אלגוריתמים לזימון תהליכים

מדדים להערכת אלגוריתם לזימון תהליכים:

- זמן שהייה מינימאלי- זמן המתנה כולל + זמן הריצה. את זמן הריצה לא ניתן למזער אך ניתן למזער את זמן ההמתנה הממוצע.
- תקורה מינימאלית- הזמן שמתווסף כתוצאה מהחלפות הקשר בין התהליכים.

לפעמים מעוניינים במטרות נוספות:

- ניצול של המעבד- כמה זמן המעבד פעיל.
- תפוקה (throughput): כמה תהליכים מסתיימים בפרק זמן מסוים.

מדדי יעילות למנגנוני זימון:

T_i - זמן השהייה של תהליך בזמן הקצר (רץ או מוכן).

t_i - זמן הריצה של תהליך.

זמן שהייה ממוצע של תהליך, תחת מדיניות זימון A : $\boxed{H_A = \frac{1}{N}\sum_{k=1}^{N} T_k}$ N -מספר התהליכים.

אלגוריתם FCFS- First Come First Served:

התהליך שהגיע ראשון לתור, ירוץ ראשון עד שיסיים. התהליך הינו non-preemptive- מקבל את המעבד עד לסיום ריצתו.

תכונות:

- מימוש פשוט באמצעות תור FIFO.
- מספר החלפות ההקשר הוא מינימאלי: מספר התהליכים פחות 1.
- נותן עדיפות לתהליכים חישוביים (CPU bound).
- ממזער ניצול התקנים
- לא מספק דרישות שיתוף (time sharing).

זמן ההמתנה:

הזמן תלוי בסדר הגעת התהליכים לטווח הקצר.

דוגמא לזמן המתנה ממוצע שונה כתלות בסדר הגעת התהליכים:

זמן המתנה ממוצע = 17 = (0+24+27)/3.

זמן המתנה ממוצע = 3 = (0+3+6)/3.

אפקט השיירה: תיתכן חסימה של התקני ק/פ עקב תהליך עתיר חישובים שתופס את המעבד שאחריו ישנם הרבה תהליכים מאוד קצרים.

אלגוריתם RR- Round Robin:

תור מעגלי של תהליכים מוכנים לריצה. מוגדר q- זמן שמותר לתהליך לרוץ, ואם התהליך עבר את הזמן הזה, הוא מופסק ומועבר לסוף התור המעגלי.

תכונות:

- preemptive- יש הפקעת תהליכים לפני סיום ריצתם.
- מימוש ע"י timer שמייצר פסיקה כל q יחידות זמן.
- לא זקוק לסטטיסטיקות ומידע על זמני הריצה של התהליכים.

©ענת עציון

- אם q קטן, הזימון הוגן
  - כביכול כל תהליך רץ במעבד משלו בקצב $1/N$ כאשר N הוא מספר התהליכים.
  - זמן התגובה (כמעט) ליניארי ב- N ו-q.
  - התקורה (הזמן כתוצאה מהחלפות הקשר) עלולה להיות גבוהה.
- אם q גדול מאוד, RR הופך ל-FCFS.
- זמן שהייה ממוצע תחת RR הוא לכל היותר פעמיים האופטימאלי.

אלגוריתם Selfish Round Robin:

זהו שילוב בין שני האלגוריתמים הקודמים. תהליכים חדשים ממתינים בתור FIFO, תהליכים ותיקים מוחזקים בתור RR ומתבצעים. כל יחידת זמן עדיפות התהליך גדלה (הזדקנות) וכאשר היא עוברת סף מסוים או כאשר תור הותיקים ריק, התהליך עובר לתור הותיקים.

תהליכים אינטראקטיביים מזדקנים "מאוד מהר" ותהליכים חישוביים מזדקנים יותר "לאט", כי הם גם ככה ירוצו זמן ארוך לאחר-מכן.

אלגוריתם SJF- Shortest Job First:

מריצים את התהליך בעל זמן הביצוע המינימאלי עד לסיומו.

הנחות:

- כל התהליכים מגיעים ביחד.
- זמן הביצוע של כל תהליך ידוע מראש.
- זימון לפי עדיפויות כאשר העדיפות היא ההופכי של זמן הביצוע.

תכונות:

- non-preemptive, כל תהליך רץ עד לסיומו.
- נותן את זמן ההמתנה הממוצע המינימאלי לכל סידור אפשרי של התהליכים.

למה: לכל מדיניות זימון עם הפקעה A עבור N תהליכים שמגיעים יחד, קיימת מדיניות זימון A' ללא הפקעה כך ש- $H_{A'} \le H_A$ (H - זמן שהייה ממוצע).

משפט: כשתהליכים מגיעים יחד, לכל מדיניות זימון A , $H_{SJF} \le H_A$.

משפט: כשתהליכים מגיעים יחד, $H_{RR} \le 2H_{SJF}$.

אלגוריתם SRTF- Shortest Remaining Time to completion First:

כאשר מגיע תהליך P_i שזמן הביצוע הנותר שלו קצר יותר מזן הביצוע הנותר של התהליך שרץ כרגע P_k , מכניסים את P_i למעבד במקום P_k .

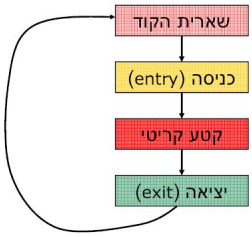
תכונות:

- preemptive
- ממזער את זמן השהייה הממוצע במערכת.
- בניגוד לSJF, נותן את הזמן האופטימאלי כאשר התהליכים לא מגיעים ביחד.
- ייתכן מצב שבו לא נגיע לטפל בתהליך (הרעבה).

©ענת עציון

תיאום בין תהליכים

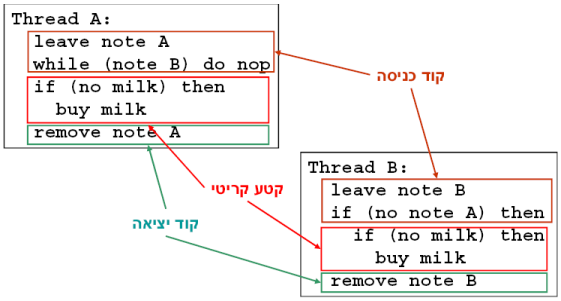
תהליכים משתפים פעולה בגישה למשאבים או העברת נתונים דרך התקן משותף. מניחים שביצוע התהליכים משולב באופן שרירותי ולכן יש צורך להשתמש במנגנוני תיאום. תיתכן גם גישה למשתנים גלובליים ובמקרה זה התוצאה אינה צפויה.



קטע קריטי: הקוד שניגש למשאב המשותף. זהו לא בהכרח אותו קוד לכל החוטים (לא בהכרח סימטרי, למשל: חוט אחד מגדיל מונה וחוט שני מקטין אותו).

את הקטע הקריטי עוטפים בקוד כניסה וקוד יציאה (של הקטע הקריטי).

דוגמא:



תכונות רצויות:

- **מניעה הדדית-mutual exclusion:** חוטים לא מבצעים בו-זמנית את הקטע הקריטי, חוטים מעוניינים מחכים בקטע הכניסה. כאשר החוט הנוכחי יוצא מהקטע הקריטי, הם יכולים להיכנס.
- **התקדמות-no deadlock:** אם אין חוט בקטע הקריטי ויש חוטים שרוצים לבצע אותו, אז חוט כלשהו יצליח להיכנס לקטע הקריטי.
- **הוגנות:** אם יש חוט שרוצה לבצע את הקטע הקריטי, לבסוף הוא יצליח (היעדר הרעבה). רצוי שיצליח להיכנס תוך מספר צעדים חסום (bounded waiting) ואפילו בסדר הבקשה (FIFO).

מנעולים:

אבסטרקציה אשר מבטיחה גישה בלעדית למידע באמצעות שתי פונקציות:

acquire(lock): נחסם בהמתנה עד שמתפנה מנעול. אחרי ביצוע הפונקציה, החוט מחזיק במנעול. רק חוט אחד מחזיק את המנעול (בכל נקודת זמן) ורק הוא יכול לבצע את הקטע הקריטי.

release(lock): משחרר את המנעול.

בעיה במימוש מנעולים: המנעולים מכילים קטע קריטי.

אלגוריתם קופת חולים/ המאפיה:

כל חוט נכנס ולוקח מספר. חוט ממתין שמספרו הקטן ביותר- נכנס לקטע הקריטי.

האלגוריתם מחכה שב-entry (קטע הכניסה לקוד הקריטי) כל החוטים יבחרו מספר. number[i]- המספר שבחר חוט i.

choosing[i]- true אם החוט i עדיין לא סיים לבחור מספר ו-false אם הוא כבר סיים לבחור.

חוט עובר להמתין לכניסה לקטע הקריטי רק כאשר כל החוטים סיימו לבחור מספר ב-entry.

```
Thread i:
  initially number[i]=0;
  choosing[i]=true;
  number[i]=max{number[1],…,number[n]}+1;
  choosing[i]=false;
  for all j≠i do
    wait until choosing[j]=false;
  for all j≠i do
    wait until number[j]=0 or
      ((number[j],j)>(number[i],i));
  critical section
  number[i]=0 // Exit critical section
```


©ענת עציון

מימוש מנעולים- חסימת פסיקות:

החסימה מונעת החלפת חוטים ומבטיחה פעולה אטומית על המנעול. בעיות:

- תוכנית מתרסקת כאשר הפסיקות חסומות.
- פסיקות חשובות הולכות לאיבוד.
- עיכוב בטיפול בפסיקות I/O גורם להרעת ביצועים.

הערה: במערכת עם כמה מעבדים, לא די בחסימת פסיקות (חוטים יכולים לרוץ בו-זמנית על כמה מעבדים שונים).

Spinlocks:

מימוש מנעול באמצעות **busy-waiting**:

- בדוק האם המנעול תפוס (ע"י גישה למשתנה).
- אם המנעול תפוס בדוק שנית.

חיסרון: מאוד בזבזני. חוט שמגלה כי המנעול תפוס, מבזבז זמן CPU. בזמן הזה, החוט שמחזיק במנעול לא יכול להתקדם.

בעיית priority inversion: לחוט הממתין עדיפות גבוהה יותר מאשר החוט המחזיק במנעול.

פיתרון: חוט שנתקל במנעול נעול היה נכנס לתור המתנה ואז החוט בעל העדיפות הנמוכה יותר יכול לרוץ ולפתוח את המנעול.

סמפור:

מעין הרחבה למנעול- מנעול עם מונה. יש כביכול כמה "מפתחות" ולכן כמה תהליכים/חוטים שונים יכולים לנעול. פעולות על סמפור:

- **wait(semaphore):** מקטין את המונה ב-1. רק אם המונה חיובי יש מפתחות זמינים לשימוש. אם המונה אפס או שלילי, התהליך נכנס לתור המתנה ולא יכול לרוץ עד שמישהו "יחזיר את המפתח". מספר הממתינים הוא ערכו המוחלט של הסמפור.
- **signal(semaphore):** משחרר אחד מן הממתינים ו"נותן לו את המפתח" לפני היציאה מהקטע הקריטי, או מגדיל ב-1 את המונה אם תור הממתינים ריק.

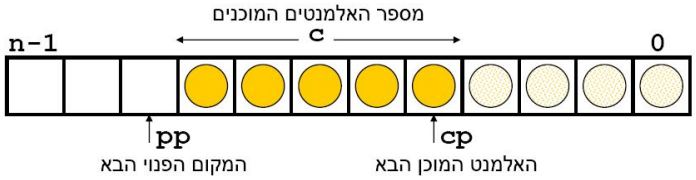
הבדלים מהותיים בין סמפור למנעול:

- סמפור שאותחל להיות בינארי, לא בהכרח ימשיך להיות בינארי, זה באחריות המשתמש.
- מנעול לא ניתן לפתוח פעמיים.
- במנעול יש בעלות, רק מי שנעל יכול לפתוח. במסמפור אין בעלות, כל אחד יכול לעשות signal ו-wait.
- סמפור אמור לעבוד בין תהליכים, מנעול עובד רק בין חוטים.

בעיית יצרן/צרכן עם סמפורים:

שני חוטים רצים באותו מרחב זיכרון. היצרן מייצר אלמנטים לטיפול. הצרכן מטפל באלמנטים.

מערך חסום מעגלי מכיל את האלמנטים המיוצרים.



©ענת עציון

שימוש בשני סמפורים שלכל אחד מהם יש מונה- כמה מקומות פנויים יש וכמה אלמנטים יש על פס הייצור.

כל אחד מקטין את אחד המונים בתחילת פעולתו ומגדיל את המונה השני בסיום פעולתו, בהתאמה.

```
semaphore freeSpace,          מספר המקומות הפנויים
  initially n
Semaphore availItems,         מספר האיברים המוכנים
  intiailly 0

Producer:                     Consumer:
  repeat                        repeat
    wait( freeSpace);             wait( availItems);
    buff[pp] = new item;          consume buff[cp];
    pp = (pp+1) mod n;            cp = (cp+1) mod n;
    signal( availItems);          signal( freeSpace);
  until false;                  until false;
```

מנעול קוראים-כותבים עם סמפורים:

יש חוטים קוראים וחוטים כותבים. מספר חוטים יכולים לקרוא בו-זמנית. כאשר חוט כותב, אסור שחוטים אחרים כתבו ו/או יקראו.

בעיות:

רק כאשר לא נותרים קוראים זה יכול להיפתח לכתיבה.

לקוראים יש עדיפות והם יכולים די בקלות להרעיב את הכותבים.

```
int r = 0;                    מונה מספר הקוראים
semaphore sRead,              מגן על מונה מספר הקוראים
  initially 1
semaphore sWrite              מניעה הדדית בין קוראים לבין כותבים
  initially 1                   (ובין כותבים לעצמם)

Writer:                       Reader:
  wait(sWrite)                  wait(sRead)
  [Write]                       r:=r+1
  signal(sWrite)                if r=1 then
                                    wait(sWrite)
                                signal( sRead)
                                [Read]
                                wait( sRead)
                                r:=r-1
                                if r=0 then
                                    signal(sWrite)
                                signal( sRead)
```

פעולות על משתני תנאי:

משתנה תנאי תמיד קשור לאיזשהו מנעול.

wait(cons, &lock): שחרר את המנעול (חייב להחזיק במנעול). המתן לפעולת signal. המתן למנעול (כשהוא חוזר מהפעולה הוא מחזיק את המנעול).

signal(cond): הער את אחד הממתינים ל-cons, אשר עובר להמתין למנעול. הולך לאיבוד אם אין ממתינים.

broadcast(cond): הער את כל התהליכים הממתינים. הם עוברים להמתין למנעול. הולך לאיבוד אם אין ממתינים.

הערה: כאן נפתרה הבעיה של unlock ו-block שיש בסמפורים כי מבצעים דברים בצורה אטומית.

בעיית יצרן/צרכן עם משתני תנאי:

משמעות ה-signal ב-producer היא להעיר את מי שמחכה לאלמנטים המיוצרים.

שימוש ב-broadcast כאן יהיה פחות יעיל- יתעוררו יותר צרכנים מכמות האלמנטים שנוצרו- בזבוז של משאבים.

```
condition not_full,
  not_empty;
lock bLock;

producer:                     consumer:
lock_acquire(bLock);          lock_acquire(bLock);
while (buffer is full)        while (buffer is empty)
  wait(not_full,&bLock);        wait(not_empty,&bLock);
add item to buffer ;          get item from buffer
signal(not_empty);            signal(not_full);
lock_release(bLock);          lock_release(bLock);
```


©ענת עציון

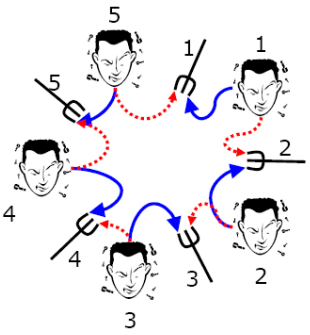
קיפאון – deadlock

קיפאון: קבוצת תהליכים/חוטים שבה כל אחד ממתין למשאב המוחזק ע"י מישהו אחר בקבוצה.

הפילוסופים הסועדים:

הפילוסופים יושבים במעגל, ויש מזלג אחד בין כל שני פילוסופים. כל פילוסוף מנסה לתפוס את שני המזלגות שמימינו ומשמאלו כדי שיוכל לאכול.

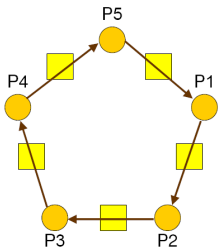
עם סמפורים: סמפור fork[i] לכל מזלג




```
wait(fork[i])
wait(fork[i+1])
eat
signal(fork[i])
signal(fork[i+1])
```

בביצוע מסונכרן כל פילוסוף משתלט כל המזלג שמימינו ואז ממתין למזלג שמשמאלו ⇦ קיפאון

deadlock: לכל מזלג שמים סמפור בינארי ואז כל אחד תופס מזלג מצד ימין ומחכה למזלג השני שכבר נתפס וכך כולם מחכים לכולם.

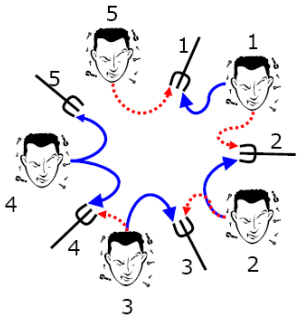


מעגל מכוון בגרף בקשות-הקצאות מתאר מצב של deadlock.

מניעת המתנה מעגלית: בקשת מזלגות בסדר עולה...

נקבע סדר בין המשאבים (=המזלגות), ונבקש אותם בסדר עולה.

הפיתרון הזה אף-פעם לא יגרום ל-deadlock.




```
if ( i = 5) then
   wait(fork[1])
   wait(fork[5])
else
   wait(fork[i])
   wait(fork[i+1])
eat
signal(fork[i+1])
signal(fork[i])
```

אלגוריתם הבנקאי (מניעת קיפאון):

כאשר תהליך מגיע למערכת, הוא מציהיר כמה עותקים ירצה (לכל היותר) מכל משאב.

תהליך לא יוכל לבקש מעבר למה שהציהיר מראש.

אם תהליך מבקש עותק ממשאב, הבקשה תיענה רק אם:

- יש מספיק עותקים פנויים
- ההקצאה משאירה את המערכת במצב בטוח: במקרה הגרוע ביותר ניתן יהיה לספק את הבקשות של כל התהליכים שכבר נמצאים במערכת עד למקסימום המוצהר לכל סוג משאב (למרות שייתכן שהתהליכים שרצים כרגע לא ירצו לנעול בזמן זה את כל מה שהם הצהירו עליו).

ברגע שתהליך מסיים לרוץ, כל מה שהוא נעל משתחרר וזמין שוב במערכת.

האלגוריתם הוא די שמרני, הוא אינו מתחשב בסדר ריצה של תהליכים, אלא רק בודק את המצב הסטאטי של הטבלאות שלו.

©ענת עציון

m- מספר המשאבים.

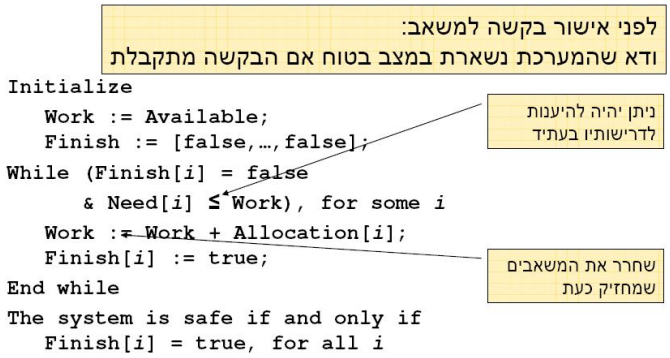
n- מספר התהליכים.

Available: מערך של מספר עותקים פנויים מכל משאב.

Max: מטריצה של הדרישה המקסימאלית של כל תהליך על כל משאב.

Allocation: מטריצה של משאבים המוקצים כרגע לכל תהליך.

Need: מטריצה של המשאבים שכל תהליך עדיין לא צרך ויכול לבקש בעתיד.

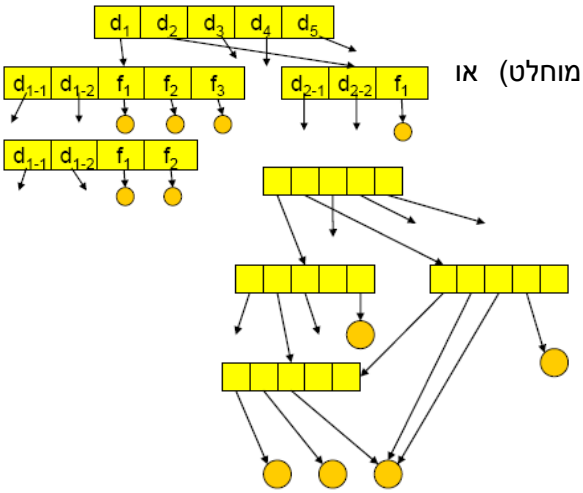


©ענת עציון

מערכת קבצים

מבנה מדריכים:

עץ מכוון:

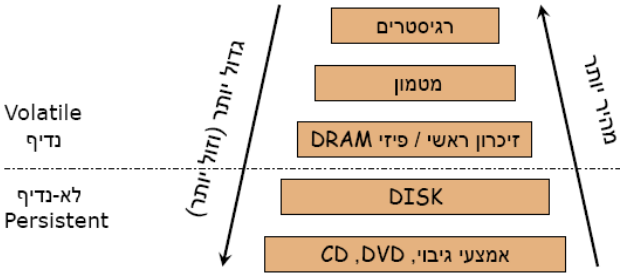


קובץ מזוהה ע"י מסלול מהשורש (path מוחלט) או מהמדריך הנוכחי (path יחסי).

גרף אציקלי:

מאפשרים למספר כניסות במדריכים להצביע לאותם העצמים.

היררכית האחסון



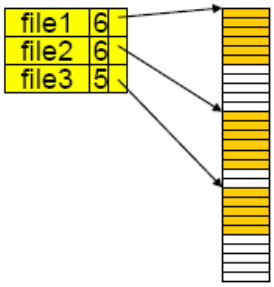
זיכרון משני: בניגוד לזיכרון הראשי, אינו מאפשר ביצוע ישיר של פקודות או גישה לבתים.

שומר על מידע לטווח ארוך (לא נדיף).

גדול, זול ואיטי יותר מהזיכרון הראשי.

מיפוי קבצים:

הקצאה רציפה:

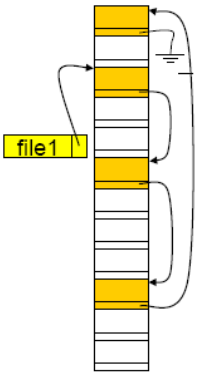


- בלוקים באורך קבוע ($4-512kb$).
- המשתמש מצהיר על גודל הקובץ עם יצירתו.
- מחפשים בלוקים רצופים שיכולים להכיל את הקובץ.
- הכניסה במדריך מצביעה לבלוק הראשון בקובץ.

יתרון: יעיל מבחינת גישה סדרתית וגם מבחינת גישה אקראית.

חיסרון: יש סגמנטציה פנימית וחיצונית. קשה מאוד להגדיל את הקובץ.

הקצאה משורשרת:



- כל בלוק מצביע לבלוק הבא.
- הכניסה במדריך מצביעה לבלוק הראשון בקובץ.

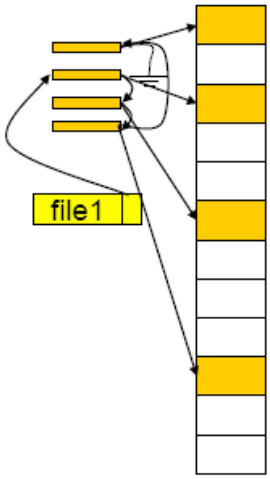
יתרון: אין סגמנטציה חיצונית. קל להגדיל את הקובץ.

חיסרון: גישה איטית, בעיקר לגישה אקראית (ישירה). סגמנטציה פנימית. שיבוש מצביע בבלוק גורם לאיבוד המידע בשאר הקובץ.

כתיבה באופן קציף כבר אינה מהירה יותר מאשר גישה אקראית.

©ענת עציון

FAT- File Allocation System:

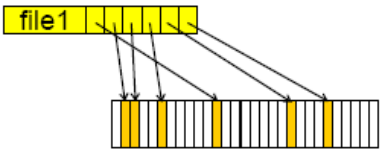


- החזקת שרשרת המצביעים בנפרד. זו למעשה טבלה שמתארת את התוכן של הדיסק כולו.
- הדיסק מחולק ל-clusters ולכל אחד מהם יש כניסה בטבלה.
- מצביע הקובץ במדריך מראה על הכניסה הראשונה.

חיסרון: כל קובץ דורש לפחות cluster אחד- מגביל את מספר הקבצים. וגם מגדיל סגמנטציה פנימית (בזבוז של 10-20% הוא שכיח).

אובדן הטבלה הוא הרה אסון ולכן הפיתרון: משכפלים את טבלת המצביעים פעמיים וכך הסיכוי לטעות- קטן.

אינדקס:



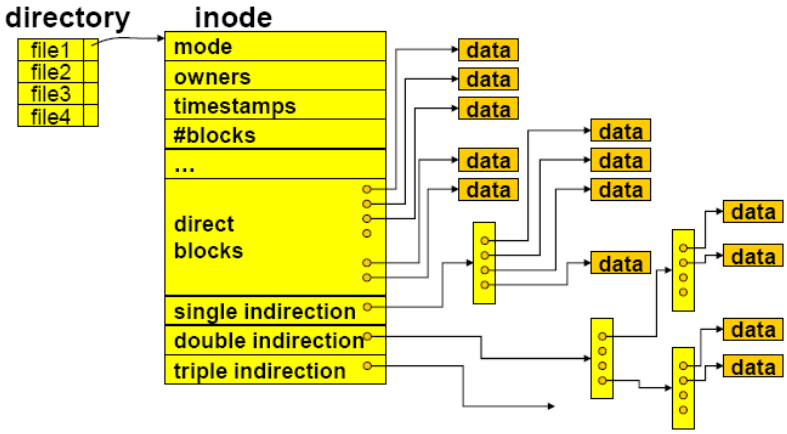
- המשתמש מצהיר על גודל קובץ מקסימאלי.
- המצביעים לבלוקים מרוכזים בשטח רצוף.

יתרון: משפר את זמן הגישה הישירה לעומת הקצאה משורשרת.

חיסרון: יש להצהיר מראש על גודל הקובץ.

index blocks: לוקחים את המתאר קובץ בדיוק המכיל הרבה מצביעים ומשתמשים במספר מצביעים לפי הצורך. שיטה בזבזנית כי יוקצו יותר מצביעים ממה שהקובץ צריך בפועל.

אינדקס מרובה-רמות:



שילוב בין כמה שיטות.

זו מערכת שניתן להגדיל בה קובץ די בקלות.

יש מתאר קובץ שנמצא בדיסק- inode. שם הקובץ לא מופיע בו.

המידע נמצא בבלוקים כך שאם נפגע מצביע, לא נאבד את המידע שמצביעים עליו בבלוקים אחרים.

עם כל מצביע-direct ניתן לייצג קובץ בגודל 256k. אם צריך מצביעים נוספים, יש בלוק-single של מצביעים המכיל 256 מצביעים. ויש גם אפשרות ל-double: $(256)^2$ מצביעים.

FFS- Fast File System:

מחלקים את הדיסק לקבוצות של צילינדרים קרובים. בכל צילינדר משתדלים לשים בלוקים של אותו קובץ, ובלוקים של אותו מדריך, כולל inodes. שומרים על $\sim 10\%$ מקום פנוי בכל קבוצת צילינדרים.

ניהול בלוקים פנויים-bitmap:

מערך ובו סיבית לכל בלוק (0-תפוס, 1-פנוי). מאוחסן במקום קבוע בדיסק ונטען תמיד לזיכרון הראשי. יתרון: קל לזהות רצף של בלוקים פנויים, אך יצריך מעבר על כל מערך הביטים. קל למימוש.

חיסרון: תופס הרבה מקום.

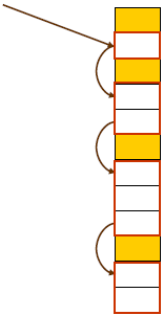
©ענת עציון

ניהול בלוקים פנויים-רשימה מקושרת:

יתרון: מציאת בלוק פנוי בודד היא מהירה.

חיסרון: הקצאת מספק בלוקים לאותו קובץ- במקומות מרוחקים, מציאתם מחייבת תזוזות של הדיסק.

מבנה הנתונים נהרס אם מצביע באמצע הרשימה השתבש (ולכן שומרים את הרשימה פעמיים).



ניהול בלוקים פנויים-קיבוץ:

זו רשימת קבוצות של בלוקים. בלוק ראשון מצביע לבלוק הראשון בקבוצה הבאה.
הבלוק הראשון יודע מי איתו בקבוצה וכל בלוק מצביע אל הבלוק הראשון בקבוצה.

יתרון: ניתן למצוא בצורה יעילה מספר גדול של בלוקים פנויים ורציפים.

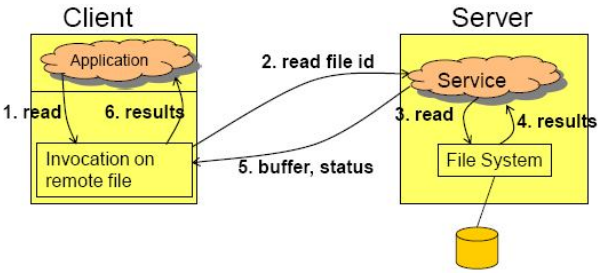
זוהי השיטה הנפוצה.

©ענת עציון

מערכות קבצים מבוזרות

תהליך רץ במחשב לקוח.
התהליך מבקש לגשת לקובץ הנמצא במחשב שרת.
מחשב הלקוח ומחשב השרת מחוברים באמצעי תקשורת כלשהו.

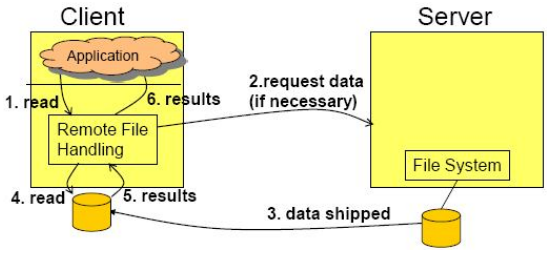
אפשרות 1-שיגור הפונקציה:



הפעולות מועברות לשרת אשר מבצע אותן לוקלית ומעביר את התוצאות חזרה ללקוח (function shipping).

יתרון: מימוש פשוט. ניתן לבצע נעילה בשרת.
חיסרון: עומס על השרת שמריץ את כל הפעולות של כל הלקוחות. המידע לא מובא בבת-אחת.

אפשרות 2-שיגור נתונים:



הנתונים מועברים מהשרת ללקוח, אשר מבצע את הפעולה באופן מקומי (data shipping).

יתרון: מוריד עומס מהשרת.

חיסרון: איך יודעים אם הנתונים עדכניים? מביא לקונפליקטים בין קבצים שלקוחות שונים מנסים לכתוב.

מדיניות משולבת- מטמון:

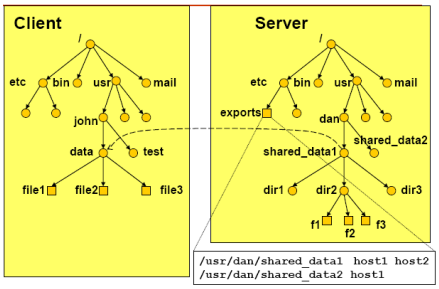
מדיניות עדכון- מתי מעבירים עדכונים מהמטמון לשרת:

- write-through: אמין, אך המטמון מנוצל אך ורק בקריאה.
- write-on-close: השינויים מועברים בעת סגירת הקובץ (לא שומר על סדר הגישה בין לקוחות).
- delayed-write: השינויים מועברים כל פרק זמן.

קונסיסטנטיות- איך הלקוח יודע שהנתונים במטמון תקפים:

- client initiated: הלקוח בודק בכל גישה, כל פרק זמן או כאשר הקובץ נפתח.
- server initiated: השרת זוכר איזה לקוח מחזיק איזה חלק של הקבצים. ואז אם שני לקוחות מנסים לכתוב, אפשר להודיע על שגיאה לאחד מהם.

NFS-Network File System:



פרוטוקול שרת-לקוח (stateless).

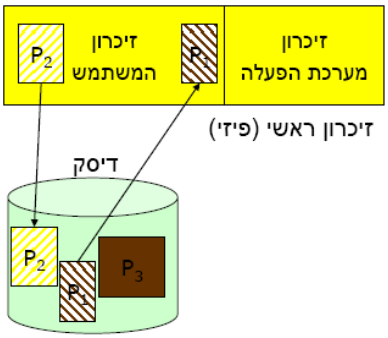
הלקוח מרכיב (mounts) תת-עץ של השרת במדריך שלו. אחרי הרצת הפקודה, המערכת תסתיר את הקבצים שב-data משמאל ומה שהלקוח יראה זה הקבצים שב-shared_data1 המצויים בצד ימין.

גישה שקופה למשתמש, כמו לקובץ מקומי.

©ענת עציון

זיכרון וירטואלי

swapping:



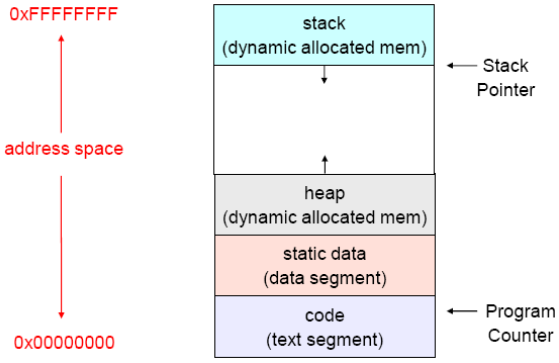
שיטה שהייתה נפוצה בשנות ה-70.

זיכרון של תהליך שאינו רץ, מועבר לדיסק (swap out).

כאשר תהליך חוזר לרוץ, מקצים לו מחדש מקום ומביאים את הזיכרון שלו (swap in).

עבור קבצים בינאריים: לא נעשית קריאה בבת-אחת של הקובץ אלא נעשה מיפוי בין הכתובות שקפיצות jmp קופצות אליהן לבין הקוד אליו נקפוץ מתוך הקובץ.

מרחב הכתובות של תהליך:



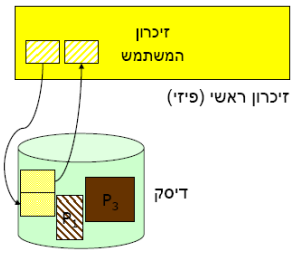
בעיקרון, כל מרחב הכתובות צריך להיות זמין בזיכרון הפיזי של המחשב כאשר התהליך רץ.

האם התהליך באמת צריך את כל הזיכרון הזה כל הזמן?

עיקרון הלוקאליות: תהליך ניגש רק לחלק מזערי של הזיכרון שברשותו בכל פרק זמן נתון.

צריך לאחסן ערכים של משתנים, גם אם לא משתמשים בהם, אבל לא חייבים לעשות זאת בזיכרון הפיזי.

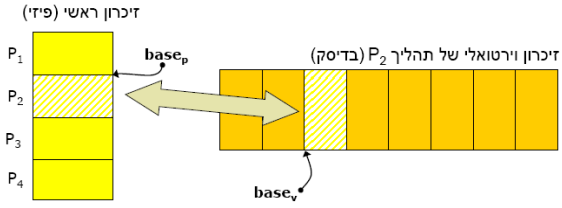
הדיסק: מכיל חלקים מהזיכרון של תהליך שרץ כרגע.



- צריך לזכור מה נמצא בזיכרון הפיסי ואיפה (ומה נמצא בדיסק).
- צריך לבחור מה להעביר לדיסק.
- צריך לזכור איפה שמנו חלקי זיכרון בדיסק, כדי לקרוא אותם חזרה אם נצטרך אחר-כך.

זיכרון וירטואלי: לכל תהליך יש מרחב כתובות מלא שיכול להיות גדול מהזיכרון הפיסי. רק חלקי הזיכרון הנחוצים כרגע לתהליך נמצאים בזיכרון הפיסי. התהליך יכול לגשת רק למרחב הכתובות שלו.

שיטה ישנה-חלוקה קבועה:



מערכת ההפעלה מחלקת את הזיכרון הפיסי לחתיכות בגודל קבוע size שיכולות להכיל חלקים ממרחב הכתובות של תהליך.

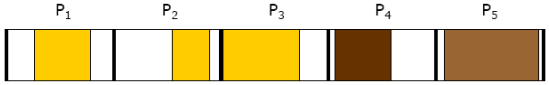
לכל תהליך מוקצית חתיכת זיכרון פיסי.

התרגום מתבצע באמצעות רגיסטר $base_p$ המחזיק את כתובת הבסיס הפיסית של התהליך שרץ במעבד, ורגיסטר $base_v$ שמציין את כתובת הבסיס של החלק ממרחב הכתובות של התהליך שרץ ונמצא בזיכרון הפיסי. כתובת va נמצאת בזיכרון הפיסי אם היא בתחום: $[base_v, base_v + SIZE)$, אחרת צריך להביא את החתיכה המתאימה מהדיסק מהכתובת: $base_p + (va - base_v)$, ומשנים את $base_v$ בהתאם לחתיכה שהובאה מהזיכרון. בהחלפת הקשר, מחליפים את $base_p$ ואת $base_v$ בהתאם.

©ענת עציון

בעיות:

- אם תהליך צריך עוד זיכרון- הוא קיבל חתיכה נוספת. הן חייבות להיות רציפות וההעתקות של הזיכרון ממקום למקום אינן יעילות.
- החתיכה מאוד גדולה ביחס למה שחלק מהתהליכים צריכים- מאוד בזבזני- סגמנטציה פנימית.



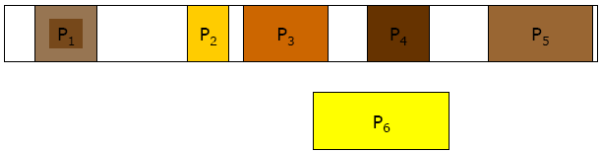
- מספר התהליכים מוגבל ע"י מספר החתיכות בזיכרון.

שיטה ישנה נוספת- חלוקה משתנה:

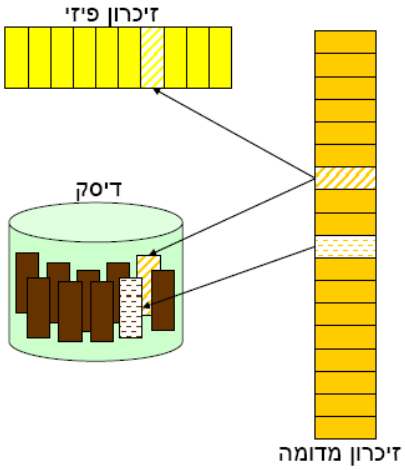
הרחבה של השיטה הקודמת ע"י תוספת רגיסטר המציין את אורך החתיכה.

יתרון: מונע סגמנטציה פנימית (אין מקום שמוקצה לתהליך שמבוזבז סתם).

חיסרון: יש סגמנטציה חיצונית- הרבה זיכרון מבוזבז עקב סידור מרווח של החתיכות ביניהן (שאריות מקום בין החתיכות שלא מתאימות לכלום).



שיטה מודרנית- דפדוף:



מחלקים את הזיכרון הוירטואלי ל**דפים** (pages) בגודל קבוע-4k.

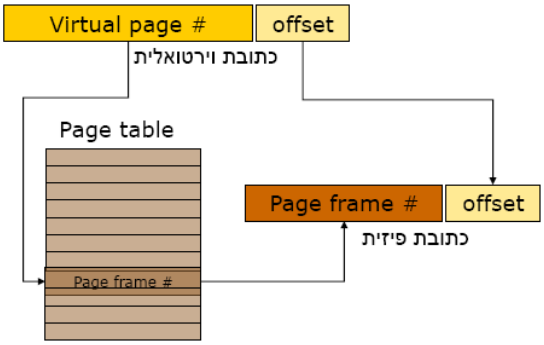
מחלקים את הזיכרון הפיסי ל**מסגרות** (frames) בגודל דף.

כל מסגרת בזיכרון הפיסי יכולה להחזיק כל דף וירטואלי.

כל דף וירטואלי נמצא בדיסק, חלק מהדפים נמצאים בזיכרון הפיסי.

לכל תהליך יש **טבלת דפים** אחת על-מנת למפות בין הזיכרון הוירטואלי לפיסי.

טבלת דפים:



כל כניסה בטבלה מתייחסת למספר דף וירטואלי ומכילה מספר מסגרת פיסית.

לכתובת יש שני חלקים:

- מספר הדף (Virtual Page Number).
- offset (מיקום בתוך הדף).

ה-VPN מאפשר למצוא את מספר המסגרת שבה נמצא הדף (physical page number).

הוספת ה-offset נותנת את הכתובת עצמה.

דוגמא:

כתובת וירטואלית של 32 ביט (כאשר מרחב הכתובות הוא $2^{32} = 4G$).

דפים בגודל $4k$ שמכילים 2^{12} כתובות.

20 ביטים לVPN ו-12 ביטים ל-offset.

©ענת עציון

כניסות בטבלת הדפים מכילות גם מידע ניהולי בנוסף למיקום הדף הפיסי:

V	R	M	protect	Page frame #

Valid bit: דלוק- הדף בזיכרון, כבוי- הדף בדיסק.

Reference bit: האם ניגשו לדף. מודלק בכל גישה לדף.

Modify bit: האם הייתה כתיבה לדף. מודלק בכתיבה.

protection bits: מה מותר לעשות לדף.

Page fault:

כאשר החומרה ניגשת לזיכרון לפי טבלת הדפים ומגיעה לדף שאינו בזיכרון הפיסי (valid bit==0) נגרמת חריגה מסוג: page fault.

בטיפול בחריגה, גרעין מערכת ההפעלה טוען את הדף למסגרת בזיכרון הפיסי ומעדכן טבלאות דפים (ייתכן שיידרש לפנות מקום או דף ממסגרת אחרת בזיכרון).

כאשר מסתיים הטיפול בחריגה, מבוצעת ההוראה מחדש (restartable instruction).

page fault יכול להיגרם גם מ:גישה לא חוקית לזיכרון, גישה לדף שלא מוקצה ועוד.

יתרונות:

- חוסך סגמנטציה חיצונית: כל מסגרת פיסית יכולה לשמש לכף דף וירטואלי ומערכת ההפעלה זוכרת איזה מסגרות הן פנויות.
- מצמצם סגמנטציה פנימית: דפים הם קטנים בהרבה מחתיכות.
- קל לשלוח דפים לדיסק: גודל הדף מתאים להעברה בבת-אחת לדיסק. גם לא חייבים למחוק, אפשר לסמן לכבות את ה-valid bit.
- אפשר להחזיק בזיכרון הפיסי קטעים לא רציפים.

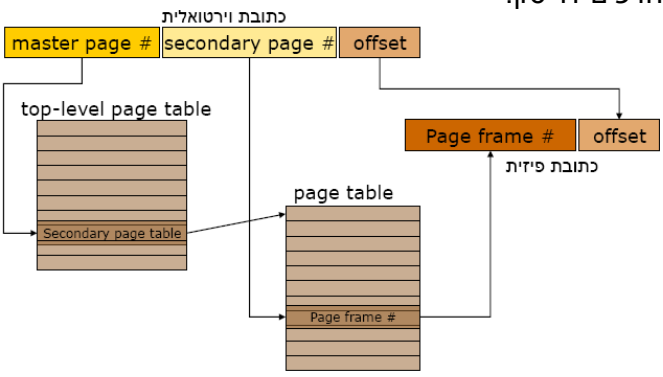
חסרונות:

- טבלת הדפים תופסת מקום רב (4MB לכל תהליך).
- תקורה של גישות לזיכרון לצורך תרגום כתובות.
- עדיין יש סגמנטציה פנימית מצומצמת (גודל זיכרון התהליך אינו כפולה של גודל הדף ולכן יש שאריות).

דפים קטנים ממזערים שברור פנימי אבל דפים גדולים מחפים על שהות בגישה לדיסק.

טבלת דפים בשתי רמות:

חוזרים על אותו תעלול ושולחים חלקים מטבלת הדפים לדיסק.



כעת לכתובת הוירטואלית יש 3 חלקים.

הטבלה ברמה העליונה נכנסת בדף אחד: 1024 כניסות, כל אחת של 4 בתים, סה"כ: 4k.

הטבלה ברמה העליונה היא תמיד בזיכרון הראשי.

חיסרון: תקורה של שתי גישות זיכרון (ולפעמים גם גישה לדיסק) על כל גישה לדף.

פיתרון: שימוש במטמון חומרה מיוחד... TLB.

©ענת עציון

TLB- Translation Lookaside Buffer:

מטמון חומרה ששומר מיפויים של מספרי דפים וירטואליים למיפויי דפים פיסיים. למעשה, הוא שומר עבור כל מספר דף וירטואלי את כל הכניסה שלו בטבלת הדפים, אשר יכולה להכיל מידע נוסף. מפתח הטבלה הוא מספר הדף הוירטואלי.

ה-TLB קטן מאוד אבל בגלל עיקרון הלוקאליות במקום ובזמן, הוא משפר ביצועים באופן ניכר.

אם כתובת נמצא ב-TLB – hit, פגיעה. יש פגיעות ב-99% מהכתובות

אם כתובת לא ב-TLB- miss, יש לעשות תרגום רגיל דרך טבלת הדפים.

בהחלפת הקשר בין תהליכים נעשית פסילה של ה-TLB כי תרגום הכתובות כבר אינו תקף (בהחלפת הקשר בין חוטים אין צורך לבצע פסילה).

אלגוריתמים לפינוי:

אלגוריתם FIFO: מפנה את הדף שנטען לפני הכי הרבה זמן.

האנומאליה של Belady: הגדלת הזיכרון מגדילה את מספר ה-page faults עבור אותה סדרת גישות במקום להקטינו. סימן לאלגוריתם "לא מוצלח". $\boxed{\exists r,d \quad PF_A(r,d) < PF_A(r,d+1)}$

אלגוריתם מחסנית: אלגוריתם שמבטיח שלכל סדרת גישות, בהגדלת הזיכרון לא יוגדל מספר ה-page faults עבור אותה סדרה. $\boxed{\forall r,d \quad M_A(r,d) \subseteq M_A(r,d+1)}$

אלגוריתם אופטימאלי: אם כל הגישות לזיכרון ידועות מראש, האלגוריתם החמדן מפנה מהזיכרון את הדף שהזמן עד הגישה הבאה אליו הוא הארוך ביותר. לא פראקטי כי אין אפשרות לחזות את העתיד.

LRU-Least Recently Used: מפנה את הדף שהגישה האחרונה אליו היא המוקדמת ביותר.

מימושים:

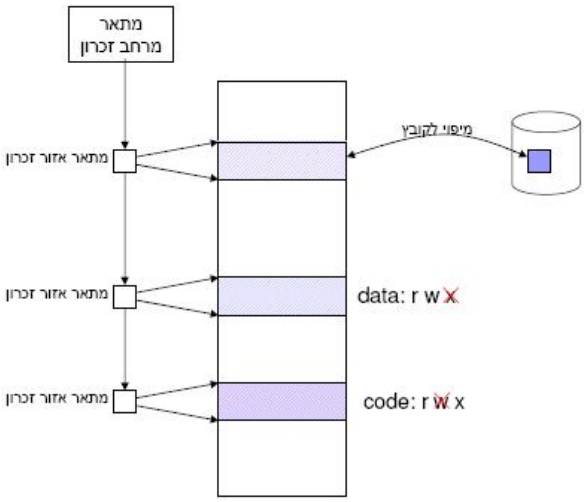
- חותמת זמן (timestamp) לכל דף. מפנים את הדף בעל החותמת זמן המוקדמת ביותר.
- ניהול מחסנית. בגישה לדף מעבירים לראש המחסנית ומפנים מהתחתית שלה.

חיסרון: יקר ודורש תמיכה בחומרה. במקרה הרע יהיה כמו FIFO.

LRU מקורב: יש reference bit לכל דף המודלק בגישה לדף. בפינוי דפים אם הוא שווה 0, הדף מפונה ואם הוא שווה 1, מאפסים את הביט. נקבל שרק אם באמת לא ניגשנו לדף זמן רב אז הוא יפונה. ניתן גם לצרף שדה גיל, כאשר מפנים לפי הזדקנות וסף מסוים, על עיקרון דומה.

©ענת עציון

זיכרון וירטואלי בלינוקס



מתאר מרחב הזיכרון:

לכל תהליך מרחב זיכרון וירטואלי משלו, מתחיל ב-0 ומסתיים ב-3G.

מרחב הזיכרון של תהליך מתחלק לאזורים בגודל כפולה של דף.

לכל אזור קיים מתאר אזור זיכרון הכולל גבולות אזור הזיכרון והרשאות.

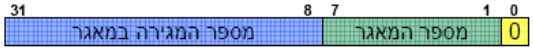
כל מתארי מרחב הזיכרון שבמערכת משורשרים ברשימה מקושרת.

טבלת דפים:

לכל תהליך יש טבלת דפים משלו המנהלת את הדפים שלו- האם הדף נמצא בזיכרון הראשי ובאיזו מסגרת.

כל כניסה בטבלת דפים מכילה ביט-present המציין האם הדף נמצא בזיכרון הראשי.

אם הוא דלוק, הכניסה מכילה את מספר המסגרת בה מאוחסן הדף בזיכרון הראשי. אם הוא כבוי, הכניסה מצביעה על המיקום של הדף בדיסק.

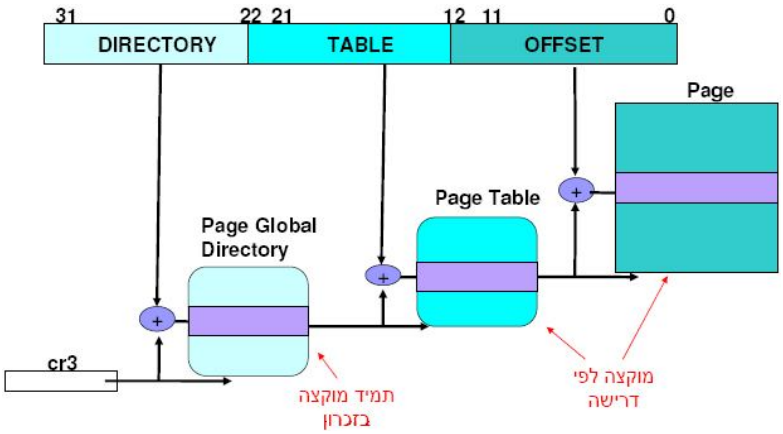


יש רגיסטר מיוחד CR3 המצביע על טבלת הדפים של התהליך הנוכחי (מתעדכן בהחלפת הקשר).

טבלת הדפים הינה בשתי רמות

נעשית כאן גישה של 3 פעמים לזיכרון, לא כל-כך יעיל.

ולכן נעשה שימוש ב-TLB.



כניסה בטבלת דפים, present=1:

ביט accessed: מודלק ע"י החומרה בכל פעם שמתבצעת גישה לכתובת בדף (referenced).

ביט dirty: מודלק ע"י החומרה בכל פעם שמתבצעת כתיבה לנתון בדף (modified).

מאגר דפדוף - swap area:

אזור מיוחד בדיסק אליו מפונים דפים מהזיכרון הפיסי הראשי.

דפים ממופים אנונימית: דפים הממופים למאגר הדפדוף.

בלינוקס ניתן להגדיר ולנהל מספר מארגי דפדוף. כל מאגר דפדוף מחולק למגירות (slots). בגודל הדפים (כל מגירה משמשת לאחסון דף מהזיכרון הראשי). דפים הממופים לקבצים, מועברים לקובץ ממנו באו כשהם מפונים (לדוגמא: הדפים של קובץ ריצה בניגוד לדפים של ה-heap או המחסנית).

©ענת עציון

KMPGD- Kernel Master Page Global Directory:

טבלת הדפים למרחב הזיכרון של הגרעין. מתעדכנת כאשר הגרעין מקצה ומשחרר דפים לשימושו. אף תהליך לא משתמש בטבלה זו אולם היא משמשת כמקור ממנו מתעדכנות טבלאות הדפים של תהליכי המשתמש לגבי דפים שבשימוש הגרעין.

מטמון דפים- swap cache:

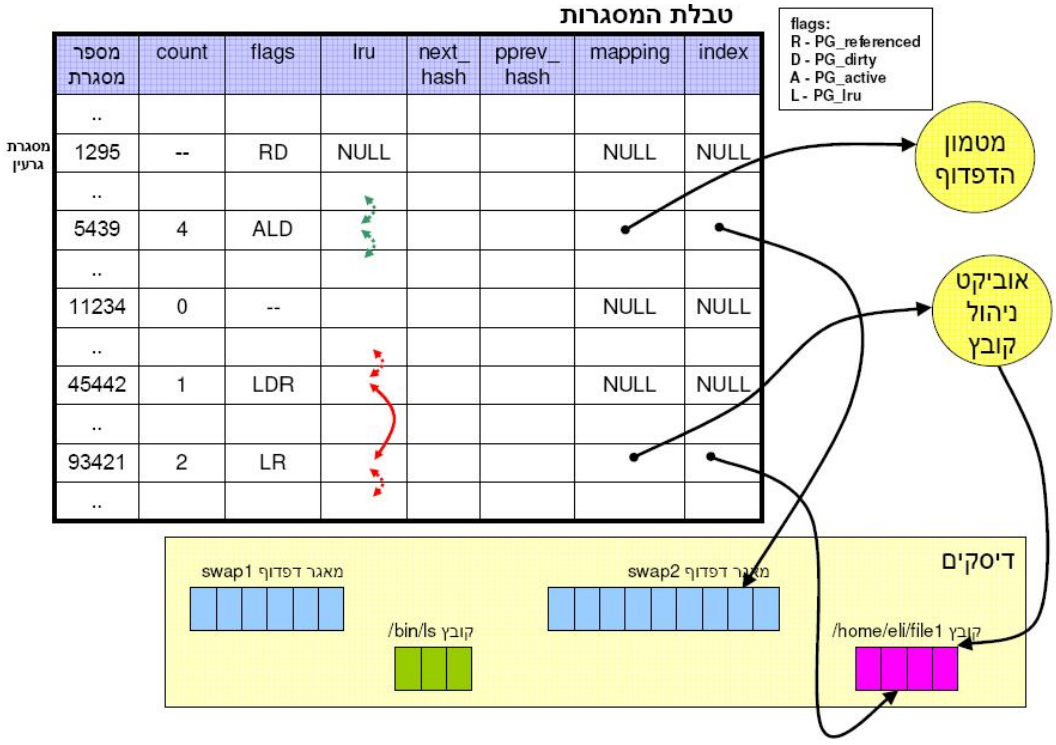
מטמון הדפים הוא המנגנון המרכזי לטעינת ופינוי דפים בין הזיכרון לדיסק בלינוקס. כל דף של מרחב הזיכרון של תהליך מפנה תמיד דרך מטמון הדפים. המטמון ממומש כחלק מטבלת המסגרות.

טבלת המסגרות:

הגרעין מחזיק מערך mem_map עם כניסה לכל מסגרת בזיכרון הראשי.

mapping- מצביע לאובייקט ניהול המכיל מידע ופונקציות לטיפול בדף שבמסגרת (פינוי וטעינה) לפי סוג המיפוי (ממופה לקובץ או ממופה אנונימי).

index- מציין את המיקום הפיזי של הדף במאגר בדיסק. עבור מידע מקובץ, את הoffset מתחילת הקובץ ועבור דף מזיכרון תהליך, את מזהה המגירה.



מטמון הדפים מכיל **טבלת ערבול** המתרגמת צירוף של mapping+index לכתובת מסגרת (אם יש כזו) מתאימה. טיפול בהתנגשויות נעשה באמצעות רשימה מקושרת כפולה.

בדומה למסגרת, לכל מגירה במאגר דפדוף יש מונה שימוש- swap_map, הסופר כמה מרחבי זיכרון מצביעים לדף המאוחסן במגירה. מבחינת מונה השימוש של מסגרת או מגירה, מטמון הדפים נחשב כמרחב זיכרון נפרד המשתמש בדף המאוחסן בה.

מטמון הדפים מחזיק את הקשר בין מסגרת ומגירה, המכילות את אותו דף ממופה אנונימי כל עוד קשר זה מתקיים.

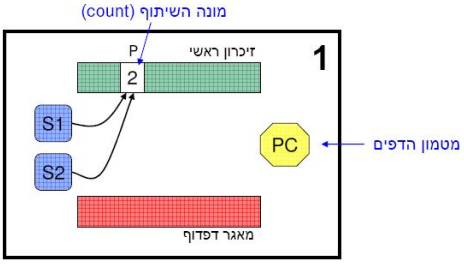
שלבי פינוי דף:

- מטמון הדפים מצביע למסגרת בזיכרון הראשי, בתוכה נמצא הדף (ה-count של המסגרת מוגדל ב-1).
- מוקצית מגירה במאגר דפדוף, אליה יעבור הדף המפונה.

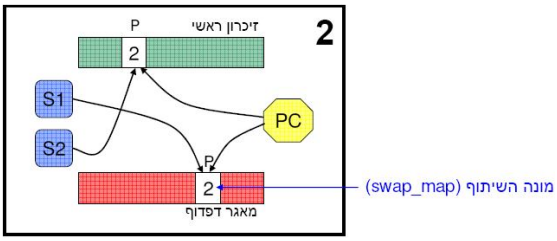
©ענת עציון

- המטמון מצביע גם למגירה זו.
- הדף נכתב למגירה במאגר הדפדוף.

<u>הערה:</u> באופן סימטרי מבוצעת גם הבאת דף ממאגר הדפדוף לזיכרון הראשי.

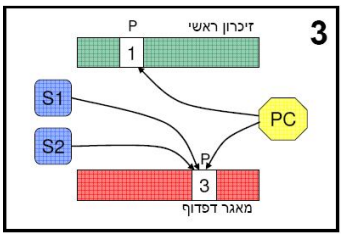


1. S1 ו-S2 הם שני מרחבי זיכרון שיש להם דף משותף P, המצוי בתחילה במסגרת בזיכרון.

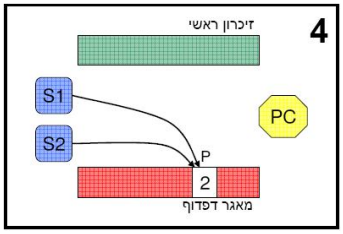


2. במעבר על הטבלאות של S1 הגרעין מחליט לפנות את P לדיסק. הדף P אינו במטמון דפים (שדה mapping במסגרת מצביע על NULL) ולכן:
   - מוקצית מגירה עבור P (אך לא מתבצעת העתקה של P).
   - המסגרת והמגירה מקושרות למטמון דפים (מעודכנים ה-mapping וה-index של המסגרת).
   - הטבלה ב-S1 עוברת להצביע על המגירה של P.
   - P עדיין לא מפונה מהזיכרון והמונה שלו נשאר 2 כי גם המטמון מצביע עליו. המונה של המגירה הוא גם 2.

הערה: אם בשלב זה S1 מנסה לגשת לדף, הוא יקבל page fault ואז נרצה לשנות את ההצבעה מהמגירה חזרה אל המסגרת. את התרגום למזהה מסגרת עושים בעזרת טבלת הערבול במטמון.



3. בהמשך הגרעין מחליט לפנות את P במעבר על הטבלאות של S2. כעת הדף כבר במטמון דפים ולכן:
   - הטבלה ב-S2 עוברת להצביע על המגירה של P ומעודכנים כל מוני השיתוף הרלוונטיים.
   - P עדיין לא מפונה פיזית מהזיכרון!



4. מאוחר יותר, מתבצע פינוי פיזי של הדף P מהזיכרון (אם ה-index=1 וגם יש מיפוי למגירה זה אומר שרק המטמון דפים מצביע ולכן ניתן לפנות).
   - תוכן המסגרת נכתב לדיסק.
   - המסגרת מוצאת ממטמון הדפים ומסומנת כפנויה (עדכון mapping ומוני שיתוף).
   - המגירה מוצאת אף היא ממטמון הדפים (עדכון מונה שיתוף).

<u>Minor Page Fault:</u> תהליך ניגש לדף שמבחינתו נמצא בדיסק אך בפועל הדף עדיין בזיכרון הראשי.

©ענת עציון