By : Lior Daniel
Email : Lior7daniel@gmail.com

סיכום הקורס מערכות הפעלה

סיכום מקיף על בסיס המצגות והקורס המועבר על ידי ד"ר קוגן קיריל.
נכתב בעיקר עבורי ובמילים שלי על מנת להקל על הקושי והערפל שנוצר נוכח הנסיבות שהביאה איתה הקורונה.
חומרי עזר מהם אספתי חומר: הרצאות בר אילן פרופ' דודי סרנה, סיכום צבי מינץ ואילון צדוק, סיכומים עמית שלו והמון Google.
אם מצאתם שסיכום זה עזר, אשמח לפרגון כוכב בגיטהאב : http://www.github.com/liordaniel

Von Neumann Architecturel

כל המחשבים פחות או יותר מבוססים על אותו עיצוב בסיסי – ארכיטקטורת ואן ניומן!
בארכיטקטורה זו יש לנו 3 דרישות עיקריות :

- המחשב יכלול ארבעה תתי מערכות:
  1. Memory – מאחסן את ה-Data וה-Instructions (פקודות\הוראות) של התוכנית.
  2. ALU – מבצעת את החישובים והפעולות הלוגיות הנדרשות על ידי התוכנית.
  3. Control Unit – אחראי על ההרצה השותפת של התהליך.
  4. I/O System – מערכת המתקשרת עם ה"עולם החיצון" (מקלדת, עכבר, מסך וכו'..).
- תהליך (תוכנית בזמן ריצה) ירוץ אך ורק על\בעזרת הזיכרון.
- תהליך ירוץ בצורה סדרתית (על מנת לפשט, כך לא נצטרך לחפש איפה ההוראה הבאה).

Random Access Memory – RAM

מאפיינים

- מורכב מרצף של תאי זיכרון, כך שכל תא הוא יחידת איחסון בגודל קבוע ובעל כתובת המורכבת מביטים.
- זמן הגישה לכל תא בזיכרון שווה.
- הגישה בזיכרון היא לתא שלם לפי הכתובת שלו. לא נוכל לגשת לביט יחיד.
- כאשר תוכנית רצה, ה-data וה-instruction שלה מאוחסנת בזיכרון.
- פועל אך ורק כשיש חשמל, במצב שאין הוא מאבד את כל התוכן שלו.

על מנת לגשת לתאי הזיכרון המעבד משתמש באוגרים (registers).

- (Adress Register) MAR – מאחסן את הכתובת.
- (Data Register) MDR – מאחסן את המידע.
- Fetch/Store signal – אות חשמלי הגורם לפעולה לקרות.

- Fetch(adress) – מקבל כקלט כתובת (מהמעבד) ומחזיר העתק של התוכן באותו תא הזיכרון.
  1. Load : הכנסת הכתובת שהתקבלה לתוך ה-MAR.
  2. Decode : פיענוח הכתובת ב-MAR (מכתובת וירטואלית לכתובת הפיזית).
  3. Copy : העתקת התוכן של תא הזיכרון ל-MDR.

- Store(adress, value) – מקבל כתובת וערך, מאחסן את הערך בכתובת (דורס את התוכן בתא).
  1. Load : הכנסת הכתובת שהתקבלה לתוך ה-MAR.
  2. Load : הכנסת הערך ש שהתקבל לתוך ה-MDR.
  3. Decode : פיענוח הכתובת ב-MAR (מכתובת וירטואלית לכתובת הפיזית).
  4. Copy : העתקת הערך שב-MDR לתוך התא שכתובתו נמצאת ב-MAR.

ALU

יחידה הנמצאת במעבד המחשב לביצוע פעולות אריתמטיות \ לוגיות.

By : Lior Daniel
Email : Lior7daniel@gmail.com

Input/Output System

מהירות הגישה לרכיבי I/O איטית בהשוואה ל-RAM, לכן נוצרת בעיה שכאשר אנו רוצים לפנות בבקשה לפעולה מרכיבים אלו, המעבד והזיכרון מוחזקים זמן רב מידי (באופן יחסי, במונחי מחשב). על מנת לפתור בעיה זו, ניתן ל"מישהו" לטפל בבעיה זו.

Device Controller למעשה לכל רכיב יש 'בקר' המנהל אותו ומתקשר בעיקר עם המעבד (גם עם הזיכרון).

לבקר זה יש התקן אחסון מקומי (Buffer).
איך הוא פותר את הבעיה?
מה שקורה בפועל, זה שהבקר של הרכיב מקבל את הבקשה לפעולה כלשהי מהמעבד. הוא "משחרר" את המעבד על מנת שיוכל להמשיך לעבוד, מנהל את הרכיב על מנת שיבצע את הפעולה ובינתיים אוגר את המידע המתקבל ב- local buffer. ברגע שיש לו את כל המידע, הוא שולח interrupt למעבד.

Interrupt פסיקה - אות חשמלי המגיע למעבד מרכיב חומרה.

Trap ה-Trap הוא Interrupt שנוצר על ידי תוכנה ולא על ידי חומרה. נוצר בעיקר על ידי שגיאה או בעקבות בקשה של תוכנית המשתמש.

ניתן לחלק את הפסיקות למספר סוגים

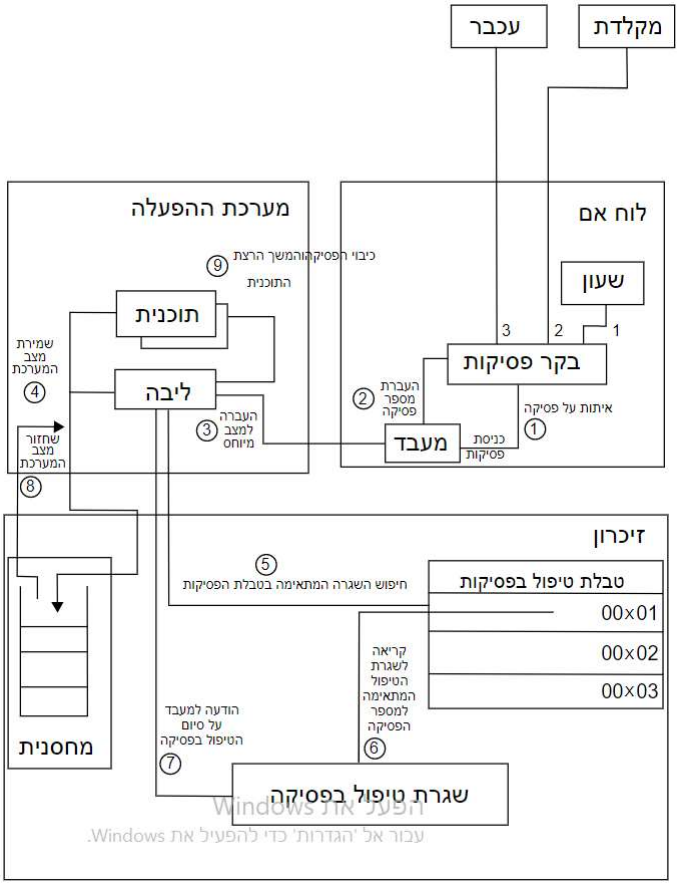
- פסיקה סינכרונית – נקראת כך משום שהיא צפויה. נוצרת ע"י חומרה או תוכנה במועד ידוע מראש, עקב ביצוע פקודה מוסימת בתוכנית. ניתן לחזור את התעוררותה כאשר יתבצע קטע הקוד המתאים.
  - פסיקת תוכנה – נוצרת על ידי תוכנה ומשמשת להעבר מידע בין תהליך למעבד.
  - חריגה \ שגיאה – נוצרת כדי לדווח על שגיאות במהלך ביצוע פקודות.
  - מלכודת – מתעוררת בעקבות הדלקת דגל כלשהו.
- פסיקה אסינכרונית – נגרמת על ידי רכיב חומרה באופן שאינו תלוי בריצת התוכנית הנוכחית.
  - פסיקה מבקר הפסיקות – מופעלת על ידי רכיב חומרה המחובר לבקר הפסיקות.
  - פסיקות שאינן ניתנות למיסוך – דורשות טיפול במיידי ולא ניתן להתעלם מהן.

Interrupt Handling

Vector Interrupt System

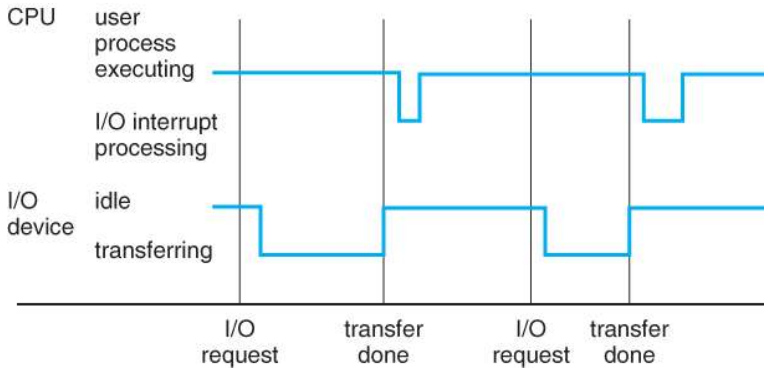
נדע להפעיל את הקוד הרלוונטי ששמור בזיכרון מכיוון שאני מקבל את כל המידע הרלוונטי עם ה-Interrupt.

- הרכיב (עכבר\מקלדת) מבקש Interrupt מבקר הפסיקות ומעביר גם את סוג הפסיקה.
- בקר הפסיקות מאותת למעבד שהיה Interrupt
- המעבד מעביר את המערכת למצב "מיוחס" על מנת לדעת שכרגע אנו מבצעים פעולה של מערכת ההפעלה ולא של תהליך מסוים.
- מתבצעת שמירה למצב המערכת (אוגרים וכו').
- כל פסיקה מגיעה עם מספר מזהה על מנת שנוכל להבין מהי סוג הפסיקה. לכן, נדע ללכת לגשת לתא בזיכרון שבו מאוחסן הקוד בטבלת הפסיקות עבור אותה פסיקה.
- נריץ את השגרה שמטפלת באותה פסיקה.
- ברגע שסיימנו נודיע למעבד על מנת לחזור לאותו המצב שבו היא הייתה לפני ה-Interrupt.
- המערכת ממשיכה להריץ את ה-Instruction מהנקודה שבה הפסיק.



By : Lior Daniel
Email : Lior7daniel@gmail.com

Interrupt Timeline



- המעבד יכול להיות בשני מצבים שונים:
  1. מריץ את התוכנית של המשתמש.
  2. מטפל ב-Interrupt.
- I/O יכול להיות בשני מצבים שונים:
  1. Idle (לא עושה כלום)
  2. קורא\כותב משהו.
- בתחילת הגרף העליון, המעבד מריץ תוכנית ולפתע מקבל מתוך ה-Instruction בקשה לשימוש ב-I/O.
- ה-I/O מקבל את הבקשה ועובר ממצב idle למצב transferring ומעביר ל-Buffer ב-Device Controller את הנתונים.
- במצב transferring ה-Device Controller מבצע את הפעולה ואוגר את המידע שצריך להחזיר אל תוך ה-Buffer שלו.
- כשהוא מסיים, הוא מעביר Interrupt וחוזר למצב idle.
- בנקודה זו המעבד שומר את מצב המערכת ועובר למצב I/O interrupt processing על מנת לטפל ב-Interrupt.
- כשהמעבד מסיים, המעבד חוזר לנקודה האחרונה בה שמר את מצב המערכת על מנת להמשיך את פעולותיו.

Control Unit

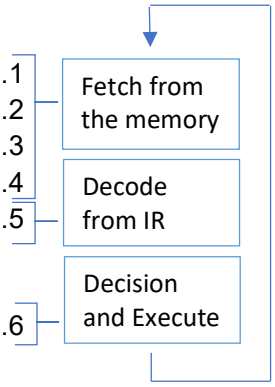
ברגע שהרצנו תהליך, הוא נשמר בזיכרון כסדרת הוראות בשפת מכונה.
תפקיד ה-Control Unit הוא להריץ את ה-instructions כתהליך ולגרום לרכיבי ה-I/O להגיב בהתאם.

ה-Control Unit עובד בשיטת Fetch -> Decode -> Execute.

מבנה ה - Control Unit

- PC (Program Counter) – אוגר מצביע, מאחסן את הכתובת של ה-instruction הבא בזיכרון.
- IR (Instruction Register) – אוגר ההוראה, מאחסן את ה-instruction שהובא מהזיכרון.
- ID (Instruction Decoder) – מפענח ההוראה מה-IR ומפעיל הרכיבים הרלוונטי.

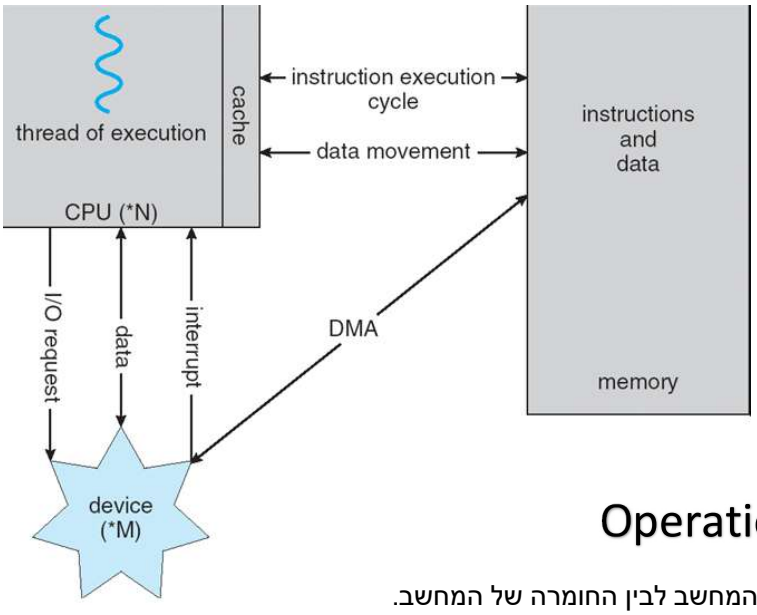
איך בפועל הכל עובד יחד? ברגע הרצת תוכנית, הכתובת של ה-Instruction הראשון מאותחל ב-PC.



1. PC -> MAR : הכתובת שב-PC נכנסת ל-MAR.
2. Fetch Signal: אות חשמלי לביצוע פעולת Fetch.
3. MDR -> IR : התוכן (ה-instruction) שנמצא בתוך הכתובת שב-MAR נכנס לתוך אוגר ה-IR.
4. PC+1 -> PC : נקדם את אוגר ה-PC כך שיצביע לכתובת הבאה (להוראה הבאה).
5. IR -> ID : נבצע Decode (פיענוח) על ההוראה המאוחסנת ב-IR ונכניס אותו לאוגר ה-ID.
   עכשיו, ה-ID ייצר את האותות המתאימים עבור המעגלים החשמליים שאמורים לתמוך בפקודה בהוראה.
6. ביצוע הפעולה (אין צורך לפרט בקורס זה).

By : Lior Daniel
Email : Lior7daniel@gmail.com

איך הרכיבים עובדים יחד?



- התוכנית שלנו מבקשת לבצע פעולה כלשהי של I/O מ-Device כלשהו.
- המעבד קורא את ה-Instruction ומבין שיש לבצע פעולה זו.
- המעבד מושך את ה-data מאותו תא בזיכרון
- המעבד יעביר שני דברים ל-Device
  1. מה הפעולה המתבקשת. נכתב לתוך הרגיסטרים של ה-Device Controller.
  2. ה-data המתבקש. נכתב לתוך ה-Buffer של ה-Device Controller.
- ה-Device יתחיל לבצע את הפעולה, וכשיסיים הוא יודיע למעבד באמצעות Interrupt.

מערכת ההפעלה – Operation System

תוכנית אשר משמשת כחוצץ בין המשתמש של המחשב לבין החומרה של המחשב.

<u>מטרות מערכת ההפעלה</u>

- הרצת תכניות המשתמש בצורה קלה.
- הפיכת השימוש במערכת לנוח יותר
- שימוש יעיל יותר בחומרת המחשב.

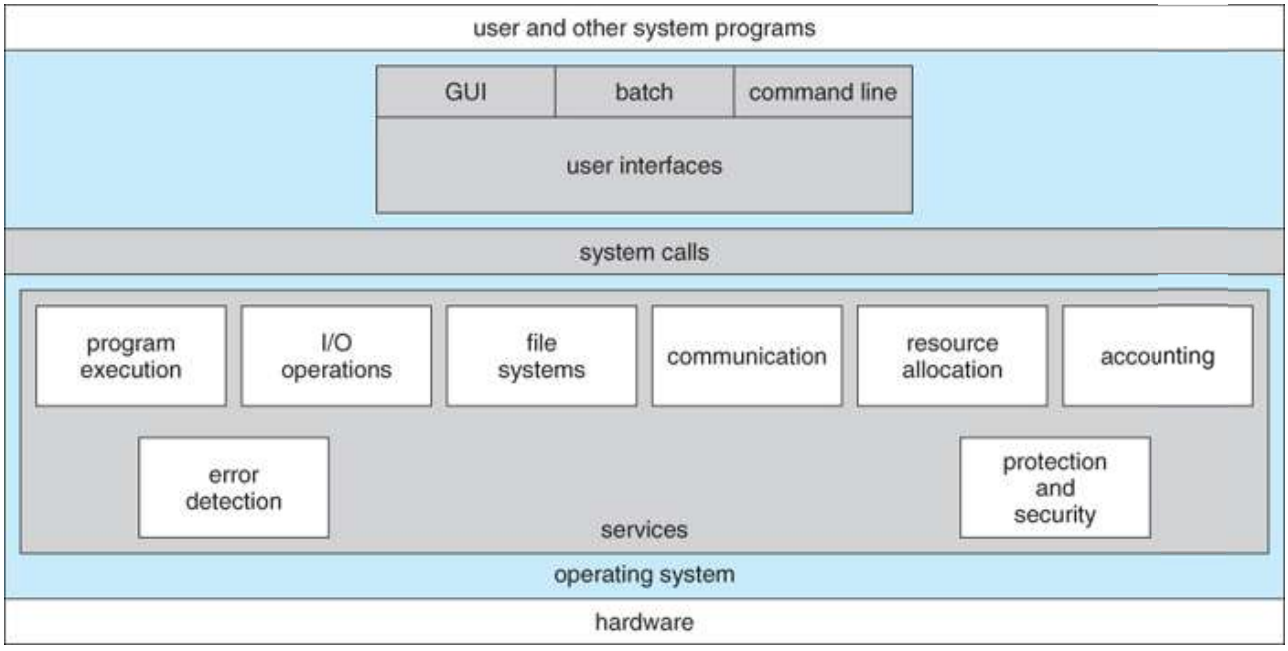
<u>למערכת ההפעלה שני סוגי התנהגויות</u>

- Resource allocator – כאשר מקצה CPU \ זיכרון \ דיסק וכו'.
- Control program – כאשר שולטת בהרצת תוכנית לשימוש נכון במחשב ומניעת שגיאות.
  דוגמא: גישה לאיזור אסור בזיכרון, System Call, שליטה על מרכיבי I/O וכו'.

<u>לכל מערכת הפעלה יש מטרות ועיצובים שונים ותכנון לפי צרכים</u>

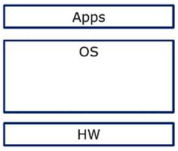
- Mainframe – ניצול מקסימלי של משאבי החומרה.
- PC – תמיכה מקסימלית בהרצת תכניות של המשתמש.
- Handheld – ממשק נוח להרצת אפליקציות (ביצועים טובים פר יחידת ניצול של הסוללה).

מבט על שירותי מערכת ההפעלה

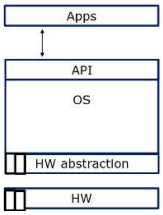


By : Lior Daniel
Email : Lior7daniel@gmail.com

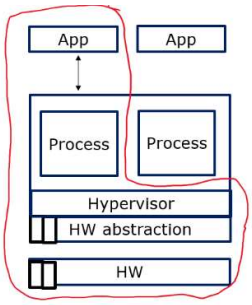
ארכיטקטורות של מערכות הפעלה.



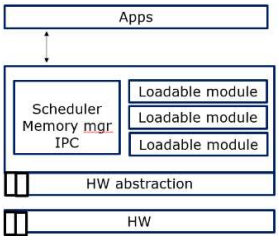
- Monolithic – כוללת בתוכנה את כל הפונקציונליות והכל כתוב כקוד אחד. זוהי אחת הארכיטקטורות הראשונות שיוצרו כדי לתת פונקציונליות ללא מחשבה על נוחות. היתרון שהביצועים יכולים להיות טובים יותר ובמקרה של שינויים קל יותר כי הכל במקום אחד.



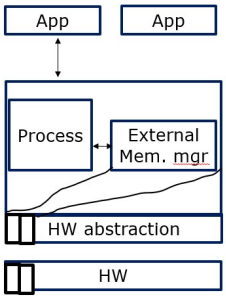
- Multi-Layer – מפתחים במערכת ההפעלה רמות, כך שכל רמה אחראית על משהו ולכן אם נצטרך שינויים הם קורים ברמה הלוקאלית. היא נוחה אבל פחות בביצועים.



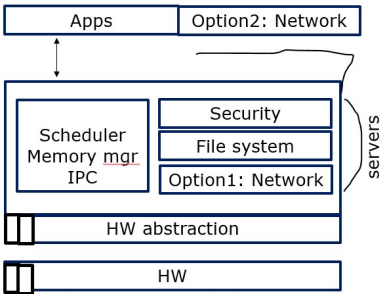
- System of VM's (virtual machine) -
הוספה של תהליכים לגרעין. התהליכים יכולים לגשת לחומרה רק דרך Hyp, אשר מנהל את הגישה לחומרה ע"י ה-kernel.



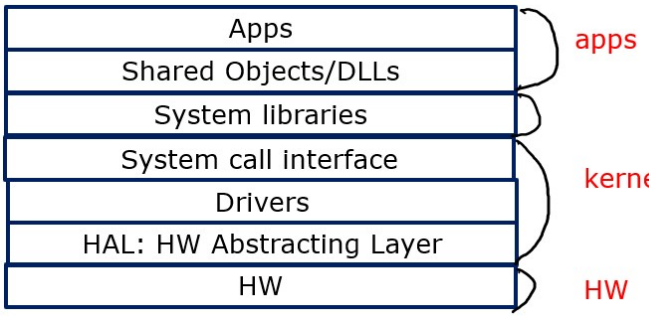
- Micro kernel – חוסך במשאבים ומאגד 3 פונקציות:
(IPC ,memory mgr ,scheduler)



- Exokernel –
חלוקה של המשאבים בצורה מאוזנת לתהליכים. מכיוון שהפונקציות ממומשות מחוץ לתהליך יש להם גישה מוגבלת לחומרה.



- Client Server – כל module נחשב server, שמטפל בתקשורת מסוג מסוים. ההבדל בין ארכיטקטורת זו ל-Micro kernel הוא שאת ה-server ניתן להריץ במרחב המשתמש. לדוג': ניתן לשים את החיבור לרשת בתוך האפליקציה.



Abstracting Layers - השכבות בצורה מופשטת ->

הערה:
לכל התקן HAL המנהל אותו בצורה מופשטת.
לדוגמא: להתקן העכבר יש HAL המנהל אותו כך שלא משנה באיזה עכבר נשתמש הפקודה להצגתו במיקום מסוים במסך קיימת בצורה מופשטת.
לעכברים מסוגים שונים קיימים Drivers עבורם היודעים לגשת ל-HAL.

משפטים נוספים –

האפליקציות יכולות לכלול Objects / DLLs.

חלק מהדרייברים יכולים לייצג פרוטוקולי (TCP/IP), אבל לא התקנים.

דרייברים ו-System Calls שייכים ל-Kernel.

By : Lior Daniel
Email : Lior7daniel@gmail.com

Kernel הקרנל הוא התוכנה במערכת ההפעלה (תהליך \ שכבת קוד) האחראית על ניהול משאבי המחשב והתקשורת עם החומרה. זהו ליבת מערכת ההפעלה.
כל שאר התהליכים שהגיעו עם מערכת ההפעלה מקוטלגים כ-System/Application programs.

Firmware קושחה היא תוכנה שצרובה פיזית על רכיב החומרה ואינה ניתנת לשינוי (הקוד על ה-ROM).

Basic Input Output System – BIOS בעת הדלקת המחשב, ה-BIOS הוא התוכנה הראשונה שפועלת, ומשמש לזיהוי\בדיקה\הגדרת התקני החומרה וטעינת הפונקציות הנדרשות מהדיסק הקשיח אל כרטיס הזיכרון להעלאת מערכת ההפעלה.

System Call בקשה של אפליקציה מה-kernel לביצוע פעולה שאינה יכולה לבצע בעצמה כמו גישה לרכיבי I/O וכו' (זהו בעצם Software interrupt).

Driver תוכנה ייעודית המאפשרת גישה לרכיב I/O מסוים.

מעבר בין User Mode ל-Kernel Mode תוכניות המשתמש רצות ב-User Mode ומערכת ההפעלה מריצה את הפקודות שלה ב-Kernel Mode כאשר רק במצב זה המעבד מבצע פקודות מוגנות (פקודות Privileged).
כשלמשתמש יש צורך לגשת לרכיב I/O, הוא יכתוב פקודת System Call שתעביר את המעבד למצב Kernel, המעבד יעשה את המוטל עליו ויחזור למצב User.
חשיבות המצבים : מערכת ההפעלה נדרשת להגן על עצמה ועל מרכיבי מערכת נוספים (למשל I/O) מפני מפני שימוש לא נכון \ זדוני.

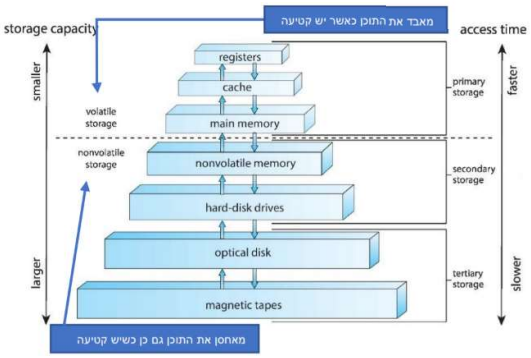
פקודות Privileged (בהרשאת ה-Kernel)	פקודות רגילות (בהרשאת המשתמש)
ניקוי זיכרון	קריאה מהשעון
כיבוי Interrupt	שליחת trap
שינוי ה-Device status table (התורים של הבקשות ל-Device'ים שונים)	מעבר מ-User Mode ל-Kernel Mode
גישה ל-I/O	

Memory Management Unit – רכיב חומרה האחראי על פעולות ניהול הזיכרון על ידי המעבד. יחידה זו ממפה בין כתובות וירטואליות לפיזיות בזמן ריצה, מגינה על זיכרון של תהליך אחד מפני אחר ובמיוחד על שטח הזיכרון של הגרעין, אחראית על הקצאות זיכרון והחלפות מידע בין 2 תהליכים.

<u>Interprocess Communication</u> – מערכת ההפעלה אחראית על התקשורת בין התהליכים עצמם.

ניהול הזיכרון

ניהול זוהי אחריות מערכת ההפעלה, העוסקת בהחלטה על מה יהיה \ ישהה בזיכרון בכל רגע. זהו מרכיב קריטי לניצולת המעבד ובעל השפעה מכרעת על זמן התגובה למשתמש.



תפקיד מערכת ההפעלה בכל הקשור לניהול זיכרון :

- מעקב ורישום החלקים בזיכרון שנמצאים כרגע בשימוש.
- החלטה על העברת תוכן זיכרון המשויך לתהליכים מ\אל הזיכרון.
- הקצאת זיכרון לתהליכים.

<u>Main Memory</u> – זיכרון ראשי.
המקום שבו תוכניות רצות.
ה-CPU יכול לעבוד איתו בצורה ישירה (ROM/RAM).

<u>Storage Secondary</u> – זיכרון משני. כיוון שיש לנו הרבה אפליקציות בו זמנית עם זיכרון משותף, יכול להיווצר מצב שחלקם לא יוכלו להשתמש בזיכרון, לכן חלקן שמורות בזיכרון הוירטואלי שמערכת ההפעלה מממשת ושמור בדיסק. הגישה אליו נעשית בעזרת Interrupt.

By : Lior Daniel
Email : Lior7daniel@gmail.com

DMA (Direct Memory Acces) תכונה המאפשרת לתת-מערכות מסוימות של המחשב לבצע קריאה \ כתיבה ישירה מ\אל הזיכרון, באופן עצמאי ללא תלות במעבד. לדוגמא, DMA מאפשר טעינה של נתונים מהכונן הקשיח ישירות לזיכרון המחשב, מבלי שהמעבד יטען כל נתון מהכונן הקשיח, ויעביר אותו לזיכרון.
היתרון בשיטת הגישה הישירה הוא החיסכון בפעולות מעבד לטובת העברת נתונים. במקום להתעסק בהעברת נתונים, פעולה שלרב אינה דורשת עיבוד או התייחסות לתוכן הנתונים, המעבד פנוי לטפל בפעולות עיבוד אחרות, ואפילו בתוכנות אחרות שרצות במחשב באותו הזמן.
החיסרון בשיטת הגישה הישירה הוא שאין בקרה מרכזית ובלעדית על הזיכרון – דבר שמגדיל משמעותית את הסיכויים למקרים של השחתת זיכרון.

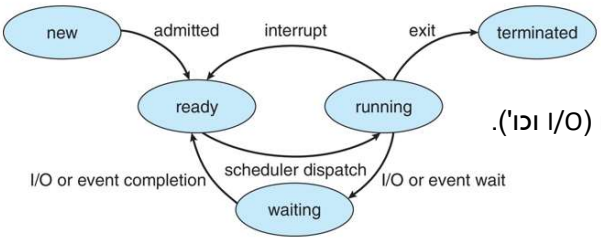
File קובץ הוא משאב מערכת לוגי המאפשר אחסון ומידע, אשר זמין לתוכניות המחשב לקריאה או כתיבה. לרב יישמר באמצעי לאחסון נתונים.

PROCESS תהליך הוא קובץ במצב הרצה.

התהליך בנוי מכמה שכבות :

- Text Section – הקוד עצמו של התוכנית.
- Status – מצב התהליך ומצב האוגרים (program counter וכו').
- Stack – המידע הזמני של אותו הבלוק (פרמטרים של הפונקציה, משתנים לוקאליים וכו').
- Data section – מכיל את המשתנים הגלובאליים.
- Heap – זיכרון המוקצה דינאמית תוך כדי הריצה.

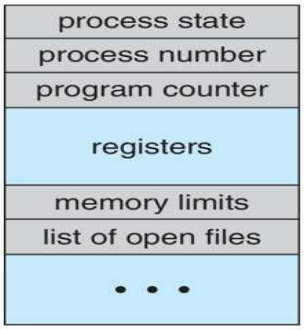
לתהליך כמה מצבים שונים :



- New – התהליך נוצר.
- Running – Instruction של התהליך רץ.
- Waiting – תהליך במצב המתנה ומחכה לאירוע מסוים (I/O וכו').
- Ready – התהליך מחכה שהמעבד יריץ אותו.
- Terminated – התהליך הסתיים.

בלוק שליטה – Process Control Block – PCB

זהו מבנה נתונים בתוך ה-kernel המייצג עבור מערכת ההפעלה את מצב התהליך ומכיל מידע הקשור אליו.

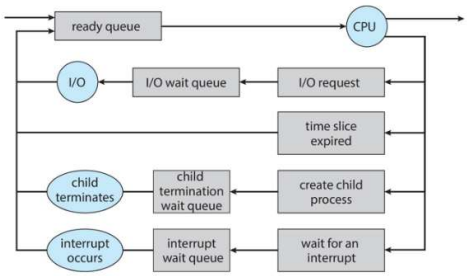


1. מצב התהליך (Ready, Waiting וכו').
2. Program Counter – מצביע לכתובת של התהליך הבא שירוץ.
3. מצב האוגרים ב-CPU של התהליך.
4. ניהול זיכרון (הקצאות דינאמיות וכו') עבור התהליך.
5. מידע על קלט פלט עבור התהליך, רשימת קבצים פתוחים וכו'.
6. מידע חישובי - מעבד בשימוש, זמן שעון שחלף מאז ההתחלה, מגבלות זמן על מצב התהליך וכו'.

הערה - לכל תהליך יש PCB. מערכת ההפעלה היא תוכנית הרצה ללא PCB, לכן היא אינה תהליך.

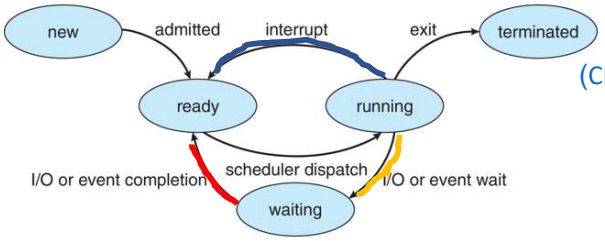
By : Lior Daniel
Email : Lior7daniel@gmail.com

CPU Scheduler



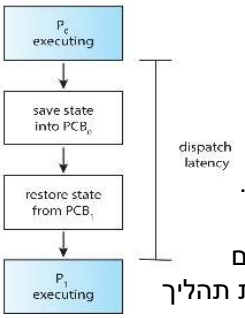
מתזמן התהליכים אחראי על הגישה למעבד,
ניצול יעיל שלו ובחירת התהליך הזמין הבא שירוץ.
המתזמן שומר על תורי תזמון של תהליכים באמצעות שתי מחסניות -
1. Ready Queue – התהליכים בזיכרון הראשי המוכנים לריצה.
2. Waiting Queue – התהליכים המחכים לאירוע (I/O וכו'..).

ברגע שה-CPU נכנס למצב "סרק", מערכת ההפעלה בוחרת תהליך מתוך ה-Ready Queue ונותן לו לרוץ על ה-CPU. החלטות המתזמן עשויות לקרות באחד מארבעת הנסיבות הבאות.



1. ממצב Running למצב Wait (לדוגמא מחכה לקלט)
2. ממצב Running למצב Ready (בתזמון פרימיטיבי, כדי לפנות CPU)
3. ממצב Waiting למצב Ready (לאחר שקיבל את מה שחיכה לו)
4. התהליך מסתיים (קורה ביוזמת התהליך).

Context Switch – מעבר בין ריצת שני תהליכים באמצעות המעבד. החלפת המעבד לתהליך אחר דורשת שמירת המצב הקיים (PCB) של התהליך הישן וטעינת המצב של התהליך החדש. תהליך זה נחשב בזבזני ולכן מערכות הפעלה מנסות לבצע אופטימיזציה בשימוש בהן.



Dispatcher – מודל זה מעביר את הבקרה של ה-CPU לתהליך שנבחר ע"י ה-Scheduler.
פעולה זו כוללת :
1. Context Switch.
2. מעבר ל-User Mode.
3. קפיצה למקום הנכון בתוכנית המשתמש במטרה להתחיל להריץ אותה (PC).

מודל זה צריך להיות מהיר ככל האפשר, מכיוון שהוא נקרא בכל פעם שמחליפים תהליך. הזמן שלוקח ל-Dispatcher להפסיק את התהליך הרץ ולהתחיל בהרצת תהליך חדש נקרא Dispatch latency.

<u>ה-CPU Scheduler משתמש בשני סוגים שונים של תזמון :</u>

1. Non Preemptive – תהליך לא משחרר את ה-CPU עד שלא מסתיים או נכנס למצב Waiting. (דוגמא 1 \ 4)
לא ייתכן מצב בו מערכת ההפעלה מפריעה לו באמצע שרץ, גם אם עבור תהליך בעדיפות גבוהה יותר.
2. Preemptive – מערכת הערה לשינויים, במיוחד עבור שני המקרים הבאים : (דוגמא 2 \ 3)
א. עדיפות : בודקת כל הזמן את מצבי התהליכים ועשויה להפסיק תהליך עבור תהליך בעל <u>עדיפות</u> גבוהה יותר. תזמון מסוג זה גורם לעלויות = זמן.
ב. זמן : המתזמן משתמש ב-Timer ומחלק לכל תהליך זמן עבור הריצה שלו, בעזרת Timer Interrupt עוצרים את התהליך ומבצעים החלפה.

<u>יתרונות וחסרונות עבור Non Preemptive \ Preemptive:</u>

- ברגע שמחליפים תהליכים אנחנו צריכים כל הזמן לעשות Context Switch. ה-PCB שומר על התהליך בתוך הגרעין כדי שיוכל לשחזר אותו בעתיד.
- ב-Non אנו יודעים מתי ואיפה נחליף את התהליכים, כך שה-Context Switch יותר מהיר.
- ב-Preem אמנם אפשר לעצור תהליך אך זה גורם ל-Context Switch להיות יותר יקר (כלומר תדירות החלפת תהליכים גבוהה מידי) כך שנבזבז המון זמן והיעילות של המתזמן תהיה נמוכה.

By : Lior Daniel
Email : Lior7daniel@gmail.com

Thread תהליכון הוא חלק מתהליך, יש לו PC, Registers ו-Stack משלו, אך כל שאר ה- PCB הם אותו הדבר כמו האב. כאשר יש כמה תהליכונים תחת אותו תהליך הם חולקים יחד את ה-Code Section וה-Data Section (נתונים שלא במחסנית כגון משתנים גלובאליים, סטטיים, קבצים שנפתחו וכו').

הערות :

- הפונקציה ()Fork לא מייצרת Thread, אלא Process (כלומר Data\Code Sections חדשים).
- תהליך יצירה של Thread הרבה יותר מהיר מאחר ולא משכפל את ה-PCB במלואו ביחס לתהליך יצירת Process.
- Thread רץ על אותו מרחב זיכרון.
- תקשורת בין תהליכים מתבצעת בעזרת Pipe \ Signal \ Shared Memory.

<u>ההבדלים בין תהליך לתהליכון</u>

תהליך	תהליכון
יש לו Data Segment ו- Heap	אין לו Data Segment ו- Heap
לתהליך יש לפחות תהליכון אחד	לא יכול לחיות בעצמו, חייב להיות חלק מתהליך
	יכול להיות יותר מתהליכון אחד בתהליך
יצירה יקרה	יצירה לא יקרה
החלפת הקשר יקרה	החלפת הקשר לא יקרה
	תקשורת יעילה
אם תהליך מת, אז כל הזכרון משתחרר וכל התהליכונים מתים💀	זכרון משתחרר במוות 💀

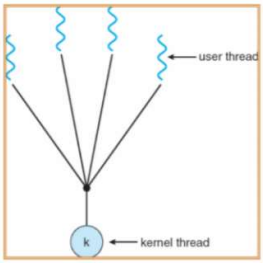
<u>ישנם שני סוגים של תהליכונים :</u>

1. User Thread – מיושם ע"י המשתמש, הגרעין לא מודע אליו.
2. Kernel Thread – מיושם ע"י מערכת ההפעלה, הגרעין מנהל אותו.

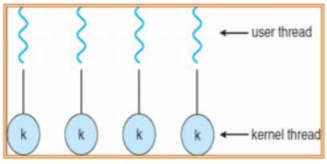
<u>ההבדלים בין השניים</u>

Kernel Thread	User Thread
איטי יותר ליצירה וניהול	מהיר יותר ליצירה וניהול
מנוהל על ידי מערכת ההפעלה	מנוהל על ידי המשתמש
דורש את תמיכת מערכת ההפעלה	לא דורש תמיכה של מערכת הפעלה ספציפית
אם נתקע תהליך אחר מתוזמן	אם תהליכון אחד נתקע כולם נתקעים
יקר ואיטי כי נדרשת קריאת מערכת להחלפת קשר בין תהליכונים	זול ומהיר כי לא נדרשת קריאת מערכת בין תהליכונים
ניתן להשתמש בכמה ליבות	ליבה אחת לכל התהליך

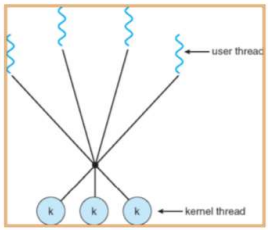
<u>מודלים של Multi-Threading :</u>



Many-To-One – כמה User Threads ממופים ל-Kernel Thread יחיד.
יתרונות : מהיר מכיוון שאינו צריך קריאת מערכת ליצירה שלהם.
חסרונות : אין מקביליות (כמה ליבות), אם אחד נתקע כולם נתקעים (לדוגמא בהמתנה ל-I/O).



One-To-One – כל User Thread ממופה ל-Kernel Thread יחיד.
יתרונות : יש מקביליות, כאשר תהליכון אחד נתקע אחרים יכולים לרוץ.
בנוסף, ביצועים טובים יותר כשיש כמה ליבות.
חסרונות : יקר, יצירה וניהול הזיכרון מערב את הקרנל בפעולות שלו וצורך את משאבי ה-kernel.



Many-To-Many – הרבה User Threads להרבה Kernel Threads.
יתרונות : גמיש.
חסרונות : בד"כ ישתמש במיפוי 1:1.

By : Lior Daniel
Email : Lior7daniel@gmail.com

קריטריונים של מתזמן

- CPU utilization (ניצול ה-CPU) – שומר את ה-CPU תמיד בתעסוקה.
- Throughput (תפוקה) – כמות תהליכים מקסימאלית שאפשר לטפל בזמן מסוים.
- Turnaround (זמן סבב) – הזמן מהרגע שתהליך נכנס להרצה ועד שהוא מסיים (כולל המתנה לתורו).
- Waiting time – הזמן שתהליך מסוים חיכה לתורו.
- Response time – הזמן שלוקח מרגע שליחת בקשה ועד קבלת תשובה.

אלגוריתמים שונים של תזמון

FCFS – First Come First Served

המעבד מטפל בתהליך הראשון שמבקש. הדרך הפשוטה ביותר למימוש האלגוריתם היא על ידי תור FIFO, המתזמן התהליכים לפי אופן הגעתם ל-Ready Queue.
זהו אלגוריתם Non-Preemptive, וזמן הממוצע שלו לרב ארוך, כי תלוי בזמן ההגעה של התהליכים.

דוגמא:

Process	Burst Time
P1	24
P2	3
P3	3

ניתן לראות כי זמן ההמתנה של P1 = 0, של P2 = 24 ושל P3 = 27.
לכן זמן ההמתנה הממוצע הוא : (0+24+27)/3 = 17.

לעומת זאת אם הסדר היה P1 -> P3 -> P2

ניתן לראות כי זמן ההמתנה של P2 = 0, של P3 = 3 ושל P1 = 6.
לכן זמן ההמתנה הממוצע הוא : (0+3+6)/3 = 3.

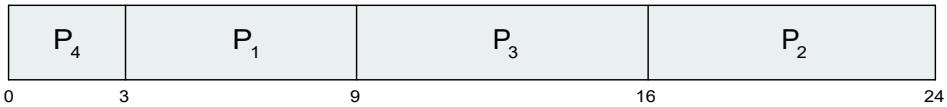
SJF – Shortest Job First

אלגוריתם אופטימאלי הנותן זמן המתנה מינימאלי. צריך לשים לב כי אם יש הרבה תהליכים קצרים ואחד ארוך, ייתכן כי הוא יחכה המון זמן. לכן, ה-Turnaround Time לא אופטימאלי כלל.

דוגמא:

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

שימוש באלגוריתם זה יתזמן את התהליכים באופן הבא: P4 -> P1 -> P3 -> P2.



זמן ההמתנה הממוצע יהיה (3+16+9+0)/4 = 7.

הבעיה : לא נוכל לדעת את אורך התהליך. איך אפשר "לנחש"?

פתרון : לפי הנוסחה $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$ ה-α משמש כמקדם כך ש: $0 \le \alpha \le 1$ (בד"כ 0.5)

הניחוש שקיבלנו לתהליך הקודם. | האורך האמיתי שהיה לתהליך הקודם ב-CPU. | הניחוש שלנו לתהליך הבא.

- α = 0 : $\tau_{n+1} = \tau_n$ - נתייחס לניחוש שקיבלנו בתהליך הקודם ולא לאורך האמיתי.
- α = 1 : $\tau_{n+1} = \alpha\, t_n$ - נתייחס רק לאורך האמיתי שלקח לתהליך הקודם.
- הנוסחה הרקורסיבית :

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

מהסיבה שה-α בין 0 ל-1, נשים לב שעל כל גורם יש השפעה קטנה יותר מאשר על אלו שלפניו.

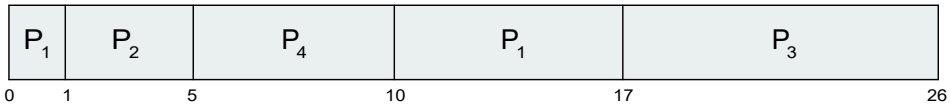
By : Lior Daniel
Email : Lior7daniel@gmail.com

Shortest Remaining Time First

גרסת ה-Preemptive של ה-SJF.
באלגוריתם זה, גם אם התחלנו לטפל בתהליך מסוים, ברגע שקיבלנו תהליך עם זמן קצר יותר – נפסיק ונעבור אליו.
כאן נוסיף פרמטר נוסף – זמן הגעת התהליך לתור ההמתנה לפי יחידות זמן.

דוגמא:

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5



הסבר: התחלנו עם P1 אבל אחרי יחידת זמן אחת קיבלנו את P2.
כרגע P1 = 7 ו-P2 = 4, לכן P2 קצר יותר והמתזמן עוצר את P1 ומטפל ב-P2.
P3 ו-P4 הגיעו תוך כדי הטיפול ב-P2, אך הם עדיין ארוכים יותר ממנו ולכן P2 מסיים את התהליך. המתזמן עובר ל-P4 הקצר אחריו, משם ל-P1 ואז P3.

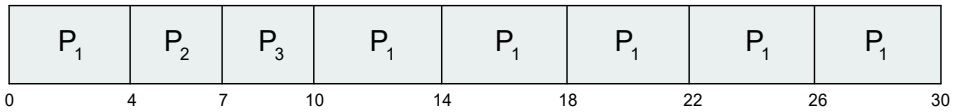
זמן ההמתנה הממוצע יהיה [(10-1) + (1-1) + (17-2) + (5-3)] / 4 = 26 / 4 = 6.5
P1 P2 P3 P4

Round-Robin Scheduling

אלגוריתם Preemptive.
כל תהליך מקבל זמן קבוע לריצה, הנקרא Quantum (בד"כ בין 10-100 מיל" שניות).
האלגוריתם מתייחס ל-Ready-Queue כאל תור מעגלי.
מתזמן המעבד עובד על התור ומקצה Quantum פרק זמן עיבוד בכל פעם לתהליך אחר, כאשר הזמן עבור התהליך מסתיים, התהליך עובר לסוף התור.
אם יש n תהליכים ב-Ready Queue אז אין תהליך שיחכה יותר מ (n-1)*Quantum יחידות זמן.
נשים לב שאם ה-Quantum יהיה גדול מידי נקבל FCFS.
לעומת זאת, נשים לב שה-Quantum לא קטן מידי מאחר וכל החלפת תהליך דורשת Context Switch ש"המחיר" עבורו גבוהה.

דוגמא:

Process	Burst Time
P1	24
P2	3
P3	3



במקרה זה ה-Quantum = 4.
המתזמן מתחיל טיפול ב-P1 ולאחר 4 יחידות זמן מפסיק ועובר ל-P2. מסיים את P2 לאחר 3 יחידות זמן ועובר ל-P3. מסיים אותו לאחר 3 יחידות זמן וחוזר ל-P1 עד שמסיים אותו.

* זמן ההמתנה הממוצע – לבדוק *

Priority Scheduling

אלגוריתם Non-Priority.
אלגוריתם זה נותן זמן ריצה לתהליך בעל העדיפות הגבוהה ביותר.
ככל שמספר ה-Priority נמוך יותר, כל העדיפות עבור התהליך גבוהה יותר.

בעיה: תהליכים עם עדיפות נמוכה עלולים לא לקבל זמן עיבוד אף פעם.

פתרון: Aging (תהליך הזדקנות) – ככל שעובר זמן ותהליך מסוים לא קיבל עדיפות, נעלה לו את העדיפות באופן יזום.

דוגמא:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

זמן הממוצע הוא (0+1+6+16+18+19) / 5 = 8.2.

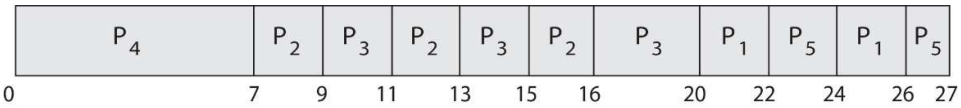
By : Lior Daniel
Email : Lior7daniel@gmail.com

Priority Scheduling w/ Round-Robin

נריץ את Priority אלגוריתם, אך במידה וניתקל בשני תהליכים עם אותה העדיפות, נתזמן ביניהם עם אלגוריתם Round-Robin.

דוגמא:

Process	Burst Time	Priority
P1	4	3
P2	5	2
P3	8	2
P4	7	1
P5	3	3



Multilevel Queue

ניצור תור תהליכים שונה עבור כל רמת עדיפות. עבור כל תור נוכל לבחור אלגוריתם תזמון שונה.
מתחילים לטפל בתור של הרמה הכי נמוכה וכשמסיימים עוברים לרמה הבאה אחריה.

Multilevel Feedback Queue

אלגוריתם הבנוי על בסיס Multilever Queue אך כאן נוסיף אינטרקציה בין התורים על מנת להימנע מתהליכים שיישכחו.
כך נוכל לבצע Aging – הגדלת העדיפות של תהליך באופן יזום עבור אחד כזה שנשכח.

EDF – Earliest Deadline First

לכל תהליך יש זמן ריצה, זמן הגעה לתור ו-Dead Line. כלומר, אפליקציה יכולה להגדיר שהיא רוצה שיטפלו בתהליך בתוך זמן מסוים. ככל שה-Dead Line יותר קרוב, כך העדיפות גדלה.
יש פרמטר נוסף שהתהליך מעביר איתו ומסמל כמה "כסף" האפליקציה מוכנה לשלם למערכת במקרה שהתהליך יטופל בזמן, אך האלגוריתם אינו מתייחס לערך זה ולכן הוא טוב עבור תהליכים שערכם בפרמטר זה שווה.

Linux Scheduling

עובד עם CFS – Completely Fair Scheduler.

nice value – עדיפות התהליך. ככל שיותר קטן העדיפות גבוהה יותר.
target latency – הזמן שבו תהליך אמור לרוץ לפחות פעם אחת (יכול להשתנות לפי כמות התהליכים).
virtual run time – כמה זמן באמת התהליך קיבל זמן עיבוד (ככל שקיבל יותר כך המספר יותר נמוך).

במימוש נכניס את התהליכים לעץ אדום שחור כך שמצד שמאל יהיו התהליכים עם ה-virtual run time הכי נמוך ובימין הכי גבוה.
במקרה בו יש כמה מעבדים, נוודא שתהליך שהתחיל במעבד מסוים גם ימשיך ויסיים איתו.

Windows Scheduling

בנוי על Preemptive ומתזמן לפי עדיפות (לכל רמת עדיפות תור משלה).
אם אין תהליך שירוץ, מריצים תהליך סרק.
תהליכים שהם real-time יכולים לעקוף כאלה שלא.

יש 32 רמות עדיפות :
0 – תהליכים של ניהול זיכרון.
Variable Class – 1-15
Real-Time Class – 16-32

Solaris

מתזמן לפי עדיפות.
בעל Class 6'ים, כך שלכל Class יש אלגוריתם תזמון משלה. כל תהליך יכול להיות ב-Class אחד בו זמנית.

Fixed Priority (FP), Fair Share (FSS), System (SYS) ,Time Sharing (TS), Interactive (IA), Real Time (RT).

By : Lior Daniel
Email : Lior7daniel@gmail.com

IPC – Inter Process Communication (תקשורת בין תהליכים)

יש שלושה דרכים בהם תהליכים יכולים לתקשר ביניהם:
1. Signals.
2. Channels.
3. Shared memory.

Signals סיגנלים הם בעצם פסיקות תוכנה (System Calls), דרכם ניתן להעביר אירועים אסינכרוניים לאפליקציה.

דוגמא: משתמש לוחץ ctrl+c, התהליך שולח סיגנל "להרוג" תהליך אחר.
ישנם כ-64 סיגנלים, ניתן לראות אותם על ידי הפקודה "kill -l".

ניתן לשנות את ההתנהגות של אחד מהסיגנלים (למעט SIGSTOP ו-SIGKILL) על ידי שימוש בפקודה signal(<SIGNAL_NAME> ,<handler_function_name>)

ניתן להתעלם מסיגנלים באמצעות הפקודה SIG_IGN במקום ה-handler_function_name.

Pipes צינור המאפשר תקשורת בין תהליכים. עובר דרך ה-kernel.

יש שני סוגי Pipes :

1. Ordinary Pipes – מאפשר תקשורת בין תהליכים בעלי קשר של אב \בן. לא ניתן יהיה לגשת לתהליך מחוץ לקובץ שיצר אותו.
נשתמש בפקודת ה-()Pipe התקבל מערך בעל 2 תאים: [0] לקריאה ו-[1] לכתיבה.
לדוגמא: בקוד שלנו נבצע את פקודת ה-()pipe לפתיחת צינור, לאחר מכן נבצע את פקודת ה-()fork לפיצול של תהליכים בתצורת אב ובן, וכאשר נרצה לכתוב משהו מתהליך כלשהו נכתוב או נקרא בהתאמה ע"י הפקודות הבאות:
כתיבה: write(fd[1], variable adress, sizeof(variable))
קריאה: read(fd[0], variable adress, sizeof(variable)
התאים לקריאה ולכתיבה ניתנים לנעילה על ידי הפקודות close\open(fd[1\0]).

2. Names Pipes – מאפשר גישה לתהליך ללא קשר של אב \ בן על ידי שימוש בשם ה-Pipe.
ניצור קובץ על ידי הפקודה mkfifo שישמש כ-named pipe, ניצור שני קבצים A ו-B הישמשו כתהליכים הרוצים לתקשר ונניח ש-A רוצה לכתוב ל-B.
שניה הקבצים יצטרכו "לפתוח גישה" ל-named pipe שלנו ע"י הפקודה open המחזירה לנו "מצביע" ל-named pipe ו-(1-) אם יש שגיאה.
לדוגמא: int fd = open("the named pipe path", 0_RDONLY \ 0_WRONLY)
לאחר מכן כל קובץ יכתוב בהתאמה את פקודה הקריאה \ כתיבה:
כתיבה:
write(named pipe adress – in our case: fd, variable adress, sizeof(variable))
קריאה:
read(named pipe adress – in our case: fd, variable adress, sizeof(variable))

By : Lior Daniel
Email : Lior7daniel@gmail.com

Synchronized Algorithms

לסדרת סרטונים בנושא הסוגרת את הרצאה 5+6 לחצו כאן (מבטא הודי, אבל מומלץ מאוד!!).

נניח כי קיימים שני תהליכונים P ו-Q, כאשר :

P:	x = 2		Q:	x = 3
	y = x – 1			y = x + 1

אנו יודעים כי המעבד מחלק את זמן העיבוד בצורה לא מסודרת, לכן, מכיוון ששני התהליכונים מתחרים על ה-'x', אנו עלולים לקבל תוצאות שונות בכל פעם. דוגמא: (3,4) \ (3,2) \ (2,3) \ (2,1).

הגדרות :

Deterministic – עבור אותו הקלט תמיד נקבל את אותו הפלט.

Non-Deterministic – עבור אותו הקלט יכולים להיות תוצאות שונות לפלט.

Race Condition – תחרות על משתנים שעלולה להיות עבור תוכנית לא דטרמיניסטית.

Mutual exclusion – הידור הקוד בצורה שבה נימנע מהתנהגות לא דטרמיניסטית.

Bernstain Conditions – אם לתהליך שלנו יש תהליכונים P ו-Q כך ש:
=> משתני הפלטים של P ו-Q שונים.
=> משתני הפלטים של P שונים ממשתני הקלטים של Q.
=> משתני הפלטים של Q שונים ממשתני הקלטים של P.
אז התהליך דטרמיניסטי.
אחרת, יכול להיות דטרמיניסטי או לא (לא בהכרח שיהיה לא דטרמיניסטי).

Critical section – קטע קוד שמשתמשים בו כמה תהליכונים במקביל, כך שכל תהליכון משנה\מעדכן משתנה משותף, נקרא קטע קריטי.
לא נרצה שכאשר תהליכון אחד משתמש בקטע הקריטי, תהליכון אחר יכנס וישנה דברים.
לכן, נרצה שכל תהליכון שירצה לגשת לקטע קריטי, יבדוק שאף תהליכון אחר לא משתמש בו.

By : Lior Daniel
Email : Lior7daniel@gmail.com

פתרונות :

1. SW Algorithm (מצגת).
2. Mutual exclusion (מצגת).
3. Progress: מי שיחליט לשחרר את קטע הקוד הקריטי הוא אך ורק התהליך שבתוך הקטע הקריטי ולא תהליך ממקום אחר בקוד.
4. Bounded waiting (מצגת).