Introduction

יום שישי 21 יולי 2017 12:49

מערכת ההפעלה היא שכבת תוכנה שמקלה על עבודת המתכנת ומאפשרת לו לא להכיר לעומק ולנהל המון מרכיבים של חומרה.
אנחנו מכירים מערכות הפעלה שונות כמו windows, Lynux וכד' אבל למען האמת, התוכנה שדרכה אנחנו מפעילים את המחשב היא לא מערכת ההפעלה עצמה. התכונה נקראת shell אם זה מבוסס טקסט ו- GUI (Graphical User Interface) כשנעשה שימוש באיקונים גרפיים. התוכנות האלה הן לא חלק ממערכת ההפעלה, על אף שהן משתמשות בה כדי לעשות את מה שצריך לעשות.

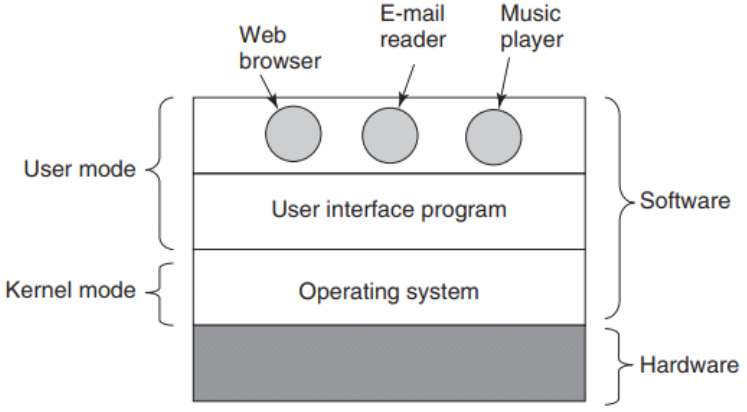
לכל רכיב במחשב יש קונטרלור. במערכת ההפעלה יש דרייבר שיכול לדבר עם הקונטרולר הספציפי של רכיב ספציפי.במדפסת למשל יש רכיב שיודע לדבר מול הדרייבר (לקבל פקודה) והדריביר יודע לדבר עם המדפסת שתבצע פקודה מסויימת (להדפיס A למשל).

מערכת ההפעלה מאפשרת לנות לקשר עם הרכיבים השונים של המחשב. היא דואגת לממשק פשוט להפעלה עם כל מה שאנחנו צריכים. מערכת ההפעלה מדבר כאמור עם רכיבי החומרה.

על מערכת ההפעלה לנהל משאבים של החומרה ולזהות מרכיבים של חומרה (למשל לזהות דיסק און קי שנכנס. אם כך, ניתן לומר שמערכת ההפעלה מסתירה ממני את רכיבי החומרה ויוצרת ממשק.

באיור ניתן לראות את המרכיבים העיקריים שעליהם אנחנו מדברים בשלב זה.
ניתן לראות שהחומרה הכי למטה והיא כוללת צ'יפים, דיסק, מסך מקלדת ועוד אובייקטים פיזיים כאלה ואחרים. מעל החומרה נמצאת התוכנה.
לרוב המחשבים יש שתי דרכי פעולה:

- User mode
- Kernel mode



מערכת ההפעלה היא ההחלק הכי בסיסי של התוכנה והיא רצה ב- kernel mode (נקרא גם supervisor mode). ה- mode הזה מאפשר גישה לכל החומרה ולמעשה יכול להריץ כל פקודה שהמחשב מסוגל לבצע. שאר התוכנה רצה ב-user mode, שבו רק חלק מהפקודות של המחשב (מכונה) נגישות. בפרט כל מה שקשור לשליטה במחשב ול- I/O לא נגיש ב- user mode.
החלוקה בין שני האופנים האלה היא לא חדה וחד משמעית, בין השאר בהתחשב בעובדה שיש מחשבים שיש להם רק user mode. אבל גם במערכות שבהן יש גם וגם, חלק לפעמים משותף (בערך).

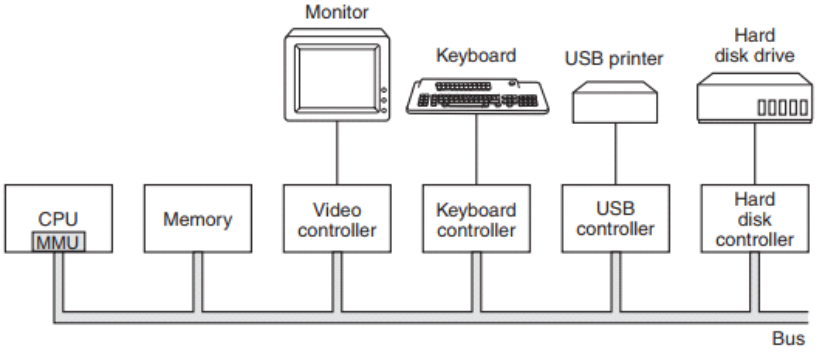
2 פונקציות עיקריות של מערכת ההפעלה:

1. יצירת מערת אבסטרקית מתוך החומרה
מערכת ההפעלה נותנת למתכנת מערכת אבסטרקטית של החומרה, בכדי שהוא לא יצטרך להתעסק ישירות עם חומרה, שהיא מאוד מסובכת לתפעול.
הפשטה היא המפתח לניהול מרוכבות. קובץ, בראוזר, קורא מיילים- כל אלה הם הפשטות שמאפשרות לנו לעשות שימוש פשוט כדי להגיע למטרה (במקום להתעסק עם ציפים פיזיים של חומרה).
אם כך, תפקידה של מערכת ההפעלה היא ליצור אובייקט מופשט יפה, נקי, עקבי וברור מחומרה שהיא ההפך מכל אלה ולנהל את האובייקט שנוצר.
חשוב לציין שהלקוח העיקרי של מערכת ההפעלה הוא התוכנות שרצות (וכמובן המתכנתים שמתכנתים אותן).

2. ניהול משאבים
מחשבים היום מורכבים כאמור ממעבדים, זכרון, דיסקים, עכבר, רשת ועוד דיוויסים רבים ושונים. דרך אחרת להסתכל על מהערכת ההפעלה היא כמי שמנהלת את המשאבים הזמינים בכל רגע נתון לתכניות השונות שעושות שימוש במרכיבים השונים. כשיש מספר משתמשים למחשב או רשת, ניהול המשאבים השונים נהיה עוד יותר קריטי.
חלוקת המשאבים מתבצעת בשתי רמות- זמן וזכרון. כשמדובר בחלוקת משאבים בהקשר של זמן, לרוב מערכת ההפעלה תיצור "תור" של לקוחות שמחכים לשאב מסויים. כשמדובר בחלוקת משאבים של מקום, לא יהיה תור, אלא כל אחד יקבל חתיכת מקום (זכרון או דיסק קשיח).

היסטוריה של מערכות הפעלה- לסכם כשיהיה זמן (כלומר אף פעם)

חומרה
מערכת הפעלה צריכות "להכיר" מאוד טוב את חומרת המחשב, או לפחות איך החומרה נראית בעיני המתכנת. לכן, כדי להבין איך מערכות הפעלה עובדות, צריך להבין ברמה כזו או אחרת את החומרה.
האיור מראה מודל מופשט של מחשב אישי פשוט.



המעבד, הזכרון וה- I/O devices מקושרים על ידי bus ומתקשרים זה עם זה דרכו. במחשבים מודרנים גם יכול להיות כמה באסים.
המעבד הוא ה"מוח" של המחשב. הוא "שולף" פקודות מהזכרון ומוציא אותן לפועל פעם אחר פעם, כמה שנחוץ.

לגשת לזכרון ולקלוט פקודה כלשהי לוקח יותר זמן מאשר להוציא אותה לפועל. לכן בכל מעבד יש רגיסטרים שמטרתם לשמור משתנים ותוצאות זמניות.
בנוסף לכל מעבד יש לרוב כמה רגיסטרים נוספים שויזבלים למתכנת (לא ברור):
Program Counter (PC) - מכיל את הכתובת של הפקודה הבאה שעל המעבד לשלוף. ברגע שהפקודה נשלפה, ה-PC מתעדכן להכיל את ההבאה.
Stack Pointer (SP?) - מצביע על הטופ של ה- memory stack הנוכחי (למלש כשפונקציה רצה).
Program Status Word (PSW) - מכיל סטטוסטים של תהליכים שרצים כרגע. לא לגמרי ברור, אבל משחק תפקיד חשוב ב- system calls (מה שזה לא יהיה...) ו-I/O.

מערכת ההפעלה צריכה להיות מודעת לכל הרגיסטרים. כשהיא עוצרת תהליך מסויים כדי להריץ תהליך אחר, מערכת ההפעלה צריכה לשמור את כל הרגיסטרים הרלוונטים כדי למשוך אותם מחדש כשהתהליך חוזר לרוץ.

בעבר מערכת ההפעלה הייתה עובדת באופן סדרתי- שולפת הוראה, מקודדת ואז מוציאה לפועל - כל פקודה לפי הסדר. מעבדים מתקדמים יכולים לטפל ביותר מפקודה אחת בו זמנית. למשל, מעבדים מסויימים מכילים שלוש יחידות נפרדות לשליפת הוראה, קידוד והוצאה לפועל. בצורה כזו, כשהמעבד מוציא לפועל פקודה אחת, בו זמנית הוא כבר יכול לקודד את הפקודה הבאה ולשלוף את הפקודה שאחריה. ארגון כזה נקרא **pipeline** (החלק השמאלי של האיור).

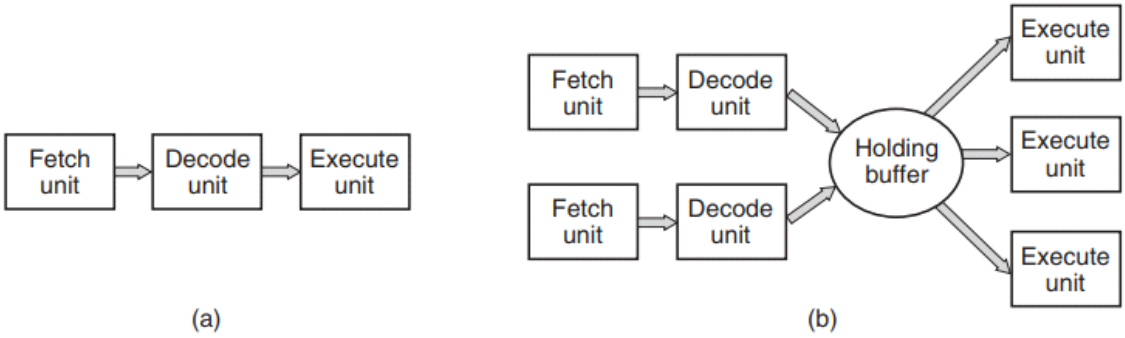


Figure 1-7. (a) A three-stage pipeline. (b) A superscalar CPU.

עיצוב יותר מתקדם מה- pipeline הוא ה- superscalar. בעיצוב הזה נקלטות ומקודדות כמה הוראות במקביל והן נזרקות לבאפר עד שאפשר להוציא אותן לפועל. ברגע שיחידת execution מסיימת עם הוראה מסויימת היא מסתכלת בבאפר לראות אם יש עוד הוראות שצריך לטפל בהן.
ממה שדיברנו בכיתה- פחות משתמשים בעיצב כזה כי הוא דורש שזמן ההוצאה לפועל של ההוראות הנוכחיות שמטופלות במקביל יהיה זהה (זה לא בדיוק מה שרשום בספר..).

כפי שכבר נאמר, ברוב המעבדים (למעט מעבדים מסויימים במערכות embedded) יש מוד של יוזר מוד קרנל. לרוב, ביט ב-PSW שולט ב-mode.
ביוזר מוד האפשרויות הן יותר מוגבלות ולרוב לא ניתן לגשת לפקודות שקשורות להגנה על זכרון או על קלט - פלט. וכמובן שלא ניתן לכייל את הביט ב-PSW.

כדי לקבל שירות ממערכת ההפעלה, תוכנת יוזר צריכה לבצע system call, ש"לוכדת" ל-kernel ומייצרת ארוע שמערכת ההפעלה צריכה להגיב לו.
ההוראה ה- trapped משתנה מ-user mode ל- kernel mode ובכך יכולה להפעיל את מערכת ההפעלה. נרחיב בהמשך על system calls, לעת עתה אפשר לראות בהן פקודות רגילות שיש להן מאפיין אחד נוסף והוא היכולת לשנות מ-user mode ל- kernel mode.
חשוב לציין שיש עוד דברים שיכולים "לגרום לארוע שמערכתה הפעלה תצטרך להגיב לו. בכל מקרה מערכת ההפעלה "תחליט" איך להגיב לארוע.

חסר - לא כולל את כל ההקדמה!!!

23 ליולי

יום ראשון 23 יולי 2017 13:32

ספולינג- אם יש לי מצב שאני פונה לרכיב איטי, אני לא רוצה לתת ל-CPU להתעקב עליו. הרעיון הוא לפנות לרכיב מהיר יותר ואז לאגור את כל מה שהוא רוצה
למשל מדפסת- זה איטי. אז בהתחלה נכתוב לזכרון ורק אחכ לא על חשבון משהו אחר, נזרוק את זה למדפסת

Processes

פרוסס זה פשוט יחידת ריצה שמקבל את כל המשאבים והזכרון של המחשב בזמן שהיא רצה. תהליך כשהוא רץ- כל משאבי המחשב נתונים לו. משאבים הכוונה היא רגיסטרים, מרחב זכרון וכו'.
המשמעשמות של כל המשאבים נתונים להליך, זה אומר שבזמן שאני רץ אף אחד לא יכול להפריע לי.
זה אומר גם שמערכת ההפעלה צריכה להגן על התהלים, כך שבזמן שהוא רץ אף אחד לא יכול להפריע.

בלינוקס- איך פרוס מרחב הזכרון של תהליך?
מרחב הזכרון של כל תהליך שרץ מקבל את מרכב הזכרון מ-0 עד 4G
מרחב הזכרון ב-32 ביט זה 4G. מרחב הזכרון זה FFFFFFFF (הקסה - לחזור לזה). כלומר 2 בחזקת 32 פחות 1.

אם כך, כל מרחב הזכרון נתון לתהליך שרץ.
איך זה מחולק?
מ-0 עד גובה מסויים זה טקסט- כלומר הקוד שכתבנו (בשפת מכונה!!). הגודל של החלק הזה תלוי בתוכנה וכמה היא צריכה והגודל הזה קבוע בכל תכנית וזה יקבע בשלב הלינקינג.
בהמשך נמצא ה- data. מה יושב שם? כל המשתנים הגלובלים, הסטטים וכד', שיש להם ערך התחלתי.
אם עשיתי:
int g_i = 5- זה יושב בדאטא.
הדאטא מחולק ל-2:
read only ו- read/ write
המשתנה למעלה יהיה ב- read write
מה יושב ב-read only?
אם יש סטרינגים שמאתחלים אותם- הם שם.
אם מנסים לשנות את ה-read only, מקבלים segmentation fault.
כשנגיע ל-memory management, נבין איך מסמנים read only

BSS- כל המשתנים שלא אותחלו. זה כולל גם את המשתנים שאיתחלתי ב- 0. הכל מחכתחילה מאופס, ולכן כשאנחנו נותנים למשתנה ערך 0, זה כאילו לא נתנו ערך התחלתי.
עד ה-heap- כל הגודל נקבע בזמן הלינקינג. הלאה משם, זה מרחבי זכרון שנקבעים באופן דינאמי בזמן הריצה

מ-4G, מלמעלה- ג'יגה שלם שייך למערכת ההפעלה.
זה כדי שיהיה לנו קשר למערכת ההפעלה. זה מרחב זכרון שלא מוקצה לתהליך שלנו. אלו שיטחי העבודה של מערכת ההפעלה- באפרים וכאלה.

אם כך, בין 3G לבין ה-BSS, זה השטח שמוקצה לסטאק ולהיפ. זה דינמי, תלוי בצורך של שניהם.
בהיפ- כל ההקצאות הדינמיות. בסטאק- כל פניה לפונקציה.

אם כך, בזמן נתון רק הליך אחד רץ כל פעם. כלומר, דיברנו על כך שיש תהליכים מקבילים. בפועל זה לא באמת מקביל, פשוט מערכת ההפעלה מייעלת את הריצה של כולם ביחד ואנחנו מקבלים את ההרגשה של ריצה כל הזמן.

כל פעם שאנחנו מפסיקים תהליך אנחנו שומרים את כל הסביבה ב-
PCB- process control block
ואז כשחוזרים לרוץ זה לוקח משם.
בצורה כזו, בכל רגע נתון רץ תהליך אחד, אז יכול להיות שהוא נעצר, הכל נשמר, מתחיל תהליך אחר, אז הוא מסתיים או נעצר ויכול להתחיל תהליך אחר או לחזור לתהליך שנעצר קודם לכן.

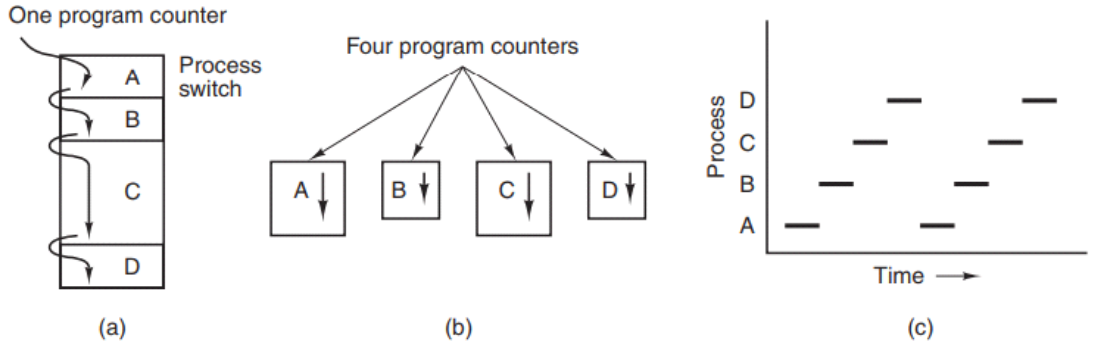


Figure 2-1. (a) Multiprogramming four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

אמרנו שיש קואנטום- סלייס זמן מסויים שבו תהליך רץ. לא מדובר בכמה זמן תהליך רץ אלא זמן החלטה של האם לתת לתהליך לרוץ או לא.
צריך להבין שמדובר בתהליך רנדומלי לחלוטין. אי אפשר לדעת איפה נקטעים (מבחינת איפה אני עומדת בתהליך שרץ), אף פעם אי אפשר לבנות על זה.
כמו כן, אם רוצים לבדוק זמן בין שתי פקודות, אף פעם א אפשר לדעת את זה, בגלל הרנדומליות של הפעלת התהליכים השונה.

יש שלושה מצבים שבהם תהליך להיות בו:
ready- תהליך נמצא במצב זה (זה ממש קיו) כשהוא מוכן לריצה. כישיגע תור התהליך יעלה מה- PCB את הסביבה שלו והוא יעבור למצב של running.
משם זה יכול או לחזור ל- ready (עד הסיבוב הבא) או שהתהליך עצמו פנה ל-I/O, ואז זה יגיע לבלוק ויחכה לאינפוט. ברגע שמגיע האינפוט אפשר לעבור רק ל-ready, אי אפשר לעבור ישירות ל- running

צריך לזכור את שלושת המצבים האלה ואיך אפשר להגיע אליהם.
שורה תחתונה וחשובה: ל- running אפשר להגיע רק מ- ready

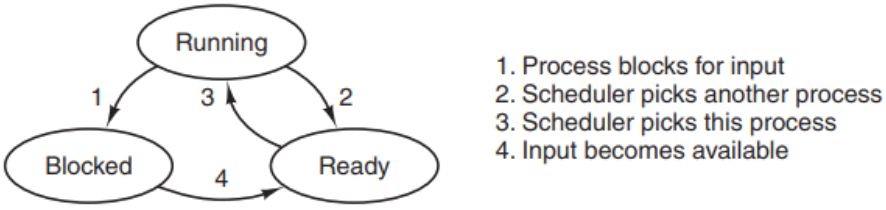


Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

מתי נוצר תהליך?
דבר ראשון, באופן טבעי, כשמפעילים איזשהי פקודה - הרצה כלשהו.
תהליכים מתחילים לרוץ גם כשמערכת ההפעלה מתחילה לרוץ.
אפשר גם להפעיל תכנית בתוך תכנית.

מתי תהליך מפסיק?
מתוך תכנית עצמה ()exit מפסיק תכנית. להבדיל מ-return זה מוציא לגמרי מהתכנית. מתי עושים את זה? אם כקרה אישזהו ארוע ממש לא טוב, צריך לצאת.
אפשרות אחרת - kill

איך מתוך תכנית שלי אני יכולה ליצור תכנית אחרת?
בלינוקס יש פקודה ()fork- מה שקורה בפקוה הזאת: בלנוקס יש היראכיה, כשעושים fork, עושים תכנית בן שהתכנית ממנה יצאתי זו תכנית האב. בהתחלה נוצרת תכנית שהיא זהה לזו שממנה יצאתי- שכפול מוחלט. הכל זהה לחלוטין. וכיוון שגם ה-program counter -PC זהה, הם ימשיכו מאותו מקום. הבן מקבל 0 כ- PID.
()fork זה system call שפונה למערכת ההפעלה
בוינדואס אגב זה עובד אחרת, אין היראכיה בין אב לבן. יש פקודה שנקראת create process ולא מדובר בשכפול, אלא בתהליך שרץ בנפרד.
כשבן מסיים, הוא מחזיר את ערך ההחזרה לאבא, ולכן האבא חייב לבצע wait כדי לקבל את הפקודה מהבן. בהמשך נראה מה קורה כשהוא לא עושה wait.

כאמור, יש היאראכיה בין תהליכים. לכל תהליך יש אבא אחד בלבד. לא יכול להיות תהליך שיש לו יותר.
כל ילד הוא בן של אבא מוגדר. כל בן יכול גם ליצור תהליכים מחדש. אין קשר בין נכד לאבא. לכל תהליך יש process ID.
בוינדוס כאמור אן היראכיה. ברגע שנוצר, אין שום קשר בין ה"אבא" ל"בן". זה משמעותי מבחינת מי יכול לשחרר, לעשות kill וכד'.

מה קורה בשלב ה-init?
כמשמערכת ההפעלה עולה התהליך הראשון שעולה נקרא **init** והוא למעשה מתחיל את כל התהליכים האחרים. וברור שהוא גם האחרון שיורד בסוף.
הבן מחזיר את ערכי ההחזרה לאבא, לכן האבא חייב להמתין לזה. מה קורה אם האבא לא ממתין? נוצר מצב שמערכת ההפעלה לא יכולה לסגור את המרחב הזכרון הזה. זה מצב שנקרא zomby - אם בן סיים והאבא לא קיבל את ערך החזרה. מערכת ההפעלה לא סוגרת אותו עד שהאבא סוגר אותו, או לחילופין אם האבא נסגר.
ברמת המערכת, שהיא האבא של כל התהליכים, היא קולטת את כל ה-"תהליכים"- היא קולטת את כל היתומים.
תהליך במצב זומבי- זה אומר שבן סיים והאבא לא סיים ולא עשה wait. אם האבא סיים, זה ישתחרר לבד.
התהליך עצמו, כשעושים PS, הוא לא נסגר.

24 ליולי

יום שני 24 יולי 2017 10:43

תחילת שיעור
code review - shell

לעשות define לכמות הפרמטרים (שאפשר לשלוח ל-shell אחרי הפקודה) ולהבהיר בדוקומנטציה שמדובר בהגבלה.
להוסיף פונקציה readCommand. לעשות לפי הפסאודו קוד במצגת.

מה עושה execvp?
הוא לא יוצר תהליך חדש (בן חדש), כלומר הוא לא עושה fork.
הוא טוען את התכנית החדשה על בסיס התשתית הנוכחית ומריץ. אם הוא הצליח לטעון, התשתית נדרסה ואז כבר אין משמעות למה שכתוב בהמשך בקוד. אם הוא לא הצליח לעשות fork, זה אומר שלא הייתה דריסה והוא יחזור לקוד וימשיך הלאה!

הערה לא קשורה:
gdb
כמעט לכל הפקודות אפשר להוסיף מספר. למשל c 20 משמע, תעשה cintinue 20 פעמים (למשל לולאה שאני רוצה להגיע לריצה ה-20 שלה).

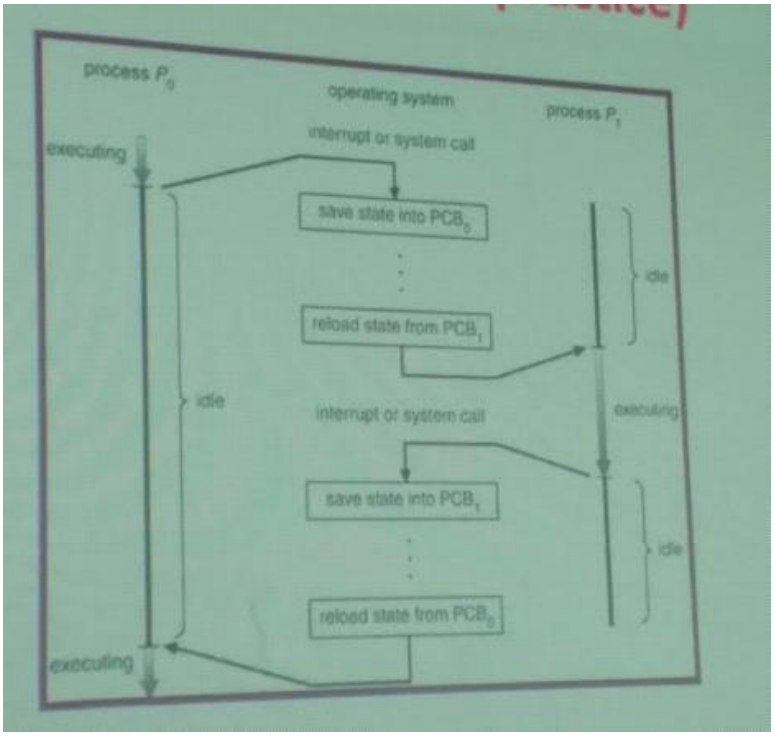
Context Switch

התהליך של הורדת תהליך אחד והעלאה של תהליך אחר.
מה שקורה בתהליך - הסביבה של התהליך הקיים נשמר (כולל כל הרגיסטרים), כשפי שאמרנו, ב-PCB כדי שבפעם הבאה שנעלה את התהליך נחזור לאותו מקום בדיוק, כאילו לא הורדנו והעלנו תהליך.
מעצם יצירת תהליך, נוצר לו PCB - כלומר, גם אם יתחיל לרוץ תהליך שלא נעצר קודם, אלא מההתחלה, פשוט מה שיהיה ב-PCB יהיה כאילו מאופס, אבל עדיין תהיה פניה ל- PCB שלו.

צריך לזכור ש- context switch זו פעולה כבדה. אנחנו לא רוצים לעשות את זה בתדירות גבוהה מדי. 2-3%
שקף- PCB

PCB זו טבלה בזכרון- שבה לכל תהליך מערכת ההפעלה זוכרת בה מה קורה בתהליך מסויים.

נניח שתהליך רץ ואז הוחלט לעשות switch. דבר ראשון עושים שמירה של הסביבה שלו לתוך ה- PCB ועושים reload לתוך ה- PCB של התהליך הבא. בכל הזמן הזה ה-CPU עובד, אבל הוא במעין השהייה (כלומר עובד "בחינם"- הוא רק מחכה. הוא כאילו לא מקדם אף תהליך. זה override של CPU לא יהיה גדול מדי. ביצועים לא מתקדמים.
מתי נקבל את זה? ככל שה-quantum קטן יותר, נבזבז יותר זמן על הסוויצ'ים. לכן זה צריך להיות מכוייל בהתאם לצרכים.



Scheduler

אותו חלק במערכת ההפעלה שהוא זה שמקבל את ההחלטה האם צריך להחליף תהליך ואת מי להעלות באותו רגע נתון. הוא עובד לפי אלגוריתם מסויים (יש כמה אלגוריתמים אפשריים (מתקשר ל-batch, real time וכו- זה קשור). הוא לא הגורם המבצע, הוא הגורם המחליט. כשמגיעים לגמר quantum, כלומר מקבלים אינטראפט של הטיימר, הוא אומר לו תגיע להחלטה.
מגיעים אליו גם כשתהליך מסויים עשה פניית I/O.
יש לו 3 סביבות שונות של עבודה:
batch- יחסית הוא עובד קל. הוא נותן לתהליך לרוץ ולא קוטע תהליך בזמן ריצה. רק כשתהליך עצר (בלוק או סיים), אז הוא נכנס לפעולה.
user interactive- הוא זה שקוטע תהליכים בזמן ריצה כדי שכולם ירגישו שיש תגובתיות טובה
real-time- גם פה, ה-scheduler כן קוטע תהליכים בזמן ריצה כדי לעמוד ב-dead-line. הוא צריך להפעיל תהליך מסויים באופן מחזורי.
העבודה הכי קשה של ה-schuduler היא ב-user interactive- כי שם ממש צריך כל הזמן לעצור תהליך. ב-real time יש פחות קטיעות, צריך לוודא שדברים קורים בקבועי זמן מסויימים.

באמת בשלושת הסביבות האלה יש מנגנונים נוספים שרצים
batch הוא non preemtive- המשמעות היא שאין קטיעה של תהליכים בזמן ריצה (רק בלוק או I/O). לכן יש אינטרולים ארוכים
האינטראקטיב הוא פרימטיב ובו יש אינטרוולים קצרים
הריאל טיים, הוא גם פרימטיב כמובן, אבל יש לו אינטרולים ארוכים. כי בזמן שלא קרה ארוע, לא צריך להחליף שום דבר

הכוונה של קצר או ארוך, זה שצריך לדעת להגיב, אבל בשגרה לא חיבים להחליף תהליכים בתדירות גבוהה (במקרה של real time).

כבר דיברנו איך תהליך יכול לעצור בזמן ריצה- יש איזשהו אינטראפט שבו נותנים למערכת ההפעלה שליטה והיא יכולה להפסיק ריצה.

קריטריונים שה-scheduler צריך לעמוד בהם
באופן כללי לכל סוגי המערכות:

- הוגנות - צריך להענות לכל התהליכים
- התחייבות למדיניות - לכל סוג מערכת יש מדיניות מסויימת. הוא חייב לעמוד במדיניות המתחייבת
- איזון - תפקידו לדאוג שאף אחד לא מתייבש, מצד שני שאין כאלה שרצים כל הזמן

באופן ספציפי לכל סוג מערכת:
Batch
Throughput- צריך למקסם את כמות התהליכים שרצים (בשעה). מצד שני, כשמגיעים לבלוק, צריך להודיע ל-cpu כה שיותר מהר שנגמר בלוק. זוהי למעשה מהות ה-batch, ברגע שתהליך מסתיים (בין אם זה בלוק או קריאה ל- I/O), צריך להגיב הכי מהר שאפשר לתהליך הבא.
Turnaround time- לדאוג לכך שכל התהליכים יסיימו במינימום הזמן האפשרי
CPU utilization- מצד אחד הוא צריך לעבוד כמה שיותר (לא להתבזבז) ומצד שני זה צריך להיות ביעילות

Interactive
רספונסביליות מהירה
פרופורציונליות- אם אני צריכה שיגיב תוך 100 מילישניות, אני לא רוצה שיגיב תוך 10 מילישניות כי זה יגרום להמון overhead (פרופורציונלי לתהליך שץ).

איפה יושב המשנה ששומר את ה-quantum?
מה קורה כששומרים משתנה גלובלי ונותנים לו ערך (5 למשל).
יש פורמט של קובץ out ויש לואדר שיודע לטעון תכנית.
מערכת ההפעלה יושבת במערכת ההפעלה. ה-bios מתניע פעולות ראשונות שמתחילות לטעון את מערכת ההפעלה לזכרון וכל התהליכים שלה מתחילים לעבוד. ה- init זה התהליך הראשון המתחיל לעבוד. בתהליך הזה מתחיל להטעין גם את כל המשתנים שמערכת ההפעלה צריכה (הם read only), בין השאר את ה-quantum או את המשתנה הגלובלי שנתנו לו ערך. כל זה יושב איפשהו בג'יגה של מערכת ההפעלה.

Real-time
צריך להגיב בהתאם לדד-ליין.
צריכה לדעת להגיב לתהליכים מחזוריים. נניח יש לי תהליך אחד שהוא מחזורי. כל 20 מילי הוא חייב לפעול. בין לבין יכול להיות מלא דברים שרצים. אבל כל 20 מילי צריך לעצור ולתת לו לעשות את מה שהוא צריך לעשות. זו גישה שונה מה- response במקרה האינטראקטיבי

בגרף אפשר לראות אינדיקציה לניצול CPU

Figure 2-6 shows the CPU utilization as a function of n, which is called the **degree of multiprogramming**.

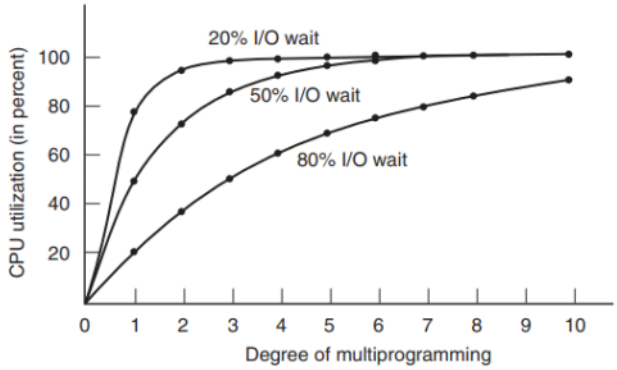
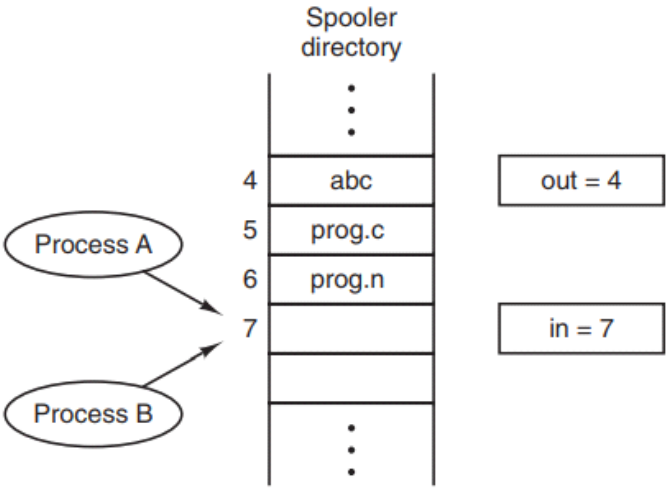


Figure 2-6. CPU utilization as a function of the number of processes in memory.

הערה:
אם יש תהליך שיש לו פניה ל-I/O- כל זמן שמחכים להקשה על המקלדת (או כל דיבייס אחר), התהליך נמצא בבלוק. מה יקרה כשיגיע הקלט מהמקלדת? זה ייצור interapt וה-scheduler ידע שצריך להעלות עכשיו את התהליך שנמצא בבלוק, התהליך יעבור ל-ready וכשיגיע תורו זה יתחיל לרוץ

Preemptive and non-preemptive



יכולה להווצר בעיה כשתהליך מספיק לא כשהוא "תכנן" לעצור. נניח שיש תהליכים שצריכים להדפיס. הם זורקים למקום כלשהו את מה שצריך להדפיס, הספולר רץ ומדספיס.
תהליך A פונה אל הספולר ומבקש מקום פנוי כדי להדפיס. הספולר אומר לו- יש מקום פנוי ב-7. אממה? מערכת ההפעלה עוצרות אותו בדיוק ברגע זה כי משהו אחר צריך לרוץ (כלומר, לפני ש-A באמת עשה שימוש במקום הפנוי). באותו רגע, כל הסביבה של A נשמרת ב-PCB, כולל ה-7. בזמן ש-A מחכה, B פונה לספולר, "שואל" אותו אם יש מקום פנוי, מקבל תשובה- 7 ונכנס לשם לחכות להדפסה. אז A חוזר אחרי בלוק, הסביבה שלו עולה, הוא יודע שיש לו מקום ב- 7, ונכנס לשם. בצורה כזו, ה-B פשוט נעלם ובחיים לא יהיה מוחזר. A דורס אותו.

זה נקרא race condition

צריך להוסיף הגנות כדי להתמודד עם מצבים כאלה

מה קורה ב-scheduler שלא קוטע תהליכים (non - preemptive)?
race condition לא אפשרי.

אבל יכול להיות "הרעבה"- תהליכים שרצים וכל השאר מחכים. ואין מה שיכול למנוע את זה. אם יש תהליך שכל הזמן פונה רק ל-cpu (כלומר אין לו i/o בכלל)- זה אומר שהוא לא רץ.

Dispatcher

החלק המבצע של ה- scheduler
הוא מבצע את ה-context switch. מה שהוא עושה- דיברנו על זה- מוריד תהליך ומחליף תהליך אחר במקומו.

quantum

המשמעות- קבוע הזמן שבו ה- scheduler מבצע החלטה נוספת של מה לעשות הלאה.
חשוב להדגיש- זה לא המקסימום זמן שתהליך יכול לרוץ, כי ה-scheduler יכול להחליט שתהליך ימשיך לרוץ.
ככל שה-quantum קטן יותר התגובתיות גדולה אבל גם ה- overhead גבוה, צריך שיהיה איזון.

הערת אגב חשובה- interapt לא יוצר context switch. כתוצאה ממנו יכול לקרות contextswitch, אבל זה לא הכרחי. לא בהכרח תהיה קטיעה של תהליך. בד"כ כשמגיע אינטאפט חומרה, זה מגיע לא בזמן שהתהליך שממתין לו רץ, כי מן הסתם התהליך הזה בבלוק.

אלגורתמים שבהם עובד ה-scheduler

first come, first served (FCFS)

האלגוריתם הכי פשוט.
בבסיס זה אלגוריתם של batch. מי שמגיע ראשון (סיים או בלוק), הוא זה שרץ.

כשיש רק תהליכי CPU אז ממש רצים לפי התור
אם יש הרבה טסקים שהם I/O bound task- אז יש יותר עצירות ותהליך אחר נכנס במקום. זה יכול להעלות את זמן הריצות של התהליכים האלה.
להבדיל מ-interactive, התגובה תהיה יחסית מהירה, כאן זה יכול לקחת הרבה זמן (כי מחכים עד שתהליכים אחרים עוצרים בעצמם)
מעצם ההגדרה שהאלגורתים הוא non- preemptive, זה אומר שזה batch

Round Robin (RR)

כמו הקודם, אבל אם יכולת preemption
כל פעם מריץ את הבא בתור ומנקה את ה- ready queue. אבל יכול לעצור כשיש צורך לתת לתהליך אחר לרוץ.

Priority Queue

אם כולם באותו לבל של פריוריטי, אפשר להשתתמש במנגנון הקודם (כי אז למעשה אין פריוריטי וזה פשוט תור).
אם לא, ניתן את העדיפיות למי שברמה יותר גבוהה. כל עוד התור של התעדוף הגבוה לא ריק, קח רק משם.
בעיה- יש כאלה שאפשר לא להגיע אליהם אף פעם.
פתרון- dynamic priority
המשמעות היא שעל אף שיש תהליכים עם תעדוף יותר גבוה, עדיין מגיעים גם לאחרים כדי לאפשר גם להם לעבור.
(לא ברור- איך זה קורה ואיך זה שונה מהאפשרות הבאה)

אפשרות אחרת- Multilevel Queue
הכוונה היא לתת לכל פריוריטי יחס חלק יחסי של פריורטי
עושים יחסיות בין תהליכים שונים. לכל פריוריטי מקבלים יחסיות של זמן, כדי לא לייבש אף אחד. השקף זה הפשטה, יש עוד דברים שצריך לקחת בחשבון- למשל איך לחשב את הזמנים אם אין אף אחד בפריוריטי הכי גבוה או בלבלים אחרים (אם אין לי בחלק מסויים מה להריץ, אני צריכה לנצל את זה כדי להריץ תהליכים אחרים).

priority change

בתחילת תהליך אפשר לשנות פריוריטי של תהליך מסוים. זו פקודה שנקרא nice()
זה בגבולות מסויימים.
כל קבוצה של תהליכים נממצאת בתוך קבוצה מסויימת של priority וה-nice יכול לשנות רק בתוך הקבוצה.

Lottery Scheduling

מעין לוטו שכל תהליך מקבל לפי הפריוריטי שלו כמות "כרטיסי הגרלה" וכך מעלה את הסיכויים לרוץ

איך נעשה הנושא של ה- priority (בלינוקס).
מ- 0 עד 139 (0 הכי גבוה).
כל התכניות שלנו רצות איפשהו בין 80 ל-100
נייס מוריד אותנו ל-99 (זו פקודה שבה מוותרים על הפריוריטי).

בוינדואס מ--1 ל-31
-1 - תהליך idle- תהליך שעושה איזשהו קאונטינג. כך הוינדוס מחשב את ה-cpu load.
כאן הפיוריטי הנמוך זה 0.

25 יולי

יום שלישי 25 יולי 2017 11:23

Threads

יחידת ריצה שהיא לא process, אלא יחידת ריצה בתוך process. יכול להיות כמה threads בכל תכנית על מנת ליצור תהליכים מקביליים.
אנלוגיה- זה יכול להיות פונקציה. כשמפעילים thread מגדירים איזה פונקציה להפעיל. תיאורטית, אותה פונקציה יכולה להיות מופעלת על-ידי כמה threads.
נניח רוצים לקרוא מלא רשומות מקובץ. זה אומר שכל פעם קוראים חתיכה, מגיעים לבלוק, מחכים לתור כדי להמשיך וכן הלאה. זה יכול לקחת מלא זמן. אם מריצים את זה ב-threads שונים אז יש כמה בקשות (ממערכת ההפעלה?) במקביל.

כזכור, יחידת ברמת הפרוסס זו יחידת ריצה שיש לה את כל משאבי המחשב זמן שהיא רצה.
ה-tread זו יחידת ריצה אבל יכולים להיות כמה בתוך תהליך אחד וכולם רצים תחת המשאבים של אותו תהליך.
לכל thread יש סטאק משלו.

כל thread מקבל 8 מגה בדיפולט ככל שיש יותר threads, יש פחות מקום בהיפ שאפשר להקצות למשהו אחר. 8 מגה- זה פרמטר קבוע מראש.
בגלל הדיפולט יש הגבלה על כמות הת'רדים שאפשר להריץ. מספר הקסם- אפשר להריץ 382 במקביל. זה בהנחה שלא הקצאנו הרבה מה- heap לפני (כלומר, אם באמת רצים 283 במקביל, ב-heap כמעט לא נשאר כלום).

מה המשמעות שיש לי stack נפרד עבור כל thread? נניח ששניים רצים על אותה פונקציה- כיוון שהסטאקים שלהם שנים, המשתנים הלוקאלים שונים ולא משפיעים אחד על השני.

יושב על כל משאבי הפרוסס שממנו הוא הופעל, פרט לזה שיש לו סטאק משלו.
כלומר, אם הגדרנו איזשהו מבנה בהיפ, כל הthreads יכוליים לגשת אליהם בלי בעיה. גם משתנים גלובלים- זה משאב משותף (כי האזור בזכרון שכולל את הדאטא, טקסט וכו- זה משותף). היתרון הוא שהעברת מידע ב-threads זה יחסית פשוט, בעוד שהעברת מידע בין תהליכים זה הרבה יותר מסובך.
איך באמת מעבירים מידע בין תהליכים? משתמשים במערכת ההפעלה. בת'ראדים אין פניה למערכת הפעלה.

כשעושים create thread זו קריאה למערכת ההפעלה, היא מקצה לו סטאק משלו ושם הוא רץ.
מערכת ההפעלה יכולה להפעיל את ה-quantum ביחס ל-thread, ולא רק ברמת התהליך.

*סטאק פוינטר- טופ של הסטאק שאני נמצאת בנקודת זמן נתונה
השני???

יש ספריה שנקראת pthreads
למשל כשרוצים ליצור:
status = pthread_create (threadID,…FuncToRun, params)

thread מתחיל מפונקציה אחת אבל בתוכה אפשר לקרוא לפונקציות אחרות.

יתרון עצום שיש ל- threads על פרוססים, שאם דיברנו על זה שקונטקס סוויץ בין תהליכים זה די כבד, במעבר בין threads, המעבר הוא פשוט. במקרה כזה צריך לשמור את כל תמונת הרגיסטרים (מזה אין מנוס), את הסטאק (פוינטרים של הסטאק) ואת המצב שלו (קרנל וכו - לא ברור). לעומת המון דברים אחרים בתהליך שקשורים למאשבים שצריך לשמור בתהליכים.

Per-process items	Per-thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Figure 2-12. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

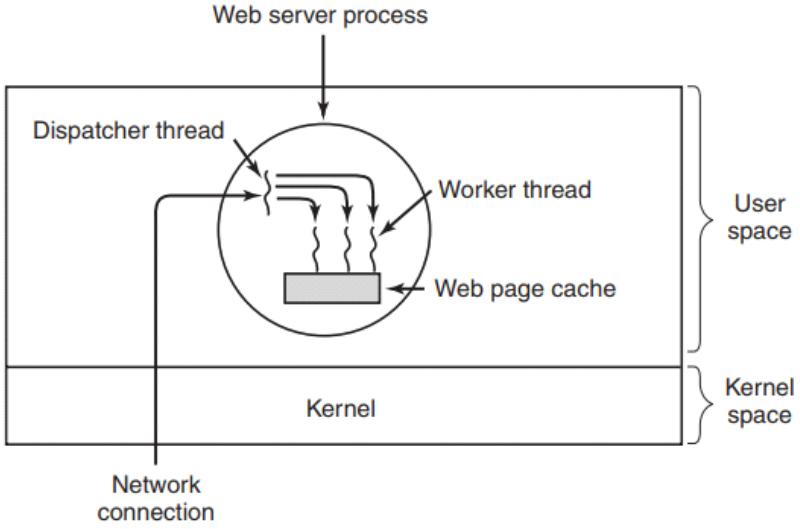
אם כך, למה נרצה להפעיל סראדים במקום תהליים?

1. אפשר לקבל את אותה תחושה של מקביליות בתוך תכנית אחת על אותו מרחב זכרון (אי אפשר בתהליכים שונים).
2. לעשות create thread זה הרבה יותר פשוט מאשר של תהליך (העתקה וכו וכו)
3. Multi- core אפשר ממש ליצור מקביליות בתכנית שלי שרצה- שאחד ירוץ בקור אחד והשני

נניח יש לי תכנית שיכולה לקבל אינפוט מכל מיני מקורות (מקלדת, רשת ועוד). בעקרון יכול להיווצר מצב שבו אני מחכה לאינפוט אחד (מהמקלדת למשל) ואז התכנית תקועה, למרות שהאינפוט שהיא מקבלת מהרשת (למשל) לא קשור ואפשר להמשיך להזרים אותו.
threads נפרדים ממש יוצרים מקביליות ומונעים סיטואציה כזו.

דוגמא- איך עובדת מערכת עם multi- thread
נניח מערכת עם web server
מה הייתי מצפה במערכת כזו? שכל בקשה תקלט ולתהיה תגובה בזמן סביר. נניח שיש רק thread אחד ויש משהו שיש לו בקשה כבדה, כולם עומדים וממתנים.

נגיד יש מנגנון שקולט בקשות ועל כל בקשה הוא פותח thread חדש ואז כל בקשה מקבלת טיפול מיידי (או לפחות בכאילו- כי אם יש קור אחד אז יש חלוקה, אבל התחושה של החלוקה עדיין נשמרת). היתרון הגדול הוא שאם יש בקשה קצרה, היא תקבל מענה מהירה. כלומר, דברים לא נתקעים.



פסדו קוד- לתהליך באיור:

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

Figure 2-9. A rough outline of the code for Fig. 2-8. (a) Dispatcher thread. (b) Worker thread.

Multi - vs. Single Threaded

מולטי:

- מולטי כמובן יוצר מקביליות
- blocking system call - מה זה אומר? כשעובדים על מולטי ו- thread אחד נכנס לבלוק, המשמעות היא שהאחרים יכולים להמשיך לעבוד, להבדיל מ- thread אחד שבו כל התהליך עוצר. דיברנו קודם על תכנית שמחכה לאינפוטים מכמה מקורות, thread אחד יתקע אותי אם אני למשל מחכה לקלט מהמקלדת.
- ביצועים גבוהים
- תכנות קל (אפשר לחלוק על זה, זה יש גם בעיות שנובעות מ-threads, צריך לדעת לסנכרן אותם כמו שצריך)

בסינגל
- אין מקבילויות (סדרתי)
- אם יש בלוק, כל התהליך נכנס לבלוק
- ביצועים נמוכים

פונקציות עיקריות

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Figure 2-14. Some of the Pthreads function calls.

exit- אפשר לסגור בכל שלב. אפשר גם לעשות return לפונקציה בתוך ה-thread ואז זה יצא ואין צורך באקזיט.
אם עושים exit לתהליך, כל הת'ראדים נסגרים.

join - פונקציה שמשמעותה שאני רוצה להמתין שת'ראד מסויים יסיים (קצת מקביל ל- wait בתהליכים). כשעושים join לת'ראד מסויים, נמצאים בבלוק עד שהוא יסיים. פה חייבים לתת את ה-ID שלו.
איפה זה עוד חשוב?
דיברנו מקודם על כך שתהליך, ברגע שהוא מסיים, כל הת'ראדי שלו מתים. הגיוני שכשתהליך מסיים, לפני זה הוא ימתין שכל הת'ראדים שהוא הפעיל יסיימו לפני שהוא יוצא.

yield- פקודה דומה ל-nice. yield אומר שאני מוותר על התור שלי- זה אומר למערכת ההפעלה לתת את התור שלי ולהתחזיר אותי לסוף התור ב-ready queue. אם יש המתנה ארוכה יחסית, נכון להשתמש ב-yeald.
נניח אני מחכה למסר כלשהו מת'ראד מסויים, אני יכולה לדגום בתדר יחסית נמוך ולעשות yield
שתי הפונקציות האחרונות- בד"כ לא נוגעים בהן, זה קשור יותר לקונפיגורציה של הת'ראדים

pthread_create כיוון שזה ממשק גנרי, הפרמטר האחרון הוא (void*) כדי שבפועל אפשר יהיה להעביר כל דבר

התכנית שהרצנו בכיתה:
במצב מסויים, נותרנים לגלובל לספור והוא לא סופר כמו שצריך (וזה אקראי לגמרי)
מה קורה?
זהו תהליך שהוא אסינכרוני (רכשהדפסנו את מספר ה-thread, ראינו כל פעם משהו אחר, לא לי הסדר).
למה הוא לא מגיע למספר שציפינו?
אם אני מקדמת קאונטר גלובלי: ++i
מה הוא עושה?
שלוש פקודות מכונה:
הוא טוען מאישהו רגיסט Ax את i
מגדיל את Ax
שם את מה שיש ב-i ב AX

ה- scheduler יכול לעצור בכל שלב בין 3 הפקודות האלה. שבין שתי הפקודות הרשונות זה נעמר
race condition כמו שראינו קודם

מה היתרונות בלהפעיל את ה-threads ביוזר מוד? כלומר לא במערכת הפעלה?
צריך לזכור שפעם מערכות הפעלה לא תמכו
ראשית, זה יותר מהיר
שנית, קל יותר לשלוט באלגוריתם של ה- scheduling

אב, אם ת'ראד אחד נכנס לבלוק, מבחנת מערכת ההפעלה כל התהליך בבלוק
אי אפשר לנהל schedule ברמה של אינטראפטים כי אין מערכת הפעלה
במילים אחרות, אי אפשר לנהל ביוזר מוד thread preemption

קרנל
blocking system calls- אם אחד נכנס לבלוק, השאר יכולים לרוץ
page fault- גם גורם לתהליך או לת'רד להכנס לבלוק, נלמד בהמשך
חסרונות:
לא כל מערכות הפעלה תומכות

באופן מעשי כל מערכות ההפעלה היום תומכות בזה וזה תמיד עובר דרכיהן
ולכן לרוב הסראדינג יעשה דרך מערכת ההפעלה

בעקרון המנגנון של threads- מאוד נוח לשימוש ברמה שכשהגיעה הודעה כלשהו, מריצים thread וזה מטופל. נקרא pop up.
יש לזה יתרון של עבודה מקבילית. החסרון של תהליך כזה הוא שכמות הת'ראדים מוגבלת וזה יכול לגרום לזה שתגיע הודעה ולא תקבל טיפול כי אין ת'ראד שיטפל בה. אבל יש דרכים להתמודד, בכל רגע נתון אני יכולה לדעת אם הגעתי למגבלה.

אם יש לי פונקציה שיש בה כמה ת'ראדים, אני לא יכולה סתם להריץ אותה בלי לקחת בחשבון. בעיקר אם יש משתנים גלובלי- יכולה להיות התנגשות. צריך לכתוב את הפונקציה בצורה שהפונקציה תעבוד בצורה נכונה ויעילה גם אם כמה ת'ראדים נצמצאים במקביל.
אם כתבנו את כל ההגנות בקוד (מתוכנן מראש לכמה ת'ראדינג)- נקרא reentrant code
קוד שלא נעשו עליו ההגנות- non reentrant code- לא מוגן מפני הפרעות או התנגשויות