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.

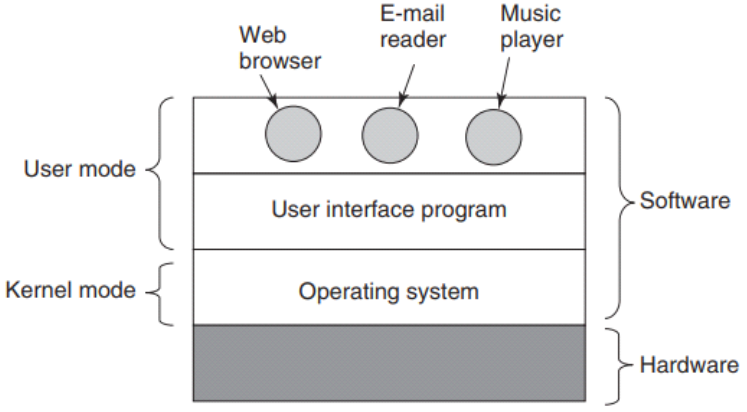
לכל רכיב במחשב יש קונטרלור. במערכת ההפעלה יש דרייבר שיכול לדבר עם הקונטרולר הספציפי של רכיב ספציפי.במדפסת למשל יש רכיב שיודע לדבר מול הדרייבר (לקבל פקודה) והדריביר יודע לדבר עם המדפסת שתבצע פקודה מסויימת (להדפיס A למשל).

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).

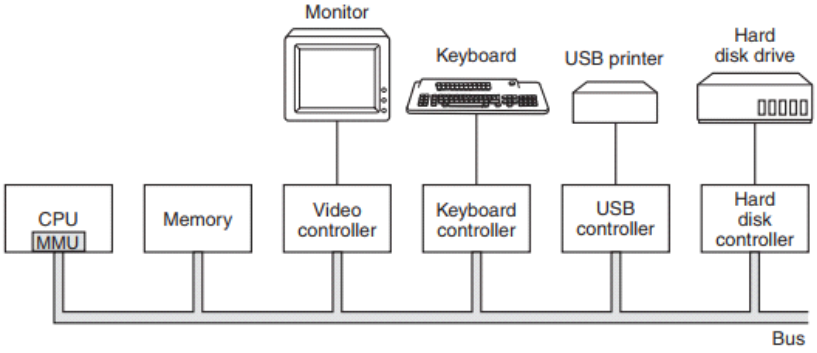
2 פונקציות עיקריות של מערכת ההפעלה:

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).

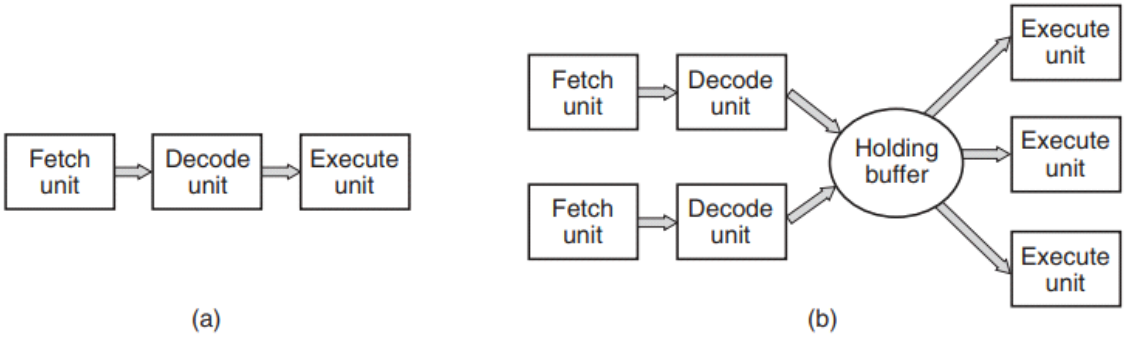


Figure 1-7. (a) A three-stage pipeline. (b) A superscalar CPU.

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.

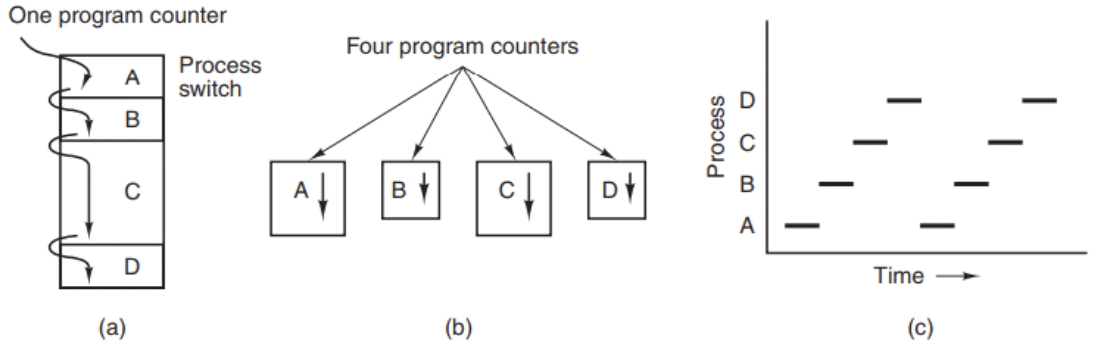


Figure 2-1. (a) Multiprogramming four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

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

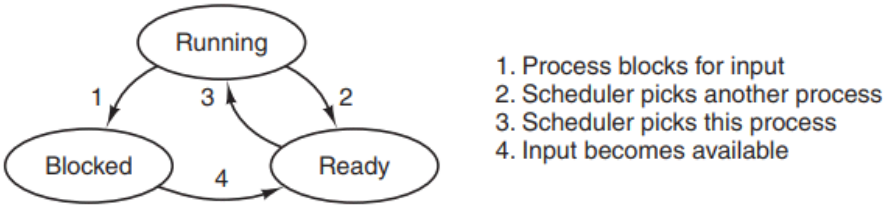


Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

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).

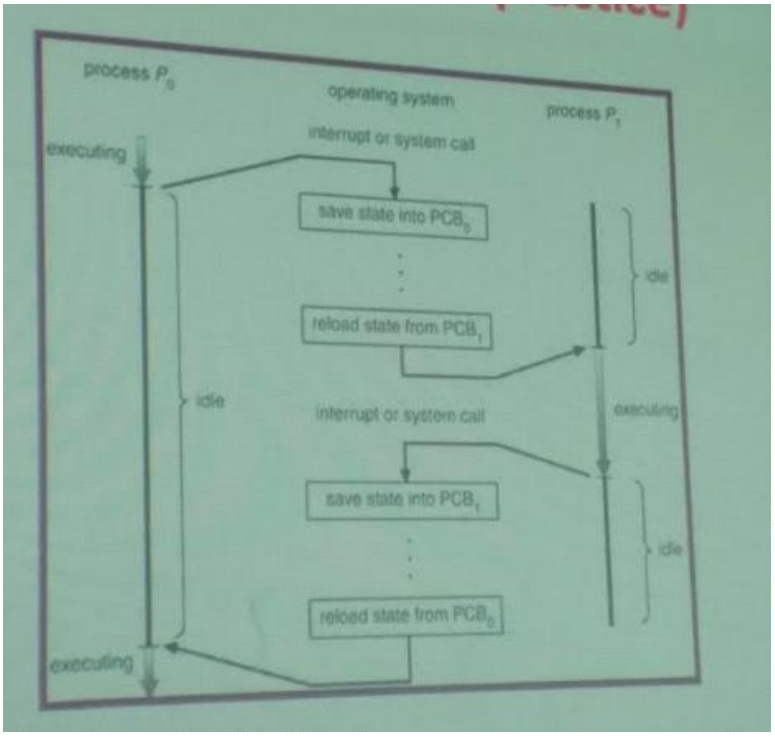
Context Switch

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).

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

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.

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

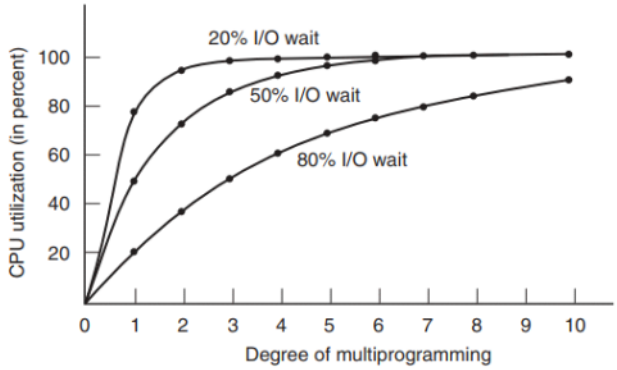
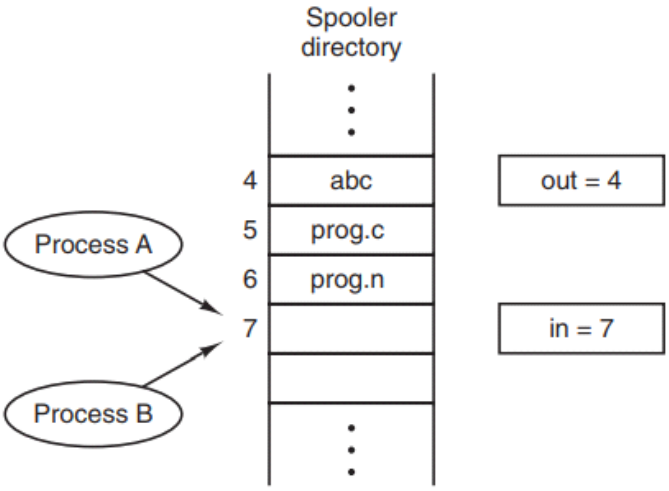


Figure 2-6. CPU utilization as a function of the number of processes in memory.

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.

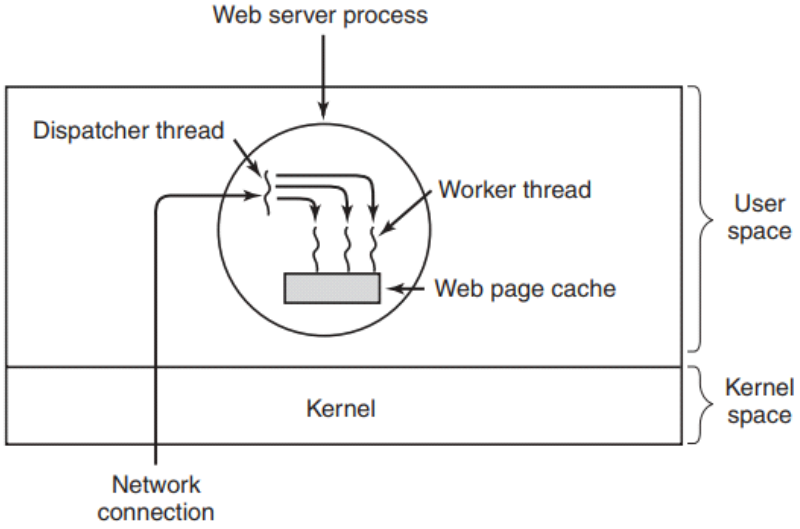
אם כך, למה נרצה להפעיל סראדים במקום תהליים?

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

26 ליולי

יום רביעי 26 יולי 2017 11:38

סיימנו את נושא התהליכים וה-threads.

IPC - Inter Process Communication

ברור שתהליכים הם לא לחלוטין בלתי תלויים זה בזה. IPC הוא שם מטעה ויש למעשה שלושה נושאים עיקריים שנופלים תחת הכותרת הזו:

1. תקשורת בין תהליכים - איך הם מעבירים אינפורמציה זה לזה
2. איך מוודאים שתהליכים לא מתנגשים ולא מפריעים זה לזה (למשל שני תהליכים במערכת בוקינג של חברת תעופה שמנסים שניהם לתפוס את המקום הפנוי היחיד שנשאר)
3. סנכרון בין תהליכים - אם תהליך A מפיק נתונים ותהליך B מדפיס אותם, הם צריכים להיות מסונכרנים (כלומר תהליך A חייב לחכות ש-B יסיים).

אנחנו לא נדבר בשלב הזה על תקשורת בין תהליכים, יהיה לנו קורס נפרד על זה (תקשורת בין threads זה לא בעיה כי הם חולקים אותו מרחב זכרון). באשר לשני הנושאים האחרים- הם רלוונטים ועובדים באותה צורה גם בתהליכים וגם ב-threads. לכן מעכשיו נדבר על threads אבל נשתמש במילה תהליך, כדי שחנה לא תצטרך כל פעם להקליד את המילה באנגלית..

מונחים:

race condition

דיברנו כבר על מה זה race condition- תהליך אחד שיכול להתנגש עם אחר

critical section

אותו קטע קוד שבו חייבים שרק תהליך אחד יהיה, אחרת הם יכולים להפריע אחד לשני

אם כך צריך לזהות את האזורים הקריטים ולדעת להגן עליהם- צריך לוודא שבאזור קריטי נמצא רק תהליך אחד.

איך אנחנו מגנים על האזור הקריטי? אנחנו צריכים למצוא דרך לא לאפשר ליותר מתהליך אחד לגשת לדאטא או כל משאב משותף אחר בנקודת זמן קריטית. במילים אחרות, מה שאנחנו צריכים זה **mutual exclusion.**

כמה תנאים צריכים להתקיים כדי שהפתרון באמת יהיה טוב:

1. רק תהליך אחד נמצא ב- critical section שלו בכל זמן נתון
2. אף פעם אי אפשר להניח הנחות של סטטיסטיקה (מה ההסתברות שיהיו לי כמה תהליכים יחד באותו אזור וכו')
3. עושים בלוק (חסימה) רק כשנכנסים לאזור הקריטי- כלומר לא חוסמים תהליך אחר ללא צורך (דהיינו מחוץ לאזור הקריטי) כדי לא ליצור מצב שבו משהו נתקע ואף פעם לא מגיעים
4. לא יכול להיות שמשהו ממתין לנצח כדי להכנס לאזור הקריטי. המנגנון חייב לאפשר לכל התהליכים לעבוד

ההתנהגות הזו מתוארת באיור. אפשר לראות שתהליך A נכנס לאזור הקריטי בנקודת זמן T1. ב-T2 B מנסה להכנס אבל נחסם, כי כרגע יש תהליך אחר באזור הקירטי ואנחנו מאפשרים רק תהליך אחד. ברגע ש-A מסיים (T3) B נכנס לאזור הקריטי ועוזב כשהוא מסיים ב- T4.

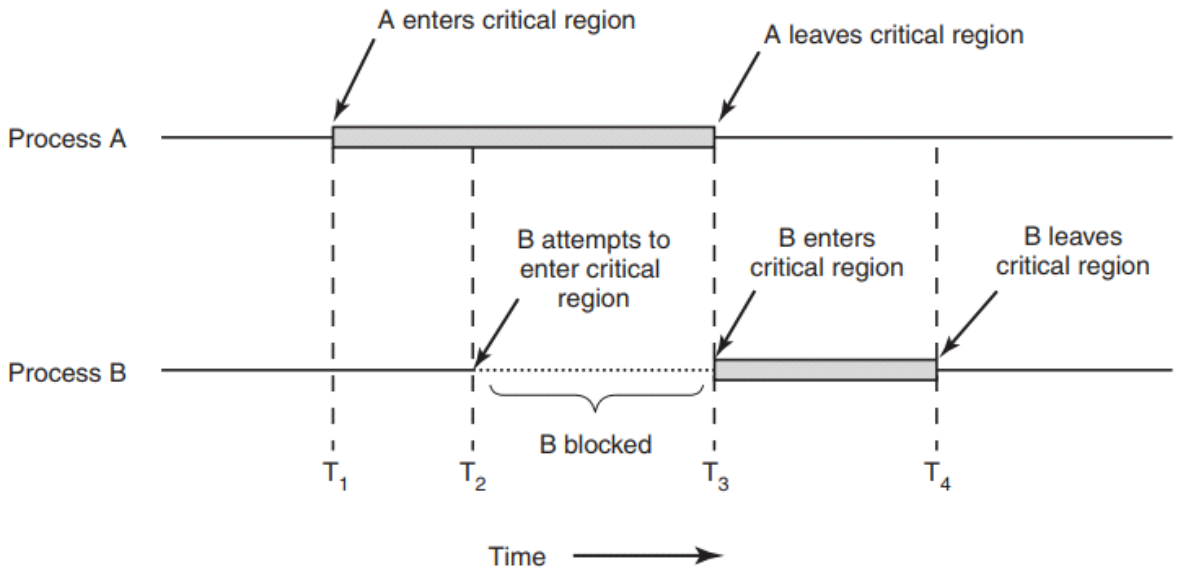


Figure 2-22. Mutual exclusion using critical regions.

ניתן לחשוב על כמה פתרונות אפשריים כדי להשיג את ההתנהגות הזו.

א. לנטרל אינטרפטים

איך scheduler יכול לעצור תהליך? אינטראפט. אם אין אינטראפטים, אף אחד לא יכול לקטוע אותי. זה אומר שאם נכנסים לאזור קריטי, בכניסה עושים disable interapts ואז אף אחד לא יכול ללהכנס.
האם זה פתרון אפשרי?
דבר ראשון ב- user mode אי אפשר לעשות את זה. בין השאר כי זה מאוד מסוכן. אם משהו נתקע בשלב הזה, המחשב מושבת. אגב, מערכת ההפעלה כן עושה את זה במקרים מסויימים (כי היא "יודעת" שזה לא יתקע).
בעיה נוספת- אם המערכת היא multi-core, אז הנטרול רלוונטי רק ל-core אחד ו-core אחר יכול להתערב בתהליך שניסיתי להגן עליו.
בעיה זו נהיית יותר ויותר רלוונטית בעולם שבו רוב המחשבים, גם הפרטיים והכי פשוטים הם כבר multi-core.

ב. Lock variable

נניח אנחנו מחזיקים איזשהו flag גלובלי שמאותחל ב-0. כשתהליך רוצה להכנס לאזור קריטי - הוא בודק קודם כל את flag. אם הוא 0, התהליך משנה אותו ל-1 (לסמן שעכשיו האזור נעול) ואז נכנס. אם ה-flag נעול (שווה ל-1), התהליך מחכה עד שהוא משתנה ל-0.
במילים אחרות 0 אומר שאין אף תהליך באזור הקריטי ו-1 אומר שיש תהליך באזור הקריטי.

מה הבעיה?
אותה בעיה שראינו עם הספולר של המדפסת (איור בעמוד 8). יכול להיות מצב שבו תהליך אחד בודק את ה-flag, רואה שהוא 0, "רוצה" לשנות אותו ל-1 אבל נקטע בדיוק בשלב הזה. לפני שהוא חוזר מהלוק, תהליך אחר רץ ומשנה את ה-flag ל-1. כשהתהליך השני יקבל שוב אישור מה- scheduler לרוץ, הוא פשוט "ישנה" את ה-flag ל-1 ויכנס לאזור הקריטי, וכך מצאנו את עצמנו עם שני תהליכים באזור הקריטי.

ג. Spin Lock

הפתרון הזה, בשונה מהפתרון הקודם מגן ב-100%.
ממה קורה פה (מה רואים בקוד)?

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Figure 2-23. A proposed solution to the critical-region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

(טעות באיור? זה אמור להיות שווה במקום שונה בתוך ה-while?
מה קורה אם 1 אף פעם לא נכנס? זה שונה ל-1 ויכול להשתנות רק אם 1 נכנס ולפני היציאה משנה ל-0

המשתנה turn שמאותחל ב-0 למעשה עוקב אחרי התור של מי עכשיו להכנס לאזור הקריטי. כרגע לענייננו יש שני תהליכים שהמספר הסידורי שלהם זה 0 ו-1.
בשלב הראשון, תהליך 0, רואה ש- turn = 0 וזה מאותת לו שהוא יכול להכנס לאזו הקריטי.
במקביל, תהליך 1 גם רואה ש- turn = 0, ונכנס ללופ שבו הוא כל הזמן בודק מתי זה משתנה ל-1.
בדיקה של משתנה כלשהו באופן מתמשך עד שהוא מקבל ערך מסויים נקראת **busy waiting**. זה מצב שבאופן כללי יש להמנע ממנו כיוון שזה מבזבז זמן CPU (זה כאילו רץ, כשמהותית, התהליך צריך להיות בבלוק כרגע כי הוא מחכה שמשהו יקרה). מאפשרים busy wait כשיודעים שזה לזמן קצר יחסית. מנעול כזה שעושה שימוש ב-busy waiting נקרא spin lock.
כשתהליך 0 עוזב את האזור הקריטי הוא משנה את turn ל-1 כדי לאפשר לתהליך 1 להכנס לאזור הקריטי שלו.
הפתרון זה עובד תמיד כי יוצרים פה מצב של תור- לכל אחד יש את הזמן שלו. אם אגב יש יותר משני תהליכים, אז יהיה לנו 0, 1, 2, 3 וכו'.
ספין לוק זה פתרון לא רצוי כשאחד התהליכים מהיר באופן משמעותי מהשני. אם 0 מהיר, בשלב מסויים הוא יוצא מהאזור הקריטי ונתקע מחוצה לו כי 1 צריך לשנות את הלוק ל-0. אבל יכול להיות ש-1 עדיין עסוק באזור הלא קריטי שלו, כי באופן כללי הוא איטי (מתקשר לשאלה למעלה).
זה מפר את אחד התנאים שדיברנו עליו קודם- שתהליך אחד חוסם תהליך אחר בזמן שהוא נמצא מחוץ לאזור הקריטי.

אם כך לפתרון הזה יש שני חסרונות עיקריים:

1. זה דורש סנכרון בין תהליכים, אם יש תהליך שפונה בקצה מהיר יותר אני בבעיה. במילים אחרות, זה מפר את אחד התנאים שפתרון צריך לעמוד בהם.
2. יש busy wait, כלומר זה מנגנון שטוחן את ה-CPU עם ההמתנות. המשמעות היא שכלפי המערכת, ה-CPU עובד, אבל בפועל הוא לא מקדם תהליכים.

מתי בכל זאת נשתמש בספין לוק?

- אם אין מערכת הפעלה (במכונות קטנות ספציפיות)
- אם האזור הקריטי מאוד קטן- לא שווה להתעסק עם מנגנונים יותר כבדים
- כשזמן ההמתנה קטן יותר מהזמן שלוקח לעשות context swotch

בפועל, זה יקרה רק אם אין מערכת הפעלה. אחרת, אני אשתמש בכלים של מערכת ההפעלה (לזכור, ספין לוק זה לא כלי של מערכת ההפעלה).

נחזור לבעיה של ה-lock variable. למה הפתרון של flag גלובלי לא יעבוד?
יש שני דברים שנעשים בנפרד- הבדיקה וההשמה. אם אפשר היה לעשות את הבדיקה ואת ההשמה בפקודה אחת, הבעיה הייתה נפתרת.
ואכן יש פקודה כזו:
TSL- Test and Set Lock
זו פקודת אסמבלי שהמעבד תומך בה. למה זה ברמת פקודות אמסבלי? כי המעבד דוגם את האינטראפט בין פקודה לפקודה, לכן פקודה אחת לא יכולה להתחך.
ה-CPU שממלא את פקודת ה-TSL חוסם את ה-bus לזכרון ובכך לא מאפשר ל-CPU אחרים לגשת לזכרון עד שהוא מסיים את הפקודה.
חשוב להבין שחסימת ה-bus זה משהו שונה לחלוטין מניטורלי אינטראפטים. קריאת פקודה וחסימת אינטארפטים לא מונעת מ-CPU אחר שנמצא על ה- bus להכנס באמצע.
במילים אחרות, אינטראפט לא יכול להכנס באמצע של פקודת מכונה, רק בין פקודות
לכן רק חסימת ה-bus יכולה לעבוד (ולכן זה כלי של מערכת ההפעלה?)

```
enter_region:
        TSL REGISTER,LOCK          | copy lock to register and set lock to 1
        CMP REGISTER,#0            | was lock zero?
        JNE enter_region           | if it was not zero, lock was set, so loop
        RET                        | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0               | store a 0 in lock
        RET                        | return to caller
```

Figure 2-25. Entering and leaving a critical region using the TSL instruction.

הסבר מפורט על הקוד- עמוד 126, פסקה 5

נניח שנחתכנו אחרי השורה הראשונה. ונגיד ה- lock היה - 0 ואז יתחוך, עדיין ה- lock הוא על 1. אי אפשר להכנס עד שזה לא ישתנה ל-1.
עדיין יוצר מצב של busy wait

מה ההבדל בין זה לבין הספין לוק מעבר לכך שזה עושה שימוש בפקודת מכונה?

Producer-Consumer Problem

פרודוסר - מייצר משהו
קונסומר- צורך
המצב הוא כזה: שני תהליכים חולקים איזשהו באפר משותף. הפרודוסר שם אינפורמציה בבאפר והקונסומר מוציא משם אינפורמציה (אפשר לתאר את אותו המצב עם יותר מפרודוסר אחד ויותר מקונסומר אחד, זה אותו מגנון).
מה קורה כשפרודוסר רוצה לשים משהו נוסף בבאפר, אבל זה מלא? הפתרון הוא שהפרודוסר "ירדם" ויעירו אותו כשהקונסומר מוציא פריט אחד או יותר מהבאפר. בצורה דומה, אם הקונסומר רוצה לקחת פריט אבל הבאפר ריק, הוא ירדם עד שהקונסומר ישים משהו בבאפר.
הבעיה היא שהמצב הזה שוב מביא ל- race condition פוטנציאלי.

כדי לעקוב אחר מספר הפריטים בבאפר, אנחנו צריכים משתנה שסופר, count. אם המספר המקסימלי שהבאפר יכול להכיל הוא N, הפרודוס קודם כל יבדוק האם count == N, אם כן הפרודוסר ילך לישון. אם לא, הוא יוסיף פריט ויעלה את N ב-1.
הקוד של הקונסומר דומה אבל הפוך- הוא קודם בודק האם N == 0. אם כן, הולך לישון. אם לא, הוא יוריד פריט אחד ויוריד את N ב-1.
כל אחד מהתהליך בודק כל הזמן האם צריך להעיר את התהליך השני בהתאם לערך של N.

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
        int item;

        while (TRUE) {                          /* repeat forever */
                item = produce_item( );         /* generate next item */
                if (count == N) sleep( );       /* if buffer is full, go to sleep */
                insert_item(item);              /* put item in buffer */
                count = count + 1;              /* increment count of items in buffer */
                if (count == 1) wakeup(consumer); /* was buffer empty? */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                          /* repeat forever */
                if (count == 0) sleep( );       /* if buffer is empty, got to sleep */
                item = remove_item( );          /* take item out of buffer */
                count = count − 1;              /* decrement count of items in buffer */
                if (count == N − 1) wakeup(producer); /* was buffer full? */
                consume_item(item);             /* print item */
        }
}
```

Figure 2-27. The producer-consumer problem with a fatal race condition.

הבעיה- אפשר להגיע למצב של הרדמות אינסופית של שני התהליכים
הסבר על הבעיה במודל- עמוד 129

27 ליולי

יום חמישי 27 יולי 2017 09:59

אינטראפט לא יוצר context switch
טיימר מעיר scheduler ואם הוא מחליט שצריך לעשות קונטקסט סויץ, אז זה יקרה.
יכול להיות context switch בתוך quantum, רק אם תהליך נכנס לבלוק בתוך quantum
אינטראפט מעביר מבלוק ל-ready

producer consumer
מודל קלאסי של סנכרון בין תהליכים
הראנו שבקלות אפשר להכנס להרדמות אינסופית
וגם יש critical section שצריך לטפל בו
שני רבדים שונים (איור שיעור קודם)

אז מה הכלים של מערכת ההפעלה שמאפשרים להתמודד עם הבעיות האלה?
כלי אחד, שנותן פתרון לשניהם (סנכרון ואזור קריטי)
semaphore
צריך לדעת שזה כלי של מערכת ההפעלה- כלומר, פונים למערכת ההפעלה. היא צריכה לתמוך במנגנון שנקרא semaphore
כל מערכת הפעלה שהיא multi- threads ו-multi processing, היא צריכה לתמוך בכלי כזה. מן הסתם בכל מערכת הפעלה, הפניה לסמפור מעט שונה

איך כותבים תכנית ב-C (למשל) שתוכל לפנות למערכת ההפעלה?
אני בתכנית שלי בונה API שיודע לפנות לסמפור. במימוש של הפונקציה שיצרתי, שם אני יכולה לעשות איזשהו switch case שמגדיר איך לפנות למערכת ההפעלה הספציפית
לינוקס למשל יכולה לעבוד עם כל מיני מעבדים, ויש לה 5% קוד שמותאם למעבד ספציפי. כלומר, מימוש ספציפי.

אז מה זה סמפור?
שאלה לפני- מה אנחנו רוצים ממנו? אנחנו רוצים שהוא יהווה מחסום שבו אני יכולה לשלוט כמה יכולים לעבור בזמן נתון. שנית, חשוב לי שבזמן שאני תקוע על המחסום, אני לא אהיה ב- busy wait, אלא אני אהיה בבלוק. כלומר, אם משהו הגיע למחסום ולא יכול להכנס, שיכנס לבלוק. המנגנון הזה יוציא אותו מהבלוק כשצריך.
כשמקנפגים סמפור, מגדירים כמה יכולים להכנס בזמן נתון. אם מגדירים אחד, זה בדיוק מנגון הגנה על critical section.
אבל, לפעמים אני רוצה לאפשר כניסה של כמה.
אנלוגיה לסמפור- תאי השירותים הציבוריים. נגיד יש שלושה תאים, מקנפגים סמפור ל-3. בכל זמן נתון יכולים להיות שלושה. הגיע הראשון, יכול להכנס, שני ושלישי גם. מגיע הרביעי- ממתין. עד שמשהו יצא. יכול גם להיות שיצאו שניים. כלומר, זה מחסום חכם עם קאונטינג.

כשמאתחלים סמפור ל-1, הוא נקרא בינארי, אזורים קריטיים דורשים סמפור בינארי.
נדבר גם על ההבדלים בין סמפור בינארי וכלי אחר שנראה בהמשך.
להוסיף דוגמת קוד

כשרוצים להכנס- עושים down לקאונטינג ו-up כשרוצים לצאת
מה קורה כשעושים down
אם s>0 (יש פחות משלושה אנשים בשרותים), מקטינים אותו
אחרת מכניסים לבלוק ואתה צריך להמתין.

סמפור בהגדרה הבסיסית המקורית לא חייב לשמור על תור אמיתי- לא בהכרח מי שהגיע ראשון באמת יכנס ראשון, זה תלוי במימוש של הכלי

מה קורה ב-up?
דבר ראשון בודקים אם יש משהו בקיו. אם יש, מוציאים אחד. אחרת, מקדמים את s באחד. אחד יוצא מהשרותים ואף אחד לא רוצה להכנס, שומרים שיש מקום פנוי כדי לדעת שאפשר להכנס להבא כשמשהו יבוא

זה לא שמפעילים תהליך ישר כשמשתחרר מקום- אלא מעבירים תהליך מבלוק ל-ready

נחזור למודל ונראה איך באמצעות סמפור פותרים את הבעיה
פותרים באמצעות שלושה סמפורים

```
#define N 100                          /* number of slots in the buffer */
typedef int semaphore;                 /* semaphores are a special kind of int */
semaphore mutex = 1;                   /* controls access to critical region */
semaphore empty = N;                   /* counts empty buffer slots */
semaphore full = 0;                    /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                     /* TRUE is the constant 1 */
        item = produce_item( );        /* generate something to put in buffer */
        down(&empty);                  /* decrement empty count */
        down(&mutex);                  /* enter critical region */
        insert_item(item);             /* put new item in buffer */
        up(&mutex);                    /* leave critical region */
        up(&full);                     /* increment count of full slots */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                     /* infinite loop */
        down(&full);                   /* decrement full count */
        down(&mutex);                  /* enter critical region */
        item = remove_item( );         /* take item from buffer */
        up(&mutex);                    /* leave critical region */
        up(&empty);                    /* increment count of empty slots */
        consume_item(item);            /* do something with the item */
    }
}
```

Figure 2-28. The producer-consumer problem using semaphores.

אפ mulex קיבל ערך 1, זה אומר שאנחנו בדיוק מדברים על אזור קריטי
יש עוד אחד קוראים לו empty ויש עוד אחד שהוא 0- אם משהו מנסה לעשות down, לא יצליח להכנס. חייבים קודם לעשות up

while True נשאר אותו דבר
דבר ראשון שעושים זה down ל-empty. מה הפרוש? מה כל אחד מחזיק- צריך להבין. ה- empty מכיל את כמות המוצרים שאני יכולה לייצר בלי שאף אחד יקנה, עד שיתמלא המחסן - כל פעם עושים down ואף אחד לא עושה up, כשאני אגיע ל-n, אני אתקע. זו המשמעות של down. הולכים לישון עד שמשהו עושה up. כלומר empty מאפשר סנכרון
אז עושים down ל-nutex, כלומר הגעתי לקטע הקריטי.
אחרי שעשיתי up לקטע הקריטי, אני עושה אפ ל-full, זה אומר שעכשיו יש מוצר שאפשר לקנות אותו (השתנה מ-0 ל-1). אם אמפטי מחזיק לי עוד כמה איברים אפשר להכניס למחסן עד שאתקע, הפול אומר לי כמה איברים יש כרגע במחסן (זה הופכי, עושים דאון לאממפטי ואפ לפול).
דיברנו על המודל- אני הולך לישון עד שיעירו אותי. ומי מעיר? האפ של האמפטי. אותו דבר הפוך- הצרכן- אם המחסן ריק, הוא הולך לישון עד שמעירים אותו. מי מעיר אותו האפ של הפול. לא לגמרי ברור.
הקונסומר עושה פעולה הפוכה.
עושה דאון לפול. מה המשמעות? אם הוא 0 הולך לישון. חוצה את האזור הקריטי ואז עושה up לאמפטי, כלומר מעיר את הפרודוסר.
כלומר, בעזרת שני פרודוסרים עשיתי סנכרון בין שניהם וגם הגנתי על האזור הקריטי.
תיאורטית יכול להיות כמות יותר גדולה של פרודוסרים וקונסומרים, וזה עדיין יעבוד ולא יתקע. יכול להיות שמשהו ימתין, אבל לא יווצר מצב ששניהם נרדמים לנצח..

חשוב לזכור- זו פונקציה של מערכת ההפעלה. לא אנחנו צריכים לממש את זה.
בלינוקס צריך לדעת מה מקביל לאפ ודאון בפניה למערכת ההפעלה

30 ליולי

יום ראשון 30 יולי 2017 11:17

בעזרת הסמפור אנחנו מסנכרנים גם את הקצוות בין הפרודוסר לקונסומר וגם מגנים על האזור הקריטי.

יש לנו כלי נוסף שנקרא Mutex
נדגיש- זהו כלי של מערכת ההפעלה
כל התפקיד שלו זה מנעול- lock, unlick
אי אפשר לעשות סינכרון
דומה לסמפור הבינארי
כלי הרבה יותר פשוט ולכן כשלא נצטרך סינכרון, עדיף שנשתמש בו

גם מוטקס- מי שעושה לוק וזה תפוס ויכנס לבלוק. גם פה אין busy wait
בהגדרה הבסיסית של מוטקס, רק מי שייצר את הלוק יכול לעשות אנלוק. ראינו במימוש של הסמפור שמשהו עושה up ומשהו אחר עושה down, על פניו אי אפשר לעשות את זה פה. אבל חייבים לדעת, שבמציאות שלנו, לפחות בלינוקס ובעוד כמה מערכות הפעלה, המגבלה הזו הוסרה- כלומר אפשר לעשות ב- thread אחד לוק ובשני אנלוק וזה עובד.
נלמד יותר מאוחר מה זה dead lock- שם צריך לשנות את עניין ההגדרה הבסיסית הזאת.

mutex זה כלי נפרד, זה לא סמפור

הבדל משמעותי בין מוטקס לסמפור בינארי: במוטקס אפשר לעשות כמה לוקים שאני רוצה ושחרור אחד משחרר את כולם. בסמפור אני יכולה לעשות לוק רקורסיבי- כלומר לוק ועוד פעם לוק ורק אחרי שתי פתיחות יצליחו לפתוח את זה. מה קורה כשאני עושה בסמפור wait, ואז עושים עוד פעם wait (semwait)
במוטקס יש רק lock ו- unlock. לחשוב על דוגמא של מתי זה שימושי הדאבל לוק

מה קורה כשמחליפים בין down(&empty) ל- down(nutex) בקוד מהשיעור הקודם? - זה נקרא dead lock, נדבר על זה בהמשך.

דוגמא, איך אפשר לממש מוטקס ל"עניים". אם אין מערכת הפעלה. זה עדיין לא בדיוק בלוק

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

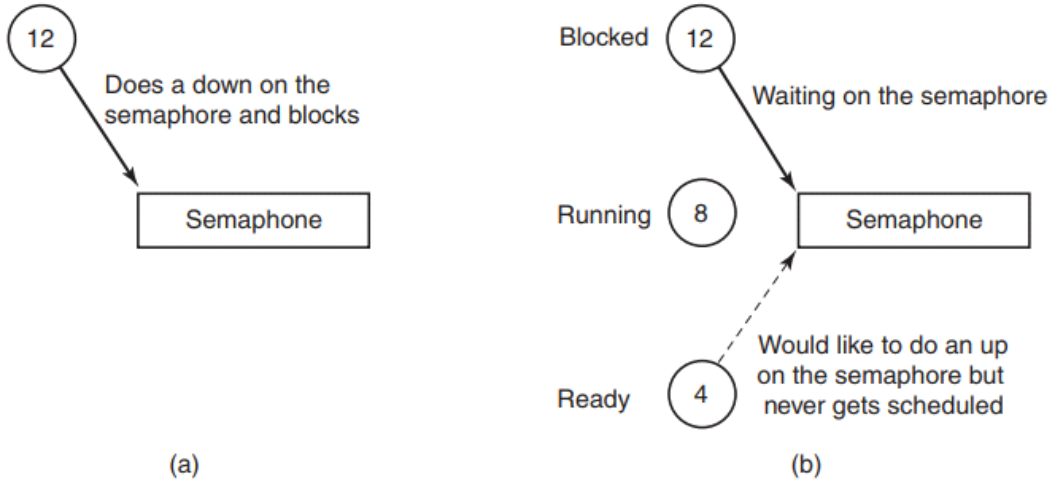
Figure 2-29. Implementation of *mutex_lock* and *mutex_unlock*.

עושים busy wait ב"תדר נמוך"- כל פעם בודקים ויוצאים. זה לא בלולאה אינסופית, אלא מאפשרים לאחרים לעבוד ודוגמים כל כמה זמן- כשמגיע תורי. האנלוק לא שונה ממה שראינו, השוני הוא בחלק הראשון.
זה גם מדגים עד כמה המוטקס הוא מנגנון פשוט, להבדיל מהסמפור שיותר מורכב.

אחת הבעיות של priority inversion

נגיד אני הליך עם תעדוף גבוה, אבל הגעתי לאזור קריטי ועשיתי לוק.
להוסיף שקף
מי שתפס את האזור הקריטי הוא בפריוריטי הרבה יותר נמוך. לכאורה אין בעיה, אם זה רק שנינו, אני אחכה ובסוף הוא יצא ואני אכנס
אבל מה יכול לקרות?
יכול להיות שיש עוד תהליכים שנמצאים בפריוריטי בינינו והם תופסים את המעבד וההוא עם הפריוריטי הנמוך שהכניס ללוק, פשוט תקוע- הוא נמצא בבלוק בזמן שהוא באזור הקריטי
לכאורה זה נמוריד אותי לפריוריטי של הנמוך

פתרון: כשמערכת ההפעלה מזהה שמשהו בפריוריטי נמוך תוקע משהו גבוה, היא יכולה לשנות את הפריוריטי של הנמוך לזה של הגבוה, עד שהוא יוצא מהאזור הקריטי



עמוד 927 בספר!!

אם כך הבעיה היא שמשהו עם פריוריטי נמוך תפס אותנו ועצר אותנו- כאילו הורידו את הפריוריטי של הגבוה
פתרון: נותנים לו זמנים פריוריטי כמו שלנו כדי שיצא כמה שיותר מהר מהאזור הקריטי

כאן סימנו את IPC, אבל יהיה לנו קורס שלם על זה

dead lock

מונחים בסיסיים:

משאב- אנחנו צריכים לתפוס אותו כדי לעבוד איתו. זה יכול להיות זכרון, מדפסת, דיסק, יכול להיות רשומה בדאטא בייס ולמעשה גם אזור קריטי. מאשב כלשהו שכדי לעבוד איתו אנחנו צריכים לקבל רשות, לא חופשי

יש שני סוגים של משאבים:

preemptable resource
משאב שיודע לנהל את עצמו. אני לא צריכה לתפוס את המשאב כדי לעבוד איתו. דוגמא: זכרון. נגיד שתפסתי זכרון- פניתי לאיזשהו משתנה. ונגיד שמהו אחר גם רוצה לפנות אליו. נגיד גם אזור קריטי. נלקח ממני המעבד, ואז מה קורה? הוא גם נתקע. מה יקרה אח"כ? אני אקבל שוב את המעבד וזה ימשיך לעבוד. אין מצב שזה יתקע

non- preemptable resource
למשל מדפסת. היא לא תשתחרר מעצמה. אם אני לא אעשה את ההגנה המתאימה, יצא זבל כי כמה יפנו אליה במקביל. במקרה הזה אני חייבת לבצע הגנה שרק אחד יכול לעבוד עם אותו משאב בזמן נתון, אחרת יהיה בלגן.

המשאבים שיודעים לנהל את עצמם. אם לא עשיתי הגנה שרק אחד יוכל להכנס, זה עדיין יעבוד כמו שצריך. זה לא משהו שאני חייבת להגן עליו ולתפוס אותו

הזכרון שונה בכך שהוא מנהל את עצמו. קונטרולר של דיסק למשל- הוא יכול כל פעם לטפל בתהליך אחד

לחילופין מדפסת- היא צריכה להיות זמינה לתהליך אחד ויחיד

בגלל שהדיבייסים השונים פונים ישירות לזכרון ואין קונטרולר, אפשר לגשת לחלקים שונים שלו.

יש משאבים שמחייבים תפיסה פיזית לפני שפונים אליהם ורק כשמחררים אחרים יכולים לגשת

אז מה זה דד לוק?
נניח יש לי משאב אחד ויחיד
נוצר מצב שכל אחד ממתין למשהו אחר ותקועים לנצח

אם יש לי משאב אחד, אפילו שיש לי 100 תהליכים, אין מצב לדד לוק. כי בכל סיטואציה שהיא מי שתפס בבוא הזמן מקבל מעבד ואז יחשרר. לא יכול להיות שהוא ממתין למשהו שאחר תפס לו ואז ממתינים לנצח. חשוב מאוד מאוד!!!!

ברגע שבזמן נתון אני תופס רק מאשב אחד בלבד, אין בעיה

זה תמיד הדדי (או שרשרת). דד לוק זה כשתהליך א תקוע כי הוא מחכה שתהליך ב ישחרר איזשהו משאב ו-ב תקוע כי הוא מחכה ש- א ישחרר משאב, אז זה תקוע פור אבר

אם כך הדרך הכי נכונה לטפל- לתכנן מערכת שבה בכל בזמם נון אני לא תופסת יותר ממשאב אחד, אני מבטיחה שלא יהיה דד לוק.
ראינו את זה קודם- לחזור לדוגמא

עוד אפשרות שבה כנראה לא יהיה אף פעם דד לוק, זה אם אני משחררת הפוך לתפיסה:

```
typedef int semaphore;
semaphore resource_1;

void process_A(void) {
        down(&resource_1);
        use_resource_1( );
        up(&resource_1);
}
```

(a)

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
}
```

(b)

לא יכולה להווצר הצלבה.
נניח שנכנסתי ל-1- זה אומר שלא יכול להיות שמשהו יהיה ב-2. כי אם הוא ב-2 הוא עוד לא שחרר את 1 ולא הייתי נכנסת לשם.

מצב הפוך- מימין:

```
typedef int semaphore;
    semaphore resource_1;
    semaphore resource_2;

    void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }

    void process_B(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }
```

(a)

```
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_2);
    down(&resource_1);
    use_both_resources( );
    up(&resource_1);
    up(&resource_2);
}
```

(b)

כשיש מספר רב של משאבים, לטפל בזה שלעולם לא יהיה דד לוק, צריך לשלם איפשהו. במקרה הזה זמן ההמתנה שלי לאזור הקריטי יהיה ארוך יותר. הביצועים נהיים פחות טובים.

להוסיף שקף- deadlock example

מהם ארבעת התנאים שרק אם כולם מתקיימים, יהיה דד לוק?

1. mutual exclusion- אם אין לי אזור שחייבם להגן עליו, אין לי בעיה
2. hold and wait- אם יש לי סטואציה שבה אני תופסת משאב ואחרי שתפסתי משאב אני יכול הלהכנס להמתנה למשאב אחר, אז יש בעיה. אם בכל זמן נתון אני תופסת רק משאב אחר, אין בעיה, אף פעם לא יהיה דד לוק
3. no preemption- כשיש משאב שאני חייבת לתפוס אותו ולהחזיק אותו עד שאני מסיימת, אז כנראה לא יהיה לי דד לוק. אני חייבת שיהיה לי משאב שאני צריכה לתפוס אותו כל הזמן עד שאני מסיימת. אם יש לי משאב שמנהל את עצמו ואני לא תופסת אותו ומחזיקה אותו בזמן שאני עובדת.
4. curcular wait- המשמעות היא שיש לי מסלול - יכול להיות בין 2 , 3 ארבעה וכו'- שיוצרים מעגל של המתנות

רק אם ארבעת התנאים יחד מתקיימים, יש לי סיכוי לדד לוק.

31 ליולי

יום שני 31 יולי 2017 13:09

הגדרה רקורסיבית של גובה- מגיעים ל-NULL זה -1, האחרון מחזיר -1 להבא, ואז מוסיפים עד שמגיעים למעלה.

שארית מהשיעור של אתמול- המשך לדד לוק
נחזור על ארבעת התנאים שרק אם כולם מתקיימים יש סיכוי שיהיה דד לוק (לא בהכרח יהיה).
כלומר, אם הצלחתנו למנוע את אחד מהם, אנחנו יכולים לדעת בודאות שאין לנו דד לוק.
נדבר הרבה על דברים תאורטיים, אבל הכי חשוב להבין את הפרקטיקה- איך לוודא שהמערכת לא נכנסת לדד לוק.

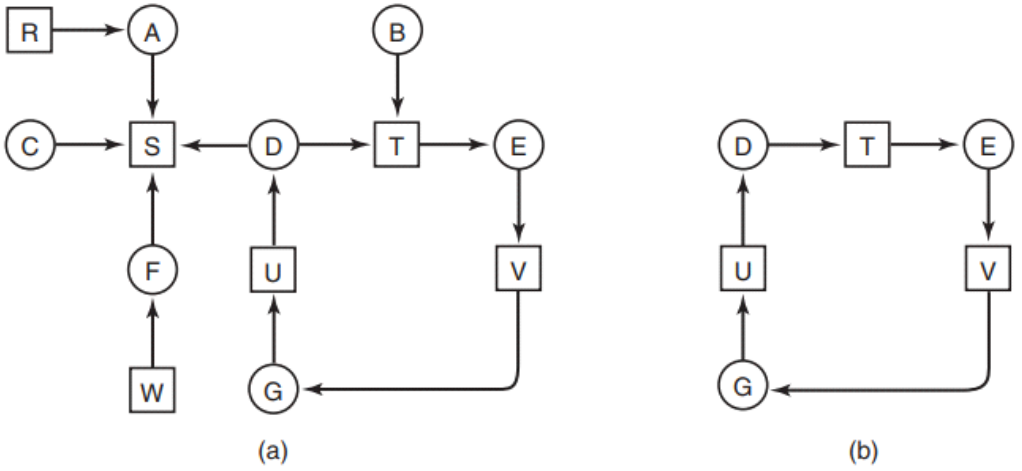
1. יש אזור שמחייב הגנה שרק אחד יכול להכנס
2. החזקה של יותר ממשאב אחד בזמן נתון. כלל מאוד חשוב- אחד האמצעים הכי נוחים שלא יהיה לנו דד לוק- לוודא שכל תהליך מחזיק רק משאב אחד.
3. יש לי משאב שאני חייב להחזיק אותו כל עוד אני עובד עליו. אם אין לי סיטואציה כזו, אז אין בעיה, בכל מצב אפשר לשחרר את המשאב
4. המתנה מעגלית בין 2 או יותר. לחזור לזה

מה האסטרטגיה שלנו בנושא של דד לוק?
אופציה אחת- להתעלם.. לא מומלץ, אבל אפשרי. אם ההסתברות שזה יקרה מאוד נמוכה והנזק לעשות ריסט למערכת לא כזה כבד. מצד שני אנחנו יודעים שכל יומיים אנחנו עושים החלפת גרסה, אנחנו יודעים שהסיכוי כל כך נמוך שהמחיר לטפל בזה גבוה מדי. בפועל, לא קורה..

אפשרות שניה- לזהות שהגעתי לדד לוק ולעשות recovery. כלומר, אני לא עושה מניעה, אני יודעת להתמודד עם זה שזה קורה

האפשרות הכי מומלצת- למנוע הגעה לדד לוק. זה הכי חשוב

כאן מה שמעניין אותי זה המעגל של תפיסת משאבים:



כלומר, כשמטפלים במניעה, צריך לדעת במה להתמקד- בסירקולציה

מתי נעשה detection של דד לוק?
או מתי יכולה להיות סיטואציה של דד לוק? כשאני מבקש משאב. כלומר, צריך להתמקד במקומות שבהם יש בקשת משאב.
אפשרות אחרת, ניתן למערכת לרוץ, כל כמה זמן אני בודקת (מערכת ההפעלה) האם יש דד לוק- עושים פעולה אחורה לסירקולציה ולראות שיש אחד ממתין לשני והפוך.

אפשרות נוספת- מה קורה כשיש לי דד לוק? אמורים לראות צניחה ב-cpu לואד, כי יש כמה תהליכים בבלוק.
רלוונטי במערכת יעודית שבה כשחלק מהתר'אים נכנסים לבלוק, אני אראה ירידה ב-CPU.

טיפול בדד לוק
אם זיהיתי שיש דד לק, מה עושים?
איך עושים רקברי?
יש כמה אפשרויות, וכולן צריכות להעשות בזהירות
הבעיה היא שאחד ממתין לשני וכן הלאה וכן הלאה וכולם תקועים. אם בצורה חכמה אפשר לדעת איזה אחד ישחרר את ה"פקק". אם כך המערכת יכולה באופן יזום לשחרר את אחד המשאבים. אפשר לשחרר בצורה חכמה- כלומר לשחרר את זה שתפס את המשאב- להביא אותו למצב שהוא גם ממתין למשאב ואז זה משחרר. צריך לזכור, שיהיה נזק כלשהי, אבל ההנחה היא, שאם עושים את זה בצורה חכמה, הנזק יהיה יחסית קטן

אפשרות אחרת- שמערכת ההפעלה תיצור לעצמה נקודות דגימה שבה היא עושה מעין מיפוי ומאפשרת לעצמה במצב של דד לוק להחזיר את הגלגל אחורה.

אפשרות אחרונה- הרסנית, פשוט להרוג הליך שתוקע
האפשרות הראשונה- לא טריויאלית למימוש

בעיני איציק, מה שצריך לעשות זה מניעה!!!
לפעמים מתקיימים ארבעת התנאים אבל אפשר להוכיח שברמת הדיזיין לעולם לא מגיעים לדד לוק. אם קשה להוכיח, צריך לטפל במניעה.
אם כך צריך שאחד התנאים לא יתקיים. נעבור אחד אחד ונראה איך אפשר למנוע אותו. אגב לא תמיד אפשר למנוע. ואז או שאני צריכה להוכיח שאין דד לוק או שאני ממש אקווה שהדיזיין שלי הוא כזה שבאמת אין אופציה להכנס לדד ליין. או שאני אצטרך להתמודד עם recovery.

אם כך, נעבור על התנאים

1. יש מצב שצריך להגן על אזור קריטי. צריך להבין שזה לא בהכרח נכון. נניח שאני רוצה להגן על קובץ כמשאב. אבל אם כל התהליכים רק קוראים ולא כותבים, האם באמת צריך להגן עליו? לפעמים אותומטית אנחנו מנסים להגן, אבל כשבודקים, מבינים שזה לא הכרחי ואפשר לשתף משאב בין תהליכים.
אפשר גם להגדיר שעושים assign למשאב רק ממש כשזה הכרחי. אבל בפועל צריך באמת למנוע

2. אם אני תופס יותר ממשאב אחד בו זמנית. איך מונעים? אפשר למנוע את זה בדרך יחסית בזבזנית אבל עובד. לפעמים אני תופסת משאב ואז תוך כדי תהליך אני תופס משאב אחר. ואז אומרים- אתה תכנס רק כשכל המשאבים פנויים. מנסים לתפוס את כולם בהדרגה (ראינו את זה אתמול) לא סידרתי. אפשרות נוספת- הכי מומלצת- לא להגיע למצב שאני תופסת שני משאבים- כלומר, לפני שאני תופסת עוד משאב לשחרר את המשאב הקודם. צריך לדעת שלא תמיד זה אפשרי. זה כנראה מה שהייתי מנסה לעשות מבין ארבעת התנאים.

3. אם אני יכולה לגרום לכך שתהליך אחר יוכל לשחרר משאב שאני תפסתי, לכאורה זה יכול לשחרר את דד לוק. ככה, אני אמנם תקועה, אבל משהו אחר יכול לשחרר כלומר, לא ליצור מצב שאני תקואעה ואף אחד לא יכול לשחרר.

4. איך מונעים סירקולציה?
ממספרים את כל התהליכים. מגדירים שפרוסס יכול לבקש איזשהו משאב רק אם כל אלה שמתחתיו פנויים. מה המשמעות? יוצרים איזשהי סדרתיות (אגב, זה מגביל מאוד). בצורה כזו מנטרלים מעגליות. לצייר ולראות שאף פעם לא יהיה מעגל. לא ברור.

כמעט כל אחת מהאופציות האלה יש לה מחיר

הנושא של דד לוק עולה כמעט בכל דיזיין מערכתי וחייבים לטפל בזה ולהראות שאין דד לוק. לרוב מה שפותר את הבעיה- מראים שאין תפיסה של יותר ממשאב אחד בו זמנית.
זה מסבך את הדיזיין, אבל לא בהכרח מאט.

דאטא בייס- מטבע הדברים אנחנו רוצים לתפוס רשומה ספציפית. הרבה פעמים אני צריכה לעבור על כמה רשומות. יכול להווצר מצב של דד לוק- אני תפסתי כמה רשומות ואז אחד ממתין לשני ולא משתחררים. מה שעושים בד"כ- תהליך לפני שהוא מתחיל לעבוד תופס את כל הרשומות שהוא צריך ותוך כדי עבודה, כשהוא מסיים עם רשומה ספציפית, משחררים.

ברמת התקשורת, יכול להווצר מצב של דד לוק מוזר. תקשורת בן מחשבים- לא במובן הקלאסי אלא נגיד ששלחתי אישזהו חבילה והיא הלכה לאיבוד בדרך (נלמד איך זה יכול לקרות), ואז נוצר מצב שאני ממתינה לתשובה. אבל מי שהיה אמור לקבל את הבקשה ממתין לבקשה ושנינו נחכה לנצח. איך פותרים- יש טיים אווט- אם לא קיבלתי תשובה, שולחים שוב.

עוד דבר שחשוב לדעת- מושג שנקרא livelock- מנגנון שדומה ל-busy wait. אני בלולואה בהמתנה למשהו, אבל זה לא בלוק- מבחנת מערכת ההפעלה אני רצה. מבזבזים CPU וחוסמים למרות שלא קורה כלום- לולאה אינסופית
לא לגמרי ברור, לא באמת קשור לדד לוק

אם ממתינים לאזור קריטי בלולאה, בלי כלים כמו mutex וכד- אז אני רצה בלולאה מבחינת מערכת ההפעלה למרות שאני אמורה להיות בבלוק במהות