

מערכות הפעלה- תוכן

שבת 12 אוגוסט 2017 11:17

עדכון גרסה:
התווסף קצת להקדמה
הושלם הפרק על זכרון
I/O - סיכום שיעור עם יגאל - ערוך (ללא סיכום מהספר)
חסר:
system calls
file system

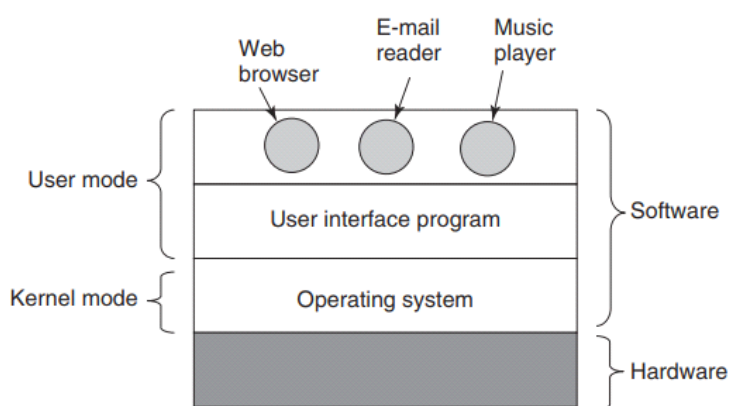
עמוד	נושא \ מושגים
2	מבוא
6	תהליכים
9	context switch
10	scheduler
11	race condition
14	Threads - Tannenbaum
16	Threads - class
20	IPC
23	producer - consumer
25	סמפורים
29	priority inversion
29	dead lock
33	ניהול זכרון- טנבאום
34	swapping
36	virtual memory
39	TLB
42	I/O סיכום שיעור

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, linux וכד' אבל למען האמת, התוכנה שדרכה אנחנו מפעילים את המחשב היא לא מערכת ההפעלה עצמה. התוכנה 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 דיבייס במחשב יש קונטרולר. במערכת ההפעלה יש דרייבר שיכול לדבר עם הקונטרולר הספציפי של דיבייס ספציפי. במדפסת למשל יש רכיב שיועד לדבר מול הדרייבר (לקבל פקודה) והדרייבר יודע לדבר עם המדפסת שתבצע פקודה מסויימת (להדפיס תו מסויים למשל).

מערכת ההפעלה מאפשרת לנו לתקשר עם הרכיבים השונים של המחשב. היא דואגת לממשק פשוט להפעלה עם כל מה שאנחנו צריכים. מערכת ההפעלה מדברת 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 החלק 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 מערכת אבסטרקטית 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, בראזר, 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 למשאב 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

מערכות 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.

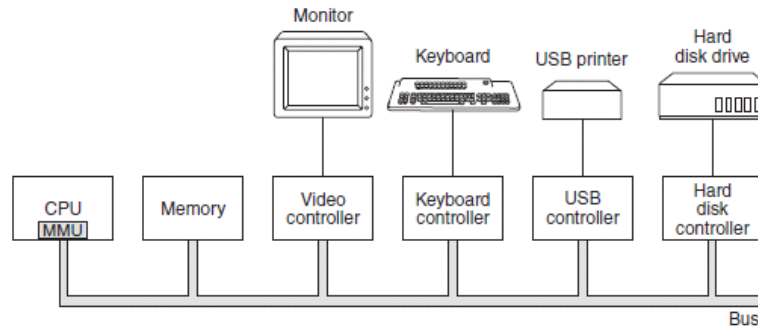


Figure 1-6. Some of the components of a simple personal computer.

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

לגשת לזכרון ולקלוט פקודה כלשהי לוקח יותר זמן מאשר להוציא אותה לפועל. לכן בכל מעבד יש רגיסטרים שמטרתם לשמור משתנים ותוצאות זמניות. לכן סט של פקודות בד"כ כולל פקודות לטעון מידע כלשהו מהזכרון לרגיסטר ולרשום מידע מהרגיסטרים לזכרון. בנוסף לכל מעבד יש לרוב כמה רגיסטרים נוספים שויזבלים למתכנת:

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). הטטאק כולל frame עבור כל בלוק שנכנס אבל עדיין לא רץ והוא מכיל את המשתנים המקומיים ואת המשתנים הזמניים שלא נמצאים ברגיסטרים.

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. הוא זה ששומר את כל המידע כשיש context switch, עליו נדבר אח"כ.

מערכת ההפעלה צריכה להיות מודעת לכל הרגיסטרים. כשהיא עוצרת תהליך מסוים כדי להריץ תהליך אחר, היא צריכה לשמור את כל הרגיסטרים הרלוונטיים 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).

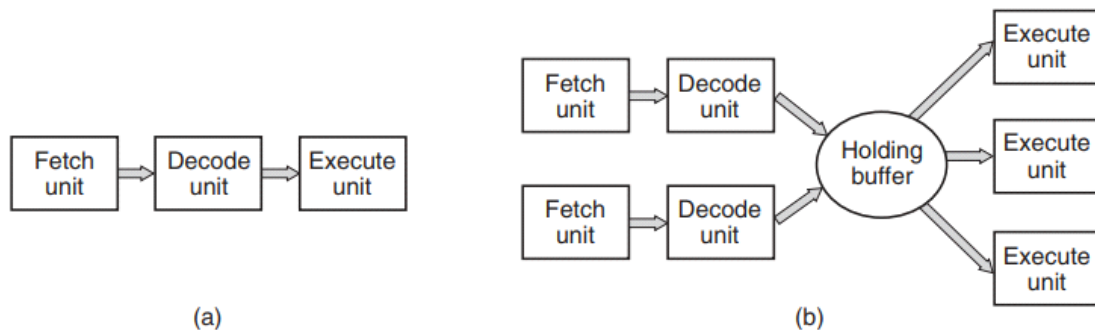


Figure 1-7. (a) A three-stage pipeline. (b) A superscalar CPU.

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 בעיצוב 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. ביזר 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"נכנסת" ל-kernel ומייצרת מאורע 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.

זכרון

באידיאל אנחנו רוצים שזכרון יהיה מאוד מהיר, מאוד גדול ומאוד זול. כל זמן שאין לנו טכנולוגיה שתשיג את האידיאל הזה, מערכת הזכרון בנויה בשכבות: השכבות העליונות בעלות מהירות מאוד גבוהה, קיבולת נמוכה ומחיר גבוה (פר ביט) מהשכבות הנמוכות.

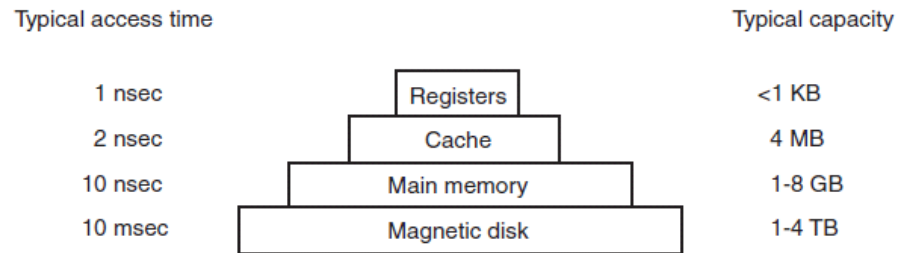


Figure 1-9. A typical memory hierarchy. The numbers are very rough approximations.

השכבה העליונה כוללת את הרגיסטרים שנמצאים בתוך ה-CPU. מבחינת חומרה הם עשויים מאותו חומר כמו ה-CPU ועל כן מהירים באותה מידה, כלומר אין דילי בפניה אליהם (לא לגמרי ברור) האם הם מהירים בגלל שהם יושבים ב-CPU והגישה אליהם מיידית, או כפי שכתוב בספר זה קשור לאופן שבו הם בנויים חומרית. הקיבולת כאמור נמוכה (פחות מ-1KB). תכניות צריכות "להחליט" איזה מידע ישמר ברגיסטרים.

הערה (מתקשר לפרק על זכרון):
כשמשתמשים בזכרון וירטואלי, ה-RAM הוא אנלוגי ל-cache, במובן הזה ששם נמצאים הפייג'ים שיותר בשימוש והגישה אליהם מהירה (באופן יחסי). אם הם לא שם צריך ללכת לדיסק. כנל לגבי ה-cache - אם הכתובת זכרון שפניתי אליה לא שם, צריך לגשת ל-RAM (למעשה צריך לגשת לפייג' וירטואלי ושם יתחיל כל תהליך ה-paging).

לאחר מכן בא ה-cache שהוא רכיב חומרתי שהוא מעין זכרון בפני עצמו, מאוד מהיר. ה-RAM מחולק למה שנקרא **cache lines** שזה בלוקים של זכרון, בד"כ בגודל של 64 בייט (צריך לבדוק) בהמשך איך זה מתקשר לפייג'ים). ה-cache lines משתמשים בהם בתדירות הכי גבוהה נשמרים ב-cache, כאשר ה-cache עצמו נמצא או ב-CPU עצמו או ממש קרוב אליו. כשיש פנייה לזכרון, דבר ראשון בודקים האם זה נמצא ב-cache.

ברמה היותר נמוכה נמצא ה-main memory (RAM). הכי נמוך בהירארכיה נמצא הדיסק.

I/O devices

בנוסף לזכרון ולמעבד, מערכת ההפעלה צריכה לנהל גם את ה-I/O devices. דיבייס לרוב כולל קונטרולר ואת הדיבייס עצמו. הקונטרולר הוא זה שמקבל פקודות ממערכת ההפעלה, אם כי לא ישירות. כיוון שכל קונטרולר הוא שונה, יש צורך בתוכנה שונה כדי לתקשר איתו. התוכנה שמדברת עם קונטרולר ספציפי נקראת device driver. יצרני הקונטרולרים (או הדיבייסים למעשה) צריכים לספק דרייברים שיתאימו לכל מערכת הפעלה (אלא אם כן הם מוכנים שהדיבייס יעבוד רק עם מערכת הפעלה מסוימת?). כדי שהדרייבר ירוץ, צריך לשים אותו במערכת ההפעלה כדי שהוא ידע לרוץ ב-kernel mode.

- תהליך הקלט והפלט ב-I/O יכול להעשות בשלוש דרכים:
- הדרך הפשוטה- תכנית מבצעת system call, מערכת ההפעלה פונה לדרייבר הרלוונטי, הדרייבר שולח את הפקודה לקונטרולר ואז הוא נכנס ללופ שבו הוא כל הזמן בודק האם הדיבייס סיים (לרוב יש ביט שאומר אם הדיבייס בפעולה או לא). כשהדיבייס מסיים, אם יש משהו להחזיר הדרייבר מחזיר את זה ואז מערכת ההפעלה מחזירה את השליטה למי שקרא לדיבייס. השיטה הזו היא למעשה busy waiting, שזה (כפי שיוסבר בהמשך) מצב שבו ה-CPU תפוס למרות שבפועל הוא לא עושה כלום- רק מחכים שהדיבייס יסיים.
 - דרך אחרת היא שהדרייבר מפעיל את הדיבייס ו"מבקש" ממנו לייצר אינטארפט כשהוא מסיים.
 - שימוש ב-DMA.

נרחיב על שלושת הדרכים בפרק הרלוונטי.

Buses

המודל באיור 1.6 (למעלה) שבו יש באס אחד עבד במשך תקופה מסוימת, אבל עם הזמן, כשהמעבדים והזכרון נהיו מהירים יותר, באס יחיד כבר לא היה מספיק כדי באמת להתמודד עם כל המידע שצריך להעביר. בהמשך לצורך הזה, התווספו עוד באסים, גם כדי להעביר מידע ל-I/O, וכדי להעביר מידע בין ה-CPU לזכרון. כיום מערכת כזו נראת כך (איור ממש לא ברור):

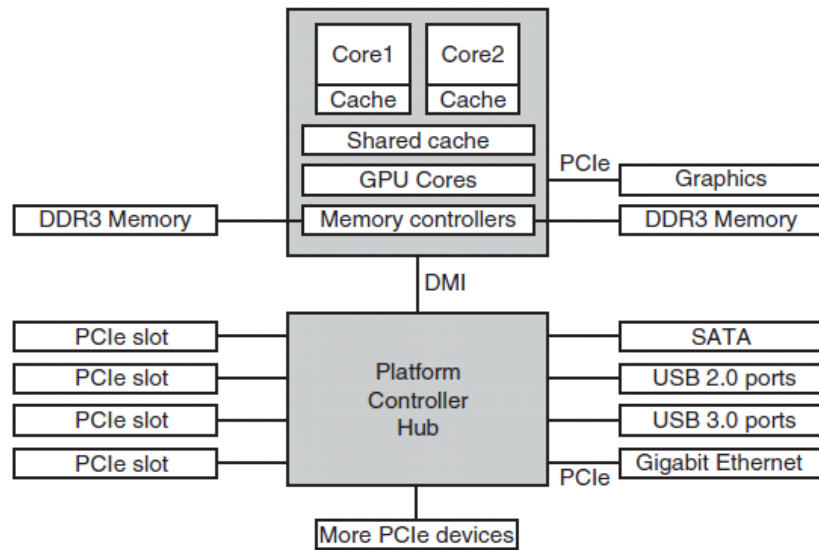


Figure 1-12. The structure of a large x86 system.

אם אני מבינה נכון, כל המשבצות הלבנות זה באסים.
הבאס העיקרי הוא **PCIe** גם לא ברור, כי יש כמה כאלה..
PCIe (Peripheral Component Interconnect Express)

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 נכתב לזכרון ורק אחכ לא על חשבון משהו אחר, נזרק 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'. המשמעות 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 פרוסס מרחב הזכרון של תהליך?
מרחב הזכרון של כל תהליך שרץ מקבל את מרחב 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/write` ו-`read only`

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 מלכתחילה 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.

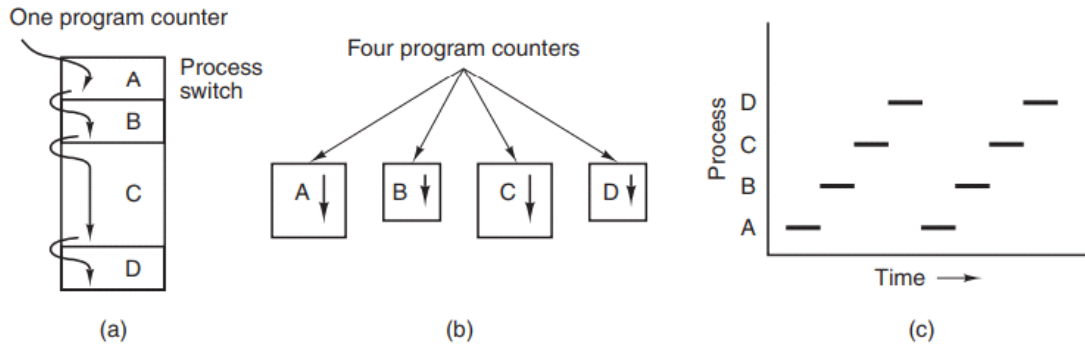


Figure 2-1. (a) Multiprogramming four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

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. כשיגיע 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

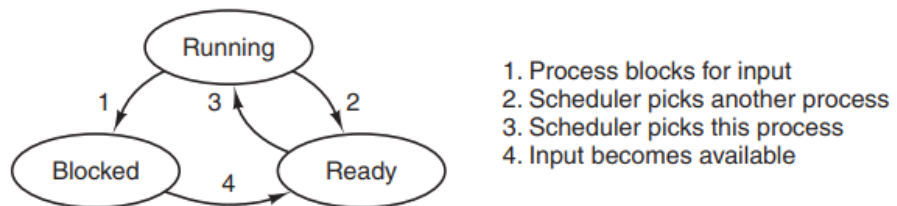


Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

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 בפקודה הזאת: בלינוקס יש הירארכיה, 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-`PC` program counter זהה, הם ימשיכו מאותו מקום. הבן מקבל 0 כ-`PID`. `fork()` זה `system call` שפונה למערכת ההפעלה. בוינדואס אגב זה עובד אחרת, אין הירארכיה 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 אין הירארכיה. 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 zomb - 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.

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 20 c משמע, תעשה 20 continue פעמים (למשל לולאה שאני רוצה להגיע לריצה ה-20 שלה).

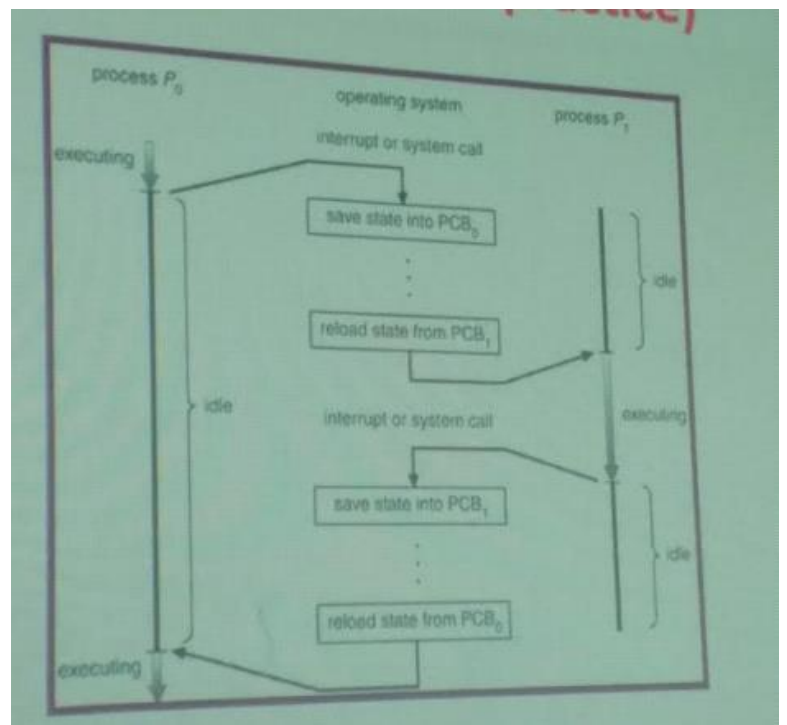
Context Switch

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% שיקף-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 מכויל 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-scheduler היא ב-user interactive- כי שם ממש צריך כל הזמן לעצור תהליך. ב-real time יש פחות קטיעות, צריך לוודא שדברים קורים בקבועי זמן מסויים.

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 אינטרולים קצרים

הריאל טיים, הוא גם פרימטיב כמוכן, 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 חייבים 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 מסוימת. 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 לו אדר

שיועד 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**.

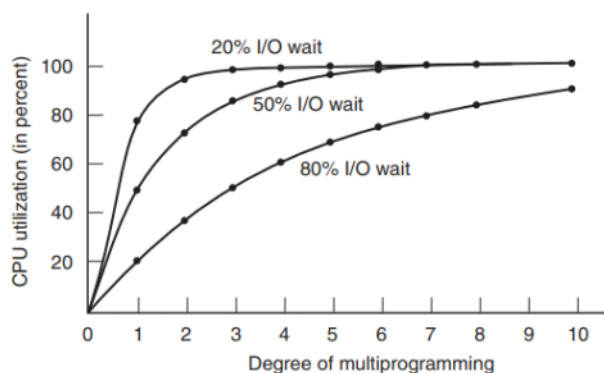


Figure 2-6. CPU utilization as a function of the number of processes in memory.

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 ומדפיס. תהליך A פונה אל הספולר ומבקש מקום פנוי כדי להדפיס. הספולר אומר לו- יש מקום פנוי ב-7. אממה? מערכת ההפעלה עוצרת 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.

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.

Dispatcher

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

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.

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 שהאלגוריתם 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 להשתמש במנגנון הקודם (כי אז למעשה אין פריוריטי וזה פשוט תור).
אם לא, ניתן את העדיפות 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 פריוריטי
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 נמצאת 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.

עמוד 97 בספר

בעקרון לכל תהליך יש מרחב זכרון ו-sigle thread of control. אבל במצבים רבים, אנחנו רוצים שיהיה כמה threads באותו מרחב זכרון, כך שכאילו יש לנו תהליכים נפרדים (למעט מרחב הזכרון שמשותף להם).

יש כמה סיבות למה נרצה כמה מיני תהליכים (threads) שרצים במקביל. הסיבה העיקרית היא שבאפליקציות רבות, יש פעילויות שונות שרצות באותו זמן (ולא בהכרח תלויות זו בזו). מעת לעת חלק מהפעילויות האלו יכולות להכנס לבלוק. אם אני מחלקת את הפעילויות השונות ל-threads שרצים כאילו במקביל, זה מאפשר לי להמשיך לתפקד- כלומר, התכנית לא תהיה מושבתת בגלל בלוק של thread אחד. זה למעשה אותה סיבה שאמרנו שאנחנו רוצים תהליכים שונים (ובשביל זה יצרנו תהליכי בן). במקרה של תהליכים, במקום לחשוב במונחים של אינטראקטים, טיימרים וקונטקסט סוויצ'ים, יכולנו לחשוב במונחים של תהליכים מקביליים.

כאן הדבר דומה, אלא שאנחנו מוסיפים אלמנט: היכולת של ישויות מקבילות לחלוק מרחב זכרון (ומן הסתם כל הדאטא שבזכרון הזה) משותף. היכולת הזאת היא קריטית במקרים מסויימים, ולכן לא ניתן יהיה לעשות שימוש בתהליכים מקבילים. סיבה נוספת לעבוד עם threads היא שהם פחות כבדים מתהליכים ולכן יותר קל (כלומר יותר מהר) ליצור ולהרוס אותם.

דוגמא לעבודה עם מעבד תמלילים שבו רצים שלושה threads- אחד קולט את הקלט מהמקלדת (היוזר כותב קובץ טקסט), השני שומר את הטקסט לדיסק (יכול לעשות את זה באופן אותומטי כל כמה שניות/ דקות) והשלישי מסדר את העמודים כל פעם שיש איזשהו שינוי באמצע (נוספה או הוסרה שורה למשל):

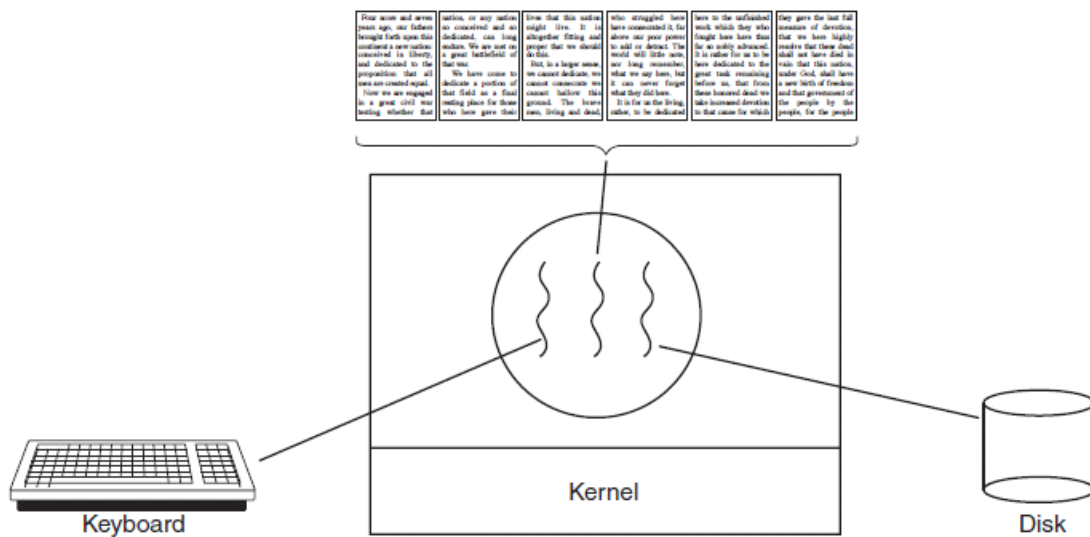


Figure 2-7. A word processor with three threads.

לו היה מדובר ב-thread אחד ויחיד, אז כל פעם שהקובץ נשמר אותומטית, לא היה ניתן לקלוט קלט מהמקלדת וכד'. אגב, בדוגמא הזו, לא ניתן היה לעבוד עם שלושה תהליכים נפרדים משום שלוששתם צריכים לתקשר עם אותו קובץ. מרחב הזכרון המשותף הוא שמאפשר ל-threads לעשות זאת. טבלה עם השוואה של סרדים ולא סרדים- מחברת??

המודל הקלאסי של Threads
המודל מתבסס על שני מונחים בסיסיים:
resource grouping
execution

לעיתים זה מועיל להפריד בין השניים, וכאן נכנסים לתמונה ה-threads. נבחן קודם את המודל הקלאסי ואז נראה איך זה בלינוקס, שמטשטשת מעט את הגבולות בין תהליכים ו-threads.

באיזשהו מקום, אם יש לנו הרבה threads תחת תהליך אחד, זה אנלוגי להרבה תהליכים שרצים במחשב אחד. כיוון שכל thread יש גישה לזכרון של הת'רדים האחרים (שהוא כאמור אותו זכרון), ברמת העקרון יכול להיווצר מצב שבו thread אחד יכול לגשת ואולי אפילו למחוק את הסטאק של thread אחר. אין שום הגנות במקרה הזה קודם כל כי זה בלתי אפשרי להגן. אבל יותר חשוב- אין צורך בכך. בשונה מתהליכים, threads שונים תמיד מופעילים על ידי אותו היוזר (שמפעיל את התהליך) ולכן לא אמור להיווצר קונפליקט בין ה-threads.

מה שמשותף:

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.

running, blocked, ready, blocked: thread יכול להיות בכל אחד מהמצבים הבאים: thread יכול לתהליך, בדומה לתהליך, thread שרץ משמע שה-CPU זמין לו. בלוק משמע שהוא מחכה שמשהו יעשה לו unblock, למשל כשמתבצעת system call לקלוט קלט מהמקלדת, ה-thread נחסם עד שמגיע אינפוט מהמקלדת (וכשזה יקרה, הוא יעבור ל-ready ויחכה לתורו). המעבר בין מצבים של threads זהה לזה של תהליכים. תזכורות, כמו שראינו בתהליכים:

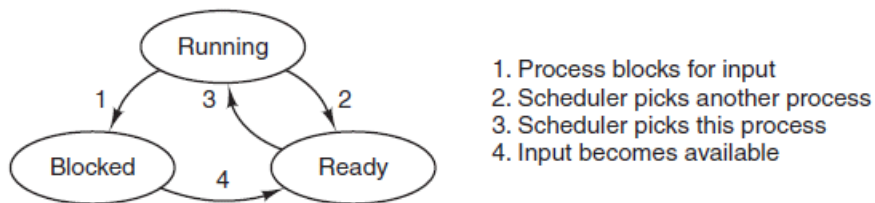


Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

מדלגת

Interprocess communication

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-threads 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 הלוקאליים 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 threads יכולים לגשת אליהם בלי בעיה. גם משתנים גלובליים- 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 או 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 שקונטקסט 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 למשאבים 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 כן, 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

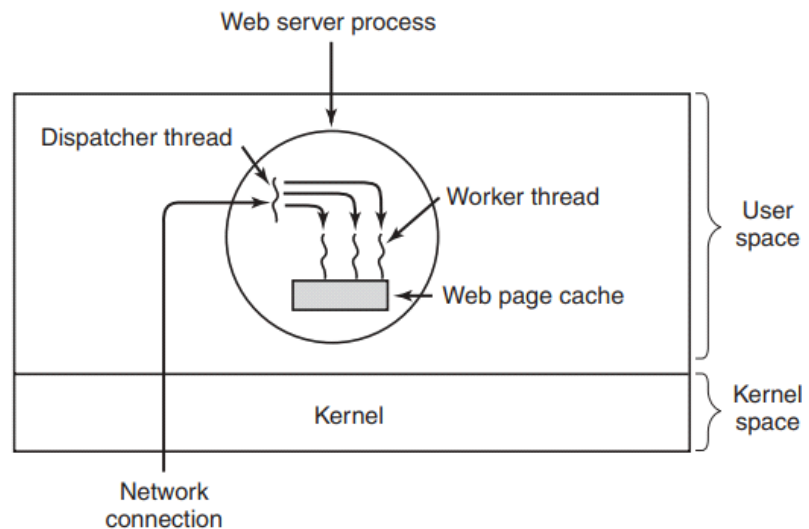
נניח יש לי תכנית שיכולה לקבל אינפוט מכל מיני מקורות (מקלדת, רשת ועוד). בעקרון יכול להיווצר מצב שבו אני מחכה לאינפוט אחד (מהמקלדת למשל) ואז התכנית תקועה, למרות שהאינפוט שהיא מקבלת מהרשת (למשל) לא קשור ואפשר להמשיך להזרים אותו. threads נפרדים ממש יוצרים מקביליות ומונעים סיטואציה כזו.

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 מגנון 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.



פסודו 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)

- אין מקביליות (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 התראדים נסגרים.

join - פונקציה שמשמעותה שאני רוצה להמתין שת'ראד מסוים יסיים (קצת מקביל ל- wait בתהליכים). כשעושים join לת'ראד מסוים, 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 התראדי שלו מתים. הגיוני שכשתהליך מסיים, לפני זה הוא ימתין שכל התראדים 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-yield. נניח אני מחכה למסר כלשהו מת'ראד מסוים, אני יכולה לדגום בתדר יחסית נמוך ולעשות yield שתי הפונקציות האחרונות- בד"כ לא נוגעים בהן, זה קשור יותר לקונפיגורציה של התראדים

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

התכנית שהרצנו בכיתה:

במצב מסוים, נותרנים לגלובל לספור והוא לא סופר כמו שצריך (זהו 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 ה-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

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 שנפלים 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 במטוס)
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 רלוונטיים 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 תהליך אחר

critical section

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

אם כך צריך לזהות את האזורים הקריטיים 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. מכניסים תהליך אחר לבלוק 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 הקריטי ואנחנו מאפשרים רק תהליך אחד. ברגע ש-A מסיים (T3) B נכנס לאזור הקריטי ועוזב כשהוא מסיים ב-T4.

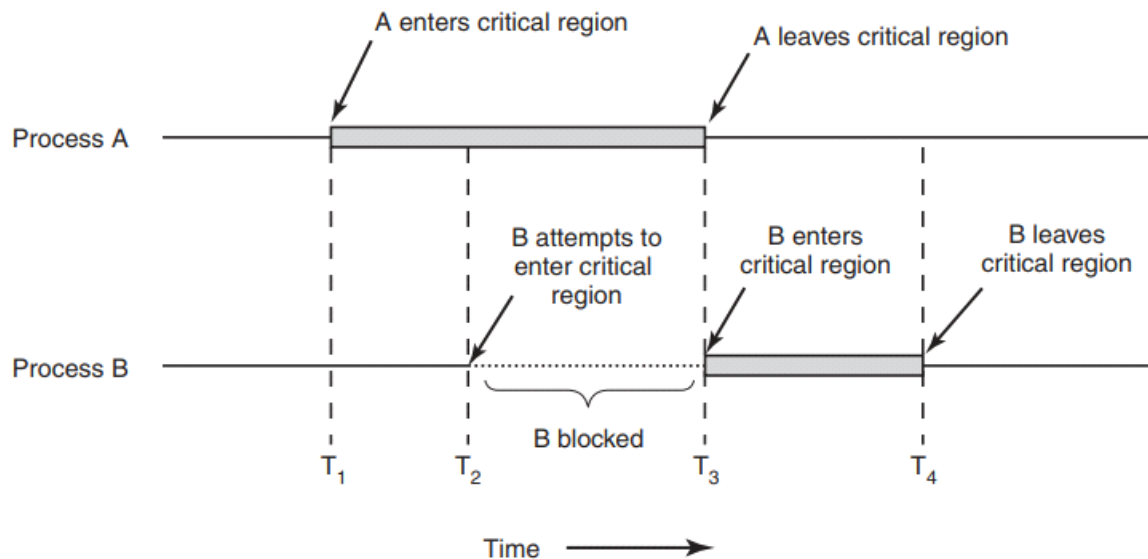


Figure 2-22. Mutual exclusion using critical regions.

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 interrupts 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 בסיכום - race condition). 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 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?

המשתנה turn שמאותחל ב-0 למעשה עוקב אחרי התור של מי עכשיו להכנס לאזור הקריטי. כרגע לעניינינו יש שני תהליכים שהמספר הסידורי שלהם זה 0 ו-1. בשלב הראשון, תהליך 0, רואה ש- turn = 0, ונכנס ללופ שבו הוא כל הזמן בודק מתי זה משתנה ל-1. 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 מסוים נקראת **busy waiting**. זה מצב שבאופן כללי יש להמנע ממנו כיוון שזה מבזבז זמן CPU (זה כאילו רץ מבחינת ה-CPU, כשמהותית, התהליך צריך להיות בלוק כרגע כי הוא מחכה שמשהו יקרה). מאפשרים busy wait כשיודעים שזה לזמן קצר יחסית. **מנועל 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 מסוים 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 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 אסמבלי? 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 אינטראפטים 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 יתחור, 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?

בעזרת ה-TSL אני לא צריכה ליצור את התורות, ואז בעיית הסינכרון נעלמת. אני יכולה לעשות הגנה אמיתית שרק אחד יכנס לאזור הקריטי בלי הסנכרון.

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.

```

1  #include <pthread.h>
2  #include <stdio.h>
3  #define BUFFER_CAPACITY 100
4
5
6  int g_count = 0;
7
8  void producer(void)
9  {
10     int item;
11
12     while (TRUE)
13     {
14         item = produce_item(); /* generate an item */
15         if ( g_count == BUFFER_CAPACITY)
16         {
17             sleep();
18         }
19         insert_item(item); /* put item in the buffer */
20         count++;
21         if (g_count == 1)
22         {
23             wakeup(consumer);
24         }
25     }
26 }
27
28 void Consumer(void)
29 {
30     int item;
31
32     while (TRUE)
33     {
34         if (g_count == 0)
35         {
36             sleep();
37         }
38     }
39     item = RemoveItem(); /*remove item from the buffer */
40     g_count--;
41     if (g_count == BUFFER_CAPACITY -1)
42     {
43         wakeup(producer);
44     }
45     ConsumeItem(item); /*do something with the item, for example- print */
46 }
47

```

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 הפרודוס 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`.

איך אפשר להגיע ל-`race condition`? נניח שהבאפר ריק והקונסומר קורא את ה-`count` כדי לבדוק האם הוא 0 (והוא כאמור 0). באותו רגע ה-`scheduler` מחליט לקטוע את הקונסומר ולתת לפרודוסר לרוץ. הפרודוסר שם אייטם בבאפר, מגדיל את `count`, בודק את `count`, רואה שהוא שווה ל-1 וקורא קריאת `wakeup (consumer)`. הבעיה היא שהקונסומר לא באמת ישן, כי הוא נקטע לפני שהוא "נרדם" והקיראה הזו למעשה הולכת לאיבוד. כשה-`scheduler` נותן לקונסומר לרוץ שוב, הקונסומר כבר קרא שה-`count` שווה ל-0 ולכן עכשיו הוא הולך לישון. הפרודוסר ממשיך למלא את הבאפר ובשלב מסוים ימלא אותו ואז יילך לישון ואז שניהם ישנים לנצח..

הבעיה טמונה בכך שקריאת ה-`wakeup` נשלחה לתהליך שעוד לא ישן והלכה לאיבוד.

חשוב לזכור: אינטראפט לא יוצר בהגדרה context switch.
אחרי הקואנטום ה-scheduler מקבל החלטה, ואם הוא מחליט שיהיה context switch, אז זה יקרה.

עוד נקודה חשובה: context switch בתוך quantum יקרה רק אם תהליך לנכנס לבלוק בזמן ה-quantum. כלומר, ה-scheduler יכול לקטוע "בכוח" רק בין quantum ל-quantum.
האינטראפט יכול להביא לקח שתהליך יעבור מבלוק ל-ready, אבל כפי שציינו קודם, לא ישירות ל-running.

ראינו בשיעור הקודם את בעיית ה-producer - consumer (בעיית הסינכרון ובעיית האזור הקריטי שלא בהכרח זהות)?
כלי אחד, שנותן פתרון לשתי הבעיות: 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, צריכה לתמוך בכלי כזה.
מן הסתם בכל מערכת הפעלה, הפניה לסמפור מעט שונה

אין כותבים תכנית ב-C (למשל) שתוכל לפנות למערכת ההפעלה? אני בתכנית שלי בונה API שיועד 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 שתהליך אחד תקוע על המחסום, הוא לא אהיה ב-busy wait, אלא בבלוק. כלומר, אם משהו הגיע למחסום ולא יכול להכנס, שיכנס לבלוק. המנגנון הזה יוציא אותו מהבלוק כשצריך.
כשמקנפים סמפור, מגדירים כמה יכולים להכנס בזמן נתון. אם מגדירים אחד, זה בדיוק מנגנון הגנה על critical section.

אבל, לפעמים אני רוצה לאפשר כניסה של כמה.
אנלוגיה לסמפור- תאי השירותים הציבוריים. נגיד יש שלושה תאים, מקנפים סמפור ל-3. בכל זמן נתון יכולים להיות שלושה. הגיע הראשון, יכול להכנס, שני ושלישי גם. מגיע הרביעי- ממתין. עד שמהו יצא. יכול גם להיות שיצאו שניים. כלומר, זה מחסום חכם שסופר ויודע בכל רגע נתון כמה יכולים להכנס וכשהוא מגיע למקסימום הוא עוצר ולא נותן לאף אחד להכנס.

הרעיון (של Dijkstra) היה לשמור את כמות ה-wakeup באמצעות כלי שנקרא semaphore. הוא הציע שלסמפור יהיו שתי פעולות- up ו-down (שהן הכללה של הפעולות sleep ו-wakeup שראינו קודם). פעולת **down** בודקת אם הערך של הסמפור גדול מ-0. אם כן, היא מקטינה ב-1 (כלומר משתמשת ב-wakeup פנוי אחד) והפעולה ממשיכה בלי לעצור. אם הערך של הסמפור הוא 0 (כל תאי השירותים תפוסים), התהליך נכנס ל-sleep ופעולת ה-down נעצרת (כלומר לא מקטינים ולא ממשיכים בתהליך).
בדיקת הערך, שינוי שלו וההרדמות הפוטנציאלית- נעשים יחד **כפעולה אטומית אחת** (לא ברור, אם זו פעולה אטומית אחת- אין יכולה להיות תגובה שונה ל-0 ו-1 אם זו פעולה אטומית).

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

פעולת **up** מגדילה את הערך של הסמפור (תא אחד בשירותים התפנה, הסמפור יכול להכניס תהליך אחד יותר ממה שהיה לו עד עכשיו).
בזמן הפעולה הזו יש תהליך (אחד או יותר) שרדומים בגלל שהם לא הצליחו להשלים את פקודת ה-down, בשלב הזה אחד מהם יבחר ויוכל להשלים את פעולת ה-down (כלומר להקטין שוב את הסמפור ב-1, ולהמשיך לרוץ). כלומר בעת up במצב שבו יש תהליכים בבלוק, הערך עדיין ישאר 0, אבל יהיו פחות תהליכים רדומים.

גם פעולת הגדלת הסמפור ב-1 והתעוררות של תהליך ישן היא פעולה אטומית. תהליך לא יכול להכנס לבלוק בזמן up.
אם אין תהליכים רדומים, אז באמת הערך של הסמפור גדל באחד, מה שאומר שהסמפור "יודע" שיש מקום פנוי נוסף.
הערה חשובה: גם כאן, תהליך רדום לא מתחיל לרוץ מייד כשמשתחרר מקום, אלא הוא עובר מ-ready ל-block. זה שוב מזכיר לנו שתהליך אף פעם לא עובר ישירות מ-block ל-running.

נחזור למודל ה-producer-consumer ונראה איך באמצעות סמפור פותרים את הבעיה פותרים באמצעות שלושה סמפורים.

#include <semaphore.h>

כתיבת תכנית שמשתמשת בסמפורים:

סמפור בינארי - סמאפור שיכול לקבל את הערכים 1 ו-0. זה מצב שבו צריך להגן על אזור קריטי שבו לא ניתן לאפשר יותר מתהליך אחד. במחינה לוגית, סמאפור בינארי זהה ל-mutex, שזה כלי שנדבר עליו יותר מאוחר.

סמפור הוא סוג של **unsigned int**. זה קאונט שלא יכול לקבל ערך שלילי. הסמפור מאותחל למספר המקומות החופשיים במשאב (כלומר, אם זה אזור קריטי, זה יהיה מאותחל ב-1). כשהסמפור הופך ל-0, זה מעיד על כך שהמשאב לא פנוי, ותהליכים שינסו לעשות לו down יכנסו לבלוק עד שהסמפור יקבל ערך שגדול מ-0.

ה- semaphore.h מגדיר טיפוס sem_t

sem_init - initialize a semaphore

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

/* LINK WITH -pthread */

Initializes the semaphore at the address pointed by sem

pshared - indicates weather the semaphore will be shared by threads of one process or between processes.

0 - threads

nonzero- processes (we will probably learn this later in IPC)

כיוון שאנחנו כרגע מדברים רק על threads, הערך של pshared יהיה 0. דבר נוסף שחשוב במקרה כזה, זה לשים את הסמפור במקום שזמין לכל ה-threads, כלומר כמשתנה גלובלי (או משתנה שעושים לו אלוקציה דינמית בהיפ).

חשוב- אתחול של סמפור שכבר אותחל- undefined behaviour

הפונקציה מחזירה 0 אם היא הצליחה ו 1- אחרת.

במקרה שלנו, איתחול סמפור יראה כך (בהנחה ומדובר בסמפור בינארי שמגן על אזור קריטי):

```
sem_t semaphore;  
sem_init(&semaphore, 0, 1)
```

sem_wait - corresponds to down

sem_post - corresponds to up

```

1  #include <semaphore.h>
2  #include <pthread.h>
3  #include <stdio.h>
4
5  #define BUFFER_CAPACITY 100
6
7  int g_slotsInBuffer = 0;
8
9  /* semaphores defined as globals */
10 sem_t g_mutex = 1; /*binary semaphore- access to critical area */
11 sem_t g_empty = BUFFER_CAPACITY; /* empty slots initializes with capacity- i.e all slots are empty */
12 sem_t g_full = 0; /*no full slots at the beginning */
13
14
15
16 void Producer(void)
17 {
18
19     while(1)
20     {
21         sem_wait(&g_empty); /* Decrement empty buffer slots (before putting something in it */
22         sem_wait(&g_mutex); /* Enter critical region */
23         g_slotsInBuffer++; /* Put new item in the buffer */
24         sem_post(&g_mutex); /* Leave critical region */
25         sem_post(&g_full); /* Increment count of full slots */
26     }
27 }
28
29 void Consumer(void)
30 {
31     while(1)
32     {
33         sem_wait(&g_full);
34         sem_wait(&g_mutex); /* Enter critical region. If mutex == 0, thread will be put in block */
35         g_slotsInBuffer--; /* Take an item from the buffer */
36         sem_post(&g_mutex);
37         sem_post(&g_empty);
38     }
39 }
40
41 int InitSemaphores(sem_t* _mutex, sem_t* _empty, sem_t* _full)
42 {
43     int res, res2, res3;
44     res1 = sem_init(_mutex, 0, 1);
45     res2 = sem_init(_empty, 0, BUFFER_CAPACITY);
46     res3 = sem_init(_full, 0, 0);
47     if ((res1 || res2 || res3) == 1)
48     {
49         return 1;
50     }
51
52     return 0;
53 }

```

שאלה- רשמתי בכיתה שזו פונקציה של מערכת ההפעלה, לא אנחנו צריכים לממש.
מה אנחנו לא צריכים לממש ומה כן?

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, unlock, אי אפשר לעשות סינכרון (דומה לסמפור הבינארי). כלי הרבה יותר פשוט ולכן כשלא נצטרך סינכרון, עדיף שנשתמש בו

פונקציונליות דומה- אם תהליך אחד עושה lock ותהליך אחר מנסה להכנס, האחר יכנס לבלוק. גם פה אין busy wait. בהגדרה הבסיסית (המקורית) של mutex, רק מי שעשה את ה-lock יכול לעשות unlock. במימוש של הסמפור ראינו שתהליך אחר עושה up ותהליך אחר עושה down. על פניו ב- mutex אי אפשר לעשות את זה. אבל, חשוב לזכור שזה היה כך רק בעבר. היום, לפחות בלינוקס (ובעוד כמה מערכות הפעלה) המגבלה הזו הוסרה- אם ננסה לתת לתהליך אחד לעשות לוק ולשני אנלוק, זה יעבוד. בהמשך נדבר על dead lock, שזה מצב שבו הכרחי לעשות unlock מחוץ לתהליך שעשה lock.

על אף הדמיון, 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

שאלה- האם היום בפועל יש הבדל בין mutex לבין סמפור בינארי. במה בפועל מתבטאת הפשטות של הכלי הזה?

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

```
void Producer(void)
{
    while(1)
    {
        sem_wait(&g_empty); /* Decrement empty buffer slots (before putting something in it */
        sem_wait(&g_mutex); /* Enter critical region */
        g_slotsInBuffer++; /* Put new item in the buffer */
        sem_post(&g_mutex); /* Leave critical region */
        sem_post(&g_full); /* Increment count of full slots */
    }
}
```

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

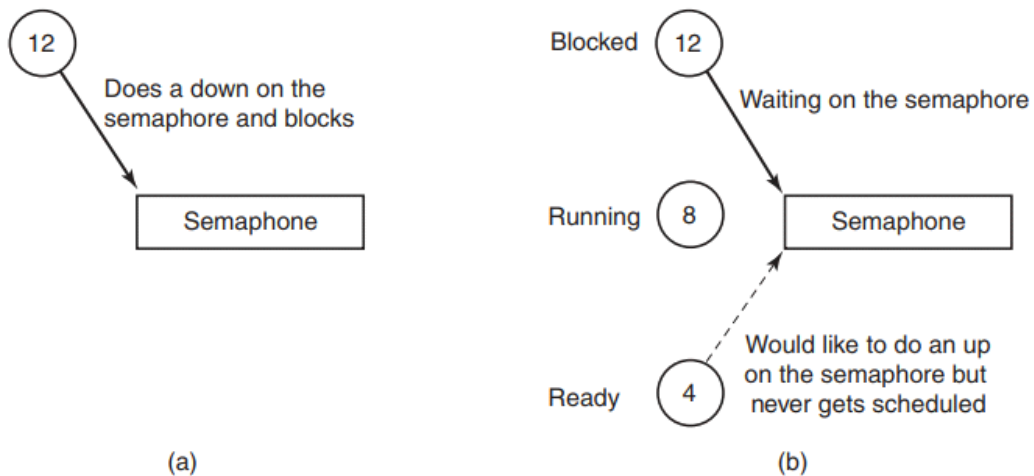
Figure 2-29. Implementation of mutex_lock and mutex_unlock.

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.
במה בא לידי ביטוי התדר הנמוך? איפה נקבע שזה לא כל הזמן בודק אלא דוגמא בתדירות יותר נמוכה?

Priority Inversion

נניח יש תהליך עם תעדוף גבוה שמגיע אזור הקריטי, והאזור הקריטי תפוס על ידי תהליך אחר עם תעדוף הרבה יותר נמוך. אם מדובר רק בשני התהליכים האלה, אין בעיה. התהליך עם התעדוף הנמוך יסיים, ישחרר את האזור הקריטי והתהליך עם התעדוף הגבוה יכנס וירוצ. אבל מה יכול לקרות?
יכול להיות שיש עוד מספר תהליכים שרצים בזמנית ורמת התעדוף שלהם נמצאת בין הגבוה לבין הנמוך שנמצא בתוך האזור הקריטי. זה יוצר מצב שהתהליכים האלה מקבל העדפה של המעבד ותוקעים את התהליך עם התעדוף הנמוך בבלוק, אלא שהוא נמצא בבלוק כשהוא בתוך האזור הקריטי ולכן הבלוק נכפה גם על התהליך עם התעדוף הגבוה מבין כולם. כלומר, התהליך עם התעדוף הגבוה כאילו עובר לרמת התעדוף הכי נמוכה.



פתרון: כשמערכת ההפעלה מזהה שמשוה בפריויריטי נמוך תוקע משוה גבוה, היא יכולה לשנות את הפריויריטי של הנמוך לזה של הגבוה, עד שהוא יוצא מהאזור הקריטי.

כאן סיימנו למעשה את נושא ה-IPC (למרות שיהיה לנו קורס שלם ספציפית על תקשורת בין תהליכים).

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, אין גישה חופשית (לכן במובן הזה אזור קריטי הוא משאב- אנחנו צריכים רשות לגשת אליו וכדי להמשיך לתפקד אנחנו חייבים לעבור דרכו).

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 שמשוה אחר גם רוצה לפנות אליו - אין פה בעיה. ה"זכרון" יודע לטפל בכולם בזמנית.
כלומר, אם לא עשיתי הגנה שרק אחד יוכל להכנס, זה עדיין יעבוד כמו שצריך. זה לא משוה שאני חייבת להגן עליו ולתפוס אותו.

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.

הזכרון שונה בכך שהוא מנהל את עצמו. בגלל שהדיבייסיים השונים פונים ישירות לזכרון ואין קונטרולר, אפשר לגשת לחלקים שונים שלו. יש משאבים שמחייבים תפיסה פיזית לפני שפונים אליהם ורק כשמחררים, אחרים יכולים לגשת. קונטרולר של דיסק למשל- הוא יכול כל פעם לטפל בתהליך אחד. כנל מדפסת- היא צריכה להיות זמינה לתהליך אחד ויחיד.

אז מה זה דד לוק?

זה מצב שבו תהליך א' תופס משאב מסויים ונמצא בבלוק כי הוא מחכה שתהליך ב' יפנה מאשב אחר, אבל תהליך ב' (שתופס את המשאב האחר) נמצא בבלוק כי הוא מחכה שתהליך א' יפנה את המשאב שהוא צריך כדי להמשיך לרוץ. הם שניהם ימתינו למעשה לנצח.

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

בכל מקרה, בלי קשר לכמות המשאבים, אם בכל רגע נתון תופסים רק משאב אחד, אז אין בעיה. אם כך הדרך הכי נכונה לטפל- לתכנן מערכת שבה בכל זמן נתון אני לא תופסת יותר ממשאב אחד, אני מבטיחה שלא יהיה דד לוק. זה יכול לקרות כשיש תפיסת תהליכים הדדית בין שני תהליכים או בשרשת בין כמה תהליכים. ראינו דוגמא גם במודל של הפרודוסר- קונסומר וגם בהקשר של priority inversion.

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.

בניח שנכנסתי ל-1- זה אומר שלא יכול להיות שמשוה יהיה ב-2. כי אם הוא ב-2 הוא עוד לא שחרר את 1 ולא הייתי נכנסת לשם. כשיש מספר רב של משאבים, חייבים לדאוג לכך שלעולם לא יהיה דד לוק, המשמעות היא שכנראה יהיה לזה מחיר כלשהו. במקרה הזה זמן ההמתנה שלי לאזור הקריטי יהיה ארוך יותר. הביצועים נהיים פחות טובים.

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. זה אחד האמצעים הכי נוחים לדאוג לכך שלא יהיה דד לוק - לוודא שכל תהליך מחזיק משאב אחד ויחיד בכל זמן נתון.
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. circular 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 (לא אומר שבהכרח יהיה, אבל אם אחד מהם לא מתקיים, בודאות לא יהיה דד לוק. כלומר, אם הצלחתנו למנוע את אחד התנאים, 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.

מה האסטרטגיות האפשריות להתמודדות עם מצבי דד לוק?

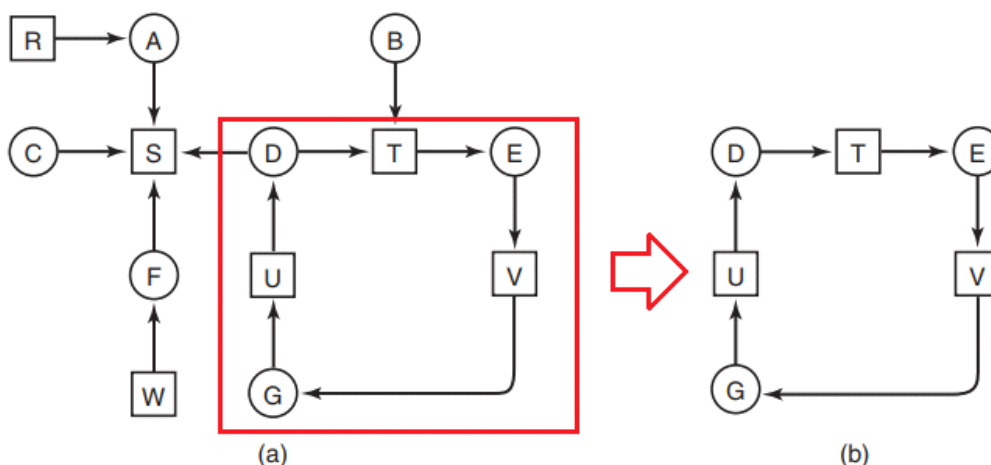
אפשרות אחת- להתעלם.

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 וזה המקום שאני צריכה להתמקד בו כדי למנוע. 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 נבדוק (כלומר מערכת ההפעלה תבדוק) האם יש דד לוק- במידה וכן, עושים פעולה אחורה אל הסירקולציה.

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

טיפול בדד לוק

אם זיהיתי שיש דד לוק, 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 (החזרת הגלגל אחורה).

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.

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 תופסת משאב אחר. ואז אומרים- אתה תכנס רק כשכל המשאבים פנויים. מנסים לתפוס את כולם בהדרגה (ראינו את זה קודם) 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 תקועה 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 ברוח.** 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 בהמשך איך זה יכול לקרות), ואז נוצר מצב שאני ממתנינה 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 בלולאה 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
אף אחד לא יודע שאני בדד לוק, כי זה במעין לופ אינסופי. כשלא נמצאים בבלוק. דד לוק על שני ספין לוקים יכול ליצור מצב של לייב לוק. כלומר, העניין העיקרי הוא שהמערכת לא מזהה שאנחנו במצב המתנה. אפשר יהיה אולי לעלות על זה בירידה של CPU

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

באידיאל, כל תכנית הייתה רוצה שתהיה לה נגישות לזכרון בלתי מוגבל בגודל ובמהירות, ובנוסף שהזכרון הזה יהיה nonvolatile - כלומר, שהתוכן לא יאבד כשהחשמל כבה. ובנוסף שגם יהיה זול..

עם הזמן התחילו לדבר במונחים של memory hierarchy. בהירארכיה הזו, למחשבים יש כמה מגה בייט של זכרון cache מאוד מהיר, מאוד יקר ונדיף (volatile), כמה גיגה של זכרון עיקרי (RAM) עם מהירות בינונית, מחיר בינוני וגם נדיף וכמה טרהבייט של אחסון שהוא זול, איטי ולא נדיף.

אחד מהתפקידים של מערכת ההפעלה, כפי שציינו בהקדמה הוא לעשות אבסטרקטיזציה להירארכיה הזו לתוך מודל יעיל שיאפשר לה לנהל את האבסטרקציה (את הזכרון).

RAM - Random Access Memory
ROM - Read Only Memory

מתן גישה ישירה לזכרון הפיזי לתהליכים שונים עלול לגרום לכמה בעיות. בעיה עיקרית היא שתכנית מסויימת יכולה בטעות (או בכוונה) לדרוס את המקום בזכרון שבו יושבת מערכת ההפעלה. כמו כן, במודל כזה זה קשה לאפשר לכמה תכניות לרוץ במקביל (או אפילו בתורות אם מדובר ב-CPU אחד).

יש צורך לפתור שתי בעיות במקרה של תכניות שרצות במקביל:

1. הגנה על זכרון של כל אחת מהתכניות
2. בעית הרפרנס היחסי לזכרון של כל אחת מהתכניות

הבעיה הראשונה ברורה.

הסבר על הבעיה השנייה:

נניח שיש לי שתי תכניות שרצות במקביל, כל אחת בגודל 16 KB (כל אחת צורכת 16,000 בייטים). התכנית הראשונה מתחילה לרוץ והפקודה הראשונה שלה (כפי שמופיע באיור) הוא לעשות קפיצה לכתובת 24 ושם לבצע פעולה כלשהי (MOV). התכנית השנייה מתחילה מקפיצה לכתובת 28 ומבצעת פקודה משלה (CMP). הבעיה היא ששתי התכניות כוללות פקודות עם כתובת ביחס ל-0 שלהן (מתוך ה-16k). אבל ברגע שתכנית אחת רצה, ה-0 של התכנית השנייה צריך להתחיל ב-16,380 (16*1024). כלומר, נקודת הייחוס צריכה להיות גם ביחס ל-0 של כל תכנית וגם ביחס של המקום בזכרון שהוקצה לכל תכנית (זזה תלוי בכמות התכניות שרצות ובגודל שלהן).

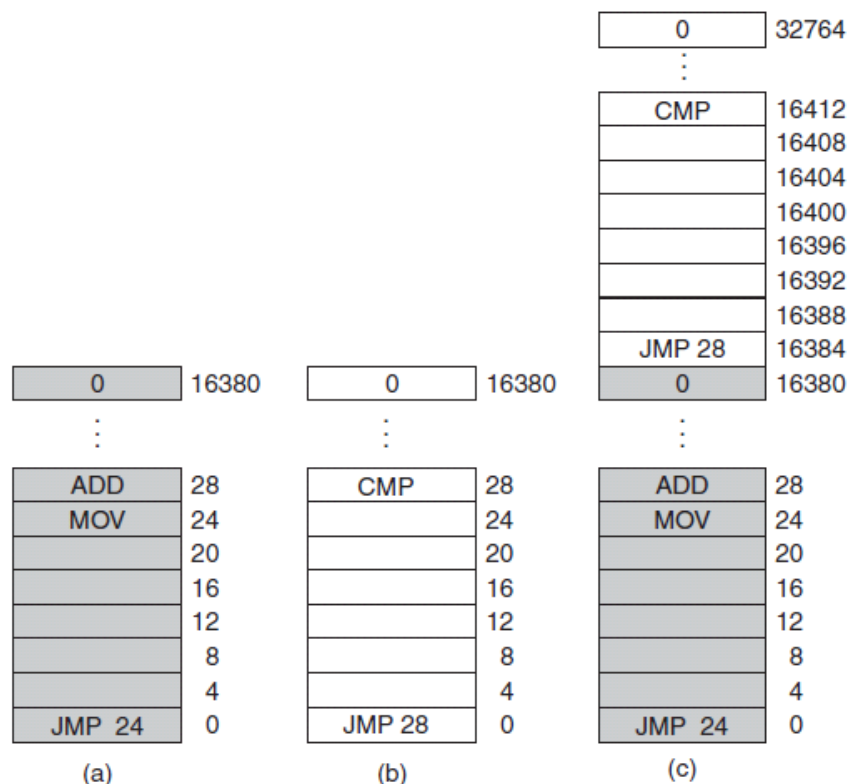


Figure 3-2. Illustration of the relocation problem. (a) A 16-KB program. (b) Another 16-KB program. (c) The two programs loaded consecutively into memory.

פתרון אפשרי אחד הוא להוסיף לתכנית השנייה קבוע של 16,380 לכל כתובת זכרון שמופיעה בתכנית, אבל הפתרון הזה הוא איטי ומסובך. באשר לבעיה הראשונה, פתרון פרימיטיבי אפשרי הוא לתת לכל תכנית מעין קוד כניסה שונה - כל פניה לזכרון מצריכה את הקוד וכך תכניות

תהליך- מעין CPU
מופשט שמאפשר להריץ
תכניות

פתרון יותר טוב- אבסטרקציה חדשה של זכרון - מרחב כתובות.
מרחב הכתובות יוצר מעין זכרון אבסטרקטי שתכניות יכולות "לחיות" בו.
מרחב הכתובות זה אוסף של כתובות שתהליך יכול לפנות אליו כשהוא צריך לפנות לזכרון. לכל תהליך מרחב כתובות משלו שלא תלוי במרחבי הזכרון של תהליכים אחרים (למעט מקרים בודדים שבהם תהליכים "רוצים" לחלוק מרחב זכרון (גם את זה יש להניח נלמד בתקשורות בין תהליכים).

כל המספרים עד 7 ספרות זה מרחב הכתובות של מספרי טלפון למשל (אם כי לא משתמשים בחלק מהמרחב הזה, למשל במספרים שמתחילים ב-0). סטרינגים באורך 2-63 תווים שנגמרים בסטרינג com. זה מרחב הכתובות של דומיינים באזור גאוגרפי מסויים.
בעיה שצריך לפתור- איך לתת לכל תכנית את מרחב הזכרון שלה- כלומר שכתובת 86 בתכנית אחת זה מקום אחד בזכרון הפיזי וכתובת 86 בתכנית אחרת זה מקום פיזי אחר בזכרון.

פתרון אחד פשוט- מיפוי מרחב הכתובות של כל תהליך למקום אחר בזכרון הפיזי. בעבר זה היה נעשה עם שני רגיסטרים ב-CPU שנקראו base ו-limit. ברגע שתהליך מתחיל לרוץ, הוא מקבל את הזכרון הפיזי שפנוי והרגיסטר base נטען בהתחלה של המקום הפיזי שהתהליך מקבל וה-limit זה האורך של התכנית. בצורה כזו, בכל פניה לזכרון במהלך הריצה, ה-CPU מוסיף את הערך של הבייס לפני שהכתובת נשלחת על ה-bus. בו זמנית הוא גם בודק האם קריאת הזכרון פונה לכתובת שמעבר ל-limit ואז יש שגיאה.
החסרון של השיטה הזו היא שבכל פניה לזכרון יש צורך לבצע את פעולת ההוספה (לבייס) וההשוואה (ללימיט). פעולת ההשוואה לרוב היא מהירה, אבל פעולת ההוספה איטית יחסית.

במידה והזכרון הפיזי הוא מספיק גדול כדי להכיל את כל התהליכים, הפתרון שתואר יעבוד פחות או יותר.
אבל לרוב, ה-RAM הכולל הנחוץ לכל התהליכים הרבה יותר גדול ממה שיש בפועל בזכרון הפיזי.
עם השנים התפתחו שתי גישות לבעיה הזו- memory overload:

1. העברת "חתיכות" זכרון של תהליך לדיסק ובכך פינוי זכרון RAM (swapping)
2. זכרון וירטואלי - מתן אפשרות לתכנית לרוץ גם אם היא נמצאת רק באופן חלקי ב-main memory

Swapping

האיור מראה את תהליך ה-swapping.

בשלב הראשון רק תהליך A נמצא בזכרון. בשלב הבא תהליכים B ו-C מתחילים לרוץ (או ממשיכים, אם מחזירים אותם בתהליך swapping מהדיסק). בשלב d מורידים את A לדיסק ומתפנה מקום בזכרון. אז D עולה ובהמשך B מועבר לדיסק. לבסוף A שוב עולה וכאן צריך לשים לב שהוא מקבל מקום שונה משהיה לו במקור ולכן משהו צריך לדאוג לכך שמרחב הכתובות שלו יתעדכן (פתרון הבייס והלימיט יעבוד סבבה).

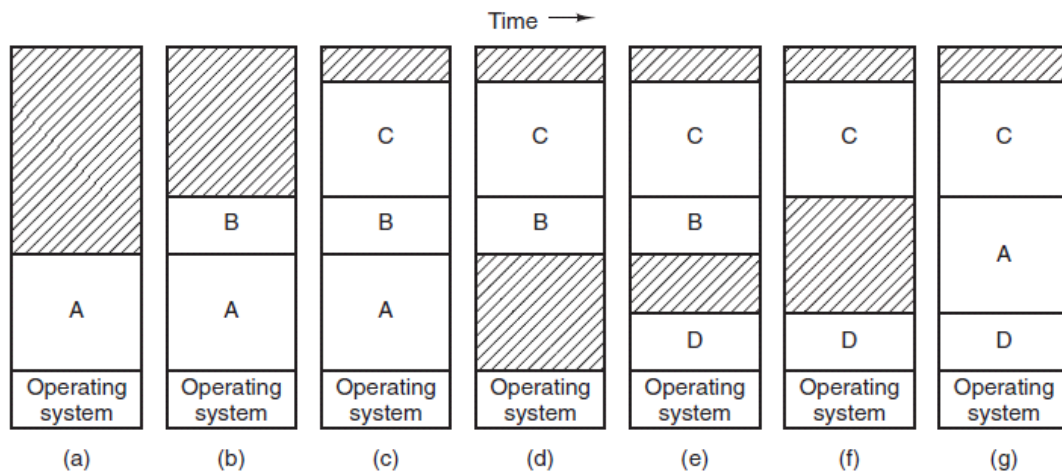


Figure 3-4. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

כשעושים הרבה החלפות, יכולים להווצר הרבה "חורים" בזכרון שהם קטנים מדי בשביל להעלות לתוכם תהליך חדש. במקרה כזה אפשר להוריד את כל התהליכים למטה (ולעדכן את מרחב הכתובות שלהם). זו שיטה שנקראת **memory compaction**. לרוב לא משתמשים בשיטה הזו כי היא דורשת הרבה זמן CPU.

שאלה שצריך לתת עליה את הדעת- כמה מקום זכרון להקצות לתהליך. אם התכנית היא קבועה וכמות הזכרון שהיא דורשת לא תשתנה, זה פשוט, מקצים את מה שהיא צריכה. אם במהלך הריצה זה יכול לגדול (אם מקצים זכרון ב-heap באופן דינמי) אז צריך לנהל את זה. אם בקוביה הסמוכה יש חור, תמיד אפשר לתת לתכנית שרצה עוד זכרון. אם אין מקום פנוי, אפשר או להקצות לה מקום לא בסמיכות לאיפה שהיא רצה, או להוריד תהליך כלשהו לדיסק ולפנות מקום. אם אזור ה-swapping בדיסק מלא, התהליך יכנס לבלוק עד שיתפנה מקום. באופן כללי, אם תהליך מאפשר הקצאת זכרון דינמי, מראש רצוי להקצות אקסטרה כדי לא לתקוע את התהליך בזמן ריצה (אבל במקרה כזה, אם עושים swapping במהלך הריצה, צריך לעשות את זה בלי האקסטרה, אחרת אנחנו מבזבזים מקום בזכרון שאפשר לנצל).

ניהול של זכרון פנוי

כשזכרון מוקצה באופן דינמי, מערכת ההפעלה חייבת לנהל אותו. יש שתי דרכים להתחנות אחר זכרון פנוי: bit maps ורשימות מקושרות. בשיטת הביטים, הזכרון מחולק ליחידות הקצאה ולכל יחידה יש ביט תואם במפת הביטים (ראה איור). אם היחידה פנויה הערך של הביט הוא 0 ואם היא תפוסה 1 (או הפוך, זה לא באמת משנה). ברור שכל שיחידות ההקצאה יותר קטנות, כך מפת הביטים תצטרך להיות יותר גדולה (זו שאלה מהותית שצריך לעסוק בה בעת הדיזיין).

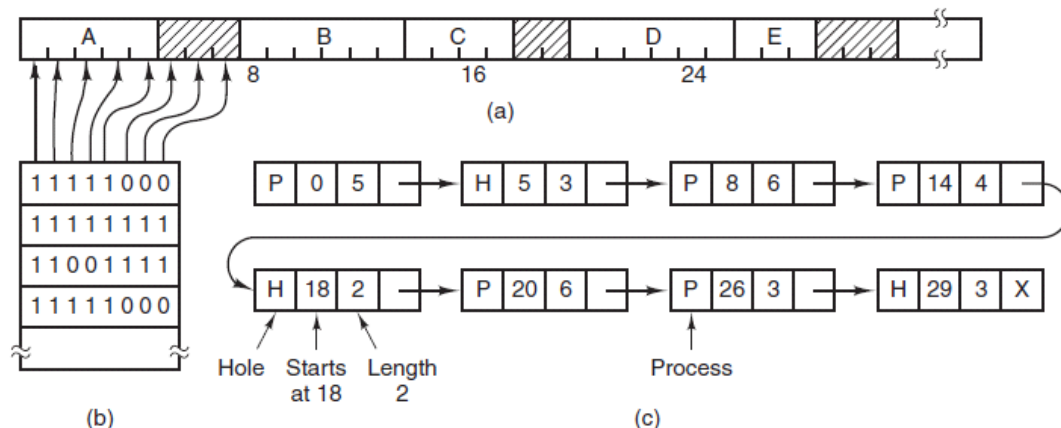


Figure 3-6. (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

הבעיה היא שכשצריך להעלות תהליך שדורש k יחידות זכרון מנהל הזכרון צריך לרוץ על המפה ולמצוא k ביטים רציפים של זכרון פנוי. ריצה כזו לוקחת זמן ועל כן זהו חיסרון של שימוש בשיטת מפת הביטים.

בשיטת הרשימה המקושרת - כל node בליסט מייצג חור (מקום פנוי) - H או תהליך, כלומר מקום שתפוס על ידי תהליך - P. כל node כזה מכיל את הכתובת שבו הוא מתחיל, האורך שלו ופוינטר לאיבר הבא. ברגע שיש swapping (או שתהליך פשוט מסיים את הריצה) עדכון הרשימה זה דבר פשוט, כיאה לרשימה מקושרת. לתהליך שסיים יש שני שכנים שיכולים להיות או process או hole ועל כן יש ארבע אפשרויות:

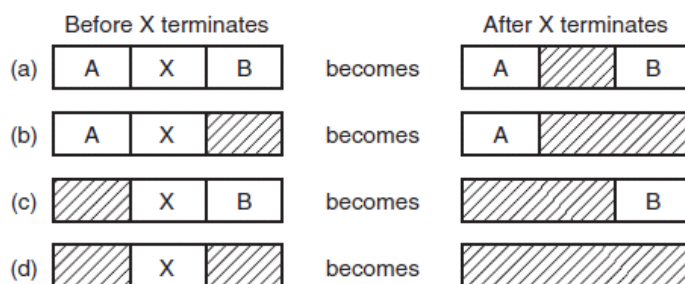


Figure 3-7. Four neighbor combinations for the terminating process, X.

אם ליד יש מקום שהתפנה, פשוט ממזגים את שני הסלטים. אגב, כדי לעשות את זה נכון יותר לנהל את זה ברשימה מקושרת דו כיוונית ולא חד כיוונית כמו באיור הקודם.

כשמשתמשים ברשימה מקושרת, יש מספר אלגוריתמים שבהם אפשר למצוא את המקום הפנוי המתאים לתהליך שמתחיל לרוץ:

first fit

האלגוריתם הכי פשוט - מנהל הזכרון רץ על הרשימה עד שנמצא סלוט מספיק גדול בשביל התהליך החדש. הסלוט (H) מחולק אז לשני חלקים - החלק שדרוש לתהליך והחלק שנותר פנוי (למעט המקרה הלא סביר שמדובר בסלוט בדיוק בגודל הנחוץ). זה אלגוריתם מהיר כיוון שהוא מחפש הכי מעט שאפשר.

שינוי קטן של האלגוריתם הזה הוא **next fit**, שזהה ל-first fit למעט העובדה ששומרים את המקום האחרון שמצאנו ומתחילים לרוץ ממנו פעם הבאה שצריך למצוא מקום פנוי. לא ממש ברור למה זה יותר טוב שהרי יכול להיות שהתהליך הבא שנצטרך להקצות לו זכרון צריך פחות זכרון וזה קיים לפני המקום שבו עצרנו בפעם האחרונה... מה גם שסימולציות בייסאניות מראות שהאלגוריתם הזה נותן ביצועים מעט פחות טובים..

best fit

רץ על כל הרשימה ולוקח את הסלול הכי קטן שמתאים למה שמחפשים. ברור שהאלגוריתם הזה איטי יותר מה-first fit. זה לא מפתיע, מה שכן מפתיע, שגם מבחינת מקום הוא פחות יעיל, כי מסתבר שהוא ממלא את הזכרון בהמון חורים קטנים שאי אפשר להשתמש בהם אח"כ. כלומר, בממוצע האלגוריתם הקודם משאיר חורים גדולים יותר.

כדי לפתור את זה, העלו גם את האפשרות של **worst fit**, שאומרת שנמצא את הסלול הכי גדול בזכרון כדי גם להשאיר אח"כ חור מספיק גדול. סימולציות מראות שגם האלגוריתם הזה לא מזהיר..

ניתן לשפר את הביצועים של כל ארבעת האלגוריתמים על-ידי שמירת שתי רשימות נפרדות לתהליכים וחורים. המחיר של זה הוא מורכבות יותר גדולה והתעסקות יותר גדולה כשסלול בזכרון מתפנה- אז צריך להוריד אותו מרשימה אחת ולהעביר לאחרת וכל הפוך.

quick fit

כאן שומרים רשימה נפרדת לגדלים הנדרשים בשכחות הכי גבוהה. כאן הביצועים הרבה יותר טובים, אבל יש את המורכבות של להעביר סלולים מרשימה לרשימה.

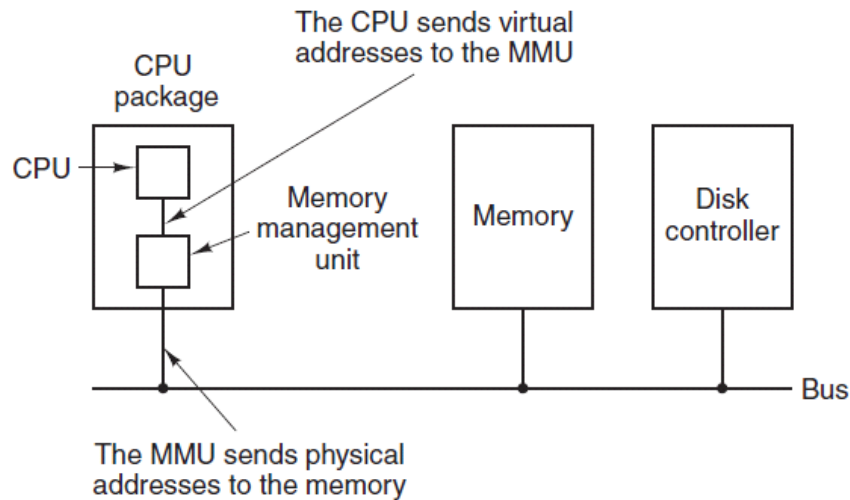
Virtual Memory

בעוד שהרגיסטים בייס ולימיט מאפשרים ליצור הפשטה של מרחב כתובות, יש בעיה נוספת שהם לא עונים עליה, והיא שאמנם כמויות הזכרון הזמין גדולות כל הזמן, גדלי התוכנות (כלומר הזכרון שהן דורשות) גודלם בקצב יותר מהיר (מונח שנקרא **bloatware**). כלומר, יש צורך להריץ תכניות שהן גדולות מדי בשביל להתאים לזכרון הפיזי וגם אם תכנית אחת מספיק לה הזכרון הפיזי, בודאי שיש צורך להריץ כמות גדולה של תכניות שיחד הזכרון הפיזי לא מספיק להן. swapping זה לא פתרון מספיק, כיוון שמהירות ההעברה לדיסק לרוב היא בערך כמה מאות מגה ביט לשניה, מה שאומר שלהעביר תכנית בגודל של 1 גיגה לזכרון ייקח כמה שניות ועוד כמה שניות להחזיר אותה. הבעיה הייתה קיימת כמעט מאז תחילת עידן המחשבים, כי תמיד היו תחומים שבהם היה צריך להריץ תוכנות שגדולות יותר מהזכרון הפיזי הזמין (מדע, הנדסה וכד').

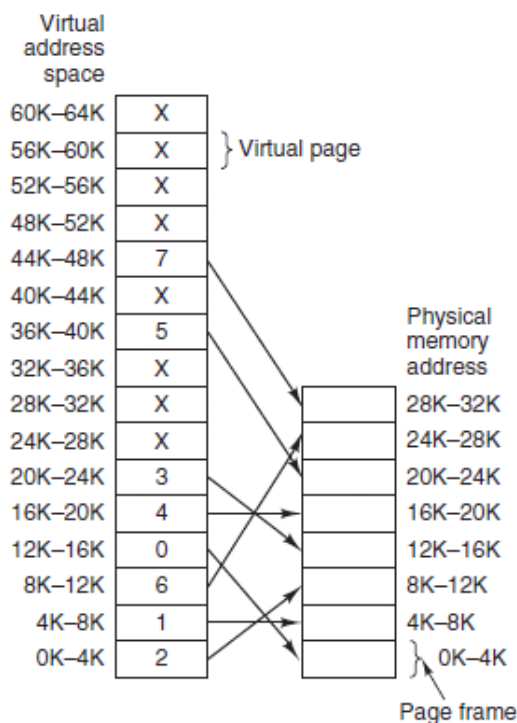
הפתרון שאומץ עוד בשנות ה-60 היה לחלק את התכנית לחתיכות קטנות שנקראות **overlays**. כשתכנית הייתה מתחילה לרוץ, חתיכה אחת הייתה עולה שהיא ה-overlay manager, והחתיכה הזו הייתה טעונת את חתיכה 0 בתכנית וכך זה היה רץ בהדרגה. ניתן היה להריץ חתיכה ליד החתיכה הקיימת או עליה, תלוי בצורך ונעשה גם שימוש ב-swapping בחתיכות שלא בשימוש. את כל זה היה מנהל ה-overlay manager. בהתחלה ה-swap עצמו היה נעשה על ידי מערכת ההפעלה, אבל החלוקה עצמה של התכנית לחתיכות הייתה צריכה להעשות על ידי המתכנת, וזו הייתה עבודה לא קלה שמצריכה זמן ורגישה לטעויות. לא עבר הרבה זמן עד שגם את התפקיד הזה העבירו למערכת ההפעלה.

הפתרון בסופו של דבר היה מה שאנחנו קוראים לו זכרון וירטואלי. המשמעות היא שלכל תכנית יש את מרחב הכתובות שלה שמחולק לחתיכות קטנות שנקראות **pages**. כל page הוא טווח רציף של כתובות. הפייג'ים ממופים לזכרון הפיזי, אבל הנקודה החשובה היא שלא כל הפייג'ים חייבים להיות על הזכרון הפיזי באותו זמן כדי שהתכנית תרוץ. כשהתכנית פונה לחלק במרחב הכתובות שלה שנמצא על הזכרון הפיזי, החומרה עושה את המיפוי הדרוש. כשהתוכנה פונה לכתובת שלא נמצאת בזכרון הפיזי, זה נכנס לבלוק והמערכת ההפעלה מקבלת אינטארפט שיש צורך להביא את התחיקה החסרה ולהריץ מחדש את הפעולה שנעצרה. למעשה זכרון וירטואלי זו הכללה של אותו הרעיון שיושם באמצעות הרגיסטרים limit ו-base. כמובן שבזמן שתהליך אחד מחכה להעלאה של זכרון, ה-CPU יכול לקדם תהליכים אחרים.

אם כך רוב מערכות הזכרון הוירטואלי משתמשות בשיטת ה-paging. כשתכנית פונה לכתובת מסויימת, זוהי למעשה כתובת וירטואלית והיא חלק ממרחב הכתובות הוירטואליות. אגב, גם במחשבים בלי מערכת זכרון וירטואלי, זה לא שונה, פשוט אותה כתובת נשלחת ישירות ל-memory bus ומשם נעשה שימוש ישיר בזכרון הפיזי. כשמשתמשים בזכרון וירטואלי, הכתובת הוירטואלית לא עוברת ישירות לבאס, אלא ל-MMU - Memory Management Unit, והיחידה הזו היא שממפה את הכתובת לכתובת בזכרון הפיזי.



אגב, כאן ניתן לראות שה-MMU הוא חלק מה-CPU, אבל זה לא הכרח המציאות. זו יחידה נפרדה ויכולה בעקרון לשב גם במקום אחר, ובעבר זה אכן היה כך.



אם כך, איך זה עובד בפועל? כאן אפשר לראות מחשב שמייצר כתובות וירטואליות של עד 16 ביט, כלומר מ-0 עד 64K. אולם, אפשר לראות שהזכרון הפיזי של המחשב הוא רק 32K (כלומר, ניתן לשלוח כתובת בגודל 32 ביט, מה שאומר שמרחב הכתובת הרבה יותר קטן). המשמעות היא שתכניות בנפח של 64K יכולות לרוץ, אבל הן לא יכולות להיות לשבת במלואן בזכרון הפיזי. אבל זה כן צריך להיות בדיסק, כדי שאפשר יהיה להעלות את החתיכות החסרות בעת הצורך בזמן ריצה. מרחב הכתובות הוירטואלי בנוי מיחידות בגודל קבוע שנקראות **pages** (כאמור. היחידות התואמות בזכרון הפיזי נקראות **page frames**).

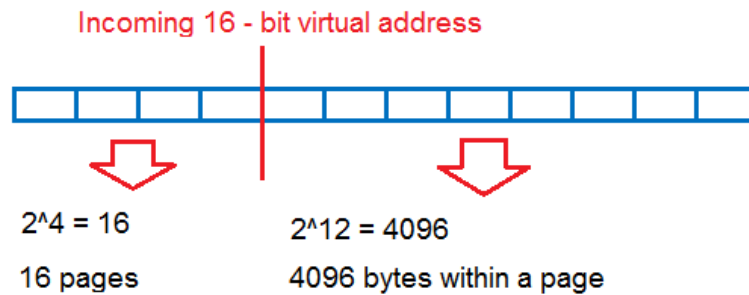
הפייג'ים וה-page frames הם באותו גודל (בשיעור עם יגאל דיברנו על כך שזה לא חייב, אבל לרוב זה כך), בדוגמא הזו (ולרוב במחשבים של 32 ביט) הן בגודל של 4K. המעברים בין ה-RAM לדיסק הם תמיד בפייג'ים שלמים. הסימון באיור מראה שהפייג' הראשון (או ה-page frame הראשון) הוא מ-0 ל-4095, השני מ-4096 עד 8191 וכו'. ברגע שהתכנית קוראת למשל לכתובת 100, הכתובת הוירטואלית 100 נשלחת ל-MMU אשר רואה שהכתובת נופלת בפייג' 0, ושהמפוי שלה הוא ה-2 של page frame (8,192 - 12,287) ועל כן הוא מעביר ל-bus את הכתובת 8192. הזכרון לא "יודע" דבר על ה-MMU, הוא רק רואה בקשה לכתוב לקרוא בכתובת 8192 ומטפל בה כרגיל.

שיטת המיפוי הזאת עדיין לא פותרת את הבעיה שמרחב הכתובות הוירטואלי גדול ממרחב הכתובות הפיזי בזכרון. ניתן לראות שהפייג'ים שמסומנים ב-X לא ממופים. בחומרה עצמה יש ביט שיועד איזה פייג' ממופה לזכרון פיזי ואיזה לא (**present/absent bit**).

מה קורה כשתכנית פונה לכתובת שלא ממופה?

ה-MMU מבחין בכך שהפייג' לא ממופה וגורם ל-CPU לאותת (ללכוד) למערכת ההפעלה. האיתות הזה נקרא **page fault**. במקרה כזה מערכת ההפעלה לוקחת איזשהו page frame שנמצא בשימוש מועט ומעתיקה את התוכן שלו לדיסק (אם זה עדיין לא מועתק לדיסק). בשלב הבא זה לוקח את הפייג' מהדיסק שהתכנית קראה לו ושם אותו ב-page frame ששוחרר, משנה את מפת הביטים או הרשימה המקושרת מאפשר לתכנית להמשיך לרוץ.

מבחינת ה-MMU, כשמגיעה כתובת, ה-4 ביט הראשונים שלה מסמנים את האינדקס של הפייג' (כלומר במקרה הזה יש לנו סך הכל 16 פייג'ים כמו שראויים באיור). וה-12 ביטים הנותרים מאפשרים לגשת לכל מרחב הכתובות שיש בתוך פייג' אחד - 4096 (אמרנו שגודל הפייג' בדוגמא שלנו זה 4K).



איור:

הכתובת שנשלחת מהפרוסס היא 8196 (שזה 001000000000100 בבינארי). ה-4 ביטים הראשונים הם האינדקס ב-page table, במקרה הזה- 2. התא ב-page table נותן שני דברים- ביט אחד בו הו-ה absent/ present bit. אם הוא 0, חל trap של מערכת ההפעלה שצריכה להביא את ה- page frame הרלוונטי מהדיסק. במקרה שלנו הוא 1. בנוסף זה מכיל את שלושת הביטים הראשונים של הכתובת הפיזית בזכרון (כלומר, מאיפה מתחילים את האופסט ב-page frame), במקרה הזה- 110. 12 הביטים האחרונים מהכתובת שנשלחה מה-process מועתקים כמו שהם ויחד מקבלים את הכתובת הפיזית בעלת 15 ביטים, כאן קיבלנו את הכתובת 24580 והכתובת הזו היא שהולכת ל-memory bus להגיע לזכרון הפיזי.

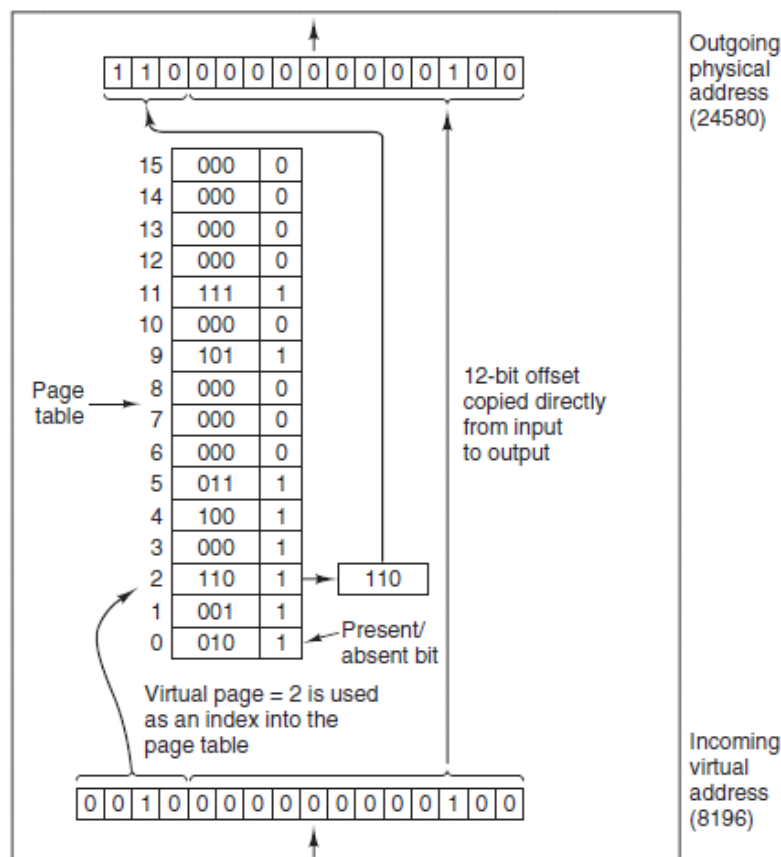


Figure 3-10. The internal operation of the MMU with 16 4-KB pages.

אם כך, תפקיד ה-page table הוא למפות כתובות וירטואליות ל- page frames. במילים אחרות ה-page table היא פונקציה שמקבלת את הכתובת הוירטואלית כפרטמט ומחזירה מספר frame בזכרון הפיזי.

נמקד עכשיו את הדיון לכניסה ספציפית אחת בתוך ה- page table. כאן נית לראות מודל של כניסה אחת:

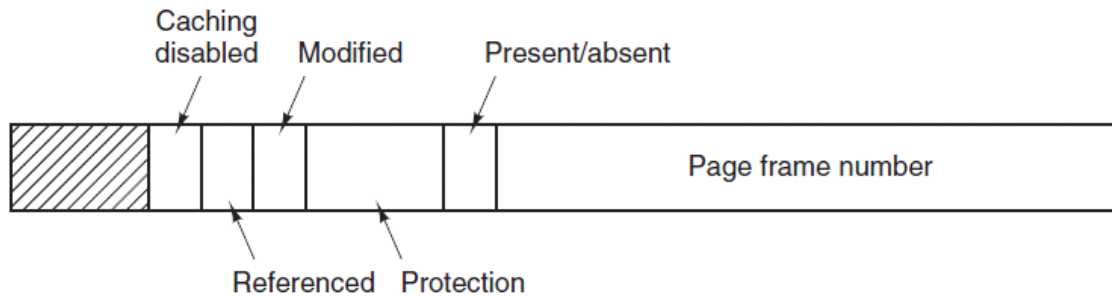


Figure 3-11. A typical page table entry.

הגודל יכול להשתנות, 32 ביט זה גודל סטנדרטי. החלק הכי חשוב זה החלק הימני- מספר ה-page frame. בסופו של דבר המטרה היא למפות לזכרון הפיזי. בנוסף יש לנו את ה-present/absent bit, כפי שאמרנו, אם הערך שלו הוא 1, ניתן להשתמש ב-frame, אין צורך בקריאה למערכת ההפעלה. אם הערך הוא 0, זה גורם ל-page fault. הביטים של ההגנה אומרים איזה סוג של הרשאה הזכרון הפיזי מכיל (במקרה הפשוט או שזה read only או שזה read/write). ה-modified וה-referenced שומרים מידע על השימוש ב-page. כשכותבים לתוך page, ה-modified אוטומטית משתנה (על-ידי החומרה) ל-1. המידע הזה חשוב כשצריך להוריד page לדיסק בתהליך של swapping. אם הוא שונה, יש צורך לעשות העתקה מחדש, אם לא אפשר פשוט לעשות שימוש, כי מה שהיה בו כבר שמור בדיסק. הביט הזה נקרא לפעמים ה-dirty bit (דרטי, במידה וכן הייתה מודיפיקציה). ה-referenced רק מציין האם פנו אל ה-page הזה, בין אם זה לכתיבה או לקריאה. המידע הזה נחמק כל כמה זמן והוא משמש לסטטיסטיקה-כשצריך להחליט איזה פייג' להוציא לדיסק כדי לפנות מקום, רצוי להוציא פייג'ים שבפרק זמן האחרון לא הייתה אליהם כל פניה. הביט האחרון מאפשר לנטרל את האופציה ל-caching- לא ברור למה זה נחוץ, להסתכל בסיכום שיעור.

באינפלמנטציה של מערכת paging יש שתי בעיות עיקריות שצריך להתמודד איתן:

1. המיפוי בין הזכרון הוירטואלי לזכרון הפיזי צריך להיות מהיר, כדי לא ליצור צוואר בקבוק. כל פקודה יכולה להכיל כמה פניות לזכרון (למשל הפקודה $x+y$ כוללת פניה לשתי כתובות זכרון).
2. אם המרחב הוירטואלי גדול, זה אומר שאנחנו צריכים להחזיק איפשהו טבלה מאוד גדולה. במחשבים מודרניים מרחב זכרון וירטואלי של 32 ביט זה נורמה, ומרחב זכרון וירטואלי של 64 ביט גם כן הופך לנפוץ (למעשה היום, כמה שנים טובות אחרי שטנגבאום כתב את הספר, זה נכראה הנורמה כבר). 32 ביט של כתובות זו טבלה עם מיליון פייג'ים, 64 זה כבר הרבה יותר.. נזכור גם שלכל תהליך יש טבלה משלו.

בגדול, המשמעות היא שאנחנו צריכים לשפר את הביצועים של מערכת ה-paging.

Translation Lookaside Buffers

באופן כללי, הפתרון שמתמודד עם המהירות מתבסס על ההבנה שרוב התכניות עושות מספר רב של קריאות זכרון למספר מועט של פייג'ים. כלומר, ברוב הזמן לא נעשה שימוש בכל מרחב הזכרון, אלא רק בפייג'ים ספורים.

הפתרון- דיבייס חומרתי שממפה כתובות וירטואליות לכתובות פיזיות בלי לגשת ל-page table, והוא נקרא **TLB- Translation Lookaside Buffer**. הוא יושב לרוב ב-MMU וכולל מספר מועט של כניסות, באיור ניתן לראות שיש לו 8 כניסות, לרוב זה לא עולה על 256 (איציק דיבר על כך שלא יותר מ-1000).

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figure 3-12. A TLB to speed up paging.

הטבלה המוקטנת הזו כוללת את המידע הבא (שתואם אחד לאחד למה שנמצא ב-page table, למעט הפייג' הוירטואלי, שלא נחוץ ב-page table):

הפייג' הוירטואלי, הפריים הפיזי, המידע על הרשאות (קריאה או קריאה/כתיבה), האם הפייג' שונה לאחרונה בסייקל האחרון והאם הפייג' בשימוש או לא.

אם כן, איך ה-TLB עובד?

כשה-MMU מקבל כתובת וירטואלית לגשת אליה, הרכיב החומרתי קודם כל בודק האם הפייג' הוירטואלי שממנו נשלחה הכתובת נמצא ב-TLB. אם כן (ואין הפרה של הרשאות), הכל נעשה כפי שראינו עד עכשיו, אבל בלי לגשת ל-page table.

אם הפייג' הוירטואלי לא נמצא ב-TLB, ה-MMU מפנה את אחת הכניסות ב-TLB ושם במקומה את הכניסה הרלוונטית מה-page table. כלומר, זה כמו עוד page table, רק יותר קטן, מהיר ונגיש.

על פניו נראה שיש פה הרבה יותר סיכוי ל-TLB fault (כלומר, שהפייג' הרלוונטי לא יהיה בתוך הטבלה המוקטנת הזו), אבל מסתבר שאפילו 64 כניסות מוריד את אחוז הפספוסים מספיק כדי שכל העסק יתנהל בצורה יעילה.

מעבר לכמות הכניסות, מערכת ההפעלה יכולה לבצע את ההחלפות בצורה חכמה מה זה אומר? היא יכולה "לנחש" הרבה פעמים איזה פייג'ים יהיו בשימוש בקרוב ולהביא אותם לתוך הטבלה לפני שיש TLB fault. למשל, כשבתוך תהליך נשלחת הודעה לשרת, רוב הסיכויים שהשרת יצטרך לרוץ בקרוב, ולכן אפשר להביא לתוך הטבלה את הפייג'ים שלו.

יש שני סוגים של פספוסים כשמשתמשים ב-TLB:

soft miss - כשהפייג' לא ב-TLB אבל כן בזכרון (כלומר שאם הייתי פונה אל ה-page table, לא היה page fault). במקרה כזה הטיפול הוא יחסית מאוד מהיר, מדובר בכמה עשרות פקודות.

hard miss - כשהפייג' עצמו לא בזכרון ויש להביא אותו מהדיסק. פספוס כזה יכול להיות פי מיליון (!!) יותר איטי מ-soft miss (וזה יכול לקחת כמה מילישניות לטפל בזה).

למעשה יש עוד מצב באמצע והוא שהפייג' לא נמצא בזכרון של התהליך שקרא לו, אבל כן נמצא בזכרון של תהליך אחר, מה שאומר שלא צריך ללכת לדיסק כדי להביא אותו.

החיפוש עצמו ב-page table נקרא **page table walk**.

פתרון ה-TLB כשלעצמו עדיין לא מספיק כדי להתמודד עם מרחבי זכרון וירטואלים מאוד גדולים.

Multi - level page tables

על איזו בעיה אנחנו מדברים?

כתובת של 32 ביט מכילה 12 ביט אופסט שמועבר כפי שהוא ו-20 ביטים נוספים של הכתובת אותה אנחנו מחפשים. 20 ביטים זה אומר 2^{20} כתובות - כלומר 1,000,000 כניסות ב-page table. כל כניסה בטבלה תופסת כ-4 בייט (20 ביטים שהם ה-page frame עצמו ועוד כמה ביטים לאינפורמציה חשובה כגון האם הפייג' שונה, האם פנו אליו, האם הוא קיים בזכרון או שצריך להביא אותו מהדיסק - כל הדברים שדיברנו עליהם (איור 11 למעלה).

אז מיליון כניסות כפול 4 בייט נותן לנו 4MB.

לא נשמע הרבה מדי, אבל זה נהיה בהחלט הרבה כשזוכרים שכל תהליך (כל תכנית) צריכה page table משלה. כלומר, אם רצות 100 תכניות במקביל (מה שקורה כל הזמן) אנחנו מדברים על 400MB רק על טבלאות שאומרות לנו איפה ללכת לחפש בזכרון את המידע הנחוץ לתכנית.

הרעיון כאן הוא להמנע משמירה של כל ה-page tables בזכרון כל הזמן. במילים אחרות, הטבלאות שלא עושים בהן שימוש, לא צריכות להיות כל הזמן בזכרון.

הפתרון הראשון הוא שימוש בטבלאות פייג'ים שיש להן סדר הירארכי.

באיור ניתן לראות מרחב כתובות של 32 ביט שמחולק באופן הבא:

10 ביטים PT1 (כלומר טבלת פייג'ים ברמה הראשונה), 10 ביטים PT2 (טבלה ברמה הבאה) ו-12 ביטים אופסט.

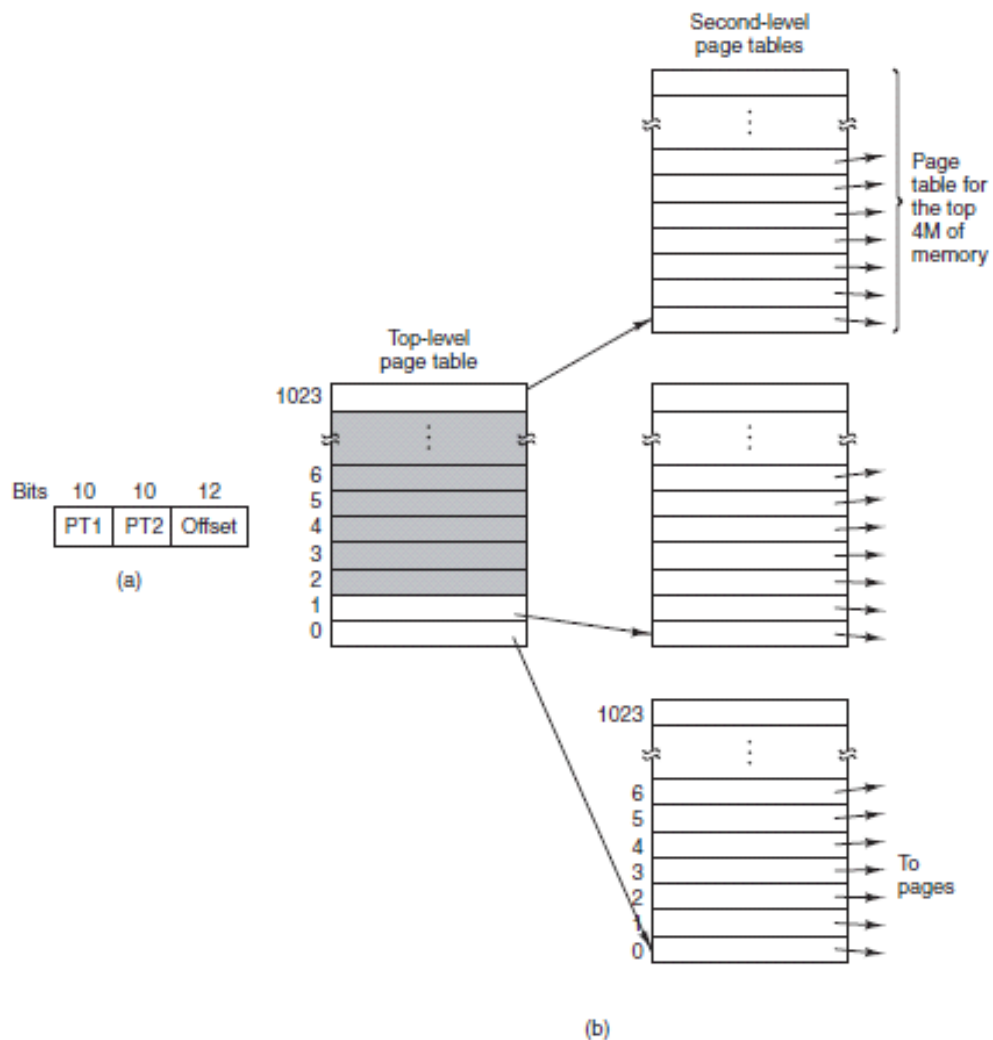


Figure 3-13. (a) A 32-bit address with two page table fields. (b) Two-level page tables.

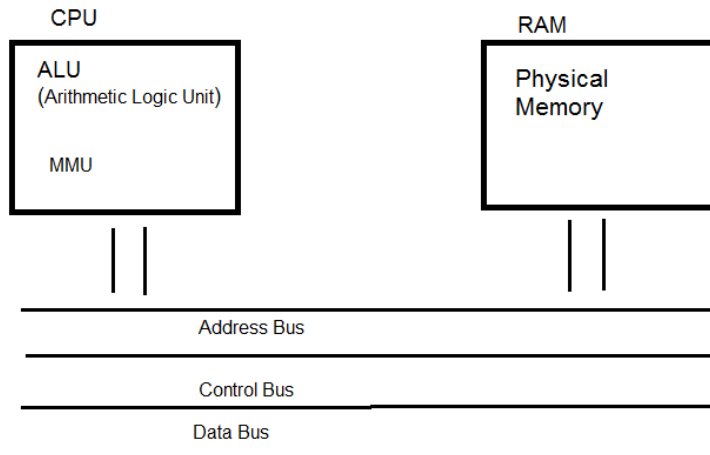
בגדול, הרעיון לא שונה מפתרון הזכרון הוירטואלי שמצאנו עבור תוכניות שצריכות יותר זכרון משבאמת יש לנו זכרון פיזי. הרעיון הוא בכל רגע נתון להחזיק בזכרון רק את מה שבאמת צריכים, ובמידת הצורך להביא מהדיסק את מה שחסר. אותו דבר כאן. אנחנו חייבים להחזיק בזכרון את הרמה הראשונה של ה-page tables. הטבלה הזו יכולה להצביע ל-page table אחר, אבל היתרון במבנה כזה הוא שה-page table ברמה הבאה יכול למעשה לשבת גם בדיסק, ולא צריך לתפוס מקום יקר בזכרון. הטבלה ברמה הראשונה מכילה 1024 כניסות, ועל כן היא מצביעה על 1024 טבלאות אחרות. אבל ברמה השנייה, כבר לא כל הטבלאות חייבות לשבת בזכרון עצמו, חלקן יכולות לשבת גם בדיסק, כי אני תמיד אדע להביא אותן על-ידי הטבלה הראשונה. כשיגשים לאינדקס בטבלה הראשונה הזו, מקבלים את הכתובת של הטבלה הבאה שנחוצה כדי לעשות את המיפוי האמיתי.

חשוב להבין- עבור כל תכנית כדי לעשות מיפוי אמיתי לזכרון הפיזי אני חייבת לפחות 2 טבלאות בזכרון- הטבלה ברמה הראשונה והטבלה שהיא מצביעה אליה- ביחד זה נותן לי את הכתובות שה-MMU ישלח לזכרון.

איך יודעים לאיזה טבלאות לגשת?

כאמור, ה-10 ביטים הראשונים נותנים את האינדקס לטבלה הראשונה. אותו האינדקס מצביע לטבלה השנייה וה-10 ביטים הבאים נותנים את האינדקס בטבלה השנייה. חיבור האינדקסים בשתי הטבלאות מאפשר לעשות את התרגום לכתובת הפיזית, ה-page frame. בפועל מחזיקים לא רק 2 טבלאות אלא 4: כניסה 0 בטבלה הראשונה מצביעה על ה-text של התכנית, כניסה 1 על ה-data וכניסה 1023 מצביעה על ה-stack. כל השאר לא באמת נמצאות בזכרון. ניתן לשלוף אותן מהדיסק במידת הצורך. זה אומר אגב שה-present/absent ביט ב-1021 כניסות הנותרות של הטבלה הראשונה הוא 0, מה שיגרום ל-page fault כשיקראו להן.

את אותו הרעיון אפשר למעשה לישים גם עם 3, 4 או כל מספר אחר של שכבות (של טבלאות).



ב-CPU יש את ה-ALU שעושה את החישובים כשמגיעות פקודות ואת ה-MMU (ועוד כמה דברים). ה-CPU מתקשר עם הזכרון הפיזי, ליתר דיוק ה-MMU.

ה-MMU מתרגם כתובת וירטואלית לכתובת פיזית ומעביר אותה דרך ה-address bus.

הוא גם מעביר את המידע של מה רוצים לעשות- לקרוא או לכתוב. אם זה לקרוא, אז שמים את המידע שיש בכתובת (הפיזית) ומעבירים חזרה דרך ה-data bus.

אם כך, כשרוצים לקרוא מידע ולעשות איתו משהו, איך זה מתבצע? ל-CPU יש סט של פקודות מכונה שאומרות מה לעשות עם המידע- לקרוא, לכתוב, להזיז וכו'.

למשל:

add add1. add2

זה אומר שצריך לבצע פעולה חשבונית (חיבור) על שני משתנים. הפעולות מתבצעות על רגיסטרים. קוראים כתובת ראשונה לתוך רגיסטר (רגיסטר זה תא נפרד ב-CPU), כנ"ל עם כתובת השניה ואז מבצעים פעולה חשבונית (ה-ALU). כל מידע עובר דרך ה-data bus וזה מתבצע ברמת החומרה.

כלומר, ה-CPU יודע לקבל מידע מהזכרון.

אני רוצה לדבר עם עכבר או עם מסך.

תנאי ראשון- צריך חיבור פיזי להתקן. בשביל לתקשר עם הדיבייסיס- הם גם צריכים לשבת על ה-bus. ה-bus יושב על לוח האם. העכבר יושב מבחוץ, אבל בדרך יש מתווך שמחובר ל-bus.

USB אגב זה דיבייט בפני עצמו.

בין המסך ל-bus יש מעין כבל RGB ויש שם גם כרטיס גרפי (שנמצא לרוב על לוח האם).

כל התקן כזה הוא I/O- אפשר לקבל קלט ולשלוח פלט. דיסק הוא גם I/O.

איך כל הדיבייסיס יודעים לדבר עם המחשב- יש להם קונטרולר- מעין כרטיס בפנים (device controller). הקונטרולר לא יושב על ה-bus (בשונה מכרטיס מסך) אלא מעבר ל-USB. זה אלקטרוניקה, במקרים רבים זה ממש מחשב קטן.

כדי לתקשר עם דיבייט, צריך interface- צריך לדבר ברמה אבסטרקטית, כלומר צריך איזשהו API. בד"כ זה סדרת רגיסטרים חומרתיים - על ידי כתיבה לתוך הרגיסטרים אני יכולה לתת פקודות ואז הדיבייט קונטרולר לוקח את הערך ששמתי ולעשות עם זה משהו. ואפשר גם לקרוא. מי שבונה את ההתקן, מגדיר את סט הרגיסטרים שאליהם אפשר לקרוא ולכתוב ומגדיר מה צריך לעשות כדי שפעולה מסוימת תקרה- כלומר זה מעין API חומרתי.

כיוון שצריך לתקשר מול הרגיסטרים, אני צריכה שתהיה לי האפשרות לכתוב ולקורא ביניהם (רגיסטר זה תא זכרון לכל דבר). לתקשר איתו זה אומר שאני צריכה לשלוח כתובת של הרגיסטר ולשים את הדאטא ולומר לו- תקרא. או להפך, להקיד שאני רוצה לקרוא. כלומר, בשביל לעבוד עם ההתקן, האם אני צריכה להבין את הרגיסטרים של כל התקן? כמובן שזה בעייתי. כלומר, אנחנו צריכים משהו שיתווך ביני לבין הקונטרולר של ההתקן. מי שמבין איך לעבוד עם הדיבייסיס ברמה האבסטרקטית- יש לי שכבות- מי יושב בין האפליקציה לחומרה? במערכת ההפעלה יש שכבה של דרייברים- והם מגדירים API. הדרייבר ספציפי לדיבייט שאני עובדת איתו.

לעכבר למשל לא כותבים- הוא צריך לדעת לקרוא מיקום. וזה מה ש-API שהדרייבר של העכבר נותן לי.

דרייבר זה לא חלק אינטגרלי של מערכת ההפעלה אלא זה add on. הוא עובד ב-kernel mode, אבל הוא לא חלק ממערכת ההפעלה, מה שמאפשר להוסיף כמה דיבייסיס שאני רוצה. וגם כאלה שמערכת ההפעלה לא יודעת על קיומם.


דרייבר זה תוכנה (מאוד יעילה). לחילופין driver controller זה רכיב חומרתי. הדרייבר כולל את ה-API כדי לממש את הדיבייט.

כלומר הצורה לתקשר עם התקן זה דרך דרייבר ומעל הדרייבר (ומתחת לאפליקציה) יש ספריות סטנדרטיות c stdlib מי שניגש אל הדיבייט זה הדרייבר דרך system call

איך הדרייבר יודע איזה כתובת לתת לדיבייט? יש כמה אופציות.

כל הטיפול ב-I/O הוא חלק חשוב במערכת ההפעלה. כל העבודה עם הדרייברים- זה מאוד חשוב, אנחנו עובדים כל הזמן עם I/O devices בד"כ דיבייסיס, בגלל שהם צריכים להיות מאוד מהירים, הם עובדים באחת משתי דרכים:

- interrupt
- polling

פקודה lspci 
מקבלים את הרשימה של הדיבייסיס. מה זה PCI?
כל ההתקנים שמחוברים ל-bus.
פקודת lsusb
-USB זה extension של ה-bus.

איך מדברים עם דיבייסיס?

- יש שלושה סוגים:
- block device
- character device
- other

נגיד מקלדת. באיזו רמה המקלדת מתקשרת עם ה-CPU- ברמת התו הבודד. המקלדת גם לא צריכה לשמור דאטא. עכבר כנ"ל. אם נקח את ארבעת החצים שיש לנו על המקלדת- זה למעשה העכבר- קוארדינטה מורכבת מתו ימינה שמאלה למעלה למטה- פשוט הוא עושה את זה מאוד מהר. אם כך, גם העכבר מתקשר ברמת התו הבודד.

מה זה block device? כשמתקשרים מול דיסק- מעבירים chunks של מידע. לכל העברת מידע יש overhead, ויש לזה מחיר CPU. מהמקלדת אני חייבת לקלוט תו תו. אבל אם יש לי אפשרות לעשות את זה בתחכום יותר גדולות- בלוקים- זה יהיה הרבה יותר יעיל.

הקטגוריה השלישית- כמו שעון - לא מעביר מידע. מעביר סיגנלים - הוא לא מעביר דאטא (דיברנו על כך שיש לפעמים עוד דיבייסיים מהסוג האחר, אבל בפועל זה רק השעון).

בבלוקים, אני צריכה להחזיק משהו מעבר לרגיסטרים כדי להעביר כמויות של מידע.

- אם כך מאפיינים של בלוק דיבייסי:
- גישה איטית יחסית (למשל קריאה של בלוק מדיסק לוקחת הרבה זמן. פקודות של CPU הן בנו).
- צריך להעביר מידע כמה שיותר מהר (עושים את זה באמצעות מנגנון DMA)
- אם מבקשים לקרוא בייט מתוך דיסק (או כל פיסת מידע שהיא קטנה יותר מהבלוק), הוא יביא בלוק שלם של מידע. הוא לא יודע לעבוד בבודדים, רק בבלוקים. כלומר, מתוך הדיסק, הוא מביא לפחות סקטור (אבל כאמור היתרון פה הוא היעילות).

- בדיבייסיים של תו בודד:
- אין באפרים
- כל פעולת עכבר גורמת לאינטראפט

דיבייסי קונטרולר- מעגל חשמלי אלקטרוני שיוזע לתפעל את החומרה של הדיבייסי. בתוך העכבר יש כרטיס קטן שיוזע לעבוד עם לייזר, כפתורים וכד'.

כדי להגיע לקונטרולרים (אוסף של רגיסטרים או האפרים) יש לי שתי אופציות:

port mapped i/o map
memory mapped i/o

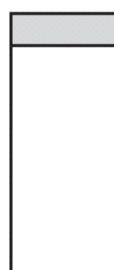
אם לוקחים זכרון של תהליך- אפשר להקטין את ה-RAM כך שלא יהיה עד 4 גיגה אלא נגיד 3.5. מה עושים עם המקום הפנוי- זה כתובת שאני יכולה לחלק לכל הקונטרולרים. זה על חשבון הגיגה של מערכת ההפעלה כי זה ב- kernel mode ה- MMU לא יודע שכשהוא הולך לכתובת מסויימת הוא הולך לדיבייסי- הוא חושב שהוא ניגש ל-RAM. בפועל זה ממפה לתוך הדיבייסי. כלומר ממפים את הדיבייסיים על חשבון מרחב הזכרון. אפשר לעבוד עם פקודות רגילות של זכרון והדרייבר עובד ישירות איתן עם הכתובת (שממפת לדיבייסי). כלומר הכתובות פיזית מגיעות לדיבייסיים. אם כך כל הרגיסטרים או באפרים בדיבייסי ממפה כאילו אני עובדת מול זכרון רגיל עם פקודות רגילות

אפשרות אחרת- לא ממפים. מוסיפים עוד מרחב זכרון וב- control bus מוסיפים עוד קו שנקרא I/O שמים כתובת. איך יודעים לאן ללכת- ל-RAM או לתוך ה-decive? הקו הנוסף אומר לי האם הכתובת היא ב-RAM או ב- i/o space ואז יש חפיפה בכתובות של שני הזכרונות אבל ה-CPU אומר לי לאן זה שייך.

אבל איך בתכנית מבדילים? יש פקודות מיוחדות שנקראות put, get - והן הולכות לזכרון של I/O

אפשרות שניה- לוקחים מהספייס של הזכרון לטובת הדיבייסיים ואז עובדים מולם בפקודות רגילות:

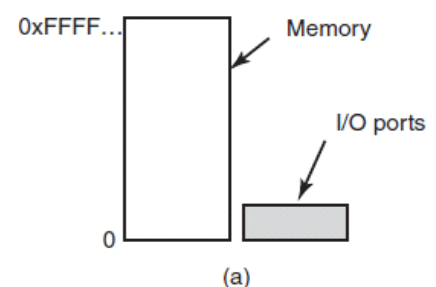
One address space



(b)

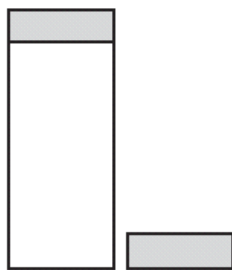
אם כך אפשרות אחת זה נפרד:

Two address



(a)

Two address spaces



(c)

האפשרות השלישית - גם וגם.

למה?

כשעובדים מול הרגיסטרים עדיף לשים אותם במרחב נפרד. **למה?**

אבל אם אני לוקחת באפר דיבייס, כלומר יש לו חתיכות זכרון- אני רוצה לעבוד עם זה בצורה דומה כמו שאני עובדת עם זכרון אחד- במקום לעבוד עם I/O, אני יכולה לשים אותם כחלק מזכרון כללי של המערכת, ואז כשפונים לבאפר של הדיסק, כאילו פונים לזכרון.

זה היתרון. לכן קונטרול בסיסי מול רגיסטרים זה בנפרד והשאר באותו זכרון. זה המודל המקובל היום.

במערכות embedded- בד"כ הזכרון הנפרד מקובל.

עבודת ה-I/O צריכה להיות ברמת ה-kernel שמאפשרות לדרייבר לעבוד. יש כל מיני פקודות כשדרייבר עולה הוא צריך לבקש לתפוס חתיכת זכרון שידועות לו:

```
request_region()
check_region()
release_region()
```

אלא פקודות שהן לא C! עושים אותן באמצעות אסמבלי. לכן הרבה מהקוד של הדרייבר זה אסמבלי.

כשעובדים עם זכרון - יש caching- זה הרבה יותר מהיר, פחות סייקלים כדי להשלים פעולה

מול חומרה לא עובדים עם caching כי כל דבר יכול להשתנות.

זה אומר שאחת הבעיות כשעובדים עם memory map, את האזור שממופה קונטרולרים- צריך להגדיר אותו page non cachible- אסור לסמוך על זה שלא משתנה. **לא ברור** למה אי אפשר לעשות cache

כש-CPU מריץ תכנית הוא שולט בביצועים. כשהוא פונה למערכת ההפעלה, היא מטפלת ויש בלוק. כשהיא מסיימת היא מוציאה מהרדמות. חומרה עובדת עם hardware interrupts

אפשרות אחרת- polling

נתתי לחומרה לעשות עבודה מסויימת. או שאני יודעת שכשהיא תסיים היא תתן לי אינטראפט או שאני צריכה כל הזמן לדגום

מאפיינים של ה-polling mode

- אינטראפט
- לא סינכרוני- האינטראפט יכל לבוא מתי שבא לו
- קוד יותר מורכב- דומה לפעולה של threads יש פה בעיית סינכרון
- חסכוני יותר ומהיר אינטראפט בא לבד

- סינכרוני- אני בודקת את זה כל הזמן
- קוד פשוט- בלופ דוגמים את מה שצריך
- איטי יחסי- צריך כל הזמן לגשת ולשאול
- מערכת הפעלה- כשאני רוצה לקרוא משהו מהדיסק, עושים קריאה מקובץ למשל, התהליך נכנס לבלוק- מה קורה כשעושים scanf? זה פוינקצית ספריה- הספריה רצה ביוזר ספייס והיא מייצרת system call כדי למשוך את תשומת ליבה של מערכת ההפעלה. היא שמה את הפוינטרים ברגיסטר ועושה טארפ למערכת ההפעלה והמספר שכתבתי זו הפקודה שאני מבקשת ממערכת ההפעלה. זה אומר שהתהליך נכנס לבלוק. הוא מחכה לתשובה- נכנס לבלוק. יש קונטקסט סוויץ ויש לזה עלויות. כשהדרייבר נותן פקודה, מי עובד? ה-device controller

אם עובדים ב-polling הדרייבר תופס CPU- הוא צריך להמשיך לדגום (busy wait).

נניח הדיסק סיים את העבודה- כלומר לקח דאטא ושם את זה לבאפר. אם דוגמים אותו, ה-CPU מעלה איזשהו רגיסטר שאומר שזה done. אפשר להעתיק מהבאפר של הדיבייס למרחב הוירטואלי של התהליך ואז אפשר להעביר את התהליך מבלוק ל-ready

אם אני במוד של אינטארפט- מה קורה כשהדיסק סיים את העבודה ונתן אינטראפט?

מה זה אינטראפט? **אחד הקיים בקונטרל באס זה אינטראפט**

אז זה מגיע ל-CPU- הוא מקבל אינטראפט, אם זה לא חסום (נדבר על מה זה) הוא מסיים א הפקודת מכונה הקודמת ואז הולך ובודק מי העיר אותו (יחד עם האנטארפט יש מספר) - המספר אומר לו ללכת לאינטראפט וקטור, שם רשומות רוטינות טיפול באינטארפט- המספר זה האינדקס בוקטור לרשומה שאומרת מה לעשות עם אינטארפט ספציפי. מי שם שם את הרוטינות? הדרייבר. זה פוינטרים לפונקציות. כל דרייבר יודע לטפל באינטארפטים של דיבייסיים אחרים. אז הוא אומר שמספר מסויים זה פוינטר לפונקציה שקשורה לדיבייסי ספציפי.

כלומר, קרה אינטראפט, מי התעורר? הרוטינה הרלוונטית שהדרייבר השתיל לטיפול באינטראפט? באינטראפט הנדלר כתוב להעתיק את מה שהיה בבאפר של הדיסק לתוך היזר ספייס

אינטרפט פלואו:

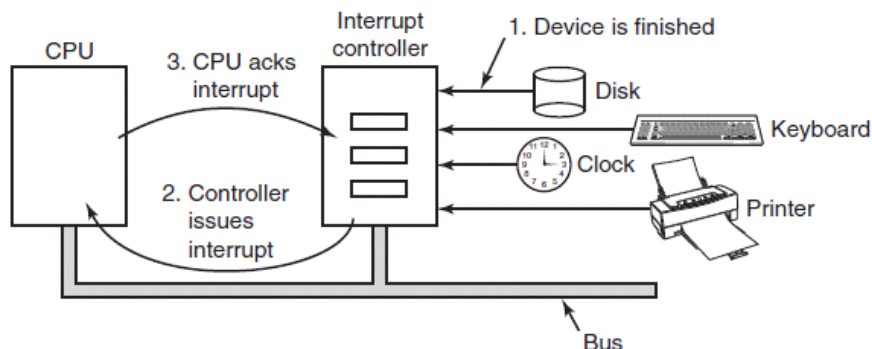


Figure 5-5. How an interrupt happens. The connections between the devices and the controller actually use interrupt lines on the bus rather than dedicated wires.

היום לרוב ה-Interrupt controller יושב ב-CPU תפקידו לזהות מאיפה מגיע האינטראפט. הוא אומר ל-CPU תעצור ונותן לו את מספר האינטראפט. ה-CPU מסיים פקודה ואז בפקודה הבאה הוא הולך ל-Interrupt vector שנמצא בתוך ה-OS ושולף משם את רוטינת האינטראפט. כלומר בתוך הכתובת של מערכת ההפעלה יש טבלה. ובאמצעות אינדקס שולפים את הרוטינה. בעקבות זה הוא עושה acknowledge - הוא אומר לקונטרולר שהוא טיפל - כלומר הוא איפס אותו וזה אומר שעוד אינטראפט יכול להגיע. אחרת, אם לא מידעים את הקונטרולר, מפספסים את האינטראפט הבא. אינטראפט קונטרולר יכול לחסום אינטראפטים. למשל הדיבייס קונטרולר יודע שאסור להפריע לו הוא עושה masking כך שאף אינטראפט לא יכול להגיע.

שקף - interrupt flow(2)

מה קורה למשל כשרוצים להדפיס טרינג?

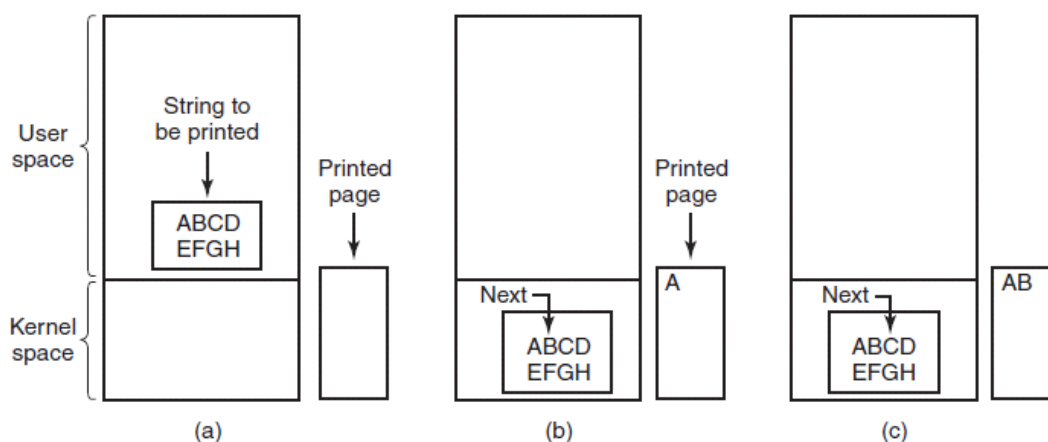


Figure 5-7. Steps in printing a string.

איורים 5-8, 5-9
קוד של פולינג ואינטראפט

DMA - Direct Memory Access

כשקוראים באפר שלם לתוך הדרייבר של הדיבייס צריך להעתיק - ה-CPU צריך לקרוא ממקום אחד ולכתוב במקום אחר - זה בזבז. צריך להעתיק כתובת מהאזור של מערכת ההפעלה, לשים ביזר ספייס ולקדם כתובת. במקום שה-CPU יעשה את זה המציאו את המנגנון של ה-DMA - בנו קונטרולר קטן שמה שהוא יודע לעשות זה לקדם כתובת, אין בו את ה-ALU של המעבד. נותנים לו כתובת של מקור ושל יעד ורוצים להעביר 1000

בייטים, הוא יודע לבד להכניס את הכובות ולקדם. הוא עושה את עבודת ההעברה בלי התערבות של ה-CPU, מה שאומר שה-CPU יכול להתעסק במשהו אחר. ברגע שהוא סיים, מעלה אינטארפט. יש המון פעולות כאלה של העתקות- קריאת פייג' מהדיסק למשל. עם סיום ה-DMA, ה-CPU יודע להעיר את מי שצריך. ה-bus מאפשר לעבוד גם עם ה-CPU וגם עם ה-DMA. הבאס הוא מאוד מהיר, אמנם זה לא באמת במקביל אבל זה מאוד מהיר, ולכן מבחינת הביצועים זה כאילו במקביל.

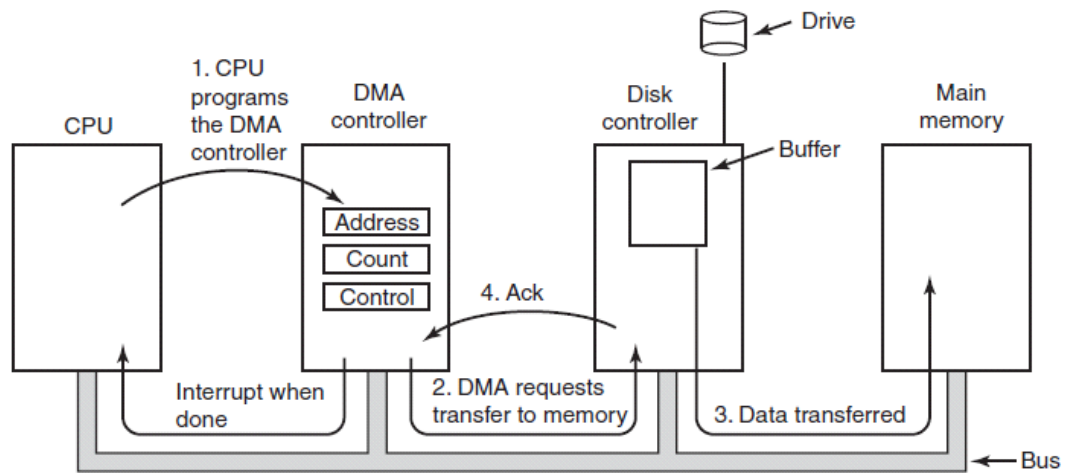


Figure 5-4. Operation of a DMA transfer.

אם כך, ה-DMA זה חומרה שמאפשרת להעביר דאטא ממקום למקום לא על ידי ה-CPU. העבודה מולה גם על ידי אינטארפטים. זה חוסך המון זמן CPU.

קצב העברת נתונים בבאס נקרא רוחב פס. קצב העברת נתונים של PCI גבוה מאוד. PCI זה שם של הבאס שמחבר בין ה-CPU לשאר החלקים במחשב. המשאב המשותף בזמן ש-CPU עובד על משימה אחת ו-DMA עובד על משימה אחרת זה ה-bus ויש מנגנון של ארביטריציה והם יכולים לחלוק את המשאב. איך עובדים: proc/interrupts אפשר לראות את האינטארפטים שהמערכת שלנו יודעת לטפל בזה. אפשר להסתכל ולראות סטטיסטיקה על אינטארפטים ועל איך הם עובדים.

Device Driver Interface

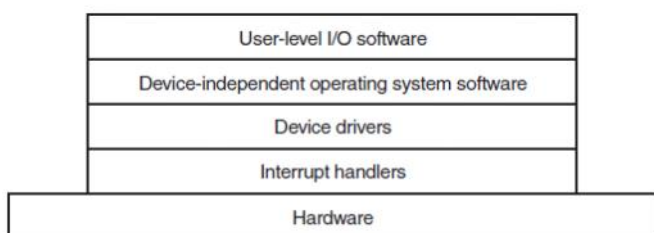


Figure 5-11. Layers of the I/O software system.

מי שיועד לתפעל את הקונטרולים זה הדרייבר.

דיבייס דרייבר ואינטארפט הנדלר זה למעשה אותה יחידה. ההנדלרים זה פונקציות של הדרייבר מעל יש את ה-device-independent operating system software מה שחשובה שזה בלתי תלוי בדיבייס. ההבדל הוא השכבה למטה- בדיבייס דרייבר. גם ללוח אם יש דיבייס דרייבר- כי בכל מחשב זה שונה וכולם צריכים לדעת עם אותה מערכת ההפעלה.

מעל חסר עוד שכבה של ספריות, אבל זה חלק מהיזר ספייס.

לא לגמרי ברור באיזה הקשר האיור הבא, צריך לקרוא בספר:

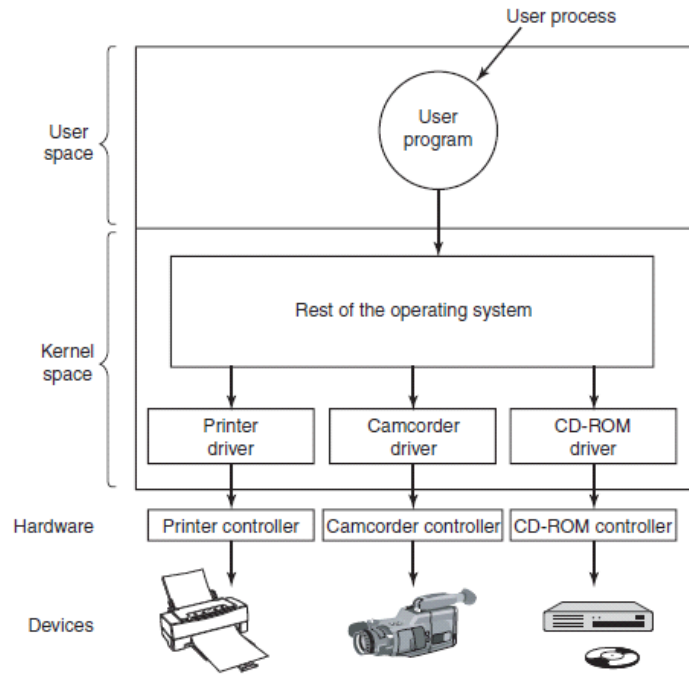


Figure 5-12. Logical positioning of device drivers. In reality all communication between drivers and device controllers goes over the bus.

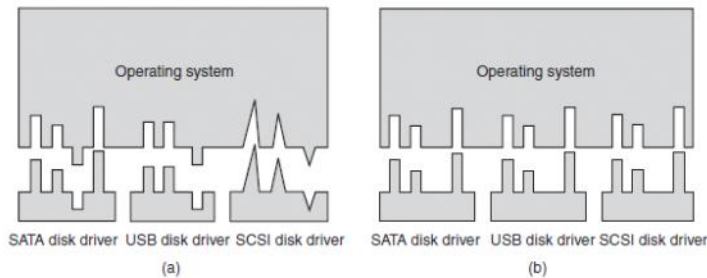


Figure 5-14. (a) Without a standard driver interface. (b) With a standard driver interface.

הדרייברים זה סוג של מערכת הפעלה.
לשים לב שדרייבר ומערכת ההפעלה- קרנדל מוד
אז איך מערכת ההפעלה יודעת לדבר עם הדרייברים השונים?
אנאלוגי לשאלה של למה אני יכולה לעבוד עם כל סטאק שהוא- כי ה-API הוא גנרי. כנ"ל פה.
מצד שמאל באיור API לא גנרי, ימין - גנרי

כשהדיבייס מתחיל לעבוד, ה-interrupt handler נכנס לפעולה.
כל הדרייברים עובדים ב-kernel mode. יש הרבה יותר אפשרויות.
בקרנל אפשר לחסום אינטארפט. לשים לב שזה מאוד מסוכן. למשל בתקשורת, המחיר של זה יכול להיות איבוד נתונים (אם לא הגבתי לאינטארפט של נתונים שנכנסו, זה לא נשמר בשום מקום).
דרייברים- יכול לקרות inverse priority ואי אפשר למנוע את זה. כי המשמעות של למנוע את זה לא לאפשר לדרייברים לחסום אינטארפטים, אבל הדרייבר חייב את האפשרות הזו.

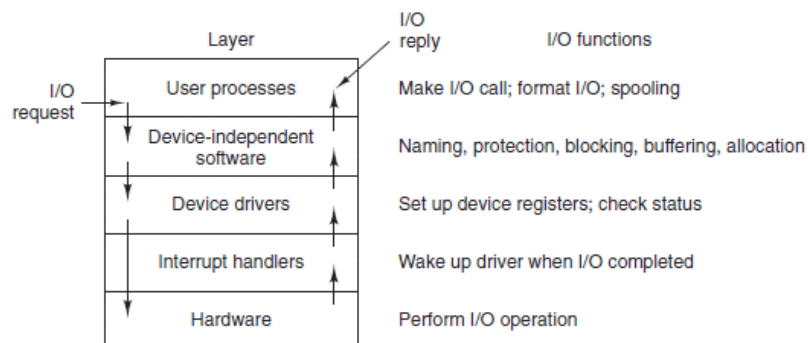


Figure 5-17. Layers of the I/O system and the main functions of each layer.

שעונים

איך המערכת יודעת לטפל בזמן?

יש התקן פיזיקלי שתחת מתח חשמלי יוצר תנודות

אפשר למדוד את כמות הפיקים ואם אני יודעת מה הזמן בין פיק לפיק, אפשר לעשות מונה שיספור את כמות הפיקים וברגע שהוא יגיע לכמות הפיקים שהגדרתי יהיה אינטאפט

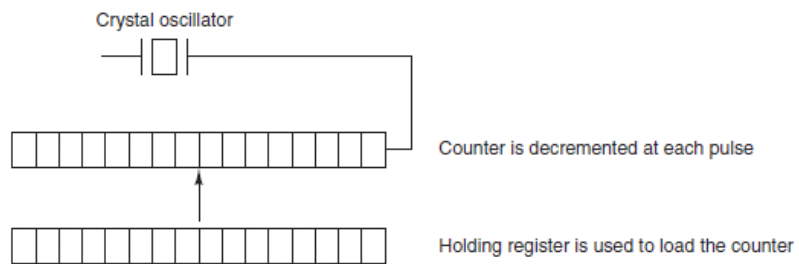


Figure 5-28. A programmable clock.

לשעון יש סוללת גיבוי קטנה- הוא ממשיך לספור גם כשאין חשמל
זמן נשמר במערכת - נשמר ככמות הפיקים מאז ה- 1 לינואר 1970

כמות התיקים-
סדר גודל של
פיקו עד ננו

מתי משתמשים ב-click-
למשל כשבדקים ביצועים
זמן ותאריך
קוונטום

אפשר למדוד את זמן ה-CPU. זמן אבסולוטי לא בהכרח קשור ל-CPU- כמה באמת ה-CPU עבד

שעון מעורר לכל מיני סיגנלים

watchdog- במערכות קריטיות, למשל שני פרוססים מדברים אחד עם השני, נפוץ במערכות צבאיות ורפואיות. אם יש שני חלקים ואני צריכה לוודא ששני החלקים עובדים. מדי פעם משדרים תשדורת דמי שכל מטרתה לוודא שהצד השני חי. אם שלחתי ווטשדוג ולא קיבלתי תשובה, זה אומר שהצד השני מת. ואז אפשר לעשות הזעקה. מנגנון תכנותי שמדי פעם מעורר את הצד השני לבדוק שהכל תקין.
סטטיסטיקות, מונטורינג