מערכות הפעלה – מושגים

* גרעין ליבה:

- קטע קוד, אחד מקטעי הקוד הראשונים שעולים בהדלקת המחשב, כאשר הראשון שעולה הוא ה-bootloader שהוא טוען את ה-kernel ולאחר מכן הוא היחידה שרצה כל הזמן על המחשב והיא ליבת מערכת ההפעלה.

- הליבה מתעוררת כאשר interrupts מתרחשים.

- שוכן בזיכרון הראשי RAM

- אחראי על תפקוד מערכת ההפעלה

- טיפול בפסיקות חומרה -ניהול פעולות קלט פלט -תזמון שימוש תהליכים במעבד -ניהול מרחב זיכרון

* תפקידי מערכת ההפעלה:

- להריץ את תוכניות המשתמש

- להפוך את בעיות המשתמש לקלות יותר

- שימוש בחומרת המחשב בדרך יעילה

* מערכת המחשב מחולקת ל-4 מרכיבים:

1. חומרה: נותנת את משאבי המחשוב הבסיסיים כגון CPU, זיכרון, IO
2. מערכת ההפעלה: שולטת ומתאמת את השימוש בחומרה בין אפליקציות שונות בשביל משתמשים שונים, תוך ניצול יעיל של מערכת המחשב
3. אפליקציות: מגדירים את הדרכים בהן יעשה שימוש במשאבי המערכת לפתירת בעיות מחשוב של משתמשים
4. משתמשים: אנשים, מכונות, או מחשבים אחרים (משתמשי קצה)

* **קריאת מערכת System Call:**

-קריאת מערכת היא בקשה שמבצעת תוכנת מחשב מליבת מערכת ההפעלה, על מנת לבצע פעולה שהיא אינה יכולה לבצע בעצמה.

-תהליכים לדוגמא מנועים מגישה חופשית ל-KERNEL ולכן צריכים לבצע SC כדי לקבל הרשאה לבצע שינויים ופעולות על ה-KERNEL.

-ממשק המקשר בין אפליקציית משתמש לשירות שמערכת ההפעלה (או ה-KERNEL) מספקים.

-SC יכולים להיות מתוייגים ל-5 קבוצות:

1. שליטת תהליכים (Process Control)

2. ניהול קבצים (קריאה, כתיבה)

3. ניהול מכשירים **?**

4.תחזוקת מידע (לדוגמא: קביעת שעה או תאריך של המערכת)

5. תקשורת

-קריאות מערכת מתבצעות בדר"כ באמצעות פונקציות מספריות C/C++

* Interrupt: אות המגיע למעבד מרכיב חומרה או תוכנה, מאפשר לשנות את סדר ביצוע הפקודות בתוכנית מחשב שלא על ידי בקרה מותנית. בעת קבלת הפסיקה משהה המחשב את ביצועה הסדרתי של התוכנית, כדי להפעיל שגרת טיפול בפסיקה. לאחר הטיפול, ממשיך המחשב בביצוע הסדרתי של התוכנית. **פסיקות משמשות כאמצעי תקשורת בין תהליכים במחשב.**

- פסיקה מחומרה: יכולה לקורת בכל שלב ע"י שליחת Signal ל-CPU.

- פסיקה מתוכנה מתבצעת ע"י System call.

- המעבד בודק אם יש פסיקות ממתינות לטיפול בין ביצוע של כל 2 פקודות מכונה.

- פסיקה אינה קוטעת ביצוע של הוראת מכונה בסיסית **?**

- במידה ויש פסיקה הממתינה, המעבד עובר ל-KERNEL MODE וקוראת לשגרת הטיפול בפסיקה **?**

* שגרת טיפול בפסיקה – Interrupt Service Routine:

בלוק מיוחד של קוד אשר מקושר עם פסיקה ספציפית.

למשל, קוראת את התו שנלחץ במקלדת ושומרת אותו במקום מתאים בזיכרון.

סוגי פסיקות:

1. פסיקה סינכרונית: פסיקת תוכנה, נוצרת ע"י המעבד כתגובה על ביצוע פקודות מכונה אי חוקיות מסויימות (חילוק ב-0)
2. פסיקה אסינכרונית – פסיקת חומרה, יכולה להגיע בכל רגע ללא קשר למצב המעבד באותו הזמן. למשל: לחיצה על מקש במקלדת.

תהליך טיפול בפסיקה:

1. החומרה מעבירה את הבקרה למערכת ההפעלה
2. שמירת מצב ה-CPU ע"י שמירת ערכי הרגיסטרים וה-PC
3. קביעת סוג הפסיקה בעזרת Interrupt Vector Table
4. הפעלת קטע המתאים עבור הפסיקה

* מעבר בין USER MODE ל-KERNEL MODE

|  |  |
| --- | --- |
| User Mode | Kernel Mode |
| תוכניות משתמש | תוכניות מערכת הפעלה |
|  | במצב זה המעבד מבצע פקודות מוגנות |
| כאשר למשתמש יש צורך לגשת לרכיב קלט\פלט הוא יכתוב פקודת System Call אשר תעביר את המעבד למצב Kernel Mode | המעבד יעשה את המוטל עליו, לאחר קריאת SC ואז יחזור למצב User Mode. |

* סוגי זיכרון על המחשב:

|  |  |
| --- | --- |
| Main Memory – המקום שבו תוכניות רצות. ה-CPU יכול לעבוד איתו בצורה ישירה. | |
| RAM – Random Access Memory  זיכרון חשמלי, נמחק עם כיבוי המחשב | ROM – Read Only Memory  זמין לקריאה בלבד. מכיל מידע הדרוש למערכת ההפעלה על מנת להעלות את המחשב. לא נמחק כאשר המחשב נכבה. מידע צרוב. |
| Secondary Storage  זיכרון משני. הרחבות של הזיכרון הראשי המספקות יכולת אחסון מאוד גבוהה. זיכרון מכני. גישה אליו נעשית באמצעות Interrupt. | |

* **Process – תהליך:**

נקודות מהמצגת של ארקדי:

- תהליך לא אמור לגשת ל-KERNEL של המחשב.

-אין לו יכולת לגשת לkernel memory או לפונקציות

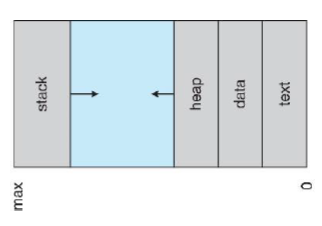
-סיבות בגללן תהליך מנוע מגישה חופשית לקוד ליבה:

1. יכול לסכן \ לפגוע בתהליכים אחרים שרצים

2.לגרום לפגיעה פיזית למכשירים (התחממות יתר וכו'...)

3. לשנות את התנהגות המערכת

\* כאן נכנס התפקיד של System Call, מאפשר לתהליכים ליצור קשר עם ה-KERNEL בצורה בטוחה ו"לבקש רשות" לפני שהוא מבצע פעולות על ה-KERNEL.

- מספר תהליכים שרצים במקביל הוא כמקסימום מספר הליבות.

- כל ליבה יכולה להריץ תהליך אחד בלבד.

- מצבים של תהליכים:

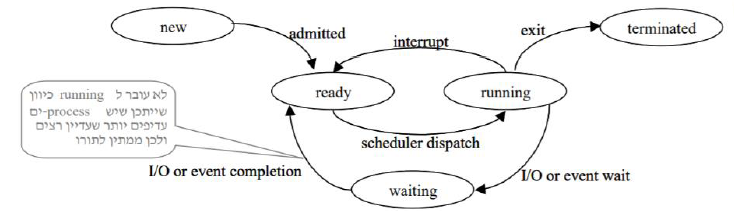
1. New – ברגע שהprocess נוצר.

2. Running- התהליך מריץ פקודות. בזמן נתון רק תהליך אחד יכול להימצא במצב זה!

3. Waiting – התהליך ממתין ל-event (לרוב IO). ממצב זה לא ניתן לעבור לריצה.

4. Ready – התהליך ממתין לקבל את ה-CPU. מוכן לעבודה.

5. Terminated- התהליך הסתיים.



פונקציית fork():

- יוצרת תהליך חדש, שהוא העתק מדוייק של התוכנית שקראה ל-fork() (מאותה שורה והלאה)

-גם האב וגם הבן יתחילו את התוכנית שלהם מהנקודה בתוכנית הראשית שקראה ל-fork()

-פונקציית fork() מחזירה child pid אם היא נקראת ע"י ההורה **או** שהיא תחזיר 0 אם נקראה ע"י הילד.

(ככה יהיה ניתן לזהות איזה תהליך ביצע איזה פונקציה)

- getppid(): תהליך ילד יכול להשיג את המספר זהות של תהליך האב.

- חיסרון אפשרי ביצירה של child process: האב מעתיק את כל התוכן שלו לתוך תהליך הבן. לפעמים זה בזבזני כי הבן לא צריך את כל הנתונים שהאב משתמש בהם.

תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי

ילדים זומבים

- כאשר תהליך מסתיים, הזיכרון והמשאבים שהוקצו עבורו משתחררים.

- עם זאת, נשאר תיעוד של התהליך בטבלת התהליכים של המערכת (שומרת על סדר הכנסה והוצאה של תהליכים לפי יכולות המערכת)

- דבר זה, מאפשר לתהליך האב לקרוא נתונים על סטטוס תהליך הבן.

-כאשר האב לא מבקש לקרוא את הסטטוס של הבן ותהליך הבן הסתיים, הוא נקרא "זומבי" כי כל מה שנשאר מימנו זה התיעוד בטבלה.

\*יש סיטואציות מסויימות בהם דליפת זיכרון זאת היא רצויה בכוונה (לדוגמא, למנוע יצירה של תהליך בן עם אותו מספר pid)

- כיצד נאתר ונאסוף את אותם תהליכים "זומבים" ?

1. על ידי פונקציה שהאב משתמש בה בשם wait().

2. ניתן לזהות אותם ע"י פונקציית ps -el

פונקציית wait():

מכריחה את התהליך שקרא לה, להמתין עד שהסטטוס של אחד תהליכי הבן ישתנה.

-עוצר את ההוצאה לפועל של התהליך הקורא

-סטטוס אפשרי של תהליך:

1. terminated

2. was stopped

3. ברגע שהסטטוס של תהליך בן נקראה ע"י תהליך אב באמצעות פונקציית wait(), רק אז התיעוד של תהליך הבן ימחק מטבלת התהליכים של המערכת.

-waitpid(): מחכה רק עבור תהליך ספציפי

-waitid(): מחכה לתהליך מתוך קבוצה של תהליכים

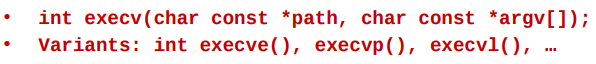
ילדים יתומים

כשתהליך האב מת לפני תהליך הבן, תהליך הבן נהפך לתהליך יתום.

היתומים "מאומצים" על ידי המערכת ומבוצע תהליך שנקרא reparenting.

(לפעמים תהליכים יתומים נוצרים במכוון, לדוגמא ליצירת daemon .. )

פונקציית exec():

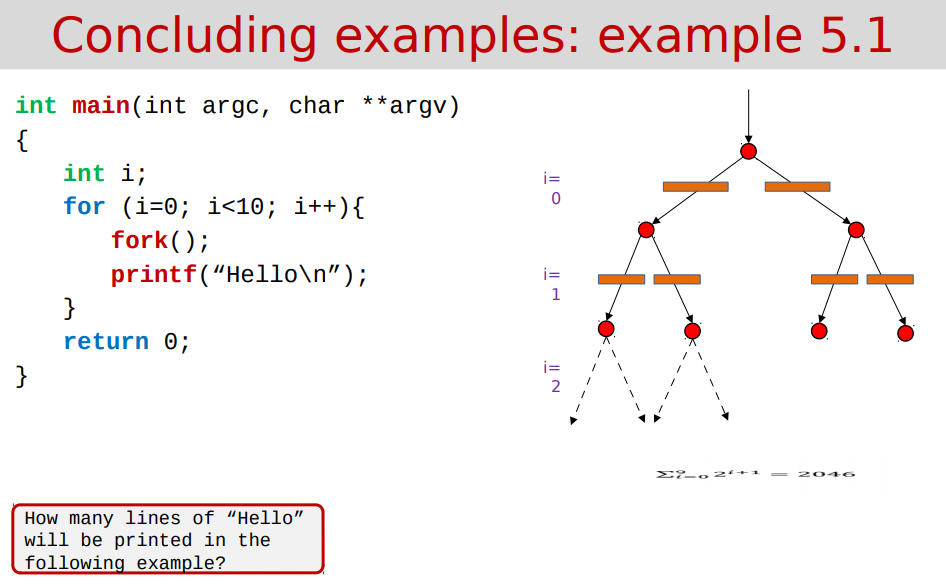


- קבוצת פונקציות מספריית C, שדורשות מתהליך קיים לעבוד על דבר אחר לחלוטין.

-כשקוראים לפונקציה זו, לא ניצור pid חדש, כי לא נוצר תהליך חדש.

פונקציית errno():

תרגיל:



* **Thread – תהליכון:**

- ייתכנו כמה תהליכונים בתוך תהליך (process) אחד

- תהליכונים מאפשרים הוצאה עצמאית לפועל של אותה תוכנית כמה פעמים (תוך שימוש משותף באותם משאבים של ה-process תחתיו הם קיימים)

- מצבים אפשריים עבור תהליכון:

1. רץ Running

2. מוכן לריצה Ready

3. חסום Blocked

4. מת – Terminated

|  |  |
| --- | --- |
| **יתרונות** | **חסרונות** |
| התהליכונים חולקים משתנים, מבני מידע, קבצים פתוחים, תהליכים בנים וכו'... | ביטחונית & יציבות: בגלל שהם חולקים את אותם נתונים (אותו מערך לדוגמא) יש סכנה של דריסת נתונים, חריגה מגבולות וכו' ... |
| תהליכונים יכולים ליצור קשר אחד עם השני ללא צורך בשימוש ב-System Calls. | Signals משפיעים על כל התהליכונים הקשורים לאותו תהליך. |
| מהיר יותר ליצור/לסיים/ להחליף בין תהליכונים מאשר בין תהליכים (process) ~ בערך פי 10 עד 100 |  |

-בלינוקס: כאשר Thread מבצע fork() אז רק אותו תהליכון מועתק, אבל הדבר גורם לבעיות (לדוגמא, כאן תהליך הבן משתמש ב-mutex variable)

User Level Thread VS. Kernel Level Threads

|  |  |
| --- | --- |
| **User-Level Thread** | **Kernel-Level Thread** |
| - ממומשים בספריות ברמת משתמש (כמו Pthread)  - החלפת תהליכונים לא דורשת קריאה למערכת ההפעלה ואינה יוצרת interrupts (אותות ל-kernel)  -אפליקציית המשתמש היא זאת שמתזמנת את זמן ניצול ה-CPU של התהליך (עבור התהליכונים הפנימיים שלו)  -מאפשר ניצול של CPU בודד, מערכת ההפעלה לא תאפשר ניצול של יותר ממעבד בודד פר תהליך. | - תהליכונים אלה חשופים בפני הליבה  - הליבה יכולה לנהל ולתזמן תהליכונים אלו  - קיימים System Calls שמאפשרים יצירה וניהול של תהליכונים |
| **יתרונות** | **יתרונות** |
| ניתן ליצור תהליכונים במערכות הפעלה שלא תומכות בתהליכונים. | הליבה יכולה לתזמן בצורה חכמה בין תהליכים עם מספר שונה של תהליכוני-ליבה. |
| כל תהליכון מיוצג ע"י program count (PC), רגיסטרים, מחסנית (stack), ויחידת בקרת שליטה. | תהליכוני-ליבה טובים במיוחד עבור אפליקציות שנחסמות בתדירות **?** |
| ניהול מהיר ויעיל: יצירת תהליכון, סינכרון והחלפה בין תהליכונים – כל אלה יכולים להתבצע ללא התערבות של ה-KERNEL, ולכן זולים יותר ובערך פי 100 מהירים יותר מאשר kernel – level – threads. | במערכת עם מספר רב של מעבדים, כמה CPU יכולים לרוץ בצורה מקבילית עם תהליכונים שונים ששייכים לאותו תהליך. |
| **חסרונות** | **חסרונות** |
| חוסר תיאום בין התהליכונים ולליבת מערכת ההפעלה (kernel): לכן, תהליך (process) יקבל זמן קצוב וקבוע על ה-CPU ללא קשר לכמה תהליכונים רצים בתוכו. המשאבים שהתהליך צורך קבועים. | יצירה, ניהול והחלפה של תהליכונים אלו היא יקרה ואיטית מאשר תהליכוני-משתמש. |
| System Call באחד התהליכונים יגרום למערכת ההפעלה לחסום את כל התהליך עצמו ולכן את כל התהליכונים שרצים בתוכו. |  |
| חוסר היכולת של הליבה להבדיל בין תהליכוני-משתמש מקשה לעצב מתזמן מותאם לתהליכונים שהם בנים של אותו תהליך (threads in a process) |  |

תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי

תמונה שמכילה שולחן

התיאור נוצר באופן אוטומטי

* מתזמן תהליכים:

בגלל שיש הרבה תהליכים שרצים במקביל ובגלל שיש יותר מעבדים מאשר ליבות, אז נצטרך מתזמן תהליכים.

- ממקסם את צריכה המעבד

- בוחר מכלל התהליכים אשר נמצאים במצב Ready את התהליך הבא שירוץ על המעבד.

* החלפת קשר – Context Switch:

מעבר בין הרצת שני תהליכים באמצעות המעבד. באמצעות החלפת הקשר, מספר תהליכים יכולים לחלוק את אותו מעבד. החלפת ה-CPU לתהליך אחר דורשת שמירת המצב הקיים של התהליך הישן וטעינת המצב של התהליך החדש.

\* תהליך החלפת הקשר נחשב לתהליך בזבזני מבחינת משאבי המערכת ועל כן מערכות הפעלה מנסות לבצע אופטימיזציה בשימוש בהן. הדיליי של החלפת קשר נקרא גם Dispatch Latency.

* מתי יש צורך בשיתוף תהליכים?

- שיתוף מידע.

- זירוז חישוביות. במקום שתהליך אחד יבצע הכול, כל תהליך מבצע פעולה קטנה.

- מודולאריות

- נוחות.

* **IPC-Inter Process Communication-Signals:**המטרה העיקרית של signals היא לייצר שליטה על תהליכים.

-אותות יכולים להישלח מהמקלדת, קומנד-ליין, באמצעות sysCalls

-אות יכולה להישלח לתהליך על ידי תהליך אחר או על ידי ה-KERNEL.

רשימת Signals אפשריים:

1. kill – l

2. sigInt – interrupts a process  
3. sigStop – stop (pause) a process  
4. sigCont – continues (resume) a process  
5. sigUsr1 – a “free” signal for developer to use  
6. sigTerm – terminates the prosess gracefully (like ctrl+z)  
7. sigKill – terminates immediately

אנחנו מתמודדים עם signals רק כאשר תהליך סיים התמודדות עם Interrupt או עם system call ולפני חזרה למצב משתמש.

-ניתן לחסום אותות. כלומר האות התקבל אבל הטיפול באות נעצר, דבר המגן על פעולות רגישות (לדוגמא: למנוע מאותות אחרים להתערב ב-signal handler רץ, למנוע מה-signal handler שינויים במשתנים גלובליים שאנחנו כרגע משתמשים בהם)  
-ניתן להתעלם מאותות לחלוטין.

-לכל תהליך יש טבלת signals

תמונה שמכילה שולחן

התיאור נוצר באופן אוטומטי

עמודה ראשונה מציינת מס' אות, שנייה האם להתעלם מימנו או לא, שלישית מציינת מה לעשות במידה וקיבלנו את האות.

|  |  |  |
| --- | --- | --- |
| Fg<PID> | Killall<signal><PNAME> | Kill -<signal><PID> |
| Resumes the execution of a suspended process by sending a SIGCONT signal. This will cause the resumed process to run in the **f**ore**g**round | can be used to send multiple signals to processes running specific commands, owned by a specified user, have a certain age etc | \*Sends the specified signal to the specified PID  \*e.g. **Kill -9** 1024 sends signal 9 (SIGKILL) to process 1024  \*If no signal is specified, the TERM signal is sent |

- לא כל תהליך יכול לשלוח סיגנלס לכל שאר התהליכים

- רק קוד גרעין וסופר-יוזר יכולים לשלוח אותות לכל התהליכים.

-תהליכים רגילים יכולים לשלוח אותות רק לתהליכים שמשויכים לאותו משתמש

-Process Group ID:

- קבוצת תהליכים היא אוסף של תהליכים הקשורים זה לזה

- לכל תהליך יש PID ו-group ID שזה ה-PGID שלו.

-סינגל יכול להישלח לתהליך בודד או לקבוצה של תהליכים

|  |  |  |  |
| --- | --- | --- | --- |
| Setgrp(int pid1, int pid2) | Setgrp() | Int getgrp() | Int getpid() |
| משייך את pid1 לקבוצה של pid2 | משנה את PGID לPID של התהליך הנוכחי | מחזיר את ה-PGID של התהליך הנוכחי |  |

Signal-handler:

-מיישמים אותו פר תהליך, פר אות.

-נציין שכל תהליך צריך להגדיר האם הוא מתעלם ל-signal / משתמש ב-signal דיפולטיבי / מגדיר signal handler

-מוציאים לפועל את חלקת הקוד של signal\_handler ב-user mode

-signal\_handler יכול לבצע syscalls

תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי

תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי

-אם משתמשים בפונקציה execv(), כל האותות יאותחלו להתנהגות דיפולטיבית, אך לרוב האותות, אם צויין שנדרש להתעלם מהם, נתון זה ישמר.

-לא ניתן להתעלם מsigkill, sigstop

* תמונה שמכילה טקסט

  התיאור נוצר באופן אוטומטיפונקציות ליצירת תהליכונים:

תהליך זומבי: תהליך שסיים אך מחכה לעדכן את תהליך האב. אף אחד לא קרא את ערך החזרה עדיין.

כאשר תהליך מסתיים, כל הזיכרון והמשאבים ששויכו לאליו משוחררים חזרה למערכת. יחד עם זאת, הרשומה של תהליך בטבלת התהליכים אינה מוסרת.

- תהליך האב שיצר את תהליך הבן שהסתיים יכול לקרוא את ערך היציאה של הבן באמצעות קריאת המערכת wait

-רק מרגע שזו נקראה תהליך הזומבי מוסר מהרשימה.

- לאחר שהזומבי מוסר מרשימת התהליכים, מזהה התהליך שלו מתפנה לשימוש חוזר בידי תהליך חדש אחר שנוצר.

- יחד עם זאת, אם ההורה לא קרא ל-wait, הזומבי יישאר ברשימת התהליכים עד סיום ריצתו של תהליך האב.

- מרגע שתהליך האב הסתיים, עובר תהליך הבן לתהליך init.

- האחרון עובר באופן מחזורי על רשימת תהליכים הזומבים ומשחרר אותם ע"י הקריאה ל-wait.

* **Scheduler \ מתזמן תהליכים:**

קריטריונים לתזמון:

כאשר אנחנו מחליטים באיזה אלגוריתם להשתמש במצב מסויים, יש לקחת בחשבון את המאפיינים של כל אלגוריתם.

הקריטריונים בהם משתמשים להשוואות אלגוריתם של מתזמני CPU הם:

1. ניצול CPU: השאיפה היא לוודא שה-CPU עסוק כל הזמן.
2. תפוקה (Throughput): יעילות הרצת התהליכים חופפים. אחת הדרכים לבדוק את עבודת האלגוריתם, היא מספר התהליכים שסיימו ביחידת זמן אחת.
3. זמן סבב Turnaround: מנקודת מבט של תהליך ספציפי, הקריטריון החשוב הוא כמה זמן לוקח להריץ את התהליך. המרווח בין הזמן שהתהליך ביקש לרוץ לבין הזמן שהתהליך יסתיים, נקרא זמן סבב. זמן זה הוא סכום של:

- התהליך המתין לזיכרון

- המתנה בתור ready

- ריצה ב-CPU

-הפעלת I\O

המדידה של זמן זה החלה כאשר הוא נכנס ל-Memory Queue

1. זמן המתנה : האלגוריתם אינו משפיע על הזמן בו תהליך רץ או משתמש ב-IO, אלא רק על הזמן שתהליך מעביר בהמתנה בתור Ready. זמן ההמתנה הוא סכום משך הזמנים הכולל שהתהליך העביר בתור ה-ready.
2. זמן תגובה – במערכת אינטראקטיבית, זמן הסבב אינו יכול להיות קריטריון מתאים, משום שלעיתים קרובות, תהליך יכול לספק חלק מהפלט די מהר ולהמשיך בחישובים אחרים בזמן שחלק מהפלט כבר מוצג למשתמש. לכן אמת מידה נוספת היא: הזמן שלוקח מרגע הבקשה ועד לתגובה הראשונה. זמן התגובה הינו משך הזמן שלוקח להתחיל להגיב, אבל לא כולל את זמן התגובה עצמו. קובע את מידת ההגינות של המערכת – זמן תגובה דומה לכל התהליכים זה ממתי שנכנס ל-ready ועד שקיבל cpu ועבר להיות running.

המטרה של אלגוריתמים אלו היא להביא את ניצול ה-CPU ואת התפוקה למצב מקסימלי, ולמזער כמה שיותר את זמן הסבב, זמן ההמתנה וזמן התגובה. הרוב המקרים מנסים לייעל את הממוצע של כל הקריטריונים, אך קיימים מצבים בהם רצוי לייעל את ערכי המקסימום או המינימום ולא את הממוצע.

לדוגמא: כאשר רוצים להבטיח שכל המשתמשים יקבלו שירות טוב, נרצה להקטין את זמן התגובה המקסימאלי.

תרגול במצגת של בן גוריון

* FCFS: First Come First Served- המעבד מוקצה ראשון לתהליך הראשון שביקש אותו.

הדרך הפשוטה ביותר לממש אלגוריתם זה הוא בעזרת תור FirstInFirstOut או בשמו האחר FIFO.

FIFO מתזמן תהליכים באופן הגעתם ל-Ready Queue, כלומר התהליך הראשון יתוזמן ראשון והתהליך שבא אחריו יתוזמן אך ורק לאחר שהתהליך הראשון סיים.

\* אלגוריתם זה הוא מסוג Non-Preemptive. זמן ההמתנה הממוצע במקרה זה לרוב הוא ארוך, תלוי בזמן ההגעה של התהליכים האחרים.

ניקח לדוגמא:

תמונה שמכילה שולחן

התיאור נוצר באופן אוטומטי

לעומת זאת, אם התהליכים היו מגיעים בסדר P2,P3,P1 אז זמן הממוצע שהיה מתקבל הוא 0+3+6/3.

באלגוריתם זה תיתכן הרעבה, ברגע שתהליך שקיבל את זמן המעבד נכנס ללולאה אינסופית.

* SJF – Shortest Job First

אלגוריתם מסוג Non-Preemptive.

-זהו אלגוריתם אופטימאלי הנותן זמן המתנה מינימאלי.

-אם יש הרבה תהליכים קצרים ותהליך אחד ארוך, אז תתכן הרעבה של התהליך הארוך.

-הקבלה לאלגוריתם זה הוא "אפשר רק לשאול שאלה קצרה?"

-ה-Turnaround Time לא אופטימלי כלל (כל הזמן שתהליך מחכה מהרגע שנוצר עד רגע הסיום)


התיאור נוצר באופן אוטומטי

* Shortest Remaining Time First: גרסא preemptive של SJF.

באלגוריתם זה, הריצה של תהליך כלשהו יכול להיעצר אחרי זמן מסוים, בעת ההגעה של כל תהליך, המתזמן את התהליך בעל זמן הריצה הקצר ביותר שנשאר כל פעימה של שעון.

* תזמון סרט נע – Round Robin Scheduling

כל תהליך מקבל זמן קבוע לביצוע, זמן זה נקרא quantum. מתייחס אל תור ה-Ready Queue כאל תור מעגלי.

מתזמן המעבד עובר על התור ומקצה זמן עיבוד בכל פעם לתהליך אחר לזמן של quantum, כאשר הזמן מסתיים, התהליך נוסף לסוף ה-Queue.

* Priority Scheduling

None-preemptive

אלגוריתם אשר מסתכל על תיעדוף בין תהליכים. המעבד יקצה זמן לתהליכים עם עדיפות גבוהה ביותר. ככל שהמספר שה-priority קטן יותר, אז העדיפות גבוהה יותר.

בעיית הרעבה: תהליכים בעלי עדיפות נמוכה יכולים לא לקבל זמן מעבד אף פעם.

פתרון: aging. ככל שעובר זמן ותהליך מסויים לא קיבל עדיפות, נעלה לו את ה-priority בצורה יזומה.

* Priority Scheduling w/ Round-Robin

שילוב של 2 האלגוריתמים, כאשר קיימים 2 תהליכים עם אותה עדיפות, נתזמן ביניהם עם Round-Robin.

* **Real Time Operating Systems – RTOS**

מערכת זמן אמת דורשת כי משימות יתבצעו בטווח זמן מסוים ולא יאוחר מימנו.

- מערכת בטיחות קריטית: סוג של מערכת זמן אמת עם תוצאות **חמורות** של אי עמידה בזמנים

- מערכת זמן אמת קשיחה: מבטיחה עמידה ביעדים עבור כל המשימות שהן RT. לאחר היעד, אין חשיבות לערך המוחזר.

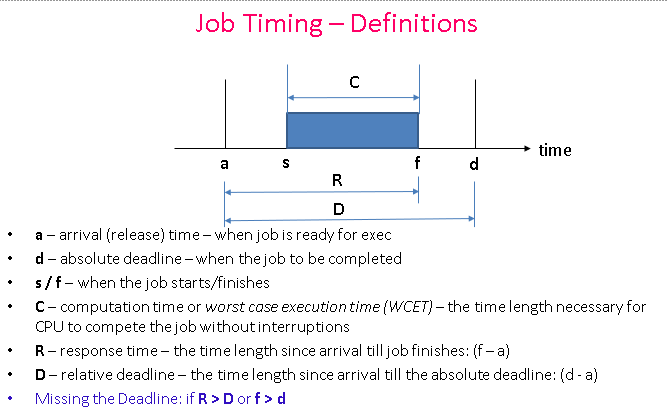
- מערכת זמן אמת רכה: נותנת תיעדוף למשימות זמן אמת, על משימות שהן לא. לרוב יש עמידה בזמנים של המשימות. קיים ערך מוחזר גם לאחר חריגה בזמנים. (לדוגמא לינוקס)

דרישות עבור מערכת זמן-אמת:

\* preemptive kernel

\* priority based preemptive scheduler

\* מינימום דיליים: מתבטא בכך שעמידה ביעדים היא הקריטית ביותר ולכן החשיבות של זה עולה על החשיבות של ניצול מקסימלי של CPU, זמן ממוצע מינימלי לturnaround, זמן תגובה והמתנה וכו'...



תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי

אלגוריתמים אפשריים:

- Earliest Deadline First:

עבור מערכת זמן-אמת, "אופטימליות" היא התכונה החשובה ביותר של המתזמן, כלומר, האלגוריתם יעמוד בכל הזמנים של המשימות כשבסך הכול, ה-utalization<number of CPUs על המחשב. נשתמש באלגוריתם זה כשמתקיימים התנאים הבאים:

1. מעבד יחיד
2. Processor preemption is allowed
3. זמן שחרור א

תמונה שמכילה טקסט, תשבץ, מקורה

התיאור נוצר באופן אוטומטי