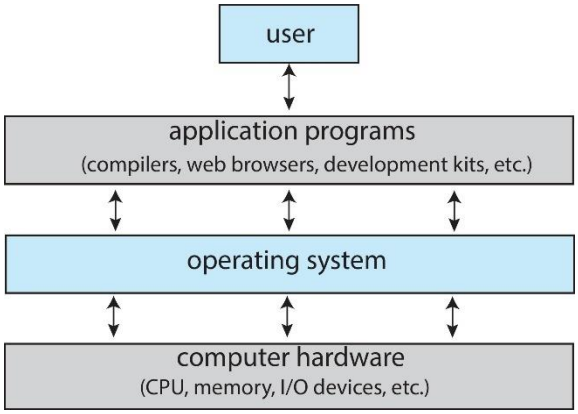
בר זמסקי

מערכות הפעלה – סיכום החומר

מערכת מחשב מחולקת ל4 מרכיבים:

1. חומרה – מספקת משאבי מחשב בסיסיים: memory, I/O devices,CPU
2. מערכת הפעלה – שולטת ומתאמת שימוש בחומרה עם מגוון רחב של אפליקציות ומשתמשים
3. אפליקציות – מגדירות את הדרך שבה משומשים משאבי המערכת לפתירת בעיות של המשתמשים – word processors, compilers, web browsers, database system, video games
4. משתמשים – אנשים, מכונות ומחשבים אחרים



מה עושה מערכת ההפעלה?

מערכת ההפעלה היא תוכנה שמטרתה לנהל את החומרה והתוכנה במחשב, אחראית על תזמון תהליכים, לגשר בין החומרה לתוכנה ומהווה תשתית משותפת לתוכנות.
מערכת ההפעלה היא זו שעושה אבסטרקציה וכך מתאימה הכול לכל סוגי החומרה.
לרוב **device driver** יהיה התוכנה המקשרת בין החומרה למערכת ההפעלה.
כיום, מערכות ההפעלה באופן כללי כוללות בתוכן גם middleware- קבוצה של תוכנות שלד שמספקות שירותים נוספים למפתחי האפליקציות כמו דאטהבייסים, מולטימדיה, גרפיקה וכו'...
Kernel- תוכנה יחידה שרצה כל הזמן על המחשב והיא ליבת מערכת ההפעלה, הגרעין. מתעוררת בעקבות אירועים.

פעילות מערכת המחשב:

התקני I/O וה-CPU יכולים להתבצע בו זמנית. כל התקן בקרה שולט על סוג התקן ייחודי. יש לו באפר מקומי. כל סוג התקן בקרה מנוהל ע"י מנהל התקן(device driver) ממערכת ההפעלה.
CPU מעביר דאטה מ/אל הזיכרון הראשי אל/מ הבאפר המקומי. I/O מעביר מההתקן לבאפר המקומי של הקונטרולרס.
מנהל ההתקן מודיע לCPU שהוא סיים את הפעולה שלו ע"י יצירת **interrupt** = אות שמגיע מרכיב תוכנה כלשהו למעבד שמטרתו לשנות את סדר הפעולה הסדיר של המעבד ע"מ לטפל בה.
הוא מעביר את השליטה ל"שירותי הפסיקה" דרך ווקטור הפסיקה שמכיל את הכתובת של כל השירותים שבשגרה. הארכיטקטורה שלו מחייבת אותו לשמור את הכתובת של ההוראה שהופסקה. בנוסף, אחראי גם להעיר את מערכת ההפעלה.

סוגי Interrupt:

- סינכרוני – המעבד מייצר (למשל כשאפליקציה רוצה להעיר את הקרנל)
- א-סינכרוניים – מגיע מחומרות אחרות שלא מסונכרנות עם המעבד

<u>טיפול ב interrupt:</u>
מערכת ההפעלה שומרת את המצב של **ה**CPU ע"י כך שהיא מאחסנת רגיסטרים וסופר תוכניות.
צריך להכריע איזה סוג פסיקה הופיעה -
Polling- לתשאל את החומרה מי עשה ואז לבדוק מה ולעשות מה שצריך
Vectored- בפסיקה יש את כל המידע ואין צורך לתשאל

- חלקים נפרדים בקוד קובעים איזה פעולה צריכה להיעשות עבור כל סוג פסיקה

<u>מושגים:</u>
caching- העתקת מידע למערכת אחסון מהירה יותר. הזיכרון הראשי יכול להיות מוצג בcache של הזיכרון המשני.
Multiprogramming (batch system) - נצרך לשיפור היעילות. משתמש יחיד לא יכול להחזיק את ה**CPU** וה**I/O** עסוקים כל הזמן. "מתזמן" את כל הקודים בכדי שמערכת ההפעלה תתעסק בדבר אחד כל פעם. זו למעשה תת קבוצה של עבודות במערכת ששמורות בזיכרון. עבודה נבחרת לרוץ ע"י המתזמן. כשאר עבודה נאלצת לחכות למשהו (נגיד ל**i/o**) מ"ה מחליפה לעבודה אחרת.

Timesharing (multitasking)- שלוחה לוגית שבה הCPU מחליף עבודות בתדירות גבוהה כך שהמשתמש יכול לתקשר עם כל עבודה בזמן שהיא רצה. זמן התגובה צריך להיות פחות משנייה. לכל משתמש יש לפחות תוכנה אחת שרצה בזיכרון (תהליך). אם כמה עבודות מוכנות לרוץ באותו זמן זה עובר למתזמן.
Dual mode- תהליך המאפשר למ"ה להגן על עצמה ועל מרכיבי תוכנה אחרים. אם יש אפליקציה שנרצה להגביל, החומרה תעשה זו ותפריד לפחות בין שתי רמות של עבודה:

- **User mode**: כל דבר שהאפליקציה רוצה לעשות חייב לעבור דרך מערכת ההפעלה
- **Kernel mode**: אין שום הגבלה

Mode bit- מסופק ע"י החומרה. אומר לנו באיזה רמה נמצא המעבד.
System call- הדרך התכנותית של התוכנה לבקש ממ"ה לבצע שירות מסויים. בדר"כ תגרור פסיקה. זהו מהלך איטי ולכן ננסה לבצע כמה שפחות.
Device driver עבור כל התקן בקשה לשליטה בs/i – מספק ממשק ייחודי בין הבקר לקרנל.
Process management- ניהול תהליכים. ככל שיש יותר תהליכים-יותר עומס ולכן יש צורך לנהל אותם. הוא מאפשר סנכרון בין התהליכים ותקשורת ביניהם.

<u>פעולות מערכת ההפעלה:</u>
תוכנת **bootstrap**- קוד פשוט לאתחול המערכת, טוען את הקרנל
כאשר הקרנל נטען, הוא טוען את עצמו ורכיבים נוספים מחוץ לליבה המסייעים למ"ה לנהל את המערכת. לאחר מכן, כל האפליקציות רצות כל הזמן. לעומת זאת, את הליבה לא נרצה להחזיק דולקת. הקרנל גורם לעצמו להפסיק לעבוד, הוא יחזור לפעולה ע"י ה**kernel interrupt driven**(חומרה או תוכנה): החומרה מופסקת ע"י אחד השירותים, התוכנה מופסקת ע"י שגיאה בתוכנה(חלוקה באפס), קריאה ל**system call** ובעיות אחרות כמו לולאות אינסופיות וכו'..

<u>מעבר בין user mode לkernel mode:</u>
כשאר המשתמש רוצה לגשת לרכיב **I/O** הוא יבקש מהקרנל. היא תעשה את המוטל עליה ותחזור חזרה למשתמש היות ואין לו גישה לs/i.
ניעזר בטיימר ע"מ למנוע לולאה אינסופית / תהליך שמחזיק משאבים סתם

- הטיימר מוגדר להפריע למחשב אחרי זמן סופי כלשהו
- שומרים קאונטר שיורד ע"י השעון הפיזי. מאותחל ע"י מ"ה
- כשהקאונטר על 0 נוצר interrupt
- הוא מוגדר לפני התהליך הבא כדי להבטיח שליטה או כדי לסיים תוכנית שעברה את הזמן המוקצה לה

Process Management:
תהליך הוא תוכנית בזמן ריצתה. זו יחידת עבודה בתוך המערכת. תוכנית היא ישות פסיבית לעומת תהליך שהוא ישות אקטיבית. תהליך צריך משאבים כדי להשלים את המשימה שלו – זיכרון, CPU, קבצי I/O ואתחול מידע. סיום תהליך דורש למעשה לבקש בחזרה כל משאב שהוא זמין לשימוש חוזר.
לתהליכים שהם single-threaded יש קאונטר המציין מיקום של הפעולה הבאה שיש לבצע. (תהליך מבצע פקודות באופן סדרתי, אחת בכל פעם).
לתהליכים שהם multi-threaded יש קאונטר אחד לכל טרד.
בדר"כ למערכת יש הרבה תהליכים, חלק של משתמשים וחלק של מ"ה

מ"ה אחראית על הפעולות הבאות ביצירת קשר עם מנהל התהליכים-

- יצירה ומחיקה של תהליכי המשתמש ותהליכי המערכת
- השהייה וביטול השהייה של תהליכים
- מספקת מכניקה עבור סינכרון תהליכים
- מספקת מכניקה עבור תקשורת בין תהליכים
- מספקת מכניקה לניהול מצבים של dead-lock

Memory Managment:
על מנת לבצע תוכנית, כל (או חלק) מההוראות צריכות להיות בזיכרון. כל(או חלק) מהמידע שהתוכנית צריכה חייב להיות בזיכרון.
ניהול זיכרון מכריע מה נמצא בזיכרון ומתי.
פעולותיו: עוקב אחרי איזה חלק של הזיכרון נמצא בשימוש כרגע וע"י מי. מחליט מתי תהליך ומידע יעברו אל מחוץ לזיכרון ומקצה ומשחרר חלקים בזיכרון כפי שצריך.

File-system Managment:
מ"ה מספקת צפייה לוגית ואחידה במידע המאוחסן בה. הגישה לקבצים מתבצע דרכה.
קבצים בדר"כ מאורגנים בתיקיות. בקרת השליטה אחראית לקבוע מי יכול לגשת ולאן.
פעולות מ"ה כוללות:

- יצירה ומחיקה של קבצים ותיקיות
- מניפולציה על קבצים ותיקיות
- מיפוי קבצים לאחסון משני
- גיבוי קבצים לאחסון מדיה

Caching:
זהו רכיב חומרתי שאומר אם פניתי למידע בזיכרון, יכול להיות שארצה להשתמש בו שוב אז אשמור אותו ב**cache** בשביל לחסוך גישות לזיכרון וכדי להאיץ את העבודה. Cache הוא חומרתי כלומר הוא לא מימוש אלא עיקרון חשוב המתבצע ברמות שונות בהיררכיה במחשב (חומרה, תוכנה, מ"ה)
המידע שבשימוש מועתק מאחסון איטי למהיר באופן זמני.
האחסון המהיר (cache) נבדק ראשון כדי לקבוע אם המידע שמה- אם הוא שם, נשתמש במידע ישירות משם. אחרת, המידע מועתק לcache ומשומש שם.
בשביל להחליף רכיב ולהכניס חדש יש צורך בניהולה של מ"ה

I/O Subsystem:
מטרה אחת של מ"ה היא להסתיר מוזרויות של התקני החומרי מהמשתמש.
למעשה זה כל הטיפול בס/i.
מערכת זו אחראית על:

- ניהול זיכרון ה**i/o** כולל באפרים (המאחסנים מידע באופן זמני), caching(מאחסנים חלקים של דאטה באחסון מהיר עבור ביצועים), spooling (הפעולה של חפיפה בין אאוטפוט של עבודה אחת לאינפוט של אחרות).
- ממשק device-driver כללי
- דרייברים של התקני חומרה ספציפיים.

Protection and Security:

Protection- כל מכניקה של שליטה בגישה של תהליכים או משתמשים למשאבים המוגדרת ע"י מ"ה

Security- מגינה על המערכת מפני מתקפות פנימיות וחיצוניות

- מ"ה מגינה על מתן הגישות וההרשאות, החל מלשמור שלא יתפסו משאבים ממקומות אסורים ועד תהליך שמנסה לגעת בחומרה.

המערכת בדר"כ מבחינה בין משתמשים כדי לקבוע מי יכול לעשות מה-

- **User ID** כולל שם ומספר חח"ע המקושר למשתמש
- ה**user id** מתקשר עם כל הקבצים והתהליכים של משתמש זה כדי לקבוע שליטת גישה.
- **Group id** מאפשרת לקבוצה של משתמשים להיות מוגדרים ולשלוט בניהול. הם גם מקושרים לכלל התהליכים והקבצים הרלוונטיים
- **Privilege escalation** מאפשר למשתמש לשנות את הID לאחד עם יותר זכויות

Virtualization:

מאפשרת למ"ה להריץ אפליקציות בעזרת מ"ה אחרות. (מכונה וירטואלית)

מ"ה באופן טבעי מקומפלת עבור CPU, כך גם מ"ה האורחות שרצות.

- **VMM (=virtual machine manager)**: מספקת שירותי וירטואליזציה. יכולה לרוץ בצורה טבעית שבצורה זו היא גם ה**host**.

Kernel Data Structures:

כמערכת זה נראה מורכב אך לרוב מדובר ברשימות מקושרות. צריך להבין מה מבנה הנתונים ומה האלגוריתמיקה שנרצה עליו.

Operating System Services:

מ"ה מספקת סביבה עבור הרצת תכניות ושירותים לתוכנות ומשתמשים.

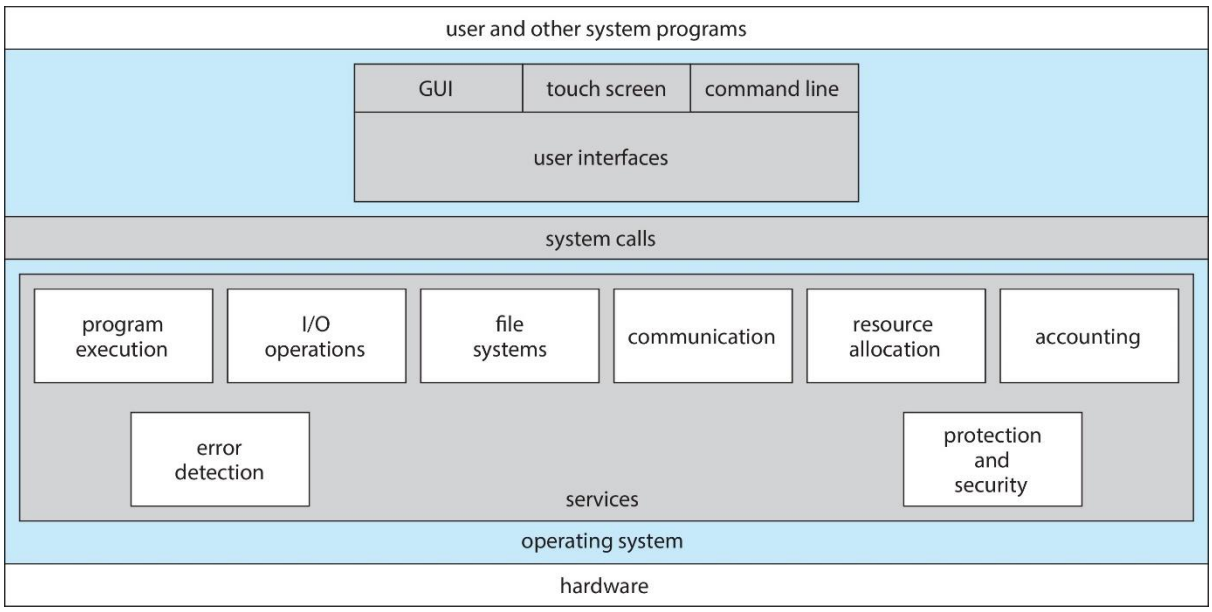
קבוצה אחת של שירותי מ"ה מספקת פונקציות שהן שימושיות למשתמש:

- (User interface) UI: כמעט לכל מ"ה יש ממשק משתמש (נבחין בין CLI, GUI, touch screen)
- ביצוע תוכנית: המערכת חייבת להיות יכולה לטעון תוכנית לזיכרון ולהריץ אותה
- תהליכי I/O: תוכנית במהלך ריצתה יכולה לבקשת I/O, שיכול לערב קובץ או התקן I/O.
- מניפולציות על מערכת ניהול הקבצים: תכניות צריכות לקרוא ולכתוב לקבצים ותיקיות, ליצור ולמחוק אותם, לחפש בהם, לערוך ברשימות וכמובן הרשאות ניהול
- תקשורת: תהליכים עלולים להחליף מידע על אותו מחשב או בין מחשבים ברשת. תקשורת יכולה להיות דרך זיכרון משותף או דרך העברת הודעות(פאקטות העוברות דרך מ"ה)
- זיהוי תקלות: מ"ה צריכה להיות מודעת באופן קבוע לשגיאות אפשריות.
  - יכולות לקרות בCPU וזיכרון החומרה, בהתקני הi/o, בתוכנות המשתמש
  - לכל סוג של שגיאה, מ"ה צריכה לנקוט בפעולות המתאימות כדי לתקן אותה
  - יכולות דיבאגינג יעזרו מאוד ליכולות המשתמש והמתכנת להשתמש ביעילות במערכת

קבוצת פונקציות נוספת של מ"ה שקיימת כדי להבטיח את היעילות שלה דרך שיתוף משאבים:

- הקצאת משאבים: כאשר מגוון משתמשים או מגוון משימות רצות במקביל, משאבים חייבים להיות מוקצים לכל אחד מהם
  - סוגים שונים של משאבים: מעגלי CPU, זיכרון ראשי, אחסון קבצים, התקני I/O

- התחברות: כדי לעקוב אחר המשתמשים- כמה ואיזה סוגי של משאבים הם משתמשים
- הגנה וביטחון: הבעלים של המידע המאוחסן במחשב עם המון משתמשים ירצו לשלוט בשימוש במידע ולגרום לכך שתהליכים הקורים באותו זמן לא יתממשקו אחד עם השני



Command line (CLI)- מאפשרת כניסת פקודות ישירה.

- לפעמים ממומש בגרעין ולפעמים ע"י תכניות מערכת
- לפעמים יש כמה ליבות - shells
- לוקחת פקודה מהמשתמש ומבצעת אותה
- לפעמים יש פקודות שמובנות ולפעמים רק שמות של תוכניות

GUI- ממשק משתמש נוח בדסקטופ

- בדר"כ עכבר, מקלדת ומוניטור
- אייקונים המייצגים תיקיות, פעולות וכ'ו..
- פעולות מגוונות של העכבר על אובייקטים בממשק הגורמים לפעולות שונות

- מערכות רבות כיום כוללות גם CLI וגם GUI

<u>קריאות מערכת – System Calls</u>

קריאה למ"ה מהאפליקציה/המשתמש. בדר"כ כתובת בשפת תכנות high-level כמו C/C++.
אנחנו נרצה לקבל גישה לדברים שהם רק דרך מ"ה(חלוקת משאבים..) כלומר, דברים שתהליך לא יכול לבצע בעצמו ולכן מופעל מנגנון זה כדי לקרוא למ"ה לעזרה. כך היא מוודאת שהדברים מתבצעים כמו שצריך ולא עוברים על מדיניות המערכת בשום שלב.

בדר"כ יש מספר המשויך לכלל קריאת מערכת. ממשק קריאות המערכת מתחזק טבלת אינדקסים בהתאם למספרים אלו. הממשק קורא לקריאת המערכת הרצויה במ"ה והקרנל ומחזיר סטטוס שלה וערכים מוחזרים אם יש.
קריאת מערכת ממומשת ע"י interrupt- עוצר תהליך ריצה נוכחי של האפליקציה ועובר למ"ה

סוגי קריאות מערכת:

- ניהול תהליכים:
  - יצירת תהליך והפסקתו
  - סיום, דחיה
  - טעינה, ביצוע
  - סיגנלים
  - ריקון זיכרון במקרה שלל שגיאה
  - מנעולים עבור ניהול גישה למידע משותף בין תהליכים
- ניהול קבצים:
  - יצירת קובץ ומחיקתו
  - פתיחה, סגירה
  - קריאה, כתיבה
  - קבלה וקביעת תכונות הקובץ
- ניהול התקנים:
  - בקשת התקן ושחרורו
  - קריאה, כתיבה, שינוי מקום
  - קבלת תכונות ההתקן וקביעתן
  - חיבור לוגי בין התקנים או ניתוק

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communications	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

System Services:

תוכנות המחשב מספקות סביבה נוחה עבור פיתוח תוכנה והרצתה.(חלק מהם הם פשוט ממשקי משתמש לקריאות מערכת וחלק יותר מורכבים).

- ניהול קבצים- כל הפעולות על קבצים ובעיקר מניפולציות על קבצים ותיקיות
- סטטוס מידע- חלק מבקשים מידע מהמערכת (תאריך, שעה, כמות זיכרון פנוי..) ואחרים מספקים ביצועים מפורטים, התחברות ודיבאגינג של המידע.
  תוכנות אלה מדפיסות את הפלט לטרמינל או להתקני אאוטפוט אחרים.
- תמיכת בשפת תכנות- קומפיילרים, אסמבלרים, דיבאגרים ואינטרפטרים לפעמים מסופקים
- טעינת תוכנה והרצתה
- תקשורת- מספקת מכניקה עבור יצירת קשר וירטולי בין תהליכים, משתמשים ומערכות מחשב.
  מאפשרת למשתמשים לשלוח הודעות למסך אחד של השני, לדפי אינטרנט, מיילים, להתחבר מרחוק וכ'ו..
- שירותי רקע- מתחילים בזמן ה-בוט. מספקים ישויות כמו בדיקת דיסק, מתזמן תהליכים, זיהוי שגיאות, הדפסה. רצות ביוזר מוד.
- אפליקציות- רצות ע"י המשתמשים, לא נחשבים חלק ממ"ה. רצות ע"י עכבר, CMD, אצבע

Linkers and Loaders:
קוד מקור המקומפל לתוך קבצי אובייקטים שנוצרו כדי להיטען לתוך כל מיקום זיכרון פיזי.
ה**linker** משלב אותם לקובץ בינארי המוכן להרצה.
הקובץ הזה מובא לתוך הזיכרון ע"י ה**loader** בכדי שיוכלו לבצע אותו.

Why Applications are Operating System Specific:

- אפליקציות המקומפלות על מערכת הפעלה אחת, בדר"כ לא יוכלו לרוץ על אחרת
- כל מ"ה מספקת את קריאות המערכת הייחודיות שלה
- אפליקציות יכולות מולטי-מ"ה:
  - אפליקציות שכתובות בשפת פסיקה כמו python הזמינות בהרבה מ"ה
  - אפליקציות שכתובות בשפה שכוללת בתוכה VM שמכילה את האפליקציה כמו java
  - אפליקציות שמשתמשות בשפה סטנדרטית כמו C, מקומפלות בנפרד על כל מ"ה

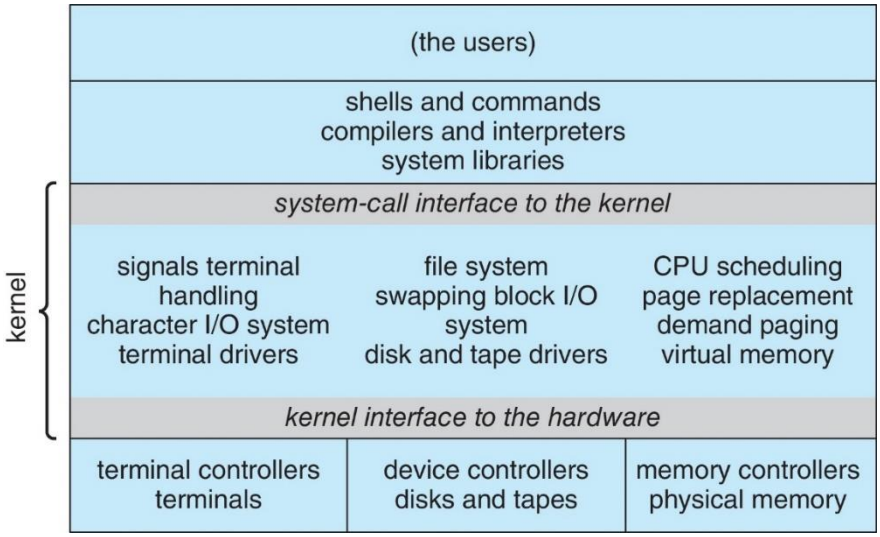
User goals- מ"ה צריכה להיות נוחה לשימוש, קלה ללמידה, אמינה, בטוחה ומהירה
System goals- מ"ה צריכה להיות קלה לתכנון, מימוש ותחזוקה כמו גם גמישה, אמינה, נקייה מטעויות ויעילה

<u>מימושים שונים למערכת הפעלה:</u>

<u>Monolithic Structure – Original UNIX:</u>
UNIX- מוגבלת ע"י פונקציונליות החומרה, למ"ה UNIX המקורית יש מבנה מוגבל.
מורכבת משני חלקים נפרדים:

- תוכנות המערכת
- הקרנל:
  - מורכב מכל מה שמתחת לממשק קריאות המערכת ומעל החומרה הפיסית
  - מספק מערכת קבצים, תזמון CPU, ניהול זיכרון ופונקציות נוספות.

הסבר: יש לי אפליקציה אחת נורא גדולה והיא עושה הכול או את הרוב בkernel mode, אולי יש לה גם אפליקציות אחרות אבל כל פעולות הליבה מתבצעות בפנים ולכן לעדכן אותה זה תהליך נורא כבד ומסובך.



Microkernels:

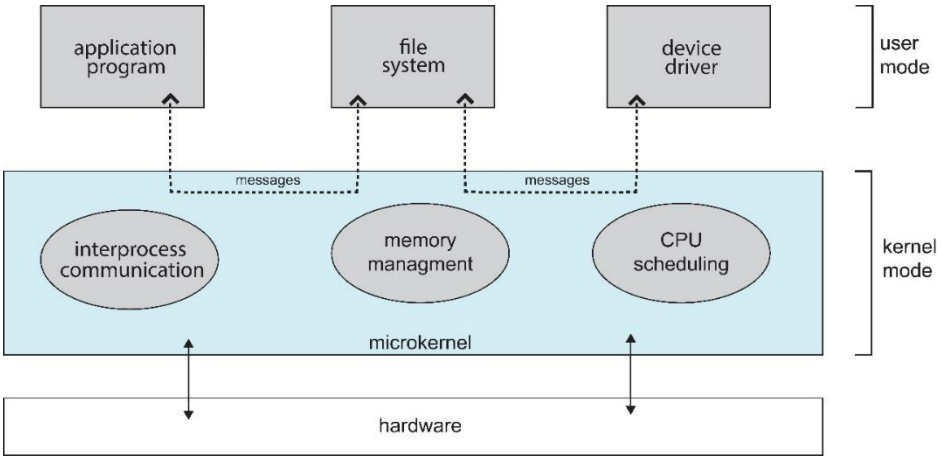
מעביר כמה שיותר מהקרנל לתוך איזור המשתמש.
התקשורת לוקחת מקום בין המודלים של המשתמש ע"י שליחת הודעות
יתרונות:

- קל יותר להרחיב מיקרו-קרנל
- קל להכניס למ"ה ארכיטקטורות חדשות
- יותר אמינה (פחות קוד רץ בkernel mode)
- יותר מאובטחת

חסרונות:

- תקורת ביצועים של איזור המשתמש לתקשורת איזור הקרנל
- ריבוי קריאות מערכת

הסבר: נבנה קרנל כמה שיותר מצומצם, מאוד קטן וקומפקטי שיעשה מעט מאוד פעולות, קצת ניהול תהליכים, זיכרון וכ'ו ואת כל השאר יעשו מערכות חיצוניות. נעשה ע"י תקשורת בין תהליכים. מדובר על דברים עם אלגוריתמיקה נורא פשוטה שנוציא מהקרנל. למשל – device driver, file system. גורם למערכת יציבה יותר – תקלה בdevice driver לא תעכב את מערכת ההפעלה, לא יגרום לכך שהכול יתקע. חוסך הרבה הפעלות מחדש של המערכת.



- **כיום רוב השרתים הם מונוליטים- היעילות ניצחה את המודולריות**

System Boot

- מ"ה חייבת להיות זמינה עבור החומרה כדי שהיא תוכל להתחיל אותה.
  - חלק קוד קטן – טוען בוטסטראפ, BIOS, המאוחסן בROM מאתר את הקרנל, טוען אותו לזיכרון ומתחיל אותו
- Bootstrap loader זה למעשה תוכנה שמגיעה עם החומרה ומטרתה לטעון תוכנה אחרת שאחראית על העלאת המערכת. בעצם טוען את הטוען של מערכת ההפעלה והיא טוענת אח"כ
  - להעלות מ"ה זה תהליך מורכב ולכן רכיבים פשוטים לא צריכים מ"ה (כמו חומרה). מ"ה באה לנהל את המחשב- לנהל משאבים וכ'ו. לכן זו לא אפליקציה פשוטה. ברגע שהיא מגיעה לרמת מורכבות כלשהי, גם הטעינה שלה מסתבכת.

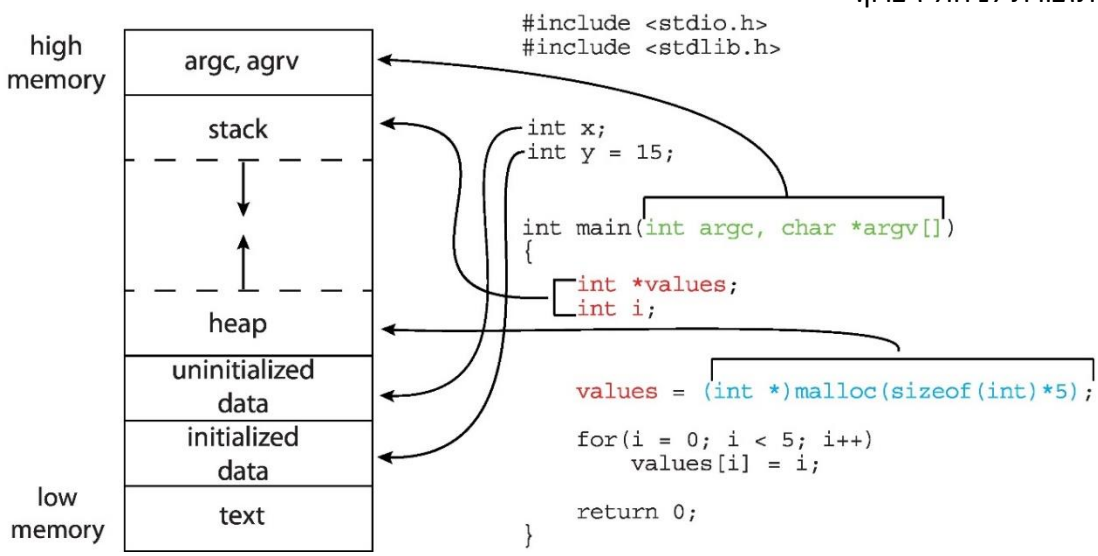
Processes:

מ"ה מבצעת מגוון רחב של תוכנות שרצות בתוך תהליך
תהליך – תוכנה במהלך ריצה. מבוצע בצורה סדרתית
חלקים רבים:

- קוד התוכנה, נקרא גם text section
- פעילות נוכחית כוללת program counter= באיזה חלק של הקוד אנחנו עכשיו
- מחסנית המכילה זיכרון זמני – פרמטרים של פונקציה, כתובות מוחזרות, משתנים לוקאליים
- Data section המכיל משתנים גלובלים וקבועים
- ערימה המכילה זיכון שהוקצה דינאמית במהלך זמן הריצה

תוכנה היא ישות פאסיבית בעוד תהליך הוא אקטיבי.
תוכנה נהיית תהליך כאשר קובץ הביצוע שלה נטען לזיכרון.
לתוכנה אחת יכולים להיות כמה תהליכים (מגוון משתמשים מבצעים את אותה תוכנה)

תזכורת לניהול זיכרון:



מצבים של תהליך:

- New: התהליך נוצר
- Running: הפעולות מבוצעות, התהליך בזמן ריצה
- Waiting: התהליך מחכה שיקרה איזשהו אירוע (נגיד הוא מחכה לi/o)
- Ready: התהליך מחכה להיות משובץ לריצה, מבקש זמן CPU מהמערכת
- Terminated: התהליך סיים ביצוע, עשה exit() והמערכת יכולה לשחרר משאבים

אם יש Interrupt, נעבור ממצב running לready.

Process Control Block (PCB):

מבנה נתונים שקיים לכל תהליך. יושב בזיכרון של הקרנל מכיל את הדברים הבאים:

- מצב התהליך
- Program counter-מיקום של הפעולה הבאה שיש לבצע
- רגיסטרים - בכל פעם שתהליך מפסיק ריצה אז הדברים שהיה לו ברגיסטר לא נשמרים, והמעבד ישתמש בזה לדברים אחרים לכן יש בPCB רגיסטרים ששם נשמור את מה שהיה וכאשר התהליך יחזור לרוץ מה שברגיסטרים יטען חזרה והתהליך ימשיך מהנק' בה עצר.
- מידע על תזמון הCPU
- מידע על ניהול זיכרון – זיכרון שהוקצה לתהליך
- Accounting information- CPU שהיה בשימוש, כמה זמן עבר מההתחלה..
- מידע על סטטוס ה-I/O: התקני I/O המוקצים לתהליך, רשימה של קבצים פתוחים

- ההחלפה הזאת, בין הרגיסטרים השמורים בPCB לרגילים, נקראת context-switch
- הקרנל הוא לא תהליך מאחר ואין לו PCB

Threads

עד כה, לתהליך היה טרד אחד של ביצוע.
נחשוב על מצב שיש לנו כמה program counters לכל תהליך -> הרבה מקומות יכולים להתבצע במקביל -> הרבה טרדים של שליטה = טרדים.

Process Scheduling:

מקסום שימוש הCPU, החלפה מהירה בין תהליכים בתוך ליבת הCPU
המתזמן בוחר מבין התהליכים הזמינים לריצה הבאה של ליבת הCPU
מחזיק תורי תזמון של תהליכים:

- Ready queue: קבוצה של כל התהליכים היושבים בזיכרון הראשי ומוכנים לרוץ
- Wait queue: קבוצה של תהליכים המחכים לאירוע

Context switch= מצב שבו הCPU מחליף מתהליך אחד לאחר
כשהCPU מחליף תהליך, המערכת חייבת לשמור את המצב של התהליך הישן ולטעון את המצב השמור עבור התהליך החדש דרך context switch.
הcontext של תהליך מיוצג בPCB שלו.
ככל שמערכת ההפעלה וה-PCB מסובכים יותר כך מתארך הזמן של ההחלפה.
הזמן תלוי בתמיכת החומרה – יש חומרות שמספקות קבוצות רבות של רגיסטרים פר CPU -> אפשר לטעון הרבה באותו זמן.

Operations on Processes:

המערכת חייבת לספק מכניקה עבור

- יצירת תהליך
- סיום תהליך

יצירת תהליך:

תהליך אב יוצר תהליך בן אשר בתורו ייצור תהליכים אחרים בצורת עץ תהליכים.
תהליך מזוהה ומנוהל דרך process identifier (pid).
אפשרויות שיתוף משאבים-

- אבא ובן שותפים בכל המשאבים
- ילדים חולקים תת-קבוצה של משאבי האב

- אבא וילד לא חולקים שום משאב

אפשרויות ביצוע-

- אבא וילדים מבוצעים במקביל
- אבא מחכה עד שהבן שלו יסיים

מרחב כתובות-

- ילד משכפל את הכתובת של האבא
- לבן יש תוכנה שנטענה לתוכו

<u>סיום תהליך:</u>

תהליך מבצע את הפקודה האחרונה ואז מבקש ממ"ה למחוק אותו ע"י שימוש ב()wait. פעולה זו מודיעה לתהליך האב שהבן סיים ע"י כך שהיא מחזירה את הstatus data שלו ובנוסף מוחקת את המשאבים שהוקצו לתהליך.

אב יכול לסיים ריצה של תהליך בן ע"י ()abort. סיבות לעשות זאת:

- הבן ניצל את כל המשאבים שהוקצו לו
- המשימה שהוקצתה לבן כבר לא רלוונטית
- האב סיים ואז מ"ה לא מרשה לבן להמשיך אם אבא שלו מת

- יש מ"ה שלא מרשות לבן להמשיך להתקיים אם אבא שלו סיים. אם תהליך מסיים אז כל הילדים שלו חייבים גם לסיים. (ילדים, נכדים וכו'..) הסיום מתבצע ע"י מ"ה. קונקרטית אין שום סיבה אמיתית לעשות את זה, הכול תלוי בצורך.

תהליך אב יכול לחכות לסיום של תהליך הבן שלו ע"י שימוש בקריאת המערכת ()wait. הקריאה מחזירה סטטוס מידע ואת הpid של התהליך שסיים. ((pid = wait(&status).

תהליך יתום- אין לו תהליך אבא. במערכות UNIX אפשר לשים לו תהליך אבא להיות INIT.

*כשהורגים אבא (סיים לרוץ לפני הבן) אז הבן ממשיך לרוץ אלא אם כן הגיע לאותה נק' סיום בקוד.

כשעושים פורק לא מריצים את שניהם ביחד, אפשר להריץ כל אחד מהם בנפרד בעזרת if .

כשאבא מגיע לexit ראשון אז הבן יגיע לinit.

אם תהליך האב לא עשה ()wait אז התהליך נקרא זומבי.

אם תהליך האב סיים בלי לעשות ()wait אז התהליך נקרא יתום.

יש תהליך אב ויש תהליך בן. כשתהליך הבן מת הוא מחזיק מידע שאמור לעבור לתהליך האבא(exit status) הוא מחזיק בנוסף גם את הpid,ppid ואי אפשר להשתמש בהם, הם שמורים בקרנל עד שהאבא עושה wait. כשהאבא יעשה ()wait, הוא ישחרר את כל המשאבים שהוקצו לבן והוא יפסיק להיות זומבי.

אם תהליך האב מת (עושה exit ואז הוא נהיה זומבי או מת)- זה שהוא סיים לא הורג את תהליך הבן, אבל מה שכן קורה זה שהמערכת הולכת לppid והופכת אותו להיות 1 כלומר, אתה לא הבן של המשפחה שלך אלא הבן של התהליך הראשון init. (התפקיד של init זה לעשות wait כל הזמן).

-מה קורה לילד של תהליך שנהיה זומבי? הופך להיות init

החלק העיקרי שמ"ה עושה כשתהליך מסיים חיים זה לשחרר את המשאבים שלו כדי שתהליכים אחרים יוכלו להשתמש בהם. (לסגור לו קבצים, לשחרר זיכרון...)

התהליך הזה לוקח זמן גם למ"ה לעשות אותו ולכן עולה השאלה: איזה סוגי תהליכים משחררים קודם ואיזה אח"כ לפי השיקולים של המערכת?

- מ"ה לפעמים תכפה סיום על תהליך כדי לקחת בחזרה את המשאבים שלו כמו זיכרון. מהחשוב לפחות:
  - תהליכי חזית
  - תהליכים נראים
  - תהליכי שירות
  - תהליכי רקע
  - תהליכים ריקים

נהרוג תהליכים עם חשיבות נמוכה יותר - משאבים של תהליך רקע שיצא אפשר לשחרר יותר מהר מאחרים וכ'ו..

ארכיטקטורה של הרבה תהליכים - היכולת להריץ הרבה תהליכים במקביל.
הרבה שרתי אינטרנט רצים כתהליך אחד כלומר, אם אתר גורם לבעיות אז כל השרת קורס.
Google chrome - מריץ 3 סוגי תהליכים:
תהליכי שרת: מנהלים UI, IO
תהליכי עזרה/הנגשה (render): מתעסקים בדפים אינטרנט, js, html
תהליכי פלאג-אין לכל סוג של פלאג אין.

תהליכים בתוך מערכת יכולים להיות נפרדים אחד מהשני וזה אומר שכעיקרון הם לא אמורים לגעת אחד בשני. הכלל הזה נשבר כשזה קורה בצורה מפורשת. יש כל מיני שיטות לעשות זאת במ"ה.
תהליכים המשתפים פעולה יכולים להשפיע או להיות מושפעים מתהליכים אחרים, כוללים שיתוף מידע.
למה זה טוב? תהליכים שרצים במקביל אז המשאבים שלהם נפרדים – לא יכולים לגעת בזיכרון אחד של השני והם לא מעבירים הודעות אחד לשני.
בדר"כ נבנה משימה שמורכבת מכמה תהליכים נפרדים שכן נרצה שיתקשרו ביניהם היות והם עובדים למען מטרה משותפת. (למשל pipe- שיתוף מידע בין תהליכים בצורה לא מפורשת)
-לכל תהליך יש את המשאבים שלו – הוא מקבל זמן עיבוד בנפרד מהאחרים, הוא מקבל משאבים משלו וכ'ו.
סיבות לשיתוף תהליכים: שיתוף מידע, הגדלת מהירות, מודולאריות, נוחות

<u>בעיית היצרן-צרכן</u>
פרדיגמה עבור תהליכים המשתפים פעולה, תהליכי ייצור מייצרים מידע שנצרך ע"י תהליכי הצרכנים.
צינור תקשורת בין יצרנים לצרכנים.
יש שני מודלים:
אנבאונדד – הצרכנים תמיד יכולים לזרוק אינפורמציה ואין מגבלת גודל על הבאפר עצמו, לא משנה כמה יזרקו המסז' קיו יוכל לאכול את זה
הבאונדד – תניח שיש ערוץ עם משאבים מוגבלים, גודל באפר קבוע, תכתוב יותר ממה שהוא יכול להכיל אז או שהוא יזרוק את המידע ואז נאבד אותו, או שהמידע פשוט יחכה וכך הייצור ייעצר.

<u>מצגת 4: Threads</u>
מוטיבציה: רוב האפליקציות המודרניות הן מולטי-טרדד. הטרדים רצים בתוך האפליקציה.
הרבה משימות באפליקציה יכולות להיות להתבצע ע"י טרדים נפרדים: עדכון, הכנסת מידע, בדיקת איות, מענה לשאילתת רשת..
לייצור תהליך משקל כבר בעוד שלייצור טרד משקל קל. טרדים מפשטים את הקוד ומגבירים יעילות.

- קרנלס בדר"כ מולטי-טרדד

ההבדל בין פרוסס לטרד – בתוך פרוסס אחד יכולים להיות הרבה טרדים (אחד מהמשאבים שתהליך מחזיק זה כמה טרדים יש לו).

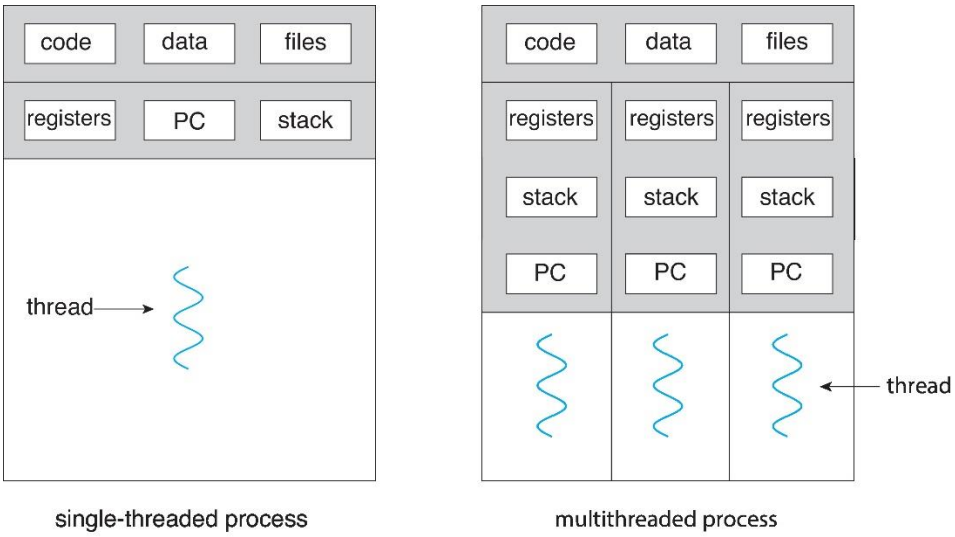
למה עושים אפליקציות מולטי-טרדד: כי אפשר

דוגמא לשימוש במולטי-טרדד: נרצה שהUI יעלה בנפרד מהמערכת עצמה.
מה יכול לתקוע UI – היות וזה IO הוא עובד יותר לאט מהעבודה של המעבד. כשאנחנו כותבים רכיבים IO מה שנרצה לעשות זה להקשיב במקביל לכמה רכיבים (נגיד לכפתורים). אם נחכה כל פעם שכל עיבוד כזה יסתיים ורק אז נמשיך זה יתקע את הUI.
כדי לפתור את זה נפריד! יהיה תהליך שיטפל רק בUI ותהליך שמטפל רק בבקשות (דוגמא ליצרן צרכן)
אם נקבל קליק אבל הכפתור במצב disable נתעלם מזה עד שהוא יחזור להיות able.

- למה לא להשתמש בתהליכים- יהיה הרבה משאבים

טרדים חיים על אותו מרחב של תשתיות ולכן יהיו בבירור מהירים יותר. (הם משתמשים באותם משאבים)
שימוש בטרדים עוזר לפשט את הקוד- התקשורת בין התהליכים תהיה מבוססת על זיכרון.

עבור מולטיטרדינג משכפלים לכל אחד את הרגיסטרים והסטאק. הבעייתיות בשיתוף הסטאק היא :
נניח שיש לנו שני טרדים ו3 פונקציות. הטרדים עושים את אותו הקוד ומריצים את 3 הפונקציות. נגיד רץ תהליך a ואז הוא עובר לb ואז לc . עכשיו עברנו לטרד b והוא דחף את החזרה של 1 ואז עוברים לשלישי והוא במקום להמשיך מ2 ימשיך ל1 כי זה מה שיש לו במחסנית.



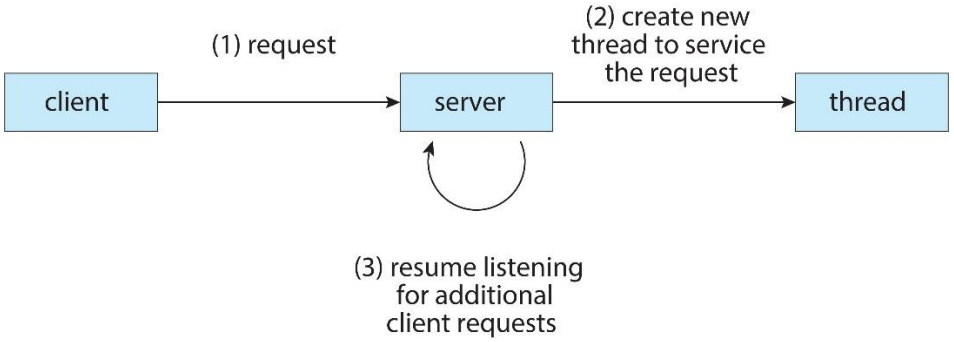
הטקסט, ההיפ והגלובל זה אזורים משותפים לכל הטרדים.

- במצב הקלאסי של סינגל טרדד – יש אחד מכל דבר בלבד. אם יש כמה ליבות (יותר ממשאב עיבוד אחד בחומרה) אז הסינגל טרדד לא ינצל יותר מחומרה אחת כי זה סתם בזבוז משאבים להעביר אותו בין ליבות.

אם נריץ כמה טרדים במקביל נוכל להריץ אותם על כמה ליבות.

- PC זה גם רגיסטר (PC=Program Counter)

<u>ארכיטקטורה:</u> יש בקשות שמגיעות לשרת ואז הוא ייתן לאחד הטרדים לטפל בבקשה והשרת יחזור לעשות listen. זה נותן לנו יעילות של הווב סרבר . הוא יכול לחלק משימות ולהמשיך לקבל משימות.



<u>יתרונות :</u>
Responsiveness (היענות) – יכול להרשות המשך ביצוע גם אם חלק מהתהליך חסום, חשוב במיוחד עבור UI
שיתוף משאבים – טרדים חולקים משאבים של תהליכים, קל יותר מזיכרון משותף או העברת הודעות (לא צריך לשכפל משאבים). במקום לייצר הרבה משאבים ולשכפל אותם נעדיף לשתף אותם.
כלכלי – זול יותר מיצירת תהליך, טרדים מחליפים תקורות נמוכות יותר מאשר context switching.
scalability – ארכיטקטורות שונות יהיה להן נוח יותר לעבוד עם טרדים. תוכנית שהייתה רצה עד היום על מעבד בודד – עכשיו יכולה לרוץ על כמה מעבדים שונים וכך יכולה לסיים יותר מהר הרבה משימות.

<u>שיעור 7</u>
<u>הקדמה:</u> עד כה דיברנו על תהליך שרץ עם טרד בודד. בתהליכי מולטי-טרדד נרצה שהתוכנית תרוץ במקביל נרצה לשכפל רק חלק מאותם משאבים, אלה הרלוונטיים לריצה- רגיסטרים, סטאק וPC נפרדים.

- למה נרצה רגיסטרים נפרדים? (רגיסטר=משאבי חישוב/עיבוד) אם נעשה למשל פעולת חיבור, כל רגיסטר יחזיק ערך, נגיד למעבד לבצע משהו ואת התוצאה להכניס לרגיסטר אחר. לא נרצה שמישהו יתערב לנו במהלך הפעולות. 2 טרדים לא יכולים לרוץ על אותם רגיסטרים. שלא ייווצר מצב שאחד קורא ואחד כותב אליו במקביל ואז נקבל תוצאה אחרת ממה שרצינו. לכן, לכל טרד יהיו רגיסטרים משלו. כלומר כל פעם שנחליף טרד, נשמור את הישנים ונטען את החדשים.

- למה נרצה להפריד את המחסנית לכל אחד? מי שמתעסק עם הסטאק זה הקומפיילר. במחסנית יהיו כל המשתנים האוטומטים המנוהלים בצורה שקופה- משתנים לוקאליים בפונקציה, סדר קריאות וכ'ו. נוכל לקרוא לפונקציה משני טרדים שונים עם ערכים שונים, ונרצה לדעת לאן לחזור. אסור שזה יתנגש עם טרדים אחרים. המחסנית יוצרת את הסדר של הטרד.

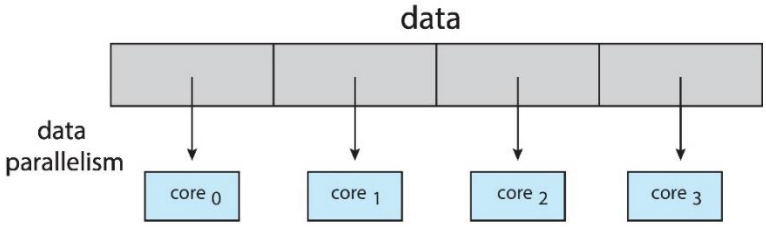
- program counter-PC: רגיסטר ספציפי שאומר לנו באיזה שורה בקוד אנחנו נמצאים. כאשר עושים פקודת call המיקום שלו משתנה.

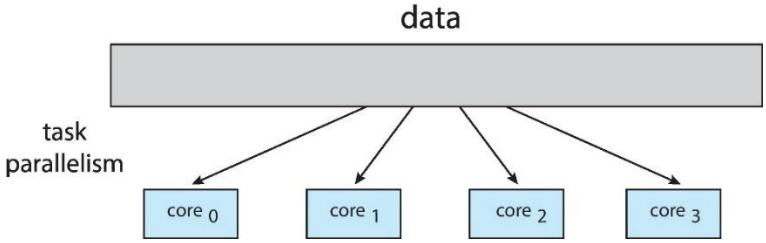
המשותף לכל הטרדים: הקוד (text), הדאטה (משתנים גלובליים) והערימה. באופן כללי זה המרחב זיכרון.
- לכל תהליך יש מחסנית משלו. המחסנית תחולק לטרדים לפי הצורך אבל כל המרחב זיכרון הוא משותף היות והוא שייך לתהליך עצמו.
- בין תהליכים שונים אין מרחב זיכרון משותף

<u>המשך מצגת:</u>
מולטי קור – כמה ליבות על מעבד אחד
מולטי פרוסור – כמה מעבדים שונים
יש לנו שיקולים למה להשתמש בכל שיטה:
שרת שיש לו הרבה טרדים נרצה שכולם ירוצו על אותו מעבד(לא בהכרח על אותה ליבה)
נרצה לדאוג שאנחנו מחלקים את האקטיבי ויוצרים איזון. זה רלוונטי לצורה שבה שנכתוב את הקוד היות ונרצה להיות אופטימליים, לפעמים נרצה להריץ שני תהליכים שיעשו אותו דבר אבל תהליך אחד יעבוד על דאטה אחד ותהליך אחר על דאטה אחר.
parallelism – היכולות לבצע יותר ממשימה אחת בו זמנית
קונקרנסי – תמיכה ביותר ממשימה אחת שמבצעת תהליך

אפשר להריץ תהליכים קונקרנטלי, שיתקדמו במקביל, ע"י כך שנחלק את המעבד בליבה אחד. זה לא בו זמנית אבל כן כולם מתקדמים אחד אחרי השני.
כשיש לנו שתי ליבות נוכל להריץ אותם בו זמנית כל אחד על ליבה אחרת.
פרלליזם – דורש יותר ממשאב עיבוד בודד. קונקרנסי אפשר גם עם משאב אחד.
כל רכיבי הIO רצים במקביל למעבד. לכן קיים פרלליזם מסויים פשוט לא בCPU.

Data parallelism: יש לנו את אותו דאטה ויש את אותה פעולות עיבוד (נגיד פעולת כיווץ) שרצה על חלקי דאטה שונים. אלו אותם אלגוריתמים שרצים על דאטה שונה.
Task parallelism: יש לנו את אותו דאטה והוא הופך להיות משאב משותף לקטעי קוד שונים. טרדים מבוזרים בליבות, כל טרד מבצע פעולה ייחודית





<u>Amdahl's Law:</u>
מאפיין ביצועים המושגים מהוספה של עוד ליבות לאפליקציה שיש לה גם מרכיבים סדרתיים וגם מקביליים.
נשאלת השאלה עד איפה נרצה ללכת עם המקביליות הזו.
המטרה – לעבוד כמה שיותר מהר.
נרצה למצוא סוג של אופטימום בין כמה מקביליות אנחנו יוצרים ובין כמה סדרתיות אנחנו יוצרים.

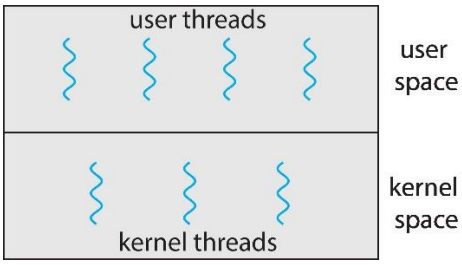
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

S- החלק הסדרתי N-מספר הליבות שלנו.
1-s זה החלק שאפשר למקבל על מקסימום N ליבות.
אם יש לי 75% מהקוד במקבול ו25% הוא סדרתי, הספידאפ המקסימלי שנקבל אם נחליף מליבה אחת לשתי ליבות הוא 1.6 (1/0.25+0.75/2)
כשN מתקרב לאינסוף, נקבל ספידאפ של בערך 1/S

זה אומר שאם הוספתי עכשיו ליבה למכונה שלי, האפליקציה שלי תרוץ במקסימום ב1.6 (היינו מצפים שזה יקטין בחצי אבל לא).
המצב הכי טוב זה ספידאפ לינארי – תוסיף עוד ליבה ונכפיל את העיבוד לפי שתיים, תוסיף 3 נכפיל פי 3 וכו..

User threads and kernel threads:

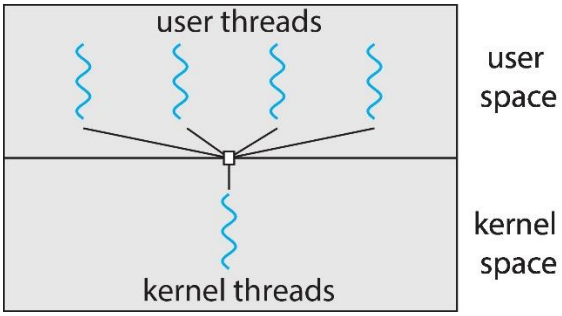
User-threads: ניהול מתבצע ע"י ספריית הטרדים ברמת המשתמש (user-level) (ווינדוס טרדס, ג'אווה טרדס..)
Kernel-threads: נתמכים ע"י הקרנל. דוגמאות: כל המטרות הכלליות של מ"ה כולל: לינוקס, אנדרואיד, IOS..



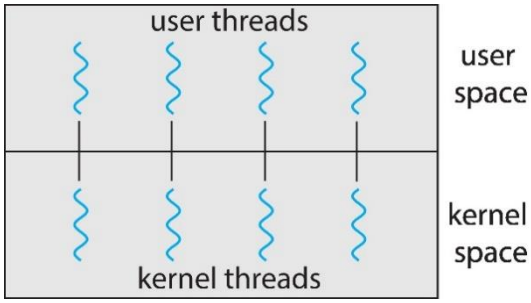
מה שיתן מקביליות אמיתית זה אם הקרנל מודע לטרדים שלנו. ברגע שהוא מודע אליו הוא ייתן לו משאבים פיזיים.

מודלים של מולטי-טרדינג:

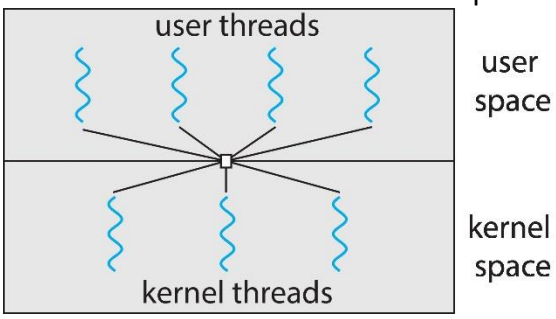
Many to one:
הרבה טרדים ברמת המשתמש הממופים לטרד יחיד בקרנל. חסימה של טרד אחד גורר שהכל ייחסם.
הרבה טרדים לא ירוצו במקביליות על מערכת מרובת ליבות בגלל שרק אחד יכול להיות בקרנל בכל זמן נתון.
מבחינת הקרנל יש לו טרד בודד. ביוזר מוד יש כמה טרדים. מבחינת הקרנל זה שקוף והוא נותן זמן לפי הטרדים שהוא מכיר שזה אחד.



One-to-one:
כל טרד ברמת המשתמש ממופה לקרנל טרד. יצירת טרד משתמש יוצרת טרד מסוג קרנל. מאפשר יותר קונקרנסי מהמודל הקודם.
כל יוזר טרד מקבל למעשה קרנל טרד. הוא מודע אחד לאחד לכל טרד שאנחנו יוצרים. כך נקבל אופטימום מבחינת משאבים.

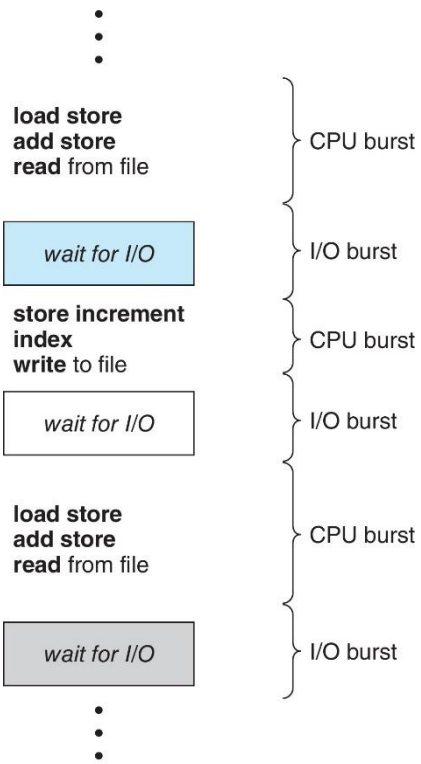


Many-to-many:
הרבה טרדים של היוזר שיהיו ממופים להרבה טרדים של הקרנל. מאפשרת למ"ה לייצר מספיק קרנל טרדס.
לא נפוץ.



<u>ספריות:</u>
כשאנחנו מייצרים היום בלינוקסים מודרנים טרדים אנחנו משתמשים בקריאת מערכת שנקראת clone – איזה משאבים הוא משתף עם התהליך שיצר אותו.
אם הוא משתף את רוב המשאבים נקבל טרד, אם הוא הולך לכיוון ההפוך ולא משתף שום דבר נקבל פרוסס.
אפשר לייצר כל מיני דברים שמשתפים מרחב זיכרון אבל לא קבצים (מיזוג של שניהם) למשל.
מהמצגת: לינוקס מתייחס לטרדים כאל משימות. יצירת טרד נעשית דרך קריאת מערכת ()clone. קריאה זו מאפשרת ל"משימת ילד" לשתף את מרחב הכתובות של משימת האב (תהליך).

<u>מצגת 5: CPU SCHEDULING</u>
עד עכשיו דיברנו על תהליכים וטרדים. עכשיו ניכנס לאזור יותר ספציפי של מ"ה המדבר על תזמון.
תזמון = יש לי מעבד ונרצה להריץ על תהליך, נרצה לבחור את התהליך הבא שבדאי לנו להריץ עליו.
המטרה: לנצל מקסימום זמן של הCPU במערכות שעושות מולטי-פרוגרמינג (הרבה תוכניות הרצות על המעבד בו זמנית ונרצה לשתף את המשאבים ביניהם). נתייחס לזה שלתהליכים (טרדים) יש נטייה לעשות מה שנקרא CPU IO burst cycle = בסופו של דבר נעשה פעולות שהן CPU ובאיזשהו שלב נלך ונחכה לIO וחוזר חלילה (פרץ של IO ופרץ של CPU).
בזמן שעושים IO אין גישה לCPU ולכן אנו ממתינים שהוא יסיים את העבודה שלו. פרץ CPU גורר פרץ IO.
מה שהכי מטריד אותנו זה לראות שאנחנו מנצלים את הCPU בצורה המיטבית (ההתפלגות).



המתזמן בוחר מבין כמה תהליכים שהולכים לרוץ מי מהם יקבל זמן CPU. התור יכול להיות מסודר בדרכים שונות.
קבלת החלטות מהמתזמן יתבצעו כאשר תהליך:
1. עובר מrunning לwaiting (מחכה לIO)
2. עובר מrunning לready
3. עובר מwaiting לready (הIO חזר)
4. מסתיים

<u>סיווג של סוגי תזמון:</u>
תזמון 1 ו4 יקרא nonpreemptive (לא עוצרים/התהליך מוותר, קורות ביוזמת התהליך)
תזמון 2 ו3 הוא preemptive (קורים ביוזמת מ"ה): נשקול גישה לדאטה משותפת, נשקול עצירה כשהוא בקרנל מוד, נשקול אינטרפט שקרה במהלך פעילות OS קריטית.

Dispatcher(אדם המשגר הוראות יציאה): נותן שליטה על הCPU לתהליך הנבחר ע"י המתזמן. זה כולל:
Switching context
Switching to user mode
קפיצה למקום המתאים בתוכנית המשתמש כדי לאתחל את התוכנית
Dispatch latency: הזמן שלקח לדיספצ'ר לעצור תהליך אחד ולהתחיל תהליך אחר.

<u>קריטריונים לתזמון :</u>
1. CPU utilization: רוצים שהCPU יהיה עמוס/עסוק כמה שיותר
2. Throughput: כמה תהליכים סיימו עבודתם בתוך פרק זמן, אינטרוול
3. Turnaround time: כמה זמן לוקח להריץ תהליך ספציפי מהרגע שהוא נוצר ועד הרגע שהוא מסיים
4. Waiting time: כמה זמן תהליך נמצא בready queue וממתין לקבל CPU
5. Response time: כמה זמן מהרגע שביקשתי להריץ את התהליך ועד הרגע שהוא מתחיל להגיב

נוכל למדוד באמצעותם כמה האלגוריתם תזמון שלנו טוב או לא טוב. יכול להיות שהוא יהיה ממש טוב במדדים מסוימים וממש גרוע באחרים, הכול תלוי באיזה קריטריון נרצה למקסם.
במערכות שיש להן user space זמן התגובה ממש קריטי וחשוב למשל, אבל במערכות אחרות הוא לא יהיה משמעותי.
האופטימיזציה יכולה להיות: מקסימום של השניים הראשונים ומינימום של שלושת האחרונים

תזמון FCFS = first-come, first-serve –
התהליך הראשון שיגיע הוא הראשון שנריץ
נדבר על מערכות שהן nonpreemptive

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:

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

אם היו מגיעים בסדר אחר:

P_2 , P_3 , P_1

- The Gantt chart for the schedule is:

- Waiting time for P_1 = 6; P_2 = 0, P_3 = 3
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

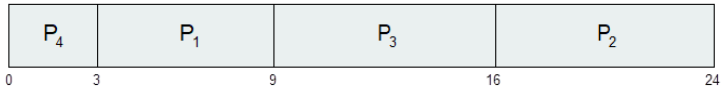
אם נרצה למקסם לפי זמן ההמתנה- סדר ההגעה הוא חשוב ולכן האלגוריתם הזה לא יהיה לנו טוב.
אנחנו יודעים כמה זמן הם ירצו לרוץ ולכן נעדיף להריץ קודם את הקצרים יותר ואז את הארוך.
אפקט השיירה: תהליך קצר שמגיע אחרי תהליך ארוך היה מסיים לפניו אבל יצרנו אפקט שיירה כי הארוך רץ לפניו ולכן הקצר ימתין הרבה זמן.
דוגמא: יש תהליך שהוא חסום ע"י הCPU ותהליכים אחרים שהם חסומי IO . הם יעשו short burst ומוותרים מאוד מהר על הCPU שהם מקבלים כי ירצו לקבל יותר IO. תהליכי שהם חסומי CPU יהיו רעבים ליותר CPU.
בדאי לנו להריץ יותר במקרה של זמן המתנה קצר יותר תהליכים שהם חסומי IO.

Shortest job first = SJF:

לכל תהליך אם היינו יודעים כמה זמן הוא הולך לרוץ, היינו רוצים לייצר מצב שיהיה יותר אופטימלי מבחינת זמן המתנה ע"י הרצת התהליכים הקצרים יותר קודם. מבחינת זמן המתנה אלגוריתם הזה יהיה אופטימלי כי נצליח להריץ יותר תהליכים בפרק זמן נתון ולכן זמן ההמתנה שלהם קצר.
במקרה של הרבה תהליכים קצרים יוצר starvation לתהליך הארוך ויכול להיות שלא נגיע אליו בכלל ואז הturnaround time שלו יהיה ממש ארוך והאלגוריתם לא יהיה אופטימלי ביחס לקריטריון הזה.

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



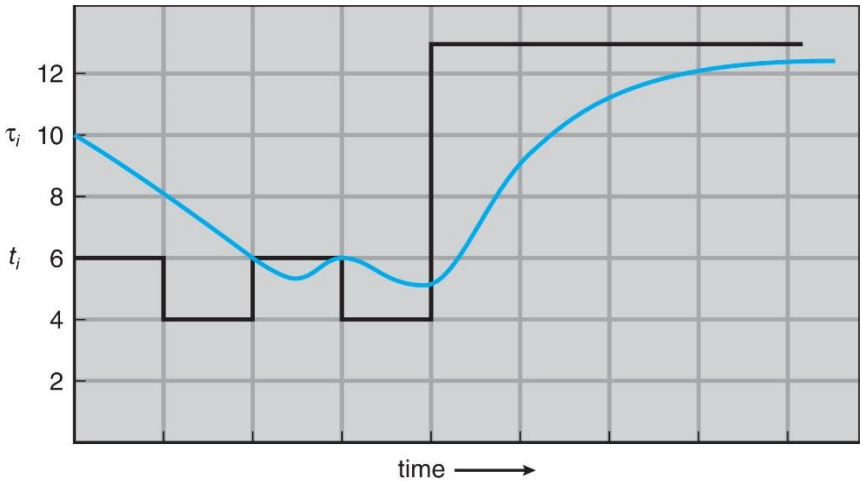
- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

איך נוכל לדעת מה יהיה הcpu burst הבא ?

מ"ה צריכה איכשהו להעריך איך לעשות את זה היות ואין לזה תשובה. אנחנו אוהבים דברים שרצים ב(1)o לדוגמא, נוכל לקחת כלים סטטיסטיים ולהגיד את הדבר הבא: נניח שתהליך מתנהג בצורה מסוימת, סיכוי טוב שהוא ימשיך להתנהג ככה גם קדימה.

t_n = actual length of nth CPU burst
τ_{n+1} = predicted value for the next CPU burst
$\alpha, 0 \le \alpha \le 1$
Define: $\tau_{n=1} = \alpha\, t_n + (1-\alpha)\tau_n$.

tn: האורך של הפרץ האחרון
tn+1: ערך שאנו חוזים עבור הפרץ הבא
בדר"כ אלפא תאותחל ל0.5
תחזית של אורך הcpu burst הבא:



דוגמא של ממוצע אקספוננציאלי:

CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\alpha\, t_{n-1} + \ldots + (1-\alpha)^j \alpha\, t_{n-j} + \ldots + (1-\alpha)^{n+1} \tau_0$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

דוגמא של shortest remaining time first:

בכל פעם שיגיע תהליך,
נעצור ונחליט אם צריך להחליף ביניהם

Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Preemptive SJF Gantt Chart

P_1	P_2	P_4	P_1	P_3

0 1 5 10 17 26

Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

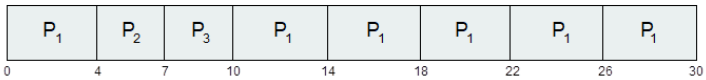
Round robin (RR)

אסטרטגיה שהיא preemptive ואומרת, בוא ניקח את הזמן CPU ונחלק אותו לקואנטומים של זמן. כל תהליך יקבל חתיכה קטנה מהזמן CPU (quantum q), בדר"כ 10-100 מילישניות. אחרי שהזמן הזה חולף התהליך מופסק ע"י מ"ה ומתווסף לסוף התור.
נסדר את התהליכים בסדר מעגלי וכך נריץ את כולם אחד אחרי השני לפרק זמן קבוע מראש.
אם יש n תהליכים בready queue ולכל אחד יש קוואנטת זמן q,אזי כל תהליך מקבל 1\n מזמן הCPU בצ'אנקים של לכל היותר q יחידות זמן. אין תהליך שיחכה יותר מq(n-1) יחידות זמן.
מנגנונים כאלה מופעלים ע"י interrupt של שעון (טיימר). כל כמות מסוימת של מילישניות הוא זורק אינטרפט ואז נבדוק האם צריך להחליף תהליך או לא.
ביצועים:
אם יהיה q מאוד גדול, נקבל FCFS (fifo) אם מגיע ראשון שהוא גדול והוא לא עובר את הq אז הוא ירוץ לפני כאלה שהברסט שלהם קטן יותר.
אם q מאוד קטן, נגיע לבעיה: בגלל שלוקח זמן למתג בין תהליכים נתחיל להתנגש עם הdispatch latency.
אם לוקח 5 מילישנייה להחליף זיכרון, הקוואנטה לא יכולה להיות אותו מספר.

דוגמא עם q=4:

Process	Burst Time
P_1	24
P_2	3
P_3	3

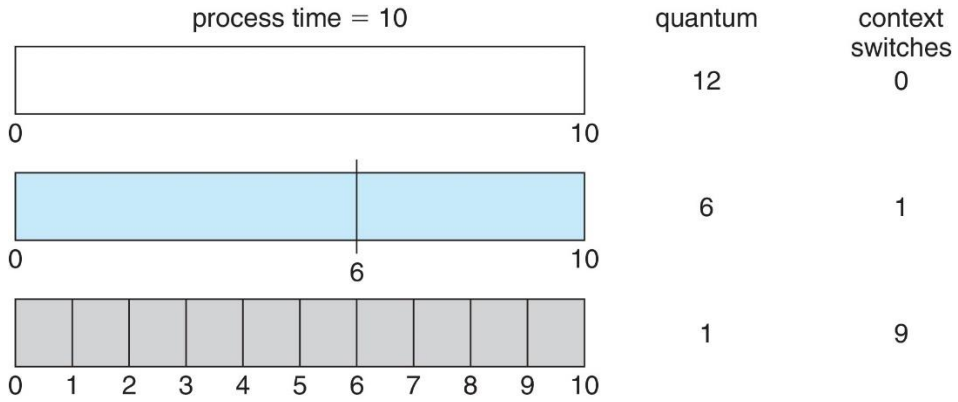
- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

- למרות שלא צריך להחליף את p1 עדיין מתקבלת החלטת מתזמן כל 4 שניות.

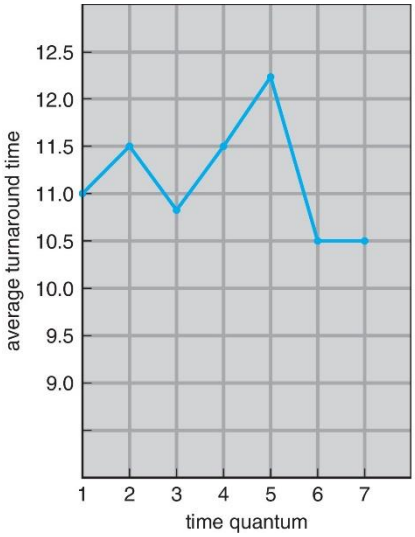
<u>Time Quantum and Context Switch Time:</u>



העיקרון: context switch צריך להיות זניח ביחד לquantum time.

איך הquantum time משפיע על ה turnaround time?
80% מהcpu burst צריכים להיות קצרים יותר מq
<u>בציור:</u> שלושת הקריטיים הם 1 3 ו6. נרצה שהq יהיה גדול יותר מ6. ברגע שזה קורה יש צניחה בturnaround time. רוב התהליכים מספיקים לסיים את הפעולה שלהם יחסית מהר ויורד זמן העיבוד שלהם.
מה נקבל מזה? כשנגיע לתהליכים ארוכים הם לא יגיעו לאפקט השיירה והם לא יעכבו את כולם אחריהם.

process	time
P_1	6
P_2	3
P_3	1
P_4	7



:Priority Scheduling

במקום לבוא ולהסתכל על אלגוריתם כללי, נסתכל על זה בצורה שנתעדף כל תהליך.
הCPU יקצה זמן לתהליכים עם העדיפות הגבוהה יותר. ככל שהמספר קטן יותר העדיפות גבוהה יותר.
בSJF- ככל שהתהליך קצר יותר העדיפות שלו תהיה גבוהה יותר
בעיה: starvation- תהליכים בעלי עדיפות נמוכה יכולים לא לקבל זמן CPU אף פעם.
פתרון: aging- תהליך הזדקנות. ככל שעובר זמן ותהליך מסויים לא קיבל עדיפות, נעלה לו את הpriority בצורה יזומה.

- תהליך שהוא io bound נעשה לו בוסט לעדיפות שלו.

דוגמא:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

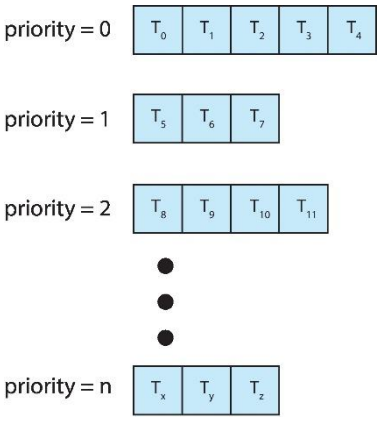
- Priority scheduling Gantt Chart

- Average waiting time = 8.2 msec

:Priority Scheduling w/ Round-Robin

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

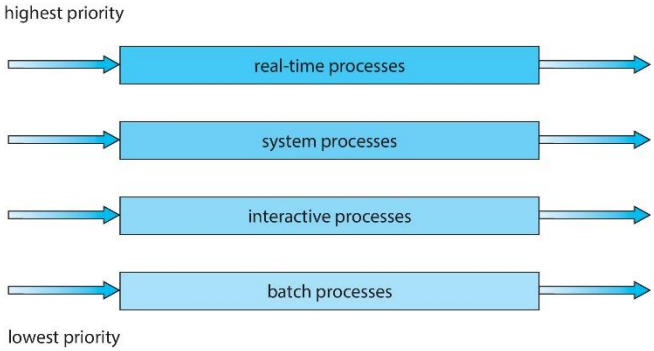
- Run the process with the highest priority. Processes with the same priority run round-robin
- Gantt Chart wit 2 ms time quantum



:Multilevel Queue

לכל עדיפות ניצור תור של תהליכים. בתוך כל תור יש למעשה אלגוריתם תזמון בין התהליכים השונים שנצטרך להחליט עליו. לכל תור נוכל לתת אסטרטגיה שונה.

איך נחליט איזה תהליך שייך לאיזו עדיפות? לפי סוג התהליך
תהליכים שהם real time (תהליכים שחייבים לסיים את פעולתם עד זמן נתון) יהיו בעדיפות הראשונה
אחריהם יבואו תהליכי מערכת שצריכים לטפל בה. אח"כ תהליכי משתמש והכי נמוכים הם batch processes.



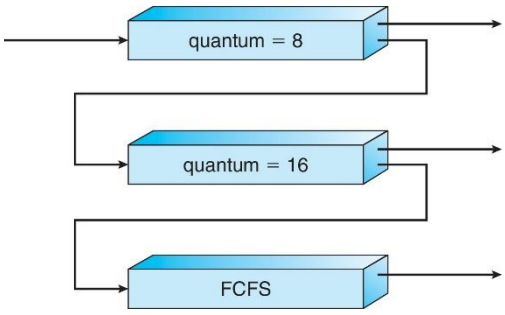
:Multilevel Feedback Queue

שילוב עוד יותר מתקדם של כמה תורים. מה שאנחנו יכולים לעשות עכשיו זה לייצר אינטראקציה בין התורים. לקחת תהליך מתור אחד ולהעביר אותו לתור אחר.
נוכל לממש כבה aging- אם תהליך היה בתהליכי המשתמש ולא קיבל זמן CPU נוכל להעביר אותו לתהליכי המערכת למשל ולקדם אותו בתור ובעדיפות.
המתזמן הזה מוגדר ע"י הפרמטרים הבאים:

1. מספר התורים
2. אלגוריתם שונה לכל תור
3. מתודה שנשתמש בה כדי להעביר תהליך בין התורים
4. מתודה שנשתמש בה לקבוע מתי להוריד לתהליך עדיפות
5. מתודה שנשתמש בה לקבוע לאיזה תור כל תהליך נכנס, מתי וכו'.

דוגמא:
יש לנו שלושה תורים. בהנחה שתהליך מסויים רץ כל הזמן ודורש קצת יותר נרצה להקפיץ אותו ל16.
אם היה שם מספיק זמן וגם 16 לא מספיק לו אולי נעביר אותו לשלישי.
ההנחה היא שבסופו של דבר הוא יוותר על הCPU כי אחרת הוא יתקע לנו אותו.

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



מצגת 9 – Main Memory

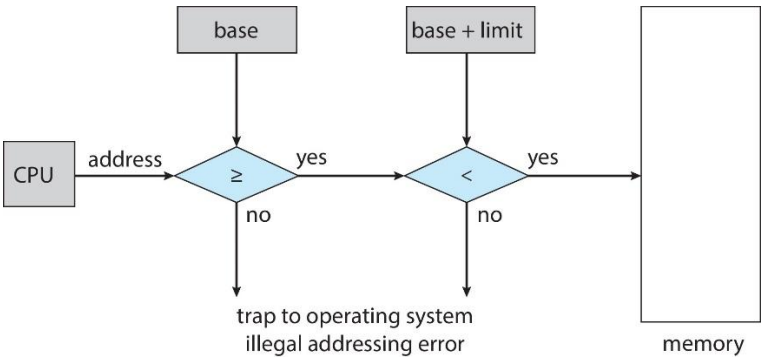
התהליך הראשון הוא להעלות תוכנית לתוך הזיכרון מהדיסק ביחד עם תהליך כדי שהיא תרוץ.
הזיכרון הראשי והרגיסטרים הם יכולים לגשת ישירות רק לאחסון CPU.
יחידות זיכרון רואות רק סטרימים של:

1. כתובות והאם זה בקשת קריאה או,
2. כתובות ודאטה והאם זה בקשות כתיבה

גישה לרגיסטר נעשית בcpu clock אחד (זמן מאוד קצר)
הזיכרון הראשי יכול לקחת הרבה סבבים (יכול להיות זמן ארוך), דבר שיגרום ל stall – הCPU מחכה עד שהמידע יגיע מהזיכרון אל רגיסטר במעבד.
איך נתגבר על הפערים? נשתמש ברכיבי חומרה – (מטמון) cache.
Chace יושבת בין הזיכרון הראשי והרגיסטרים של הCPU.
הגנה על הזיכרון נדרשת כדי להבטיח תפקוד תקין.

Protection

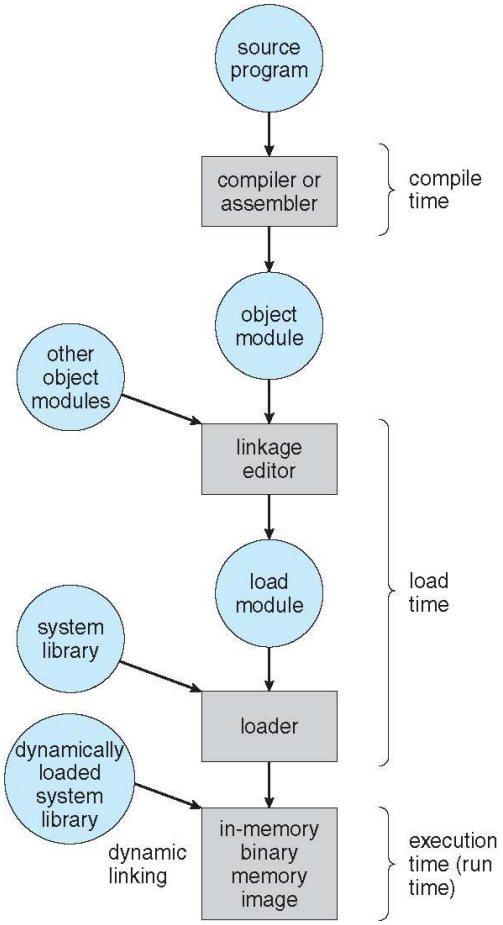
צריך להגן על זיכרון של תהליך אחד מזיכרון של תהליך אחר. שהם לא יוכלו לגעת אחד בשני.
בסופו של דבר כשאר אנחנו קוראים/כותבים אנחנו בונים על זה שאף אחד לא שינה את זה – נכונות משיקולי אבטחה – אם יש תהליך זדוני שמנסה לגנוב מידע נרצה שיהיו הפרדות ברמת הזיכרון בין התהליכים.
הדרך הכי פשוטה לעשות זאת- לכל תהליך ניתן כתובת בסיס וכתובת עליונה (limit) וכך לכל תהליך יהיה מרחב בזיכרון. החומרה היא זאת שאחראית על הקינפוג הזה. הקרנל קובע את הערכים של הבסיס והגבול.
איך נראה המנגנון? – מקבלים כתובת מהCPU (זו כתובת שנמצאת בתוכנית שלנו), ומעבירים אותה לתוך מנגנון חומרה (חלק מהמעבד) שאחראי לוודא שהכתובת נמצאת בטווח שהוקצה לתהליך. במידה וכן המידע יעבור ישירות לזיכרון, אחרת נרצה להפסיק את ריצת התהליך ע"י interrupt, נקבל חריגה על שימוש בכתובות לא חוקיות.
הערה: הכתובת ששלחנו נוצרה במהלך זמן הריצה שלנו.
הערה: כל התהליך הזה קורה ברמת החומרה. מי שמטפל בבסיס ובגבול הוא הקרנל. מה קורה אם יגיע תהליך וירצה לשנות את הערכים האלה? יש מה שנקרא privilege instructions כלומר הפקודה לכתוב לבסיס ולגבול היא פקודה שדורשת הרשאות גבוהות, צריך שנהיה בקרנל מוד. אם ננסה לכתוב לשם ולא נהיה בקרנל מוד נקבל חריגה.



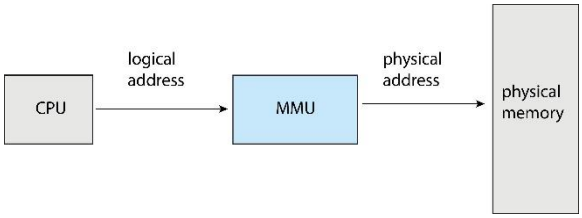
Address Binding:
אנו מעלים אפליקציה שקומפלה מהדיסק לזיכרון שיושבת בinput queue. אם אף אחד לא הודיע מראש ואין תמיכה היא תעלה לכתובת 0000. יש לנו בעיה שהקומפיילר לא יודע באיזה כתובת אנחנו הולכים לרוץ ולכאורה היה צריך לקבוע לכולם איזה כתובת קבועה דבר שהיה יוצר אי נוחות לשים את הכתובת הפיזית של תהליך המשתמש הראשון תמיד בכתובת הזאת. איך נתגבר על זה?
כתובות מוצגות בדרכים שונות בכל שלב שונה של חיי התוכנית. כתובות מקור בדר"כ סימבוליות, כתובות של קוד מקומפל מוצמדות לכתובות חדשות relocatable (למשל 14 ביטים מההתחלה של הקטע הזה).
הלינקר או הטוען יצמידו את הכתובות החדשות האלה אל כתובות אבסולוטיות (למשל 74014).
כל binding כזה ממפה מרחב זיכרון אחד לאחר.

אנחנו מבינים שהכתובות נטענות בצורה קצת יותר דינאמית מזה.
אנחנו מתייחסים לאיזה כתובת בסיס שנקבעת כאשר אנו טוענים לזיכרון.
Bind- (=לקשר) היכולת לשים קוד שמקמפלים פעם אחת ואפשר לשים במקומות שונים בזיכרון.

קישור כתובות של הוראות ומידע לכתובות זיכרון יכול לקרות בשלושה שלבים שונים:
Compile time: אם מיקום הזיכרון ידוע מראש, ניתן יהיה לייצר קוד אבסולוטי. חייב לקמפל מחדש את הקוד אם יש שינויים במיקום. לא קורה אף פעם.
Load time: חייב לייצר קוד הניתן לשינוי מקום אם מיקום הזיכרון לא ידוע בזמן קימפול. רוב המערכות עושות את זה
Execution time: פעולת הקישור מתעכבת עד זמן ריצה אם התהליך בזמן ביצועו יכול לזוז מסגמנט אחד בזיכרון לאחד. (צריך תמיכה של החומרה בשביל מיפוי הכתובות – רגיסטרי בסיס וגבול) כשאני מתחיל להריץ חלקים שונים של הקוד עצמו.

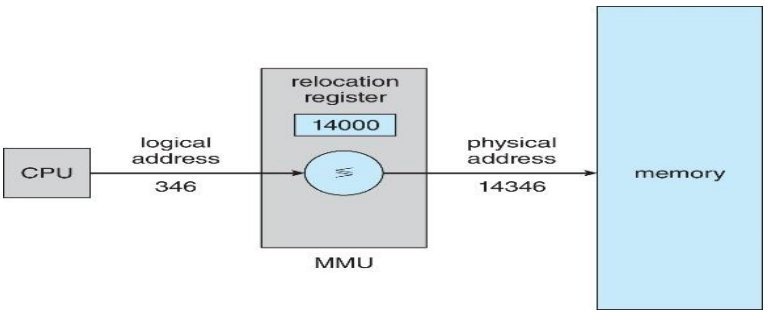


<u>מרחב כתובות לוגי VS פיזי</u>
הקונספט של מרחב כתובות לוגי שמחובר למרחב כתובות פיזי נפרד הוא דבר מרכזי בניהול זיכרון תקין.
כתובות פיזית - הכתובת שיושבת באמת בזיכרון (תא במערך), נראית ע"י יחידת זיכרון
כתובות לוגיות – לא מה שקורה בערוצים הפיזיים של החומרה, אלא מה שהתוכנית רואה. נוכל לתת כתובת לוגית ולמפות אותה לכתובת פיזית. מיוצרות ע"י הCPU. (נקראות גם כתובות וירטואליות)
כתובות לוגיות ופיזיות הן אותו דבר בסכמות של קישור כתובות (binding addresses) של זמן קמפול וטעינה. הן נפרדות בסכמות של זמן ביצוע.
מרחב כתובות פיזי הוא קבוצה של כל הכתובות הפיזיות שנוצרו ע"י התוכנית.
מרחב כתובות לוגי הוא קבוצה של כל הכתובות הלוגיות שנוצרו ע"י התוכנית.



בזיכרון הפיזי יש הרבה זיכרון וכתובות פיזיות. כל תוכנית שפונה למשל לתא 1000, תשב בתא 1000 בזיכרון. נבנה מנגנון מיפוי שימיר כתובת לוגית לכתובת פיזית.
דוגמא למנגנון MMU: משתנה i ומצביע אליו. נניח שהמשתנה יושב בכתובת 1000. נניח שיש מנגנון שממפה את הכתובת הלוגית 1000 ל2000. אז תהליך יכנס עם הכתובת 1000 ויגיע ל2000, תהליך אחר יגיע לתא אחר.
כלומר, שני התהליכים משתמשים באותן כתובות לוגיות אבל פיזית הם משתמשים בכתובות שונות.

בסכמה המאוד פשוטה, מה שנעשה זה לבוא ולקחת כתובת לוגית מהמעבד ונשים לזה relocation register- קח את הכתובת שקיבלת ותוסיף לה offset (מספר כלשהו). הערך שברגיסטר הזה יתווסף לכל כתובות שנוצרה ע"י תהליך המשתמש בזמן שהיא נשלחת לזיכרון. תוכנית המשתמש מתעסקת עם כתובות לוגיות ואף פעם לא רואה את הכתובת הפיזית האמתית.



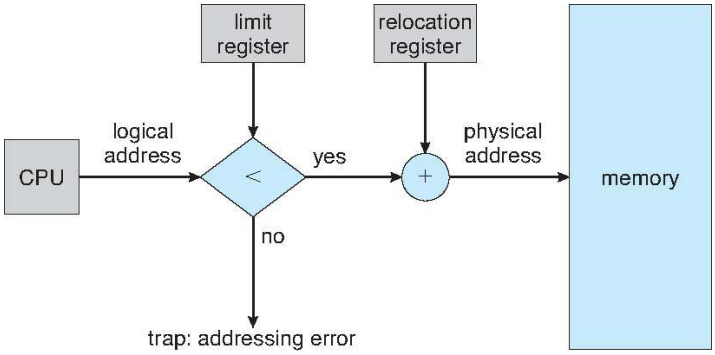
<u>Dynamic Loading:</u>
כל התוכנית צריכה להיות בזיכרון על מנת שתהיה מבוצעות. שום דבר לא נטען עד שלא קוראים לו דבר שגורם לשימוש טוב יותר של מרחב הזיכרון- תוכנית שלא בשגרה לא נקראית אף פעם.

<u>Dynamic Linking:</u>
Static Linking- מעלים את כל הקוד בפעם אחת. הספריות והקוד מתחברים ע"י הטוען לתוך תמונת תוכנית בינארית.
Dynamic Linking – שימוש במיוחד עבור ספריות. הקישור מושהה עד זמן הביצוע. יודעים לטעון חלקים מהקוד תוך כדי זמן ריצה. בשביל לייצר דינאמיות נשתמש בstub – חתיכות קוד קטנות המשומשות כדי לאתר את הספרייה המתאימה שצריך עכשיו. נשים רק את הפונקציה בלי המימוש האמיתי אלא יש שם פונקציה שאומרת לטעון איזה ספרייה בזמן ריצה. הstub מחליף את עצמו עם הכתובות של הroutine ומבצע אותה. מ"ה בודקת אם הroutine נמצאת בכתובת הזיכרון של התהליך. אם לא – מוסיפה אותה לשם.
נטען את הפונקציות הנ"ל רק אם צריך להשתמש בהן בזמן ריצה, אחרת לא.

Contiguous Allocation:

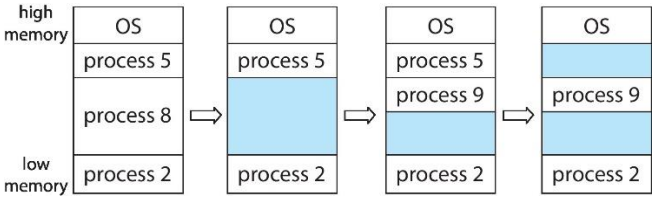
הזיכרון הראשי חייב לתמוך גם במ"ה וגם בתהליכי המשתמש. משאבים מוגבלים חייבים להיות מוקצים ביעילות. "הקצאה רציפה" היא מתודה אחת. הזיכרון הראשי מתחלק לשתי מחיצות:
מ"ה מוחזקת בדר"כ בזיכרון נמוך עם ווקטור interrupt. תהליכי משתמש מוחזקים בזיכרון גבוה.
כל תהליך מוחזק בסעיף יחיד בצורה רציפה בזיכרון.
Relocation registers נועדו כדי להגן על התהליכים אחד מפני השני ומשינויים של הקוד והמידע של מ"ה.
רגיסטרי הבסיס מכילים ערך של הכתובות הפיזית הכי קטנה
רגיסטרי גבול מכילים טווח של כתובות לוגיות – כל אחת חייבת להיות פחות מרגיסטר הגבול.
MMU ממפה כתובות לוגיות בצורה דינאמית.
בקישור דינאמי כזה יכולות לצוץ בעיות כמו למשל שהקוד של הקרנל מתחלף או שהוא משנה את הגודל שלו. אנחנו צריכים לשמור על רצף בזיכרון.
מבחינת הכתובות ברמה הלוגית יכולה להיות דרישה לבנות משהו רציף. צריך לנהל את הזיכרון בצורה יעילה ולוודא שיש יכולת להיות יותר גמישים.

- במודלים שראינו עד כה הזיכרון צריך להיות רציף



Variable Partition (מחיצות משתנות):

הקצאות מרובות חלקים. הרמה של המולטיפרוגרמינג מוגבלת במספר החלוקות.
גודל המחיצות המשתנות כגודל שהתהליך הנתון צריך.
חור – בלוק של זיכרון זמין. חורים בגדלים שונים מפוזרים בזיכרון.
כאשר תהליך מגיע מוקצה לו זיכרון מהחור שמספיק גדול כדי לתמוך בו. תהליך שיוצא משחרר את החלק שלו דבר שמאפשר למחיצות צמודות להשתלב.
מ"ה שומרות מידע על – מחיצות שמוקצות, וחורים.



לפי המודל הזה הזיכרון חייב להיות רציף. אם תהליך מסויים סיים את חייו יש לנו חור בזיכרון, איך נבחר לאן להכניס את התהליך שיבוא אחריו?
יש 3 אסטרטגיות:

1. First-fit: נקצה לתהליך את החור הראשון שהוא מספיק גדול בשבילו. ירוץ כנראה ב(O(1
2. Best-fit: נקצה את החור הכי קטן שהוא מספיק גדול עבור התהליך. חייב לחפש בכל הרשימה כל עוד היא לא ממוינת לפי גודל. בוא נחפש מסט הלוקציות הקיימות את הלוקציה שהיא בדיוק בגודל שלו או עם הרווח המינימלי. תמיד יעבור על כל החורים. דורש הכי הרבה משאבים (רץ (O(n
3. Worst-fit: בוא נמצא את האזור (החור) הכי גדול ונדחוף אותו שם (בעצם נייצר עוד חורים בזיכרון). יעבוד ב(O(1.

פירסט ובסט טובות יותר מworst במונחים של מהירות ושימוש בזיכרון.

<u>Fragmentation (חלוקה, פיזור)</u>:

1. פיצול חיצוני: כל מרחב הזיכרון קיים כדי לבצע בקשות אבל הוא לא רצוף. יש מרווחים בין המקטעים הרציפים שלי. החורים שבין התהליכים שלי.
2. פיצול פנימי: הקצאת הזיכרון יכולה להיות גדולה טיפה יותר ממה שצריך. נקצה גדלים קבועים תמיד. החיסרון- בתוך אותם מקטעים קבועים, אם התהליך לא מנצל את כולו יש לנו חור בתוך התהליך שלא נוכל למלא אותו. היתרון- תמיד נמצא מקום פנוי.

נפחית פיצול חיצוני ע"י compaction (דחיסה).
נערבב את הקשרי הזיכרון כך שימוקמו בכל הזיכרון הפנוי בבלוק אחד.
הדחיסה אפשרית רק אם ההקצאה היא דינאמית ומבוצעת בזמן ביצוע.

<u>Paging-דפדוף</u>
עוברים מהמודל שהזיכרון הפיזי רציף למנגנון עוד יותר דינאמי שאינו רציף.
מוקצה לתהליך זיכרון פיזי בכל פעם שהדף זמין. מונע פיצול חיצוני ובעיות של צ'אנקים בגדלים משתנים.
מנגנון שמבוסס על זה שאנחנו מחלקים את הזיכרון הפיזי למקטעים בגודל קבוע (= frames, הגודל הוא חזקה של 2 בין 512 ביטים לmbytes16), נחלק את הזיכרון הדינאמי לבלוקים באותו גדול הנקראים דפים.
נהיה במעקב אחר כל הframes הריקים.
כדי להריץ תוכנית בגודל N דפים, צריך למצוא N פריימים ריקים ולטעון את התוכנית.
בסופו של דבר נבנה טבלת מיפוי כדי שאומרת איך ממפים כל מקטע כזה לframes.
Page table- טבלת דפים שעושים מיפוי בין כתובת לוגית לכתובת פיזית

<u>מה היה בשיעור קודם?</u>
התחלנו לדבר על הזיכרון הראשי. בגדול דיברנו על שיטות איך ממפים זיכרון לוגי. בסופו של דבר נרצה שתהליכים לא ידרסו אחד את השני.
הגנה- לכל תהליך יש איזור זיכרון משל עצמו. נותנים לו כתובת בסיס וגבול. כל תהליך מקבל טווח ששונה מהטווח של תהליך אחר.
יש מנגנון חומרתי בעל שני רגיסטרים-אחד נותן כתובת בסיס והשני את הגבול. החומרה תוודא שהכתובת שאליה נרצה לפנות נמצאת בטווח.
בעיות – לא בהכרח יודעים מה יהיה הטווח כתובות של התהליך הpriory. בגלל זה נצטרך להשתמש בbinding – התוכנית כתובה על בסיס של איזה כתובת בסיס שאותה היא מקבלת בשלב יותר מאוחר (שלב הbinding). קורה בזמן ריצה.
הקומפיילר בונה את התוכנית, החל משלב הלינקר נוכל לעשות אדרס ביינדינג.

אופציה נוספת – הפרדה בין כתובת לוגית לכתובת פיזית. בשביל לעבור מכתובת לוגית לכתובת פיזית נשתמש במנגנון מיפוי כזה או אחר.

דיברנו על מנגנון שבו נכנסת כתובת לוגית לCPU, תעבור אל הMMU, נקבל כתובת פיזית ואז ניכנס לזיכרון.
למה זה טוב – פחות נדרשים לעשות binding וכולי כי אפשר לתת לכמה תהליכים שונים את אותן כתובות ובתהליך התרגום נתרגם אותם לכתובות שונות. אם נרצה להזיז דברים בזיכרון, נוכל לעשות זאת בגלל שאנחנו שולטים על המנגנון הזה.

שיטה נוספת היא להשתמש בrelocation register והוא למעשה מוסיף כתובת בסיס לoffset. למשל הכתובת הלוגית היא 346, הרגיסטר מוסיף 14000 אז הפיזית תהיה 14346.

טעינה דינאמית – היכולת לטעון חלקים מהתוכנית בזמן ריצה ובפרט לעשות קישור דינאמי – נטען את החלקים בזמן ריצה בשתי דרכים: או בצורה מפורשת או שנגיד ללינקר שיטען רק מה שצריך. נעשה ע"י stub.

הקצאה רציפה – אם נסתכל על המודל הקודם (עם הגבול וה-relocation), הזיכרון שם של כל תהליך רציף בזיכרון הפיזי.

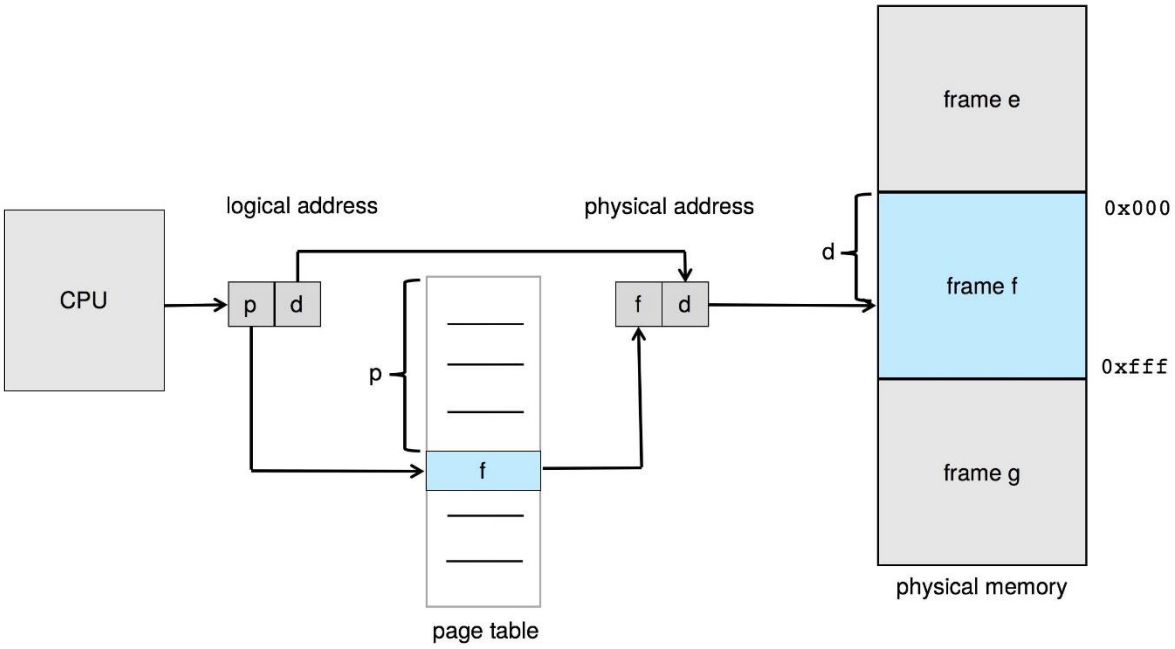
<u>קיטועים – fragmentation</u>
האלגוריתם שתיארנו עם הholes יוצר בעיות של קיטועים – בגלל שהתהליכים נטעמים ומוצאים מהזיכרון הזיכרון הפנוי "נשבר" לחתיכות קטנות.
יש שני סוגים:
external – כשיש מרחבי זיכרון רציפים. כשהתחלנו להכניס ולהוציא, המרווחים בין התהליכים בזיכרון לא מאפשרים להכניס תהליכים, נוצרים מרווחים מאוד גדולים בזיכרון.
נפתור את זה ע"י compaction: תהליך שמצמיד בין תהליכים ומצמצם מרווחים בזיכרון.
Internal- במקום להתייחס לזיכרון רציף, נחלק אותו לחלקים בגודל אחיד. ואז יכולים להיווצר חורים בתוך מרחבי הזיכרון כי אולי תהליך לא יכסה את כל המקטע שנתנו לו.
במקרה הממוצע הקיטוע הוא 0.5 מגודל הבלוק.

<u>המודל הבא : paging</u>
פתרון נוסף לבעיית הexternal fragmentation.
מחלקים את הזיכרון הפיזי (RAM) לחלקים בגודל קבוע הנקראים frames(מסגרת). המקבילה שלהם בזיכרון הלוגי זה page. אנחנו רוצים להמיר frames לpages.
אם נרצה להריץ תוכנית בגודל N pages, נצטרך למצוא N frames פנויים.

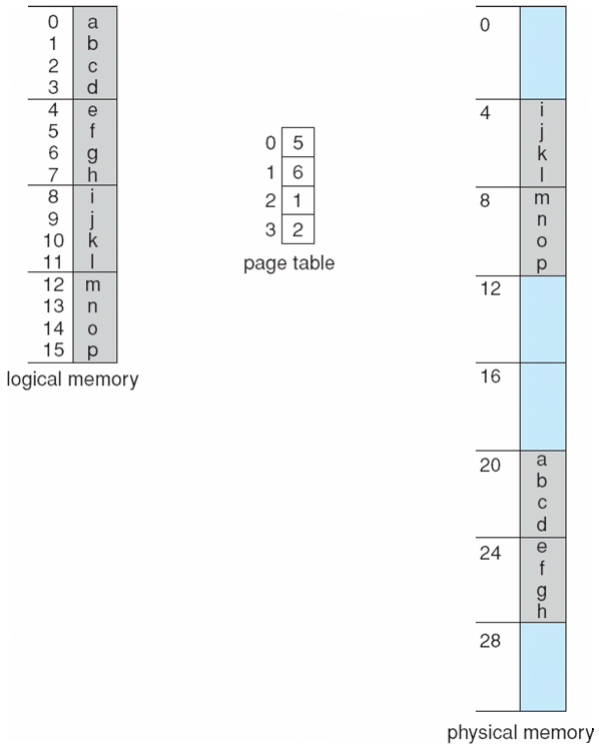
<u>טבלת דפים –</u>
(p) Page number: משמש כאינדקס לטבלת הדפים שמכילה כתובת בסיס של כל דף בזיכרון הפיזי.
(d) Page offset: משולב עם כתובת הבסיס כדי להגדיר את כתובת הזיכרון הפיזית שנשלחת ליחידת הזיכרון.

page number	page offset
p	d
m -n	n

<u>איך זה נראה:</u>
חילקנו את כל הזיכרון לדפים.
קיבלנו כתובת לוגית, לקחנו את הoffset ושמנו בצד. ניקח את הpage number המשמש כאינדקס לטבלת הדפים. נכנסים לתא המתאים ובו יש כתובת פיזית של הframe. נעביר אותה לf ונשרשר את הoffset.
נמשיך ונגיע לתוך הframe המתאים.



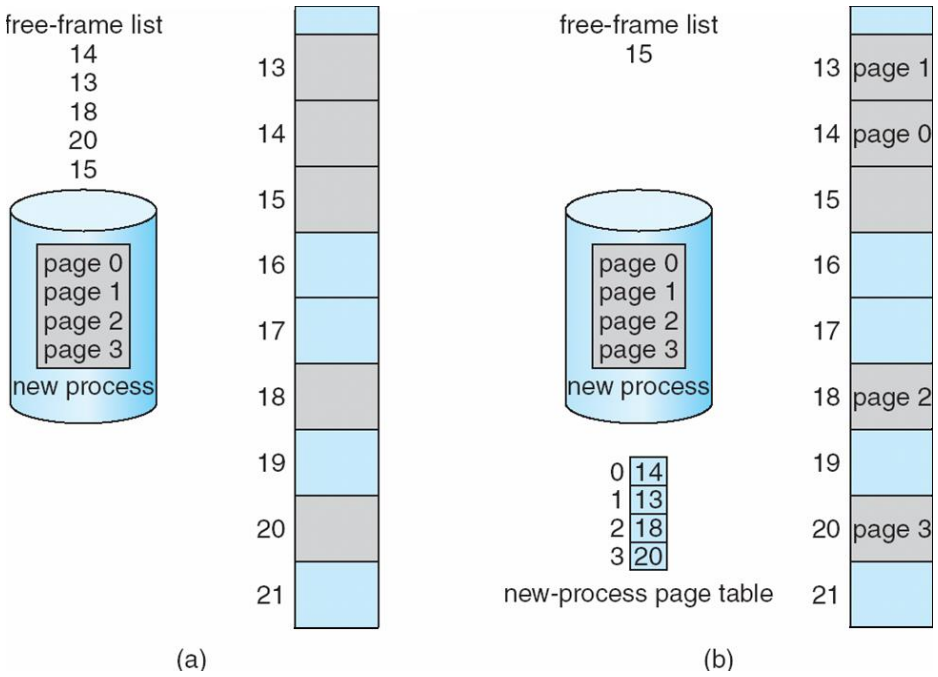
דוגמא לדפדוף:



מהו גודל הframe שטוב לנו?
פריימים גדולים מדי יצרו יותר מדי קיטועים, עבור פריימים קטנים נצטרך להגדיל את גודל טבלת הדפים כי מספר הכניסות הפוטנציאלי הולך וגדל.

Free frames:

אם במצב הקודם שמרנו רשימה של הקצאות חופשיות (חורים), עכשיו אנחנו שומרים רשימה של frames שהם חופשיים. כל פעם שתהליך מגיע ודורש frame אחד או יותר הוא יקבל אותה מהרשימה הזו.
אם נדרוש 4 דפים, נבנה טבלה רציפה עבורם ונקצה להם את הפריימים הפנויים.



- אין צורך לשמור על רציפות בזיכרון הפיזי, פשוט מקבלים רשימה של מה פנוי ומקצים לתהליך.

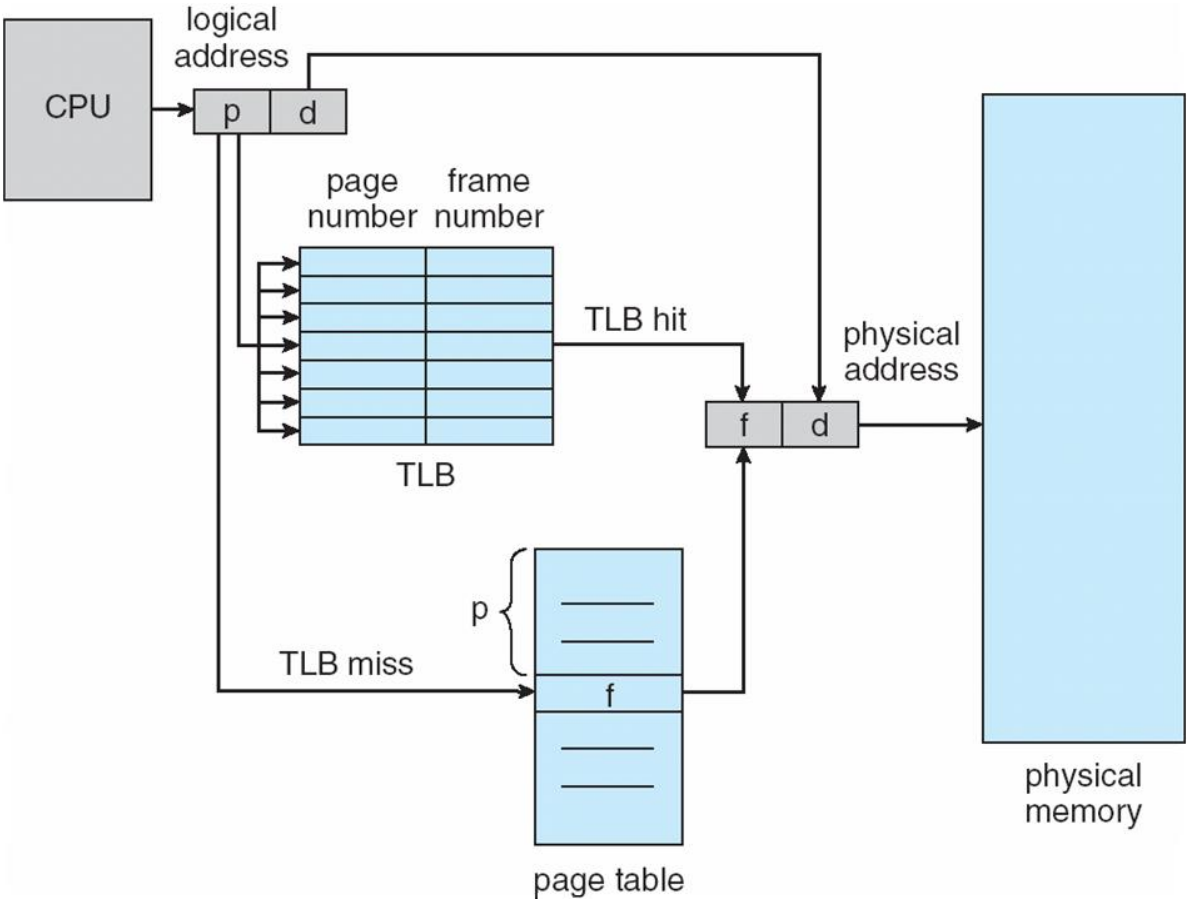
מימוש של טבלת דפים:

טבלה הנמצאת במרחב הזיכרון של הקרנל. כל פעם שנעשה context-switch, מ"ה צריכה להגיד לחומרה באיזה טבלה להשתמש עבור כל תהליך. בPCB של התהליך יש שני רגיסטרים:
PTBR: אנחנו נותנים לחומרה, למעבד את כתובת הבסיס לטבלת הדפים שלנו. עכשיו הוא יודע איפה היא נמצאת ויכול לגשת אליה ישירות.
PTLR: גודל טבלת הזיכרון

המעבד יודע בזמן ריצה להגיע לטבלת הדפים ולהגיד האם חרגנו- האם הגענו למספר דף שחורג ממה שהוקצה לו.

מה הבעיה פה? זמן הגישה לזיכרון כי כל מידע/הוראה שרוצה גישה דורשת למעשה גישה לשני זיכרונות – אחד עבור טבלת הדפים ואחד עבור המידע/ההוראה.
הבעיה יכולה להיפטר ע"י שימוש בcache מיוחדת שנקראת TLB (transaction look-aside buffer).
נבנה אוסף של רגיסטרים בזיכרון מהיר במיוחד. הם מכילים מספר מסויים של כניסות מטבלת הדפים.
כאשר המעבד מייצר כתובת לוגית מספר הדף שלה מושווה מול הרגיסטרים. אם הוא נמצא ניתן לקבל מהרגיסטר את מספר הframe המתאים ולפנות ישירות לזיכרון. אם המספר לא מופיע – נחפש את הframe בטבלת הדפים ונעדכן את אחד הרגיסטרים בערך הזה.
למעשה הTLB הוא cache שמוצמד לזיכרון והרעיון הוא שכשאר נוציא דפים מהזיכרון יתקיים: אם אלה דפים ששינינו נשמור אותם, אחרת פשוט נכתוב עליהם.

- כשמבקשים דף, מבקשים אותו מהדיסק וכדי לקבל אותו יש תהליך שלוקח זמן. הרעיון של TLB הוא שדפים שיוצאים נשמרים בו בעזרת אלגוריתם מסוים שמהיר יותר מהזיכרון ולכן כאשר רוצים דף חדש, נבדוק קודם אם הוא מופיע בTLB ולכן זה חוסך את הגישה לדיסק.

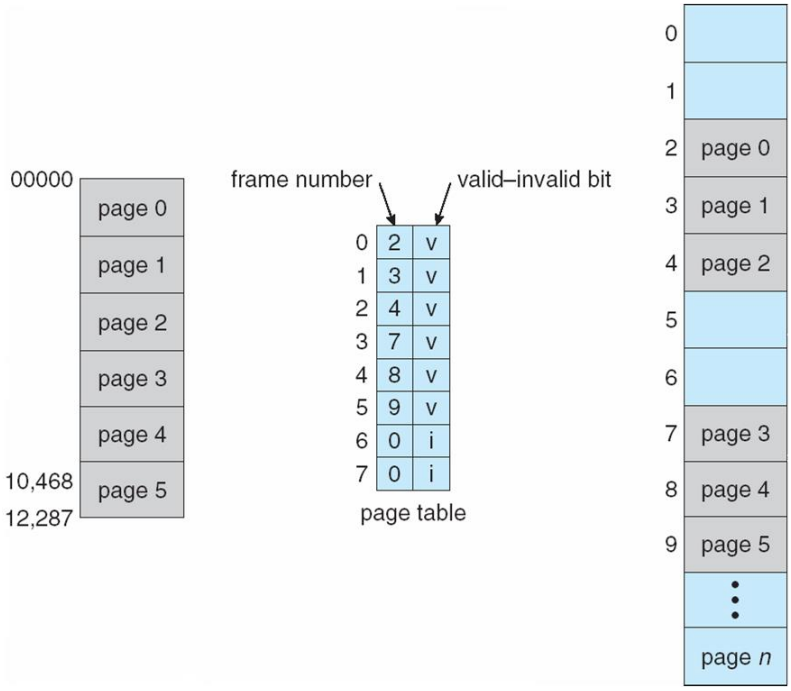


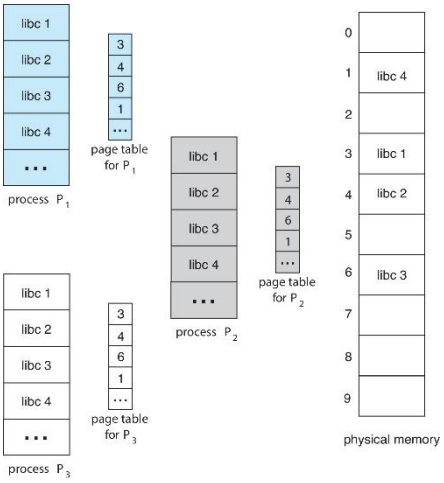
<u>שיעור 12:</u>

נשאלת השאלה – איך נחשב את זמן הגישה שלנו? נרצה לדעת פחות או יותר כמה זמן הולך לקחת לנו לגשת לזיכרון.
נעשה הערכות ולפי זה נדע זמני גישה.
Hit radio = אחוז הפעמים שמספר דף נמצא בTLB
נניח שבהסתברות של 80% יקח לנו 10 ננו-שניות לגשת וב20% ההסתברות היא שתהיה פניה כפולה לזיכרון.
הEAT(=effective access time) של זה = 10*0.8 + 20*0.2 = 12.
ככל שיש לנו יותר hits בTLB הזמן גישה שלנו ירד.

<u>Memory protection:</u>

אם עד עכשיו מיפינו דף לפריים, השלב הבא יהיה להוסיף לכל דף גם ביטים. הם יגידו אם מותר לגשת לדף או אסור. (v=valid, i=invalid).
אם נבוא למפות את דף 3, נלך לטבלה שלו ונראה שהוא ממופה לפריים מספר 7 שהוא v ולכן הכל בסדר.
אם נמפה עד דף 5 ועכשיו נפנה לדף 6 נקבל i ולכן לא ייגשו אליו כי הוא לא נמצא בזיכרון ואז היא תזרוק exception ותעיר את מ"ה.
חשוב לזכור שהחומרה היא זו שעושה את המיפוי. מ"ה ממלאת את הטבלה הזו.



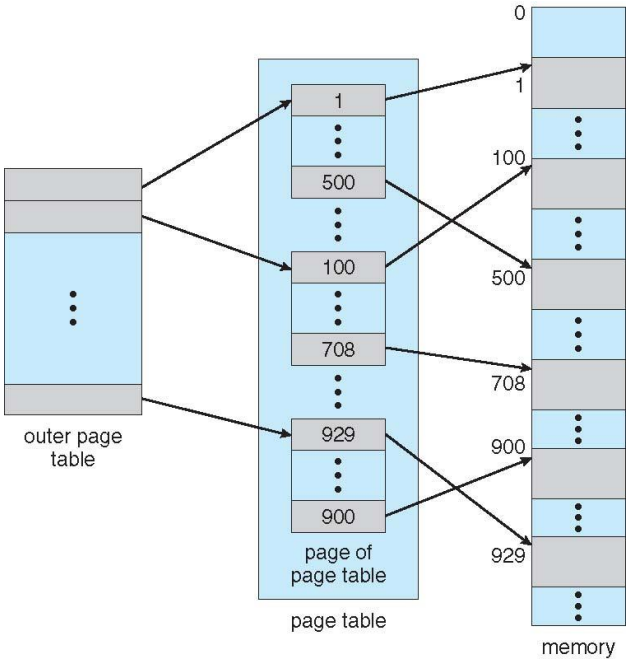


<u>דפים משותפים:</u>

קוד משותף – העתק אחד של קוד שהוא לקריאה בלבד שמשותף בין תהליכים. דומה לכמה טרדים שחולקים את אותו מרחב תהליך.
אפשר לעשות את זה גם על דפים שמכילים דאטה. כזה בין התהליכים נצטרך לעשות את זה בצורה מפורשת (תהליך לא ישתף עם אחר אלא אם כן הוא ביקש ממנו במפורש).
קוד ודאטה פרטי – כל תהליך שומר העתק נפרד של הקוד והדאטה.
הדפים של הקוד הפרטי והדאטה יכולים להופיע בכל מקום במרחב הכתובות הלוגי.

<u>היררכית טבלת הדפים:</u>

נוכל לחלק את טבלת הדפים למספר טבלאות דפים, למעשה נייצר היררכיה.
שובר את מרחב הכתובות הלוגי אל הרבה טבלאות דפים. טכניקה פשוטה היא two-level page table.
הטבלה החיצונים מצביע להרבה טבלאות דפים שהן יכילו את המיפוי לפריימים. כך נוכל להחזיק ברגע נתון רק חלק מהטבלה בזיכרון ולא את כולה.



<u>two-level paging:</u>
כתובת לוגית מחולקת ל- מספר דף שמורכב מ22 ביטים וoffer של דף המורכב מ10 ביטים.

page number		page offset
p_1	p_2	d
10	10	12

P1 זה האינדקס לouter page table וp2 זה המיקום של הדף בטבלה הפנימית.
איך נחשב את הזמן גישה במקרה הזה? EAT
ב80% פנינו ב10 ננו-שניות. בהסתברות של 20% יש לנו גישה לטבלה החיצונית, לפנימית ואז לדף עצמו = 30.
EAT = 0.8*10 + 0.2*30 = 14

<u>ארכיטקטורה של 64-ביט:</u>
מרחב הזיכרון הפוטנציאלי הוא עצום. אם גודל הדף הוא KB4 (2^52) : לטבלת הדפים יש 2^52 כניסות, בסכמה של שתי הרמות לטבלה הפנימית יכולה להיות 2^10 כניסות. הכתובת תראה כך:

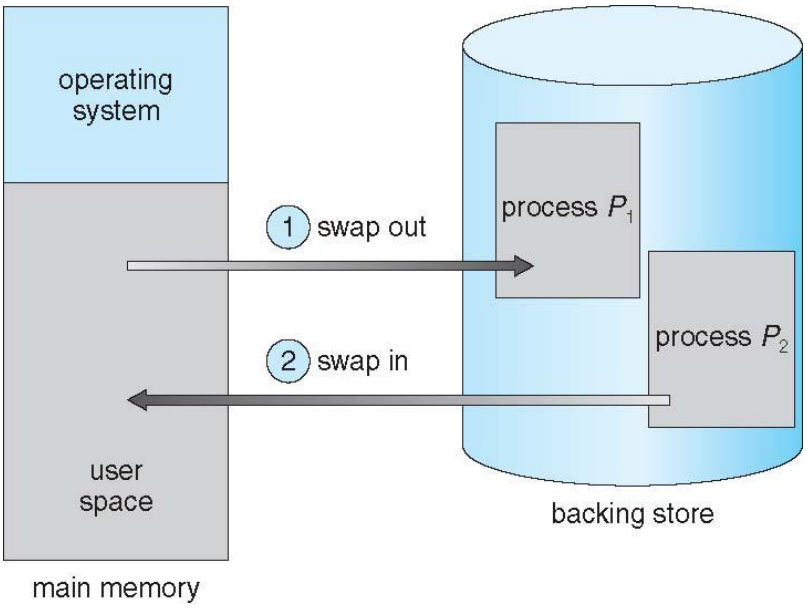
outer page	inner page	offset
p_1	p_2	d
42	10	12

outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

<u>Swapping:</u>
תהליך יכול להיות מוחלף באופן בזמן מחוץ לזיכרון לbacking store ואז ייבאו אותו חזרה לזיכרון כדי שימשיך לרוץ. ניקח תהליכים שהם לא זמינים/לא משתמשים בהם ונוציא אותם לדיסק במקום שיהיו בRAM.
Backing store – דיסק מהיר הגדול מספיק כדי להכיל עותקים של כל תמונות הזיכרון לכל המשתמשים. חייב לספק גישה ישירה אל תמונות הזיכרון הללו.
Roll out, roll in- הם מוחלפים בעזרת אלגוריתמי תזמון. ההחלפה נעזרת בעדיפויות. תהליך בעדיפות נמוכה יוחלף כדי שיכנס תהליך בעדיפות גבוהה.
המערכת מתחזקת ready queue של התהליכים המוכנים לרוץ שיש להם תמונה בדיסק.

- האם התהליך שהוצא החוצה צריך לחזור חזרה אל אותה כתובת פיזית? תלוי בשיטת ה address binding ונשקול גם תלות בIO.



אם התהליך הבא שאמור להיכנס לCPU לא בזיכרון, צריך להוציא תהליך ולהחליף ביניהם. זמן של context awitch יכול להיות גבוה מאוד. MB100 לעשות swapping לתהליך לדיסק קשיח עם שיעור העברה של 50mb. (זמן הswap out ms2000, פלוס זמן של swap in, סה"כ קיבלנו קונטקסט-סוויצ 4000ms)
נוכל להפחית אם נדע כמה זיכרון באמת בשימוש.
דבר שנוסף שצריך לקחת בחשבון כשאר עושים swapping- אם תהליך מחכה לIO אי אפשר להוציא אותו כי הIO שלו יגיע לתהליך הלא נכון.

בדר"כ לא נוציא את כל התהליך מהזיכרון אלא רק חלק מהדפים שלו כאלה שהוא כמעט לא משתמש בהם.

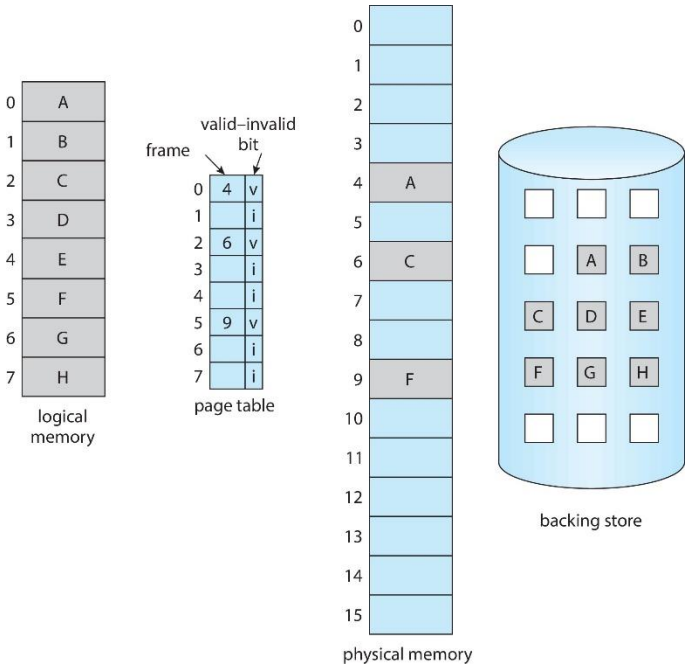
<u>מצגת 10: זיכרון וירטואלי</u>
הנחה: לא כל התוכנית בזיכרון ואם כולה בזיכרון היא לא ברצף.
זיכרון וירטואלי זו טכניקה המאפשרת הרצת תהליכים שאינם נמצאים בשלמותם בזיכרון, ומה שנמצא בו לא נשמר באופן רציף. היתרון המרכזי – הזיכרון הלוגי יכול להיות גדול יותר מהזיכרון הפיזי.
יתרונות: תוכנית לא מוגבלת בכמות הזיכרון הפיזי הפנוי, משתמשים יוכלו לכתוב תוכניות למרחב כתובות לוגי גדול יותר שיקל על העבודה, יותר תוכניות יוכלו לרוץ בו"ז ויהיה דרוש פחות IO לטעינה והחלפה של תוכניות משתמש. כל התוכניות ירוצו מהר יותר.

ניתן ליישם זיכרון לוגי בשתי שיטות-

1. <u>Demand paging</u>

יכול להביא את כל התהליך לתוך הזיכרון בזמן טעינה. או לחילופין להביא דף לזיכרון רק כשצריך אותו. מערכת הדומה למערכת paging המשתמשת בswapping. תהליכים נשמרים בזיכרון המשנה. כאשר נרצה להריץ תהליך נכניס (swap) אותו לזיכרון. בנוסף נשתמש בlazy swapper- הוא לעולם לא יכניס לזיכרון תהליך אלא אם כן יש בו צורך. נשתמש במונח pager במקום בswap.
כאשר נרצה להביא תהליך לזיכרון, הpager ינחש אילו דפים יהיו בשימוש לפני שהתהליך יוצא שוב אל מחוץ לזיכרון. במקום להכניס את כל הזיכרון הpager מביא רק את אותם דפים הדרושים. כך למעשה אנחנו נמנעים מקריאת דפים שאין בהם צורך בכלל ובכך מפחיתים את זמני ההחלפות ואת כמות הזיכרון הפיזי הנדרשת.

בשיטה זו צריך לתמוך בחומרה שיודעת להבדיל בין דפים בזיכרון לדפים שנמצאים בדיסק. נוסיף לכל דף ביט valid-invalid . ערך valid מצין שהוא חוקי ונמצא בזיכרון. ערך invalid מציין שהוא לא חוקי או לא נמצא בזיכרון. כשהערך הוא חוקי, הכניסה המתאימה לדף בטבלת הדפים מעודכנת כרגיל אבל כאשר הוא לא חוקי הכניסה מסומנת בinvalid או מכילה את כתובת הדף בדיסק.



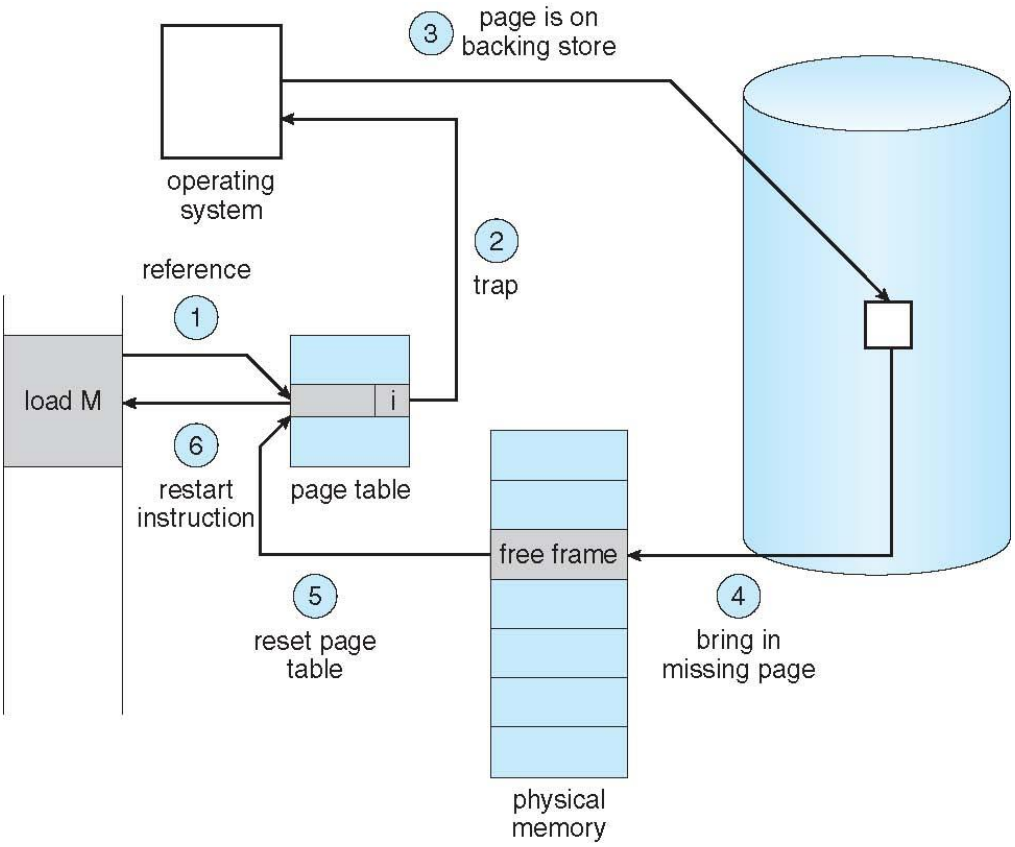
<u>Page fault:</u>

גישה לדף המסומן כinvalid גורשת לשליחת page fault trap אל מ"ה. Trap הוא תוצאה של ניסיון של מ"ה להביא דף רצוי לזיכרון והוא שונה משגיאת כתובת לא נכונה שהיא תוצאה של ניסיון גישה לכתובת לא חוקית. צריך להבין בגלל מה נגרמה השגיאה.

טיפול-

1. מ"ה תסתכל בטבלה הפנימית של התהליך (נמצאת בPCB) כדי לקבוע האם ההפנייה לזיכרון חוקית או לא.
2. אם לא חוקית נפסיק את התהליך. אם היא חוקית והדף לא נמצא-נביא אותו.
3. נמצא frame פנוי
4. נתזמן פעולת דיסק שתקרא את הדף המתאים אליו
5. כאשר פעולת הדיסק הושלמה, נעדכן את הטבלה הפנימית של התהליך וכן את טבלת הדפים שתצביע על כך שהדף נמצא בזיכרון
6. נאתחל את ההוראה שהופרעה ע"י הtrap. התהליך יוכל כעת לגשת לדף המתאים כאילו תמיד היה שם

שלבים בטיפול בpage fault:



<u>פתרון מבחן לדוגמא מועד א:</u>

<u>שאלה 2-</u>

Waiting time: כמה זמן התהליך נמצא מחוץ לCPU. כמה זמן הוא בready queue לעומת running

ציר הזמן – round robin
בזמן 0 יש רק את תהליך P1
בזמן 2 מגיע P2, נכניס אותו ל2 קוואנטות וP1 נכנס לready queue
בזמן 3 P1 רץ, ו(1)P2,P3 בready queue
בזמן 4 P2 רץ וP1 P3 בready queue
בזמן 5 P3 רץ ו(1)P1,P2,P4 בready
בזמן 6 P1 ירוץ 2 שניות וP2,P4,P3 בready
בזמן 7 P2 ירוץ שניה וP4, P1 בready
P4 יכנס לעוד 2 שניות ואחריו P1 (עד שניה 16) ואז נשאר להריץ רק את P4 עד הסוף.

- הזמן הממוצע הוא – נבדוק לגבי כל תהליך כמה זמן היה בwaiting queue.
  P1: 9 שניות, P2: 6 שניות, P3: 3 (הגיע ב5 והתחיל ב8), P4: 5 שניות
  סה"כ = 5.75 שניות

ציר הזמן – SRTF (כל פעם שיגיע תהליך חדש, נבדוק כמה זמן נשאר לו לרוץ ולפי זה נכניס)
תהליך 1 רץ 2 שניות ואז מגיע תהליך 2, לשניהם נשארו 5 שניות לרוץ לכן נשאיר את P1 שימשיך לרוץ. הוא ממשיך עד 5 שניות ואז מגיע P3, הוא צריך 2 שניות ולתהליך 1 גם נשארו 2 שניות לכן P1 יסיים. בזמן 7 נכניס את P3 כי יש לו רק 2 שניות. בזמן 9 מגיע תהליך 4, לו ול2 יש אותו זמן ביצוע ולכן נכניס את 2 כי הוא היה קודם והוא ראשון בתור. תהליך 2 ירוץ 5 שניות ואז יכנס תהליך 4 לעוד 5 שניות.
P1 = 0, P2 = 7, P3 = 2, P4 = 5. סה"כ זמן המתנה ממוצע: 14/4=3.5

ציר הזמן multi-level
תהליך 1 רץ למשך 3 שניות, תהליך 2 מגיע אחרי 2 שניות ונכנס לQ=3. תהליך 1 נכנס גם לשם.
תהליך 2 רץ 3 שניות וחוזר לתור, באמצע נכנס לתור תהליך 3 . אח"כ תהליך 1 יוצא רץ עוד 3 שניות ונכנס לתור Q=4. תהליך 4 מגיע ב9 ונכנס לQ=3. היות והתור השני לא ריק נבחר את תהליך 1 משם והוא יחזור לרוץ ויסיים. התור Q=4 ריק ולכן נבחר מQ=3. תהליך 3 רץ 2 כי זה מה שנשאר לו. תהליך 2 יכנס וירוץ עוד 2 שניות ויסיים ואחריו תהליך 4 עד הסוף.

- זמן ההמתנה: p1 = 3, p2 = 7, p3 = 5, p4 = 5. סה"כ 20/4 = 5.

<u>שאלה 3-</u>

Two-level paging: יש לנו טבלה חיצונית ופנימית.
נתונים: גודל דף 8 ק"ב, הגודל של הTLB הוא 128 כניסות, זמן גישה לזיכרון ns50, זמן גישה לTLB הוא ns1.
יש תהליך עם 40 מ"ב של זיכרון.
מה הEAT?
דבר ראשון נשאל כמה פריימים יש בתהליך: 40mb/8k = 5k. (דפים)
הTLB זוכר מתוך הK5 רק 128 דפים.
ההסתברות לtlb hit:1/40 = 128/5k (ההסתברות שדף שמור כבר בTLB)
EAT = p(hit) * AccessTime + 1-p(hit)*3AccessTime
= 0.025*51 + 0.975*150 = 1.275+146.25 = 147.525

שאלה 4 מועד ב -

MemoryAccessTime = 0.8*5ns + 0.2*50ns = 14

EAT = 0.8*(1ns + MAT) + 0.2*(2*MAT) = 12 + 5.6 = 17.6

בחלק הראשון יש רק פניה לRAM ובחלק השני יש שתי פניות: פנייה לטבלת הדפים ופנייה לRAM

שאלה 5 –

א. P1 :

T0 = 5, T1 = 0.5*t0 + 0.5*T0 = 0.5*3+0.5*5 = 4

T2 = 0.5*5 + 0.5*4 = 4.5

T3 = 0.5*2 + 0.5*4.5 = 3.25

T4 = 0.5*4 + 0.5*3.25 = 3.6125

T5 = 0.5*6 + 0.5*3.6125 = 4.8

ב. ניקח את התוצאות מהטבלה בא'. כל התהליכים הולכים לרוץ 5 שניות

עבור SRTF – p3(5), p1(5), p4(5), p2(5)

P3 p3 p3 p3 p3 p1 p1 p1 p1 p1 p4 p4…..

עבור RR – p1(3) p2(3) p3(3) p4(3) p1(2)…

P1 p1 p1 p2 p2 p2….p4 p4 p4 p1 p1 p2 p2 p3 p3 p4 p4

תרגולים

קריאות מערכת בלינוקס (תרגול 1):

אחד מהשירותים שעלינו לבצע קריאת מערכת על מנת לבצע אותו זה טיפול בקבצים. כל תהליך מחזיק אוסף את מזהים לקבצים שהוא עושה בהם שימוש. מזהה זה נקרא file descriptor.
הטיפול בקבצים יעשה בעזרת ייבוא ספרית fnctl.h

- פתיחת קובץ: הפונקציה open() תיצור קריאת מערכת ע"מ לפתוח את הקובץ.
  Int open(const char* path, int oflags, mode_t mode)
  - הערך המוחזר הוא המזהה של הקובץ
  - Path- הכתובת של הקובץ
  - Oflags- דגלים המייצגים את המטרה שלשמה פתחנו את הקובץ
- סגירת קובץ – int close(int fd)
- קריאה – size_t read(int fd, void* buf, size_t nbytes)
  - Fd- מזהה הקובץ שאותו נרצה לקרוא
  - Buf- לאן לקרוא
  - Nbytes- כמה ביטים לקרוא
  - יחזיר את מס' הבתים שקרא במקרה של הצלחה ו-1 אם לא הצליח לקרוא, 0 אם לא היה מה לקרוא
- כתיבה – size_t write(int fd, void* buf, size_t nbytes)
  - Buf- שמור שם בפוינטר המידע שאותו נרצה לכתוב

דוגמאות חיוניות לפרמטר oflags:
קריאה בלבד – O_RDONLY
כתיבה בלבד – O_WROBLY
כתיבה וקריאה – O_RDWR
יצירת קובץ – O_CREAT
להוסיף – O_APPEND

דוגמאות חיוניות לפרמטר mode:
הרשאות קריאה לבעל הקובץ – S_IRUSR
הרשאות כתיבה לבעל הקובץ – S_IWUSR

ניהול תהליכים בלינוקס (UNIX) : (תרגול 3)

תהליך הינו מופע של תוכנית. תוכנית מורכבת מאוסף של פקודות ותהליך הוא ההפעלה שלהן.
לדוגמא, אם נריץ כמה פעמים את אותה תוכנית ייווצרו לנו כמה תהליכים עבורה.
כדי לאפשר למשתמש להשתמש בהרבה תהליכים, יבצע המתזמן חלוקה של זמן המעבד.
לכל תהליך יש מספר מזהה שנקרא process ID. לכל תהליך יש מצב יציאה (exit status) שנותן אינדיקציה לתהליך האב האם הוא סיים את עבודתו בהצלחה או נכשל.
תהליך שנשאר חי אחריי שסיים עבודתו ייקרא "זומבי".
אם האבא סיים את עבודתו לפני הבן, גם הבן ימות יחד איתו ולכן יכול להיווצר מצב שהבן לא סיים את עבודתו.
מזהה של תהליך מזוהה ע"י הטיפוס pid_t (שייך לספריה sys/types/h)
Fork() – קריאת מערכת היוצרת תהליך חדש (משכפלת את זה שקרא לה).
תחזיר ערך שלילי במידה ונכשל, ערך 0 עבור התהליך החדש וערך חיובי המייצג את הID של הבן אל האבא.
Exec() – קריאת מערכת המשומשת לאחר פורק כדי להחליף את התהליך עם תוכנה אחרת

- אבא קורא wait() ומחכה עד שהילד שלו יסיים לרוץ

תקשורת בין תהליכים- Pipes

צינור המשמש לתקשורת **חד כיוונית** שבו תהליך אחד כותב לתוכו ותהליך אחר קורא ממנו.
חתימת הפונקציה: int pipe(int fd[2]). הפונקציה תחזיר 0 במקרה של הצלחה ו-1 בכישלון.
fd- מערך בגודל 2 מסוג int , המקום ה0 מייצג כתובת לקריאה וה1 לכתיבה.

I/O REDIRECTION (תרגול 5)

```
Int main {
Int x,y
Scanf("%d",&x)
Scanf(("%d",&y)
Printf("%d plus %d equals %d",x,y,x+y)
Return 0
}
```

תוכנית פשוטה לחישוב סכום של שני מספרים. נרצה לחשוב על אפשרות שבה לא נכניס את הקלט דרך stdin ואת הפלט נקבל דרך הstdout.
נקמפל את התוכנית : gcc plus.c -o plus.out
./plus.out < input.txt
הקובץ input.txt יכיל את שני המשתנים עבור הקלט לתוכנית.
זה למעשה input redirection- במקום להכניס דרך הstdin נכניס דרך קובץ.
באותו אופן נוכל לבצע output redirection ע"י הרצת הפקודה ./plus.out > output.txt

איך עושים זאת בשפת C?

שתי קריאות מערכת שיעזרו לנו:

1. Int dup(int o_fd) – מקבלת איזשהו מזהה של קובץ שפתחנו. הקריאה הזו תשכפל את המזהה של אותו הקובץ ותחזיר אותו בreturn value שלה או -1 בכישלון. כלומר יהיו לנו שני מזהים המכוונים לאותו קובץ.
   - מ"ה תקצה fd חדש בנוסף למקורי. הfd החדש יהיה המספר הנמוך ביותר שעדיין לא נעשה בו שימוש. שני הfd יהיו ברי שימוש. ובסיום השימוש נצטרך לבצע close על שניהם.
2. Int dup2(int o_fd, int new_fd) – ההבדל בין הקריאות הוא שפה אני מאפשר למשתמש לבחור מה יהיה הfd החדש שיתייחס לאותו קובץ שהfd המקורי מתייחס אליו. ערך החזרה יהיה המזהה החדש שבחרנו או -1 בכישלון.

- נקודות חשובות:
  - מה יקרה במידה וnew_fd כבר תפוס? הקריאה תסגור את השימוש הקודם בfd לפני שהיא תשתמש בו שוב.
  - אם old_fd יהיה לא חוקי – הקריאה לא תבצע כלום וכמובן גם לא תשנה את מה שnew_fd התייחס אליו קודם לכן.
  - אם o_fd וnew_fd הם אותם ערכים – הקריאה לא תבצע כלום ותחזיר את הnew_fd.

שינוי הקוד ההתחלתי:

```
Int main {
Int x,y
Int fd = open(…,O_RDONLY)
If(dup2((fd,0) == -1) {error}
Scanf("%d",&x)
Scanf(("%d",&y)
Printf("%d plus %d equals %d",x,y,x+y)
```

```
Return 0
}
```

SIGNALS (תרגול 6)

לכל תהליך יש רשימת סיגנלים. כל סיגנל הוא ייעודי למטרה מסוימת ויש לו פונקציה המקושרת אליו שהיא תבוצע כאשר התהליך יקבל את הסיגנל הזה.
ספריה: <signal.h>

MACRO	Signal number	description
SIGABRT	1	כאשר צריך לבצע סיום חריג של התהליך
SIGFPE	2	פעולה חשבונית אסורה
SIGILL	3	הוראה לא חוקית
SIGINT	4	כאשר תהליך מקבל אות מהמשתמש
SIGEGV	5	כאשר קוראים/כותבים מחוץ לאזור שהוקצה לכך

איך זה קורה במערכת ההפעלה?
מ"ה מקצה שני מספרים שלמים לכל תהליך (כל מספר בגודל 32 ביטים). שני המספרים האלו ישמשו:

1. לצורך מעקב אחר סיגנלים שהתהליך קיבל וצריך לטפל בהם
2. לצורך מעקב אחר סיגנלים חסומים

כל תהליך יכול להיות מזוהה עם לכל היותר 32 סיגנלים.

תרגול שבוע 10 – Threads

נשתמש בספריה POSIX threads או בקיצור pthreads. נייבא את הספריה בתחילת התוכנית וכאשר נרצה לקמפל נרשום gcc filename -pthread.

יצירת חוט חדש בתוך תהליך :

Int pthread_create (pthread_t *thread, pthread_attr_t attr, void*(*start_ratine)(void*), void(*args))

thread* – מצביע למזהה של הטרד החדש שניצור
Attr – דגלים/תכונות שאיתם נאתחל את הטרד ביצירה שלו
פונקציה – מצביע לפונקציה שאותה הטרד יתחיל לבצע
args* – ארגומנטים שנשלח לפונקציה

הפונקציה תחזיר 0 במידה והטרד נוצר בהתחלה, -1 במקרה של שגיאה.

<u>סיום חוט בתוך תהליך :</u>
Void pthread_exit(void *retval)
כאשר נרצה לסיים את פעילותו של הטרד אנו נקרא לפונקציה הזו מתוך אותו הטרד שאנו מעוניינים לסיים.
בסיום הטרד נוכל לשלוח את ערך המוחזר מהטרד.
חשוב מאוד לשים לב שהמצביע retval לא יצביע למקום באיזור הזיכרון של הטרד שאנו מסיימים. (כשאנו מסיימים את הטרד, כל האיזור שלו במחסנית נסגרת ולא מוגדרת יותר ולכן לא נוכל לשלוח לשם מצביע)

Int pthread_join(pthread_t thread, void **retval);
פונקציה זו מחכה לסיום של הטרד הראשי עד ששאר הטרדים המשניים יסיימו.
נשלח מזהה לטרד שאנו נרצה שיחכה ומצביע למצביע שמצביע על הערך המוחזר מהטרד שאנו מחכים לו.

<u>גישות לניהול טרדים:</u>

1. User threads – אלו טרדים שמנוהלים בuser space ע"י ספרייה. הליבה לא מודעת לטרדים האלו. הטרדים בגישה זו יהיו מהירים יותר כי לא צריך התערבות של מ"ה בכדי לנהל אותם.
אם אחד התהליכים בטרדים האלו ירצו לבצע קריאת מערכת, כל התהליך יחסם ויושהה ולא רק הטרד הנ"ל כי מבחינת מ"ה יש רק טרד אחד והיא לא יודעת שיש הרבה טרדים לאותו תהליך.
מתזמן התהליכים שנמצא בקרנל לא ייתן תעדוף לתהליך עם 100 טרדים כי מבחינת מ"ה הכל אותו דבר.
2. Kernel threads – בתוך הkernel space יש גם את הpcb וגם את הtcb(=כמו של תהליכים רק לטרדים). כלומר, הניהול של החוטים נעשה ע"י מ"ה. זה בתחום אחריותה ולא נעשה ע"י ספרייה.
בגלל שמ"ה יודעת כמה טרדים יש בכל תהליך היא אולי תדע להתחשב בתהליך שיש בו הרבה חוטים לעומת תהליך שיש בו מעט. אם יש תהליך שרוצה לבצע קריאת מערכת, היא תדע שהחוט הספציפי ביקש ולא כל התהליך ייחסם. פחות מהיר מהגישה הראשונה.

<u>בעיות שעולות בשימוש בטרדים:</u>

- מה קורה כשמשתמשים בfork בתהליך עם יותר מטרד אחד? האם כל התהליך משוכפל עם כל הטרדים? אם משכפלים הכל ויש בתוכנית גישה לקובץ, כל הטרדים ינסו לגשת אליו...
מה שקורה בפועל זה שכל התהליך משוכפל אבל יהיה רק טרד אחד וזה הטרד שביצע את הfork.
- מה קורה אם טרד צריך להסתיים בגלל שגיאה? כל התהליך מסתיים.

<u>תרגול שבוע 11 – מנגנוני סנכרון</u>
נושא שחשוב לקחת בחשבון כאשר מתכננים מערכות שהן multithreading.

<u>קטע קריטי –</u> קטע קוד שמתייחס למשאב שמשותף ליותר מגורם אחד. היות והוא משותף ליותר מגורם אחד אנחנו צריכים להיזהר מתוצאות שאינן רצויות (קריאה וכתיבה בו"ז...)
<u>Mutual exclusion -</u> מונח שמתייחס למצב שבו בכל נקודת זמן יהיה לכל היותר תהליך/טרד אחד בתוך CS
<u>פעולה אטומית –</u> פעולה שלא תופסק פעילותה החל מרגע שבו התחילה ריצה ועד לרגע בו סיימה בעקבות הפרעה מגורם חיצוני.
<u>סמפורים –</u> סמפור זה מונה מספרי שמתוחזק ע"י שתי פעולות אטומיות שנקראות UP וDOWN.
יש שני סוגים של סמפורים:

1. סמפור בינארי – סמפור שקיימים בו שני ערכים אפשריים. הערך 0 מציין שכבר יש מישהו שביצע DOWN וכעת נמצא בתוך הCS ולכן הוא ייחסם ולא יוכל להיכנס. הערך 1 מציין שניתן להיכנס לCS.
2. סמפור סופר/שלילי – סמפור שניתן לאתחל אותו לכל ערך חיובי טבעי כאשר x מציין את כמות הטרדים/התהליכים שיכולים לבצע עליו DOWN

ספרייה – POSIX, נעשה include <semaphore>
הצהרה על סמפור תהיה ע"י שימוש בטיפוס הנתונים sem_t.
אתחול של סמפור יתבצע ע"י קריאה ל sem_int
Sem_int (sem_t *sem, int pshared, unsigned int value)
הפונקציה תאתחל סמפור שsem מצביע עליו עם הערך value. הערך של pshared מציין האם הסמפור יהיה משותף לתהליכים או לחוטים של תהליך (כאשר הערך 0 יציין עבור חוטים ו1 עבור תהליכים).
הפונקציה תחזיר 0 בהצלחה.

על מנת לבצע DOWN על סמפור נשתמש ב int sem_wait (sem_t *sem)
למעשה מקבלת מצביע לסמפור שנרצה להוריד. אם יהיה אפשר להוריד היא תחזור מיד, אחרת לא תחזור עד שהתהליך שקרא לה ישוחרר.

על מנת לבצע UP על סמפור נשתמש ב int sem_post(sem_t *sem)
מקבלת מצביע לסמפור שנרצה לעשות עליו UP.

Int semdestroy(sem_t *sem) – אחרי שנסיים להשתמש בסמפור מסויים נשחרר אותו. צריך לשים לב שאין עוד תהליכים שממתינים לו לפני שמשחררים. כדי להשתמש בו שוב נעשה init מחדש.

Mutex- מנגנון סנכרון שדומה מאוד לסמפור בינארי מלבד הבדל חשוב אחד: במנגנון זה יש בעלות על מנעול. כלומר, אני זוכר מי עשה DOWN והוריד את המונה. ע"י כך לא אוכל לאפשר למישהו שאינו בעל המנעול לשחרר את המנעול.

<u>תרגול 11</u>

<u>דוגמא לשימוש בסמפור בינארי –</u>
נניח שיש שני תהליכים A,B וכל אחד צריך לבצע איזשהי פעולה A,B בהתאמה. כדי לגרום לפעולה של A להתבצע לפני הפעולה של B נוסיף DOWN לפני הפעולה של B וUP אחרי שA מסיים.

<u>מקרי קיצון ב-MUTEX-</u>

1. בעל המנעול נעל את המנעול פעמים נוספות
2. תהליך שאינו בעל המנעול מנסה לשחרר את המנעול
3. ניסיון לשחרר מנעול משוחרר

הטיפוס שיגדיר את סוג הmutex: pthread_mutexattr_t
בעזרת הפונקציה pthread_mutexattr_settype נוכל לאתחל את הטיפוס (נשלח אותו כפרמטר)
סוגי mutex שנשלח בפונקציה הזו:
-PTHREAD_MUTEX_NORNAL

1. התהליך (הבעלים) יחסם (deadlock)
2. התנהגות בלתי מוגדרת
3. כמו 2

-PTHREAD_MUTEX_ERRORCHECK

1. תוחזר שגיאה (2,3 כנ"ל)

-PTHREAD_MUTEX_RECURSIVE

1. בעל המנעול יצטרך לשחרר את המנעול ככמות הנעילות
2. תוחזר שגיאה
3. כנ"ל

-PTHREAD_MUTEX_DEFAULT

1. לא מוגדר
2. לא מוגדר
3. לא מוגדר

הטיפוס עבור mutex יהיה pthread_mutex_t
אתחול של mutex:
Int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr)

נעילת mutex:
Int pthread_mutex_lock(pthread_mutex_t *mutex)
שחרור מנעול:
Int pthread_mutex_unlock(pthread_mutex_t *mutex)

<u>תבניות סנכרון פופולאריות:</u>

1. The rendezvous pattern
תבנית זו היא מקרה כללי יותר של השימוש הסטנדרטי בסמפור בינארי.
נניח ויש לנו שני תהליכים וכל אחד רוצה לבצע שתי פעולות. נגדיר לכל תהליך סמפור בינארי.
היינו רוצים לדאוג ש SA2 תתבצע אחרי SB1 וגם ש SB2 תתבצע אחרי SA1. איך יראה הקוד?

Process A :
SA1
UP(semA)
DOWN(semB)
SA2

Process B
SB1
UP(semB)
DOWN(semA)
SB2

2. The barrier pattern
נניח שיש לנו n>=2 תהליכים וכולם צריכים לבצע שתי פעולות.
היינו רוצים שאף אחד מהתהליכים לא יבצע את s2 לפני שכולם סיימנו את s1.
(הקוד שלו במודל)