  לאחר שהטהיליך הראשון מסתים הוא מבצע  $(up)$  ובשורה הראשונה הוא צריך לעשות  $(S1)$   $(down)$  אבל התהיליך 3 כבר הוריד אותו והגענו למצב שהנעילה לא משתחררת.

Process 0:  
 $S1=S2 = 0$ ,  
 $S = 1$   
 $S1=S2 = 1$ ,  
(CS)  
→ Context switch  
Process 1:  
 $S1 = S2 = 0$   
 $S = 0$   
 $S1 = 1$   
→ Context switch  
Process 2:  
 $S1 = 0$   
Cannot proceed / enter the CS  
→ Context switch  
Process 0: finish CS. Call Up:  
 $S1 = 0 \rightarrow$  cannot release the lock!

## הפתרון הנכון (solution of Barz)

```
down($){  
    down($$2);  
    down($$1);  
    $int--;  
    if($int > 0)  
        up($$2);  
    up($$1);  
}
```

```
up($){  
    down($$1);  
    $int++;  
    if ($int == 1)  
        up($$2);  
    up($$1);  
}
```

## 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 נכון) שnitן לעבוד בעזרתו את כל העבודות.

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

- אי אפשר ליצור מספר אינסופי של תהליכיים כיון שהקיים limit כלשהו.

לכל תהליכון מוקצה זיכרון וזוהי הקצתה מבוצבת לחלווטין כי רוב הזיכרון לא ימוש.

נניח שיש לנו מיליון תהליכיים במעבד עם 16 ליבות, הscheduler יצרך לעבוד מאוד קשה כדי לתת לכלם זמן בCPU ובתוספת הזמן זמן יתבצע על context switch.

פתרון לא טוב - יצירת thread בודד שיטף בכל העבודות:

- thread יעבוד רק על ליבה אחת במקום לנצל את כל N הליבות שיש במעבד.
- פוגע בעיקרונו multiprogramming.

פתרון טוב – שימוש בסיסוף Threadpool:

- ניצור מספר אופטימלי של threads  $N * k$ ,  $k \geq 1$  כאשר N זה מספר הליבות ו k הוא פרמטר התלוי באחיזה העבודות הדורשות CPU.

עבור בעיות שהן דורשות 100% CPU מספיק לנו N תהליכיים כדי לנצל את כל המעבדים, עבור בעיות שהן 50% CPU ו/או 50% CPU 50% נדרש 2N תהליכיים כדי לנצל את כל המעבדים.

כעת נכנס עמוק בבעיות 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 יצרך לוודא שמבנה הנתונים לא מלא (Full).
2. כאשר ה Consumer ירצה לעבוד עבודות הוא יצרך לבדוק שמבנה הנתונים לא ריק (Empty).
3. כאשר ניגשים לממבנה הנתונים, נרצה שהכניסה תהיה בטוחה ושרק תהליכיון אחד יוכל לבצע שינוי (הכנסה/הוצאתה) 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);  
        down(&empty);  
        down(&UseQ);  
        enter_item(item);  
        up(&UseQ);  
        up(&full);  
    }  
}
```

```
void consumer(void){  
    int item;  
  
    while(1){  
        down(&full);  
        down(&UseQ);  
        consume_item(&item);  
        up(&UseQ);  
        up(&empty);  
    }  
}
```

## הסבר הקוד:

ראשית נציגו שאנו עובדים עם שני `Producers` ואחד של `Consumers`. `Threadpools`, אחד של `Producers` ואחד של `Consumers`. בנוספּ הקוד כתוב בשפת C ויש להתייחס אליו כל פסודו-קוד ואין להתעכּב על האלמנטים של C לצורך הדוגמה נשתמש במבנה נתוני מסווג תור.

שלבי העבודה של `Producers` כאשר הוא רוצה להכניס עבודה חדשה לתור:

1. יצירת העבודה באמצעות הפונקציה `produce_item(&item)`.
2. הורדת counter של מספר המיקומות הריקים בתור ע"י הפונקציה `empty` שמקבלת את semaphore, `empty` semaphore 1 מוריידה `empty`, במידה ו- `empty` כבר על 0, זה אומר שאין מספיק מקומות פנויים בתור ולכן ה- `Producers` הספציפי הזה ימתין עד שייהי מקום בתור להכנסת עבודות חדשות.
3. לאחר שהעלת העלה את הערך `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 `Producers` יכנס את העבודה לתור ע"י `enter_item(item)` ופתח את השימוש בתור ליצרנים/CRCNs אחרים באמצעות `up(&useQ)`.
5. לאחר שהכניס את העבודה לתור ה- `Producers` יעלה את מספר המיקומות המלאים בתור באמצעות הפונקציה `up(&empty)`.

שלבי העבודה של `Consumers` כאשר הוא רוצה להוציא עבודה מהתור:

1. הורדת counter של מספר המיקומות המלאים בתור ע"י הפונקציה `full` שמורידה 1 semaphore `full`, במידה ו- `full` כבר על 0, זאת אומרת שאין עבודות בתור ולכן ה- `Consumers` הספציפי הזה ימתין שם עד שעבודה חדשה תיכנס לתור.
2. לאחר שהורד `consumer` את `full` ב-1, הוא ינסה לקבל גישה לתור באמצעות mutex `useQ` ע"י הפונקציה `down(&useQ)`, במידה והצליח לקבל גישה הוא עובר לשלב הבא, אחרת הוא ימתין.
3. כעת `Consumers` יצא את העבודה מהתור ע"י `consume_item(item)` ופתח את השימוש בתור ליצרנים/CRCNs אחרים באמצעות `up(&useQ)`.
4. לאחר שהוציא את העבודה מהתור ה- `Consumers` יעלה את מספר המיקומות הריקים בתור באמצעות הפונקציה `up(&empty)`.

מספר דגשים:

1. ה- `Producers` מנסה להוריד את ה- `empty`, כשיגע לו אין מקום לעבודות נוספות ולכן ה- `Producers` יילכו לישון.
2. ה- `Consumers` מנסה להוריד את ה- `full`, כשיגע לו אין עבודות בתור ולכן ה- `Consumers` יילכו לישון.

```
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 1: מה יקרה אם נבצע החלפה כמתואר בתמונה בין שורות הקוד?

תשובה: הרכינה להריד לא תהיה מאובטחת כי יכולם להיות עד 100 תהליכי ( Producers & Consumers ) בקטע הקוד זהה וכולם מנסים לשנות את התור במקביל ללא הבטחה שרק אחד יקבל גישה לתור בכל פעם.

הוכנסו לתא בכל זהה אומר לקבל ערך שגוי.

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 mutex של התור.

```

void consumer(void) {
    int item;

    while(TRUE) {
        down(&Use0);           /* enter critical section */
        down(&full);          /* decrement count of full */
        remove_item(&item);   /* take item from buffer */
        up(&Use0);            /* leave critical section */
        up(&empty);           /* update count of empty */
        consume_item(item);   /* do something... */
    }
}

```

**שאלה 2:** מה יקרה אם נבצע החלפה כמפורט בתמונה בין שורות הקוד?

**תשובה:** ה Consumer קיבל גישה לתור וינעל אותו אך כיוון שלא נעשה בדיקה שקיימות בכל עבודות בתור יכול ליזכר מצב שהוא נכנס לתור ואין בעודות פנויות ולכן Consumers יילך לישון ורק הגענו במצב של Deadlock – Deadlock

### – 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 להעיף את התהילה מהמשאב).
  - Circular wait – שני תהליכיים או יותר המסתננים למשאים המוחזקים ע"י תהליכיים אחרים (לדוגמה, תהליך A מוחכה למשאב שתפוס ע"י תהליך B, ותהליך B מוחכה למשאב שתפוס ע"י משאב A).

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

- Mutual exclusion – לא תמיד ניתן למנוע, לדוגמה אם תהליך מסוים תפס mutex, רק הוא משתמש בו כי זה "הופי" של mutex (היעדרות עלולה לגרום לאובדן שליטה במשאים ולהפרות מציאות).
- Hold and wait – לא תמיד ניתן למנוע, לצורך ביצוע עבודות מסוימות צריך להציג ב-B,C,A ולא רק במשאב אחד (היעדרות יכולה לגרום לתוכניות לפעול לזמן ארכויים יותר עם זמן המתנה רב יותר).
- No preemption – בדומה ל mutual-exclusion , קיימים משאים שבהם "הופי" שלהם (היעדרות עלולה לגרום לאובדן נתונים חשובים שלא גובו).
- Circular wait – כמעט בלתי אפשרי ליישום, ואם כן, לא ניתן להרחבה.

### דוגמאות מטפוריות לdeadlock:

- איתי וליאב, כל אחד בנפרד צריכים לבצע עבודה שחיבר בה גם פטיש וגם את החפירה, איתי תפס את הפטיש וליאב תפס את החפירה.
- מעבר על התנאים לבדוק אם יש כאן deadlock.
- יש כאן mutual-exclusion - כי איתי משתמש בפטיש ורק ליאב משתמש באת החפירה.
  - יש כאן Wait and Hold - כי איתי לוקח את הפטיש ומוחכה גם את החפירה ובאותו אוף ליאב לוקח את האת ומוחכה לפטיש.
  - יש כאן preemption so - נניח שאין כאן בו שיכל לקחת מהם את הכל עבודה וכן התנאי הזה גם מתקיים.
  - רק איתי יכול לשחרר את הפטיש ורק ליאב יכול לשחרר את האת.
  - יש כאן Circular wait – איתי מוחכה למשאב המוחזק ע"י ליאב וליאב מוחכה למשאב המוחזק ע"י איתי (dependency).
  - ולכן נוצר כאן deadlock.

air הדוגמה תראה בקוד?

#### Itai's Code:

```

Take p
Hold-And-Wait
Take m

```

#### Liav's Code:

```

Take m
Hold-And-Wait
Take p

```

### פתרונות אפשריים לבעה:

1. נאחד את המשאים למקטע אוטומי, נשתמש ב mutex.

Itai's Code:	Liav's Code:
Mutex.lock	Mutex.lock
Take p	Take m
Take m	Take p
Make the job	Make the job
Mutex.unlock	Mutex.unlock

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

Itai's Code:	Liav's Code:
Take m (or wait)	Take m (or wait)
Take p	Take p
Make the job	Make the job

כאן לא יוצר deadlock כי הראשון שייקח את האט ייקח גם את k. בנוסף נשים לב שגם אם איטי ייקח ראשון את k והאחר scheduler יכנס בנקודה זו את לייב, לייב יתקע בה כי הוא כבר תפס, ואז הזרור בחזרה לאיטי שייקח את k ויתחיל לבצע את העבודה. בפתרון זה שברנו את העומס circular dependency.

כדי למנוע מצב deadline צריך לוודא שלפחות אחד מ-4 התנאים איננו מתקיים.  
נכזה משאים רק אחרי שנודא שהם "בטוחים" (safe), כלומר המשאים לא יכולים להוביל למצב של deadlock.  
נמצא גרפ מוגלי של תהליכיים וממשאים.  
ונסה לעשות recover ע"י להרוג את התהיליך או לנקות ממנו את המשאב.

### שאלה 1:

יהיו  $R_n, R_2, R_1$  משאים במערכת.  
ונניח שהמשאים ייחודיים – קיימים מופיע היחיד לכל משאב.  
הוכחה שאם ניתן לתהליכים לבקש משאים רק על פי הזמןתם קיימים מצב שבו מקבל deadlock.  
טהיליך P יכול לבקש משאב R<sub>a</sub> בזמן שהוא מחזיק משאב R<sub>a</sub> רק במידה וג>b.

### הוכחה:

נניח שמערכת כוללת תהליכיים  $P_1, P_2$  ומשאים  $R_1, R_2$ .  
נניח כי  $P_1$  מחזיק במשאב  $R_1$  ו  $P_2$  מחזיק במשאב  $R_2$ .  
עכשו  $P_1$  מבקש את  $R_2$  ועכשו הוא ממתין ל  $P_2$  לשחרר אותו.  
כדי שיהיה deadlock במערכת,  $P_2$  צריך לבקש את  $R_1$  אבל הדבר מנוגד להנחה כי ניתן לבקש משאים רק בסדר עולה.  
תנאי 4 (Circular wait) נמנע.

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

נגדיר מצב  $P_j \rightarrow P_i - R_k \rightarrow P_i$  המציין שהטהיליך  $P_i$  מבקש משאב  $R_k$  המחזיק על ידי הטהיליך  $P_j$ .  
כדי שיתרכש deadlock תת קבוצה של  $\{P_1, P_2, \dots, P_m\}$  העומדת בתנאי הבא בהכרח קיימת:  

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

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

ולכן  $J_s < J_{s-1} < \dots < J_1 < J_m < \dots < J_2 < J_1$ .

לכן מקבל  $J_m < J_{m-1} < \dots < J_1 < J_2 < J_1$ .

המצב של deadlock אנו מסיקים  $J_1 < J_m < \dots < J_1$  וכך מתקיים deadlock, אך מכיוון שהה בלאי deadlock-free, deadlock-freewait לא תתרחש, והמערכת היא circular wait.



### 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) נעה מיד את כל הבקשה למערכת? אילו תהליכיים, אם בכלל, עלולים להיות deadlock אם כל הבקשה זו תוענק באופן מיידי?

תשובה:

באו נראה מה עלול לקרות אם הבקשה של  $P_3$  תישא 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

שינונו את A ל  $(0, 0, 2)$  ואת השורה של  $P_3$  בcolumn still need ל  $(2, 5, 6)$ .

עכשו  $P_1, P_4, P_5$  יוכלים להסתיים (terminate).

וקטור 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  $P_3$ .  
תהליכיים  $P_2$  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

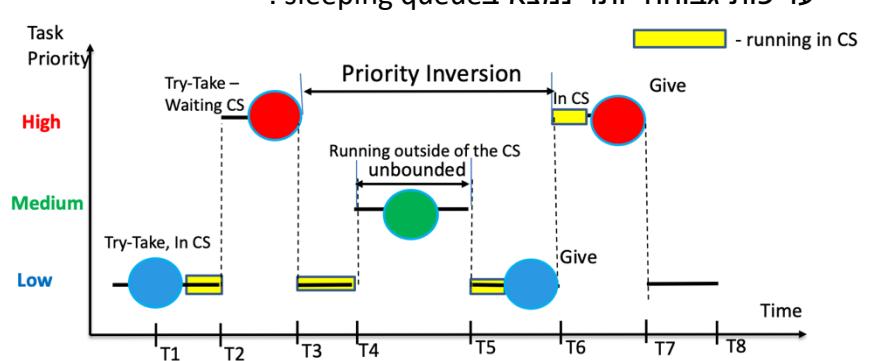
נניח ואנחנו נמצאים במערכת real-time ויש לנו 3 משימות עם priority שונה (low, medium & high). High וmedium High היי משתמשים (נניח שהמתינו ל0/0) ובינתיים low התחליל לעבוד והגיא ותפס את mutex, וכוכנו לCS. בניתוחים high סיים את המתנה ל0/0 והוא רוצה להתחליל לעבוד באותו CS אך mutex נעל ע"י低 low. לכן high נכנס לתור המתנה (sleeping queue) של mutex. בזמן זה medium סיים ורצה להיכנס לשימוש אחרת ומכוון שיש לו עדיפות גבוהה יותר מאשר low CPU החולף בינהם.

נשים לב שהhigh priority הוא unbounded taskswitch כיון שהוא ממליך medium וlow ויתקעו medium priority inversion.

ורק לאחר שהmedium סיים את העבודה, low יוכל ויסיים את העבודה低 וירוץ ויכוכן ל- CS.

שורה תחתונה – priority inversion עם עדיפות נמוכה נמצאת CPU בעוד שתחליל אחר בעל עדיפות גבוהה יותר נמצאת בענין sleeping 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 protocol שנקרא thread ownership ?  
 באמצעות priority inheritance protocol שמיון mutex רק על משאב זה יש ברגע שתפקידו של mutex, במידה והעדיפות שלו גבוהה מהעדיפות של התהיליך שנעל את mutex, מערכת הפעלה מעלה את העדיפות של התהיליך שנעל את mutex עד לרמה של התהיליך שמחכה בתור (boost priority) וברגע שהוא יסייע מושך CS מערצת הפעלה תחזיר לו את העדיפות הקודמת שלו. אם נתיחס לדוגמה הקודמת, בזמן שההight priority נכנס לתוך מערכות הפעלה היה מבצע priority boost low priority ואז medium יכול לא להיות מחליף אותו וכך הhigh priority נכנס לפני הlow.

dagshim:

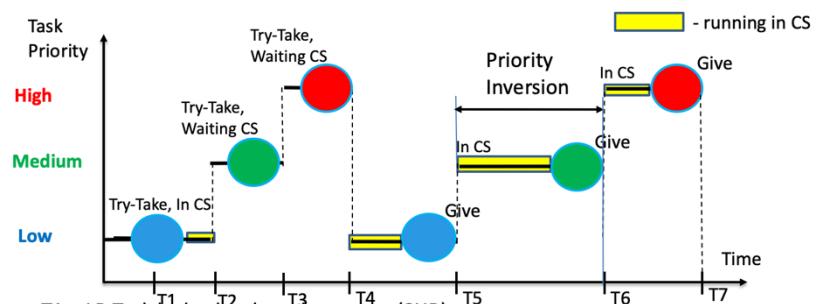
- לא ניתן להשתמש בפרוטוקול זה על semaphore.
- פרוטוקול זה אינו deadlock free.

### - Bounded Priority Inversion

נניח ואנחנו נמצאים במערכת real-time ויש לנו 3 מישימות עם priority שוניה (low, medium & high). medium High הוא העסוקים (נניח שהמתקינו ל/ו) ובינתיים low התחיל לעובוד והגיע ותפס את mutex, ונכון לCS. ביןתיים medium סיים את הרמתנה ולכן context switch מכנים אותו (יש לו עדיפות גבוהה יותר) וכשהוא רוצה להתחל לעובוד באותו CS הוא מגלה שהmutex נעל ע"י low וכאן medium נכנס לתוך המתנה (sleeping queue) של mutex. בזמן זה high סיים את הרמתנה העדיפות שלו גבוהה משל medium וכאן context switch מכנים אותו וכשהוא רוצה להיכנס ל- CS הוא גם מגלה שהוא נעל ולכן גם הוא ממתין. נשים לב שהmedium והhigh נמצאים בשעון FIFO. context switch מחזיר את low עד שהוא מסיים לעבוד ב- CS, medium מקבל זמן בCPU עד שהוא מסיים ולאחר מכן high נכנס.

priority inversion מסווג זה הוא bounded כי high ממתין זמן מוגבל (לכל מי שלפניו תור) ולעומת זאת ב-unbounded תהליכיים נוספים יכולים להגיע ולהיכנס לפניו ולכך זמן הרמתנה 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 – HP-Task is ready. Context switch to HP-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 מושימה בעדיפות גבוהה יותר מבקשת את אותו משאב, עדיפות הביצוע של התהיליך שמחזק את המשאב מוגברת (boosting) לרמת העדיפות של המשימה המבקשת. המשימה חוזרת לעדיפות הקודמת שלה כשהיא משחררת את המשאב. העיה: פרוטוקול זה אינו deadlock-free.

## – Ceiling Priority Protocol (CPP)

ישן שתי גרסאות לפרוטוקול:

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

פרוטוקול זה מונע bounded Priority Inversion (מצב של task שולב task עם deadlock).

הגדרות הפרוטוקול:

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

B. (i) Task עם עדיפות מסוימת יכולה לנעול (lock) semaphore רק עם העדיפות של task הגבוה מהceil של כל semaphoreים שנעולים כרגע ע"י task אחרים.

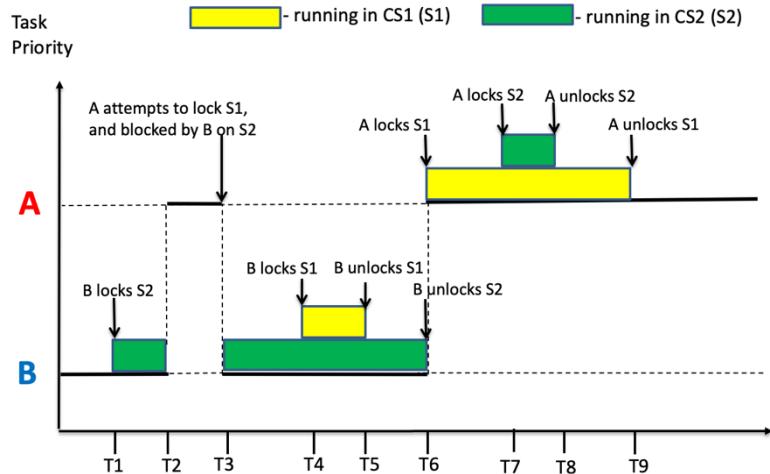
C. אם B לא נוכח, task תיהיה נעולה על ידי semaphore אחר \* S (ceil(S) > ceil(Semaphore)).

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

דוגמא:

Task Name	Time	Priority	Action	Sem. Ceiling
A	50	3	lock (S1) lock (S2) ... unlock (S1) unlock (S2)	ceil(S1) – 3 ceil(S2) – 3
B	500	2	lock (S2) lock (S1) ... unlock (S1) unlock (S2)	

נתונות 2 משימות A וB, בעלי זמני ביצוע שונים ועדיפויות שונות ולכל אחת יש משימה אחרת. כאשר ceil של כל אחד משני semaphoreים שקיימים במשימות מוגדרים להיות העדיפות של A כיוון שגם עם task זהו עם העדיפות הגבוהה ביותר שימושת בכל אחד מהsemaphore'ים.



הסבר: B מנעה את S<sub>2</sub> ומצליח כיוון שאף משימה אחרת לא מחזיקה אותו.

לאחר מכן משימה A נכנסה והזא עובר אליה כיוון שהעדיפות שלה גבוהה יותר, היא מנסה לנעול את S<sub>1</sub> אבל לא מצליחה כיון שהיא 3 היא לא גדולה ממש מהceil של כל semaphore מהceil(S<sub>2</sub>) שנוולים כרגע ע"י task אחרים (במקרה זה S<sub>2</sub> נעולה ע"י B והוא ceil(S<sub>2</sub>)). ולכן A מנעה את S<sub>1</sub> ומצליח כיוון שאף משימה אחרת לא מחזיקה אותה.

הזא עובר לB שמשיך לעבוד ולאחר מכן מנסה לנעול את S<sub>1</sub> ומצליח כיוון שאף משימה אחרת לא מחזיקה אותה. נשים לב שזמן העבודה של B בתוך S<sub>1</sub> זה בדיקת inversion bounded priority.

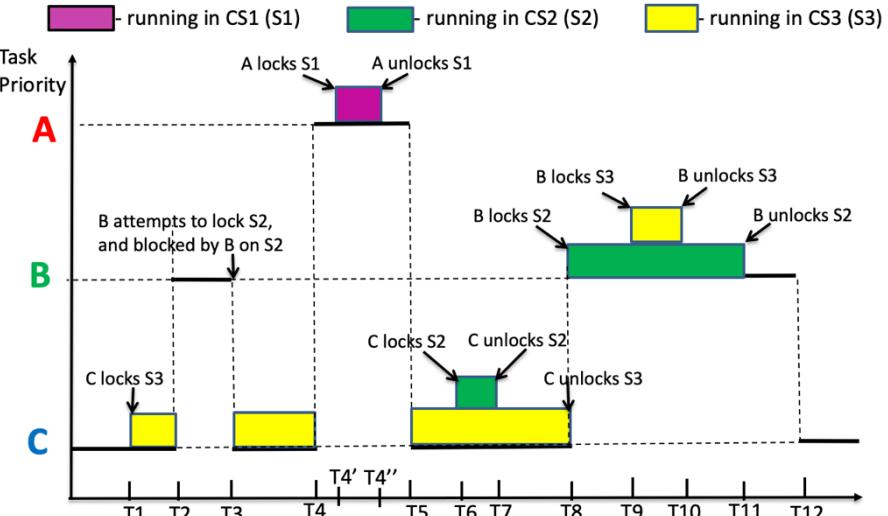
B משחרר את הנעילה על S<sub>1</sub> ולאחר מכן גם על S<sub>2</sub> והעדיפות של B מתעדכנת בחזרה ל-2. בcut A מקבל זמן CPU לבצע את העבודה שלו (מנעל את S<sub>1</sub> כיוון שאף אחד לא מחזיק אותו וככל לגבי S<sub>2</sub> משחרר אותו ומסיים).

### דוגמה נוספת:

Task Name	Time	Priority	Action	Sem Ceiling
A	50	3	lock (S1) ... unlock (S1) ...	ceil(S1) - 3 ceil(S2) - 2 ceil(S3) - 2
B	500	2	lock (S2) ... lock (S3) ... unlock (S3) ... unlock (S2) ...	
C	3000	1	lock (S3) ... lock (S2) ... unlock (S2) ... unlock (S3) ...	

נתונות 3 משימות A, B ו C בעלי זמני ביצוע שונים ועדיפות שונות וכל אחת יש משמעות אחרת. כאשר ceil של כל אחד משלושת semaphore'ים שקיימים במשימות מוגדרים להיות העדיפות הגבוהה ביותר מ בין כל המשימות שימוש 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**

נ噙ת פילוסופים יושבים סביב שולחן עגול וכולם רוצים לאכול. כדי להתחיל לאכול הם צריכים להחזיק 2 צ'ופסטיקים (forks), כאשר אין מספיק לכולם.

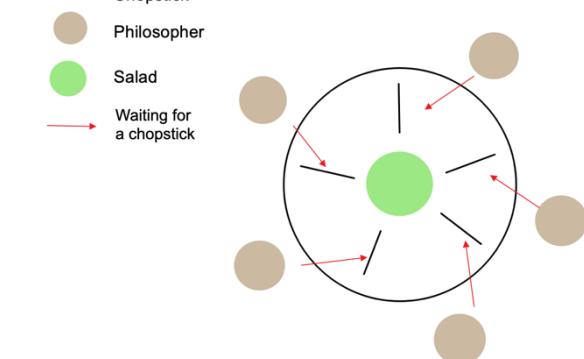
באյור ניתן לראות שאם כל אחד מהפילוסופים יקח את הfork שמייננו קודם ונגיע למצב של deadlock כיוון שכל הפילוסופים נמצאים במצב של Hold & Wait.

פתרונות: Textbook

נגדיר mutex וmutex semaphore עבור כל אחד מהפילוסופים. כאשר פילוסוף ירצה לאכול יunlock את mutex וינסה לחתת 2 מזלות וינסה לחתת הוא לו מזלו לחתת הוא נכנס למשך המתנה וברגע שהמזלות משני הצדדים שלו יתפנו את mutex, במידה ולא היו לו מזלו לחתת הוא נכנס למשך המתנה וברגע שהמזלות משני הצדדים שלו יתפנו יעירו אותו והוא יתחל לאכול.

ניתן לראות בקוד שאנו מגדירים N סועדים, 3 סועדים ימני ושמאלי של כל אחד מהסועדים ו3 מצבים שבהם סועד יכול להיות (חושב – ממתיין, רעב – ינסה לחתת מזלות ואוכל). מגדירים מערך של N semaphore'ים (אחד לכל סועד), mutex מרCCI ומערך של סועדים שמחזיק את המצב שבו נמצא כל סועד.

הפונקציה philosopher() מוגדרת כך: תחשב (זמן המתנה מסוים), תנסה לחתת מזלות, תאכל ותחזיר את המזלות.



```

#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;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i)
{
    while (TRUE) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
  
```

```

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]);
    }
}

```

```

/* i: philosopher number, from 0 to N-1 */

/* enter critical region */
/* record fact that philosopher i is hungry */
/* try to acquire 2 forks */
/* exit critical region */
/* block if forks were not acquired */

/* i: philosopher number, from 0 to N-1 */

/* enter critical region */
/* philosopher has finished eating */
/* see if left neighbor can now eat */
/* see if right neighbor can now eat */
/* exit critical region */

```

הfonקציה (**take\_forks**) מוגדרת כר' תנען את mutex עדכןiani שמי במצב רעב, תבדוק האם אני יכול ללקחת מזלות (test), תשחרר את mutex ותבצע (down) את semaphore שלו.

מתי semaphore שלו ננען? הfonקציה (test) בודקת שאנו אכן במצב רעב והמזלות מימני ומשמאלי פנויים (אם הסעודים מימני ומשמאלי לא במצב "אוכל" אז המזלות פנויים), במידה והם פנויים במצב שלו מתעדכן ל"אוכל" ואני מבצע (up) semaphore שלו.

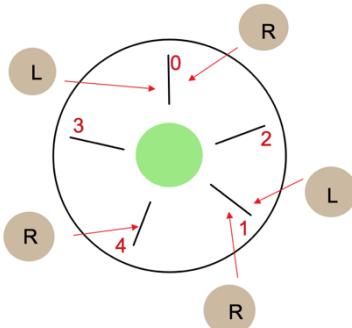
במידה והמזלות לא פנויים לא יבוצע (up) semaphore וכתוצאה לכך הסעוד ית��ע ב(down).

בfonקציה (**put\_forks**) תחילת נועלם את mutex, מעדכנים את המצב ל"חושב" כיוון ששימיתי לאוכל, ושולחים לפונקציה () test את השכן הימני והשמאלי שלנו – פה טמון הרעיון העיקרי של הקוד.

כיוון שהסעוד שחרר את המזלות שלו במידה ואחד השכנים שלו "תקוע" וממתין למזלות הוא "עיר" אותו כיוון שהוא יעדכן אותו במצב "אוכל" ויבצע לו (up) ולכן הוא יוכל לבצע (down) בfonקציה () take\_forks() לשחרר.

בסוף הפונקציה () put\_forks נשחרר את mutex.

היתרון בפתרון זה הוא שהוא מונע starvation Ciou שטמיד מישחו עיר אותו כשיתפנו המזלות. החסרונו בפתרון זה הוא שבמצב של המונ פילוסופים יהיו לנו התנגשיות והמנ פילוסופים שיתקעו בגל mutex המרכזית (צבוני מאוד).



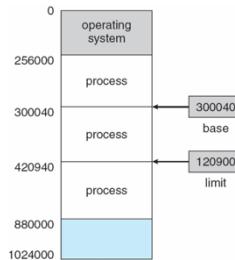
פתרון LR: כאן לא משתמש בmutex מרכזית ולכן פתרון זה הוא פחות צבוני. נגדיר שני סוגים של פילוסופים, ימניים ושמאלים. סוד ימני נסה ללקחת קודם את המזלג שמייננו והשמאלי ינסה ללקחת תחילת את המזלג שמשמאלו. נקצת לפילוסופים את היד (שמאלי או ימני) אחד אחריו השני לחילופין (אחד שמאל, אחד ימני וכן הלאה). מספר מזלג שירוטי להיות 0, ונמספר את השאר כמתואר בתמונה. וכך כל פילוסוף יוכל ל千古 מזלג בסדר עולה (או יורד).

אלגוריתם זה הוא גם starvation-free Ciou שאם פילוסוף A ממתין למשאב המוחזק ע"י פילוסוף B, אז כאשר B שחרר את המשאב הוא לא יוכל לתפוס אותו שוב לפני שא' יקבל אותו. מכיוון שהאלגוריתם הוא deadlock-free, פילוסופים ימשיכו ללקחת ולשחרר משאבי, בסופו של דבר B ישחרר את המשאב Ai יוכל להשיג אותו. בנוסף אנו לא צריכים להניח שהמזלות הם "הוגנים" (fair) Ciou שבל זמן נתון רק פילוסוף אחד מכחילה למשאב.

## זיכרון (memory) –

בעבר מתקנות הווינטוס היו כוכבים תוכנה לכתובות פיזיות (physical addresses) וברגע שניתנו להריץ את התוכנה במחשב אחר הכתובות זיכרון שם היו שונות ולכן התוכנה לא הצליחה לרש ציון שכתובות פיזיות תלויות בגודל הזיכרון (RAM size), סוג הזיכרון (RAM device) ובמעבד (CPU).

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



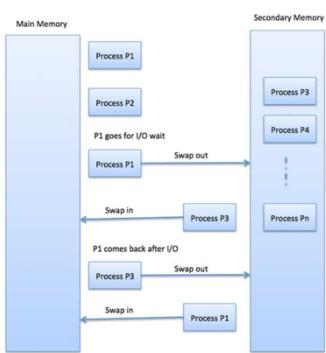
צמד register בסיס ומגביל שמנגנים את שטח הכתובות עבור התהיליך חסרון:

1. נצטרך לדעת מראש את כמות הזיכרון שהPROCESS צריך.

2. לנוול את כל processes במקביל בזיכרון – לא ניתן להכיל את כל התהיליכים בזיכרון.

3. המעבד צריך לבדוק כל ניסיון של התהיליך לגשת לזכרון כדי לוודא שהוא עומד במגבלתו שלו.

### פתרון 2 : swapping



מנגן שבו ניתן להחליף (להעביר) תהיליך באופן זמן מהזיכרון הראשי לאחסון משני (דיסק) ולהפוך את הזיכרון לזמן לתהיליכים אחרים. בשלב מאוחר יותר, המערכת מחליף את התהיליך מהאחסון המשני לזכרון הראשי. אולם הביצועים מושפעים בכך כל מתהיליך החלפה אך הוא מסיע בהפעלת מספר רב של תהיליכים גדולים במקביל וזה הסיבה שהחלפה ידועה גם כטכנית לדחיסת זיכרון.

חסרון:

1. הזמן שלוקח החלפה כולל את הזמן שלוקח להעביר את התהיליך כולו לדיסק משני ואז להעתיק את התהיליך חזרה לזכרון - פעולה כבדה לתהיליך שיש לו שימוש גדול בזיכרון.

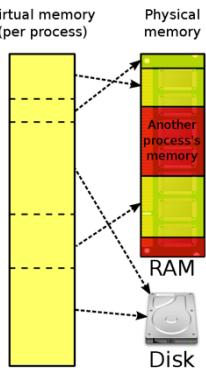
2. לא ניתן לדעת מראש כמה זיכרון התהיליך צריך.

3. במהלך הזמן, החלפה יוצרת "חרום" של כמויות קטנות של זיכרון זמן במאפיין זיכרון.

### Paging and Virtual Memory

לבסוף כדי לפתור את הבעיה הומצא מנגן Hardware Independent Memory, המנגנון ממיר את הכתובות הפיזיות שבזיכרון לכתובות וירטואליות (addressing) וכן המתכוון לעבוד מול כתובות וירטואליות ולא פיזיות כאשר:

$\text{Virtual address} = \text{Physical address} + \text{Normalization offset}$   
היתרון המשמעותי הוא שנוכל לתרום במרחב כתובות שאינו תלוי בזכרון הפיזי (במחשב של 32 סיביות יש  $2^{32}$  כתובות במרחב הכתובות הוירטואלי).



### :Page

כל תהיליך מקבל מהזיכרון הראשי גוש זיכרון וירטואלי באורך קבוע, המתוארך על ידי ערך ייחיד בטבלת העמודים (page table). זהה ייחידת הנתונים הקטנה ביותר לניהול זיכרון במערכת הפעלה של זיכרון וירטואלי. באופן דומה, מסגרת עמודים (page frame) היא גוש זיכרון הפיזי הצמוד ביותר באורך קבוע, אליו ממופה דף זיכרון על ידי מערכת הפעלה.

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

מערכת עמודים בגודלים קטנים משתמשת ביוטר עמודים, ודורשת טבלת עמודים שתווסף מקום רב יותר. לדוגמה, אם  $2^{32}$  כתובות וירטואליות ממופות לעמודים של 4KB ( $2^{12}$  bits),

מספר העמודים הוירטואליים הוא  $= 2^{32} / 2^{12} = 2^{20}$ .

עם זאת, אם גודל עמוד גדול ל-8KB ( $2^{15}$  bits) נדרש רק ל-  $2^{17}$  עמודים.

### :paging

מערכת הפעלה של המחשב, באמצעות שילוב של חומרה ותוכנה, ממנה את הכתובות הזכרון המשמשות תוכנית, הנקראות כתובות וירטואליות, לכתובות פיזיות בזכרון המחשב. אחסון ראשוני, כפי שנראה על ידי תהיליך או משימה, מופיע ככתובות רציף או zusätzlich של פלחים רציפים. מערכת הפעלה מנהלת רוחני כתובות וירטואליים והקצת זיכרון אמיתי (פיזי) לזכרון וירטואלי.

## Page Table

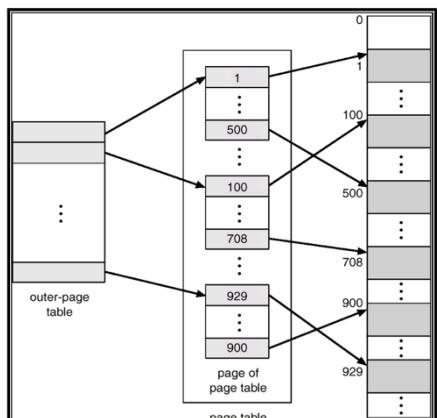
טבלת העמודים היא מבנה נתונים המשמש מערכת זיכרון וירטואלית לאחסון המיפוי בין כתובות וירטואליות לכתובות פיזיות. טבלת העמודים היא מרכיב מרכזי בתרגום כתובות וירטואליות אשר הכרחי לטובת גישה לנוטונים בזיכרון הראשי.

כיצד נוכל להתמודד עם טבלאות גדולות מדי? ישנו מספר תוכנות ומכרזים שצרכיהם להתקנים מלבני קשור לגודל הטבלה. פתרון קיומי אחד הוא שכל טבלאות המיפוי יהיו בחומרה, זה לא אפשרי מכיוון שאמנם הגישה מאוד מהירה אבל זה יקר מאוד, במיוחד עבור טבלאות גדולות, פתרון קיומי אחר הוא להזניק הכל בזכרון הראשי, זה יהיה מוגן עם המונט מצביעים וregisterים וזה גם יקר מאוד כי אז אנחנו מכפילים כל reference reference לדיזכרון. וכן נבצע paging על ה-table page עצמו.

שים קולים לגבי טבלת העמודים:

- יכולה להיות גדולה מאוד – מיליון דפים עברו 32bit בגודל 4KB.
- חייבת להיות מאוד מהירה.

כדי להימנע מלהשמור טבלאות דפים שלמות בזכרון - נהפוך אותן לריב-שכבותיות, ונימנע מההפנות לזכרון הראשי בכל הוראה על ידי שמירה במתמון (cache).



page number	page offset
$p_1$	$p_2$
10	10

12

:Two Level Paging במקום להזניק מערך רציף עם המונט כניסה (entrees) של כל הכתובות הירטואליות נחזיק outer-page table שציביע למערכים של ה-table page וכך לא כל החלקים שבטבלה החיצונית יהיו מאוכסלים אלא רק החלקים שבתפסה. כתובת לגות במכונת 32bit עם 4K גודל עמוד מחולקת ל:

מספר העמוד המורכב מ-20bit offset, 12bit המורכב מ-10bit מספר העמוד מ-10bit עבור הטבלה החיצונית ול-10bit 2 של עבור הטבלה הפנימית.

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 נוצרבים במקרה זה (יתכן ולמשל ה-table code segment לא יופיע בתוכף בזיכרון ועוד נוצרך יותר טבלאות פנימיות).

### טבלאות דפים הפוכות

– Inverted page tables :Page table רגיל לא פרקטני עבור מוכנות 64bit

4K page size /  $2^{52}$  pages x 8 bytes → 30M GB page tables!

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

1 GB of RAM & 4K page size → 2 MB table

טבלת העמודים הפוכהichert משמשת לכל התהליכים הנמצאים כעת בזכרון.

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

משתמשים בתаблицת hash table כדי למנוע חיפוש לינארי לכל דף וירטואלי.

בנוסף לתаблицת hash table, רשומי TLB משמשים לאחסון ערכי טבלת העמודים שהשתמשו בהם לאחרונה (cache).

הערה: אם הטבלה משותפת לכל התהליכים, יתכן מצב שני תהליכיים ממפים בטבלה לאוטו קטע קוד (רצים על אותה תוכנית למשל), כדי שכל תהליך מיפוי יידע מי מבצע את השאלתה הזאת, אנחנו נגיע לטבלה עם שני משתנים: מדחה תהליך (pid) וכותבת 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:  $2^{(64 - 12)} = 2^{52}$ 

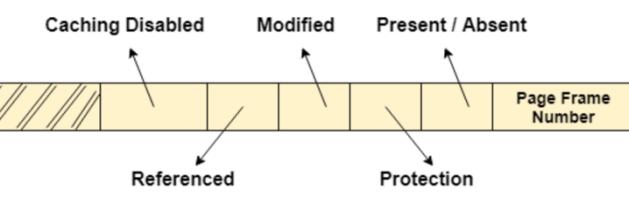
# of pages in the memory: 4GB / 4KB = 1M → addr. Of page in mem. Requires 20bits.

Total size:  $(2^{52}) * 2^6$ 

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**

- 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 עצמה, ובמצעת בעיקר תרגום כתובות זיכרון וירטואלי לכתובות פיזיות.

**– Translation Lookaside Buffer (TLB)**

מאג'ר מבט לתרגומים (TLB) הוא מटמן זיכרון המשמש להפחתת הזמן שלוקח לגשת למיקום זיכרון של המסתמש. זהה חלק מיחידת ניהול הזיכרון (MMU), TLB מ&actionן את התרגומים האחוריים של זיכרון וירטואלי לזכרון פיזי. למעשה אפשר לקרוא לו מטען לתרגום כתובות.

תהליך :resolving

1. תהליך kernel צריך לתרגם כתובות וירטואליות.

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

3. במידה והכתובת לא קיימת בTLB:

3.1 אם הכתובת חוקית, כלומר קיימת בTable Page Table:

3.1.1 נבצע PT Walk נמצאת הכתובת ונעדכן את TLB.

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

3.2.1 במידה זו שגיאת Soft: זה אומר שהכתובת קיימת אך ממופה לתהליך אחר, נמפה את הדף עבור התהליך הנוכחי ונעלה ב1 את ה usage counter.

3.2.2 במידה זו שגיאת Hard: זה אומר שהכתובת כלל לא קיימת, נבצע אלוקציה, נמפה את הדף ונכניסו לpage table.

הערה חשובה: לעיתים נדרש ליצור עותק של TLB באופן לוקאלי (לאחר עדכון TLB), הסיבה שיוצרים עותק לוקאלי בرمת ה process היא שכאשר מtbody flush cache context switchTLB השוכן והprocess החדש שנכנס מחליף את TLB שב MMU לשול TLB.

שאלות:

- 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?**

פתרונות:

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.

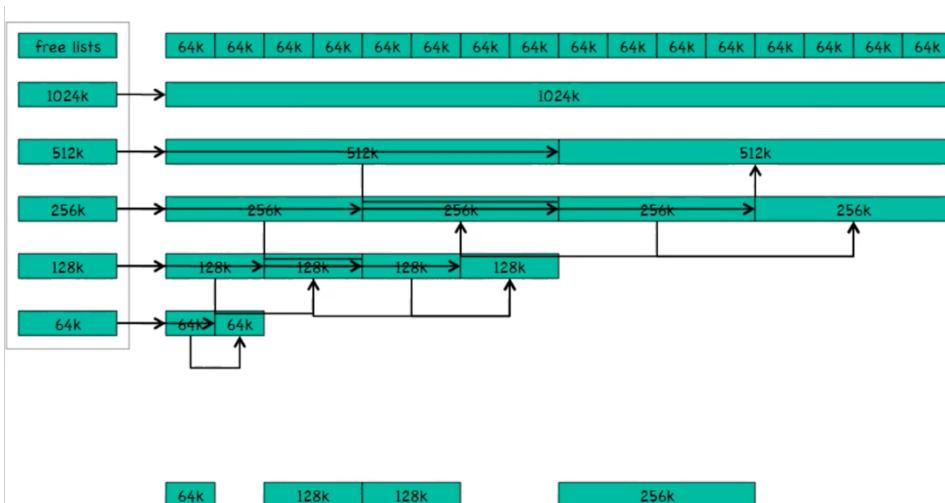
The number of pages in the system is  $2^{32}/x$ .

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 פניה למספריה שנקראת GLIBC (זו לא ספרייה בkernel), במידה ויש מספריה זיכרון היא תחזיר אותנו, אחרת, היא מבקשת מהגרעין בזיכרון הבאה: היא קוראת לפונקציה (`mmap2`) שהיא מחזירה את הגודל שהוקצה ביחידות של `page`. לאחר מכן מתבצעת לוגיקה של Knuth's System Call כדי להקצתות מתוך הדפים כתובות נצרכות. GLIBC מוחזיר רק את כמות הכתובות שנדרשו ע"י `Buddy Allocator`, ושומר את יתר עברו הקריאות הבאות.()



### – Knuth's Buddy Allocator

טכניקת הקצתה זיכרון של Buddy היא אלגוריתם של הקצתה זיכרון המחלק את הזיכרון למחיצות כדי לננות לספק בקשה זיכרון בזיכרון המתאימה ביותר. מערכת זו עשויה שימוש בפיצול הזיכרון לחיצאים כדי לננות לחתך זיכרון בזיכרון הטובה ביותר.