

מערכות הפעלה

"מחשב ללא מערכת הפעלה הוא כמו מכונית ללא מנוע."

-ביל גייטס

תוכן עניינים

4	הרצאה 1 – מבוא
4	מבנה המחשב
5	מבנה מערכת מחשב
8	הרצאה 2
8	ממשק משתמש
8	פונקציונליות שעוזרות למשתמש
9	System Calls
9	Virtual Machine
11	הרצאה 4
11	שיטות ל-scheduling
12	חיזוי על פי תצפיות עבר
12	Exponential smoothing
12	Priority scheduling
12	Priority Inversion
14	הרצאה 5
16	הרצאה 7 – Deadlock
16	התמודדות עם Deadlock
17	Safe state
17	אלגוריתם Banker
18	הרצאה 6
19	Paging
20	הרצאה 8 – זיכרון וירטואלי
20	Demand Paging
21	Page Fault
21	Copy-On-Write
21	Page Replacement
22	Frame Allocation
24	הרצאה 10 – File System Interface
24	Open file locking
24	Access Methods
25	Partitions
25	מבנה הקבצים במערכת
26	Protection
27	הרצאה 11 – ניהול זיכרון דיסק
27	הקצאת זיכרון בדיסק
27	Disk scheduling
28	הרצאה 12 – Threads
28	User/Kernel Threads

הרצאה 1 – מבוא

מבנה המחשב

המחשב מבוסס על מודל Von Neumann. הארכיטקטורה מבוססת על 4 חלקים :

1. ALU – Arithmetic/Logical Unit
2. CU – Control Unit
3. Memory Unit
4. I/O – Input/Output devices

Memory Unit

ניתן לבצע שתי פעולות מרכזיות :

- Fetch – מקבל ומעתיק מידע מהזיכרון. קריאה בלבד.
- Store – מאחסן מידע בזיכרון. יכול לדרוס מידע קיים.

I/O

הגישה למכשירי קלט/פלט היא איטית בהרבה, אך חשובה כדי לקבל מידע מהסביבה ומהמשתנה.

ALU

ה-ALU מיועד לבצע פעולות מתמטיות ולוגיות. ה-ALU מורכב מ:

- מעגלים לביצוע פעולות אריתמטיות/לוגיות.
- רגיסטרים לאחסון תוצאות חישוב ביניים.
- אפיק (Bus) שמחבר בין השניים.

Control Unit

התכנית שלנו מאוחסנת בזיכרון כפקודות בשפת מכונה. תפקידה של יחידת הבקרה הוא להריץ את התכנית על ידי חזרה על השלבים :

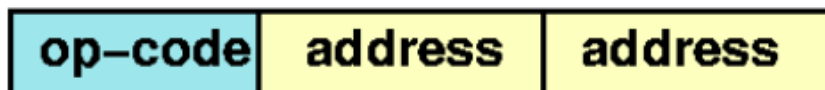
1. Fetch – שליפת הפקודה הבאה לביצוע מהזיכרון.
2. Decode – קביעת הפעולה שיש לבצע.
3. Execute – ביצוע הפעולה על ידי שליחת אותות מתאימים ל-ALU, לזיכרון ולתתי המערכות קלט/פלט.
4. ממשיך עד לפעולת HALT (עצירה).

פקודות בשפת מכונה :

מבנה פקודה

פקודה מורכבת מ :

- קוד פעולה : מציין איזו פעולה יש לבצע.
- שדות כתובת : מציינים את כתובת הזיכרון של הערכים שעליהם פועלת הפעולה.



מבנה ה-CU

ה-CU בנוי מ :

- Program Counter (PC) - שומר את הכתובת של הפקודה הבאה.
- Instruction Register (IR) – שומר את הפקודה שנשלפה מהזיכרון.

מבנה מערכת מחשב

ניתן לחלק את מערכת המחשב לארבעה מרכיבים :

1. אפליקציות
2. מערכת ההפעלה
3. חומרה
4. משתמש

הגדרת מערכת ההפעלה

מערכת ההפעלה היא **תכנית** שחוצצת בין המשתמש של מערכת המחשב והחומרה.

מערכת ההפעלה יכולה להימצא בכל מקום, החל ממחשבים ופלאפונים ועד מכונות ובסביבות ענן.

המטרות של מערכת ההפעלה הן הרצת תכניות המשתמש בצורה קלה, הפיכת השימוש במחשב לנוח יותר, ושימוש יעיל יותר בחומרת המחשב.

לכל מערכת הפעלה מטרות שונות, ובאופן כללי כל מערכת הפעלה בוחרת בין יעילות לנוחות.

מטרה מרכזית: חלוקת משאבים, והחלטה מה לעשות במצב של קונפליקט, על מנת שיהיה שימוש הוגן ויעיל במשאבים. כמו כן מערכת ההפעלה אחראית על ההרצה של התוכניות השונות.

הגדרה: החלק במערכת ההפעלה שרץ כל הזמן, ללא הפסקה, נקרא **kernel**. כל שאר התהליכים מקוטלגים ל-System Programs או ל-Application Program.

הפעלת המחשב

בהדלקת המחשב עולה bootstrap program, שמטרתו לטעון את מערכת ההפעלה. זהו הקוד הראשון שרץ עם הדלקת המחשב.

Device Controller

כל Device Controller אחראי על סוג מסויים של מכשיר.

המידע מ/אל המכשיר מנוהל באמצעות local buffer.

מעבדים

ברוב המערכות יש מעבד אחד או יותר מסוג general purpose.

בנוסף יש מעבדים מסוג special purpose – המותאמים לפעולות ספציפיות. למשל DSP או GPU.

Multiprocessors

מערכת עם שני מעבדים ויותר החולקים Bus ולפעמים גם clock וזיכרון.

ישנם שני סוגים של מערכות Multiprocessors :

1. א-סימטריות: משימות מסוימות מוקצות למעבדים מסוימים בלבד. ייתכן למשל שמעבד יחיד יהיה אחראי על כל פעולות הקלט/פלט וכו'.
2. סימטרי: מתייחס לכל רכיבי העיבוד במערכת באופן זהה.

Clustered Systems

כמה מערכות שונות שעובדות ביחד.

- בד"כ חולקים זיכרות דרך אחסון בשם SAN.
- מספק שרתים עם פניות גבוהה ששורדת כשלונות.
 - Asymmetric Clustering – מכונה אחת במצב המתנה חס.
 - Symmetric Clustering – כמה קודקודים שרצים על אפליקציות ומכוונים אחד את השני.
- חלק מה-clusters מכוונים ליעילות גבוהה, כאשר אפליקציות בהם חייבות להיכתב עם שימוש במקבול.

פסיקה

פסיקה (Interrupt) היא אות המתקבל במעבד מרכיב חומרה.

הפסיקה נוצרת על ידי מכשיר, והיא מעבירה את השליטה במעבד ל-interrupt service routine. בעצם המעבד עוזב הכל ושומר את כתובת ה-interrupted instruction ושל כתובת הרגיסטרים.

מכשיר -> בקר פסיקות -> מעבד -> ליבה -> זיכרון.

משם נחזור לליבה ונריץ את הקוד ששמרנו בזיכרון על מנת לטפל בפסיקה.

Trap

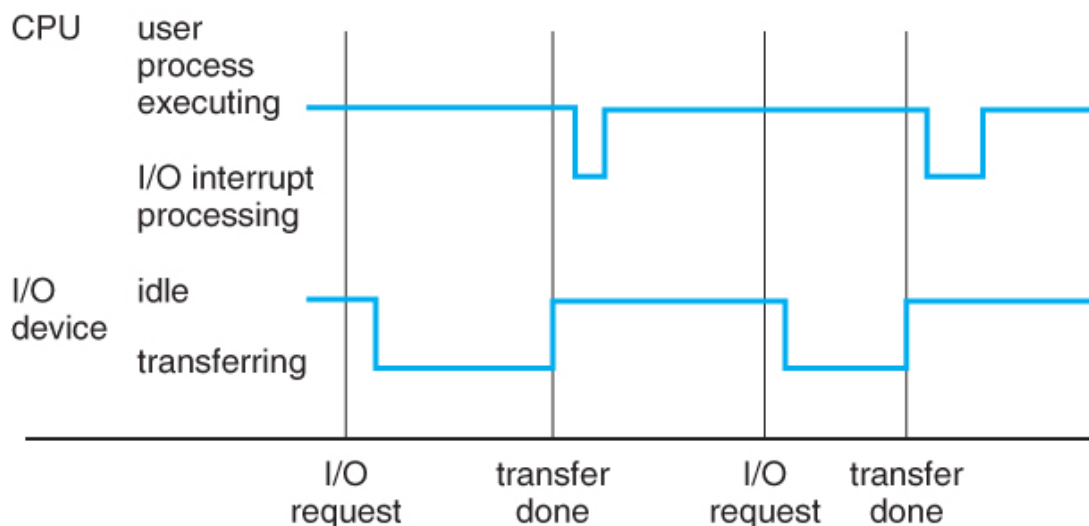
Trap הוא כמו פסיקה, רק שייצרה אותו תוכנה ולא מכשיר. Trap נגרם על ידי שגיאה או בעקבות בקשה של משתמש.

תחילה יש לקבוע איזה סוג של interrupt קרה (תזוזה של העכבר, מקלדת, וכו').

לאחר מכן ישנן שתי דרכים לטפל בפסיקה:

1. Polling - נעבור על סוגי ה-interrupts האפשריים ונבצע את הקוד שנרצה.
2. Vector - נשמור מה נצטרך לעשות עבור כל סוג של interrupt. מהיר יותר.

טיפול בפסיקות שונות:



I/O בקשות

ישנן שתי דרכים לטפל בבקשות I/O: סינכרונית וא-סינכרונית.

א-סינכרונית: רק כאשר תסתיים הפעולה של ה-I/O, נשלח את הפסיקה.

סינכרונית: נשלח בתחילת הפעולה, והשליטה חוזרת רק כאשר תסתיים.

אם יש כמה בקשות מאותו המכשיר, נשרשר את הבקשות לפי סדר בתור.

ניהול זיכרון

עוסק בהחלטה מה יהיה בזיכרון בכל רגע.

מערכת ההפעלה אחראית על מעקב ורישום החלקים בזיכרון שנמצאים כרגע בשימוש, החלטה על העברת תוכן זיכרון המשוך לתהליכים מ/אל הזיכרון.

זיכרון משני (אחסון)

דיסק חיצוני בנוסף לזיכרון ה-RAM.

משמש לשמירה של זיכרון רב/זיכרון שרוצים לשמור לאורך זמן.

המהירות בהקשר זה תלויה בדיסק.

ככל שעוברים לזיכרון קרוב, כך הוא גם מהיר יותר (אך גם נדיף יותר).

Caching

העברת מידע מאחסון מסויים לאחסון מהיר יותר. תמיד נבדוק קודם את האחסון המהיר ביותר, על מנת לבדוק האם המידע נמצא שם. אם לא, נמשיך לחפש בהיררכיית הזיכרון עד שנמצא אותו. לבסוף כשנמצא אותו המידע יועתק ל-cache וישמש משם.

כאשר יש מספר מעבדים, נאלץ לספק עקביות בחומרה על מנת שכל המעבדים יראו את הערך העדכני ביותר ב-cache שלהם.

DMA – Direct Memory Access

מחבר בין הזיכרון למכשיר.

משתמש להתקני קלט/פלט מהירים, המסוגלים להעביר מידה במהירות הקרובה למהירות הזיכרון. כך אין צורך ב-interrupt כל פעם שנרצה להעביר מידע מהזיכרון לאחסון חיצוני או להיפך. רק נזדקק ל-interrupt בסיום.

לא טוב להשתמש בו במקומות בהם מעבירים דברים חשובים ומעטים. למשל לא נרצה להשתמש בו עבור הקלדות מקלדת, מכיוון שכך הוא ישמור את כל ההקלדות ובבת אחת יקליד את הכל בסיום.

Multiprogramming

בעיה: כאשר יש לנו תכנית בודדת, היא לא יכולה לשמור את המעבד וההתקנים יחד בניצולת מלאה. התכנית רצה בצורה סדרתית ולפעמים מבצעת חישובים ולפעמים פונה ל-I/O.

אידיאלית היינו רוצים שבכל רגע נתון תהיה לנו תוכנית שתרוץ על המעבד.

בעזרת Multiprogramming נאפשר הרצה של מספר תוכניות במקביל (יותר מכמות המעבדים).

Timesharing

עדיין עושה Multiprogramming, אך מחליף בתדירות גבוהה כל כך, שהמשתמש לא ישים לב שהתוכנית הפסיקה.

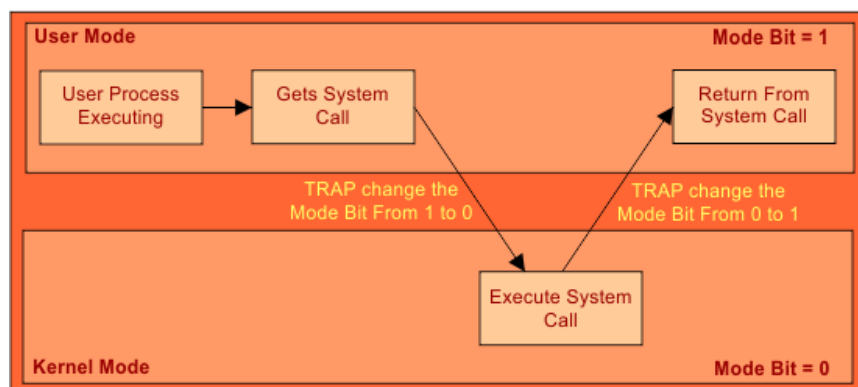
טיפול בבעיות זרימה (Flow)

מערכת ההפעלה מבצעת הגנה מפני לולאות אינסופיות, תהליכים שמנסים לשנות זיכרון של תהליכים אחרים וכו'.

מעבר בין User Mode ו-Kernel Mode

הבעיה – מערכת ההפעלה נדרשת להגן על עצמה מפני שימוש לא נכון/זדוני.

הפתרון: נייצר ביט שיושב על המעבד. כשהיה דלוק זה סימן שאנחנו מריצים קוד של המשתמש. אחרת זה סימן שאנחנו פועלים בשם מערכת ההפעלה. אם הוא על kernel מותר לו להריץ הכל. אם הוא על User הוא לא יאפשר פעולות מסויימות על מנת להגן על עצמו.



הרצאה 2

הערה לפתיחה: ההבדל בין מערכת הפעלה לממשק משתמש פשוט היא בעושר פעולות שניתן לבצע.

ממשק משתמש

כמעט לכל מערכות ההפעלה יש UI, המיועד לקלוט את רצון המשתמש ולהפוך אותו לפעולות. בפרט, יש במערכות הרבה פעמים ממשקי CLI – command line. מטרתו היא לקלוט פקודות שהמשתמש רושם. הממשק הנפוץ ביותר כיום הוא GUI – ממשק גרפי. VUI – ממשק קולי (אלקסה, סירי וכו') ישנם עוד ממשקים רבים, שמטרתם זהה – לקלוט את רצון המשתמש ולהמיר אותו לפעולות. נרחיב בעיקר על ממשק ה-CLI.

CLI

המשתמש מזין פקודה, ה-CLI לוקח את הפקודה והופך אותה לרצף של system calls. ישנן כמה דרכים לממש CLI:

1. כחלק מה-kernel. (תמיד נמצא בזיכרון, ה-command interpreter קופץ למקום המתאים בזיכרון).
 2. באמצעות תכניות השמורות בדיסק, כשכל פקודה היא למעשה שם של תכנית.
- זמן הריצה מהיר יותר ב-kernel. לעומת זאת בדיסק קל יותר לתחזק (הוספת feature בשיטה השנייה לא דורשת שינוי של ה-shell).
- הרבה פעמים יהיו לנו כמה interpreters במערכת ההפעלה – נקרא להם shells.

פונקציונליות שעוזרות למשתמש

הרצת תכנית

1. טעינת התכנית לזיכרון.
2. הרצת התכנית.
3. סיום הרצה – בין אם התכנית מסיימת בעצמה או מתוך שגיאה/הפרעה.

טיפול ב-I/O

תמיד נרצה שהגישה ל-I/O תעבור דרך מערכת ההפעלה משיקולי בטיחות.

טיפול בקבצים

כתיבה וקריאה של קבצים, מחיקה ויצירה שלהם, חיפוש, הדפסת המידע שלהם, הרשאות.

תקשורת בין תהליכים

העברת המידע בין שני תהליכים או יותר – מדובר בתקשורת גם בין אם התהליכים הם באותו מכשיר וגם אם לאו.

טיפול בשגיאות

שגיאות במעבד ובזיכרון, שגיאות בהתקני I/O, שגיאות בתכניות שרצות. לכל סוג שגיאה על מערכת ההפעלה לנקוט בפעולה המתאימה כדי לטפל בה, על מנת להבטיח חישוב נכון ועקבי.

הבטחת יעילות של המערכת

הקצאת משאבים חכמה, כאשר כמה משתמשים מחוברים במקביל.

Accounting

רישום של מערכת ההפעלה של כל השימושים במשאבים. פעול כמו Log.

אבטחה

להבטיח שכל גישה למשאבי המערכת תקינה ותחת שליטה של המערכת.

System Calls

System calls הם מנגנון המאפשר לתכנות שרצות על המחשב לבקש שירותים מסוימים ממערכת ההפעלה.

הן אלו שמהוות את ממשק התכנות שלנו לשירותים המסופקים על ידי מערכת ההפעלה. המימוש שלהן בוא בדרך כלל באמצעות מספר שמיוחס לכל system call.

לרוב הקריאה ל-system calls לא יתבצעו על ידי קריאה ישירה אלא על ידי Application Programming Interface – API.

ה-API מגדיר את סט הפונקציות הזמינות.

יתרונות לקריאה דרך API :

1. פשוט יותר לשימוש : אין צורך לדעת את כל מה שקורה מאחורי הקלעים.
2. ניידות וגמישות : שינוי במבנה או הוספה ל-system calls לא תשפיע על מי שמשמש ב-API.

חסרונות לקריאה דרך API :

1. ביצועים נמוכים יותר, מכיוון שצריכים לעבור קודם ב-API שעובר לאחר מכן ב-syscalls.
2. שליטה פחות מדויקת. נובע מהעובדה שבחרנו לא לדעת את כל הפרטים הקטנים מאחורי הקלעים.

דוגמא : נניח שמשמש קרא ל-syscall של open(). בפועל הוא קורא ל-API, שימצא את ה-syscall interface. הוא ימצא את המימוש של open ויחזיר אותו.

דוגמא נוספת : printf היא פונקציית API מתקדמת, שמשתמשת ב-formatting, מכינה טקסט וכו'. היא משתמשת ב-write מאחורי הקלעים. write עצמה היא פונקציה פשוטה יותר שמדפיסה את הטקסט. היא משתמשת ב-system_write מאחורי הקלעים.

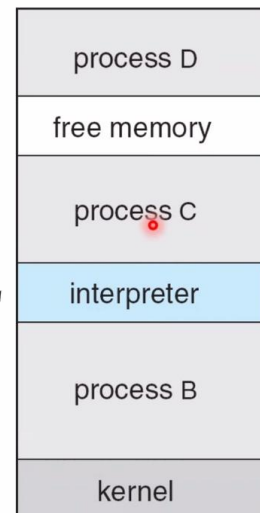
אם נדרשים להעביר פרמטרים ל-syscall, אנחנו יכולים להעביר דרך רגיסטר, או לשים את המידע בזיכרון ולהגיד למערכת ההפעלה איפה בזיכרון המידע נמצא. כמו כן אנחנו יכולים גם כמובן להעביר דרך ה-stack.

מבנה UNIX :

Example: FreeBSD

- Unix variant
- Multitasking
- User login -> invoke user's choice of shell
- Shell executes fork() system call to create process
 - Executes exec() to load program into process
 - Shell waits for process to terminate or continues with user commands
- Process exits with:
 - code = 0 – no error
 - code > 0 – error code

Interpreter is always running because of the multiprogramming



Virtual Machine

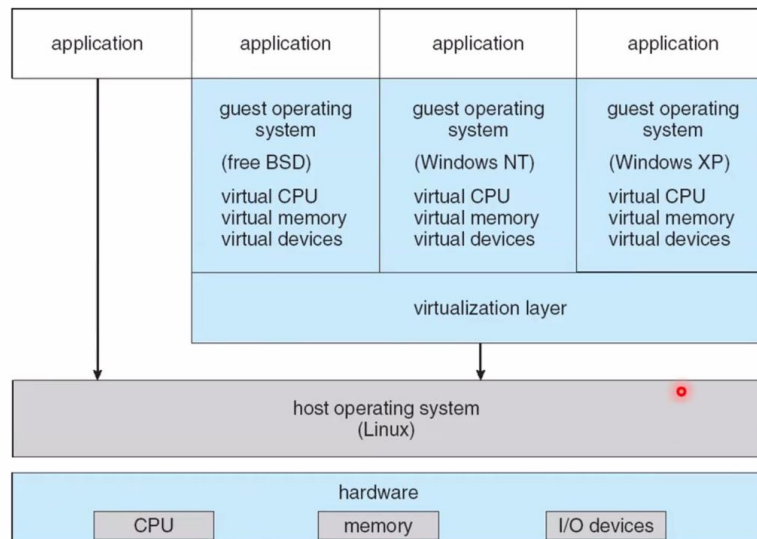
מתייחס לחומרה ולמערכת ההפעלה כאילו שניהם יחד הם רק חומרה. מערכת ההפעלה הקיימת היא המארח.

ה-VM משתמש ב"שכבת וירטואליזציה" על מנת לדמות מערכת הפעלה אחרת ממה שקיימת בפועל.

מה ששכבת הוירטואליזציה עושה בפועל זה לתרגם את פקודות ה-API של ווינדוס למשל לפקודות של לינוקס, ואז להשתמש בלינוקס שמותקן כבר כדי להריץ אותן.

הארכיטקטורה של ה-VM נראית כך :

VMware Architecture



Debugging

המחשב שומר Log Files לגבי מידע השגיאה. למשתמשים אין הרבה מה לעשות עם זה, אבל הרבה פעמים ניתן לשלוח את המידע ל-Microsoft והם ידעו לפתור את הבעיה.

הרצאה 4

מחזור החיים של תהליך בנוי מרצף של מעברים בין CPU execution ו-I/O wait.

מחזור החיים תמיד מתחיל ומסתיים ב-CPU burst.

CPU scheduler

ה-CPU scheduler אחראי על לבחור איזה תהליך מבין התהליכים שנמצאים ב-ready queue יקבל את המעבד.

מצבים בהם כדאי להפעיל את ה-scheduler :

1. כאשר תהליך עובר ממצב running ל-waiting.
2. כאשר תהליך עובר ממצב running ל-ready.
3. כאשר תהליך עובר ממצב waiting ל-ready.
4. כאשר תהליך מסיים את פעולתו.

אם ה-scheduler פועל רק באירועים מהסוג הראשון והרביעי, אנחנו אומרים שהוא nonpreemptive. במקרה כזה, המעבד מחזיק בתהליך כלשהו עד שהוא מסיים או עובר למצב waiting.

Dispatcher

ה-Dispatcher בפועל מבצע את ההחלטות שה-scheduler קיבל. מעביר את השליטה על המעבד לתהליך שנבחר.

תפקידו :

1. ביצוע context switch
2. מעבר ל-user mode
3. עוד

קריטריוני Scheduling

לפי מה נקבע כמה טוב ה-scheduling?

1. ניצולת המעבד – נשאף לניצולת גבוהה ככל האפשר.
2. Throughput – מספר התהליכים המסיימים ריצה בפרק זמן.
3. Turnaround time – משך הזמן הכולל להרצת התהליך. כולל:
 - הזמן שהמתין עד שהוכנס לזיכרון.
 - המתנה ב-ready queue.
 - זמן ההרצה על המעבד.
 - הזמן שבילה בביצוע (או המתנה ל-I/O).
4. Waiting time – הזמן היחיד שבאמת מושפע מה-scheduling.
5. Response time – הזמן מהרגע שהתהליך ביקש את המעבד ועד הרגע שקיבל אותו לראשונה.

קריטריון minmax

נעדיף לא רק למזער את הזמן, אלא שכל התהליכים יקבלו זמנים יחסית שווים. נעדיף זמן של 20 שניות בממוצע, כשכל תהליך מקבל שניה, מאשר 15 שניות בממוצע שיש תהליך אחד שלוקח 10 שניות.

שיטות ל-scheduling

first come-first serve. מי שמגיע קודם מקבל שיבוץ למעבד קודם.

Shortest job-first – בוחר את התהליך שאורך ה-cpu burst שלו הוא מינימלי. ניתן לממש זאת בשתי דרכים :

1. Nonpreemptive – ברגע שתהליך שובץ, אפילו אם יגיע אחד עם זמן קצר יותר מהזמן שנותר לתהליך הנוכחי, נחכה שיסתיים התהליך הנוכחי.
2. Preemptive – אם מגיע תהליך חדש עם CPU burst קצר יותר מהזמן שנותר לתהליך הנוכחי, ניתן לו לרוץ ואז נחזור.

ה-sjf מבטיח את הזמן המתנה הממוצע המינימלי עבור כל סט של תהליכים – כאשר הוא preemptive וזמן ה-context switch זניח. זה מתבסס על ההנחה שהחלפה של תהליך ארוך בקצר תביא בהכרח להקטנת זמן ההמתנה הממוצע. הבעיה העיקרית בשיטה היא שאנחנו לא באמת יודעים את משך ה-burst הבא.

חיזוי על פי תצפיות עבר

דרך שכיחה לשער כמה זמן ייקח ה-burst הבא היא על פי תצפיות עבר.

Exponential smoothing

השיטה מספקת ממוצע משוקלל נע של כל תצפיות העבר. מתאים בעיקר למידע שאין לו מגמה (עלייה או ירידה). המטרה היא להעריך את הממוצע הנוכחי ולהשתמש בה כאומדן לעתיד. באופן פורמלי:

$$F_{t+1} = \alpha y_t + (1 - \alpha)F_t$$

כאשר F_{t+1} הוא החיזוי ל-burst הבא. α הוא קבוע ה-smoothing. y_t הוא הערך הנצפה בסדרה במחזור t . F_t היא התחזית הקודמת.

בעצם כל פעם "מתקנים" את התחזית הקודמת בהתבסס על הערך החדש שנצפה בפועל. α קובע כמה נרצה להסתמך על התצפית האחרונה/על התחזיות הקודמות.

הנוסחה שקולה לנוסחה:

$$F_{t+1} = \alpha y_t + \alpha(1 - \alpha)y_{t-1} + \dots + \alpha(1 - \alpha)^t y_0$$

ערך α גבוה – אנחנו מסתמכים יותר על התצפיות האחרונות.

ערך נמוך – יציבות בחיזוי. נסתמך יותר על התחזיות הקודמות.

Priority scheduling

לכל תהליך מצמידים מספר שמייצג את החשיבות שלו. המעבד מוקצה לתהליך עם החשיבות הגבוהה ביותר. ניתן לבצע את השיטה הזו בעזרת צורת preemptive או nonpreemptive.

ה-SJF הוא סוג של priority scheduling, שבו ה-priority הוא משך ה-burst הבא.

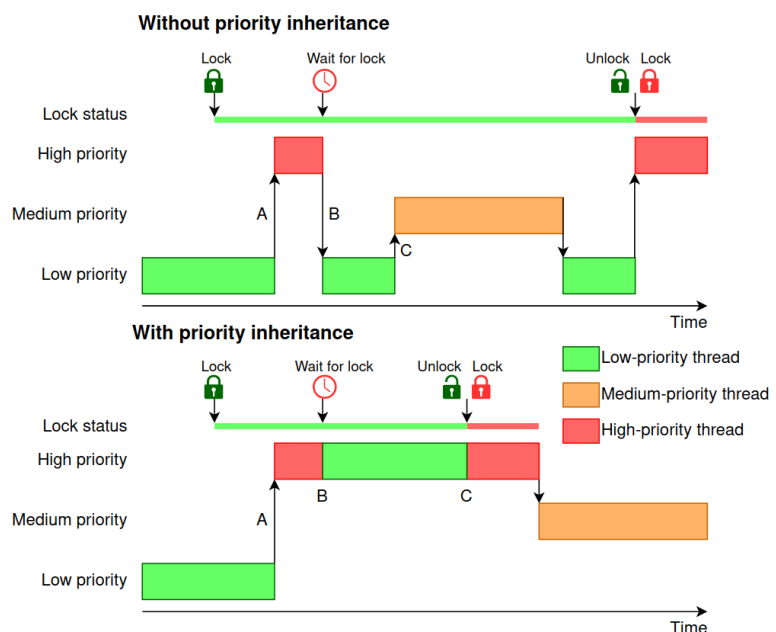
הבעיה העיקרית בשיטה זו היא Starvation/Aging.

Priority Inversion

נניח שתהליך עם חשיבות נמוכה פותח קובץ, ואז אנחנו עוברים לתהליך עם חשיבות גבוהה. כעת התהליך עם החשיבות הגבוהה ניסה לפתוח את אותו הקובץ. נצטרך לחזור לתהליך הקודם, להפסיק להסתיים בקובץ, ורק אז לחזור לתהליך עם החשיבות הגבוהה.

בעיה: אם כשנחזור לתהליך עם החשיבות הנמוכה ייכנס עוד תהליך עם חשיבות בינונית, נצטרך לחכות עוד זמן נוסף עד שנוכל לחזור לתהליך החשוב.

פתרון: נשנה את החשיבות של התהליך הנמוך לחשיבות של התהליך הגבוה שתלוי בו.



Round Robin

שיטה המיועדת במיוחד למערכות time-sharing.

כל תהליך מקבל זמן יחסית קצר על גבי ה-CPU. אם תהליך לא הסתיים, הוא מועבר חזרה לסוף התור.

Multilevel Queue

נחלק את ה-ready queue לכמה תורים נפרדים. לכל תור נקצה אלגוריתם scheduling משלו (לא בהכרח זהים). ה-scheduling מבוצע בין התורים. למשל נוכל להחליט שאנחנו תמיד נעדיף לשרת תהליך מתור ה-foreground.

כמו כן, כל תור יקבל כמות מסויימת של זמן מעבד שאותה יוכל לחלק בין התהליכים. למשל:

80% מהזמן ל-foreground עם אלגוריתם RR. 20% מהזמן ל-background עם FCFS.

ב-MLFQ, הרמה הנמוכה ביותר בדרך כלל תתנהל בעזרת FCFS.

Scheduling עם יותר ממעבד אחד

נפריד בין שני מצבים:

1. Asymmetric multiprocessing – רק מעבד אחד פונה ל-data structures הקשורים בתזמון ומנהל את התזמון.

2. Symmetric processing – כל מעבד מתזמן לעצמו.

Processor affinity – לתהליך יכול להיות affinity (צמידות) למעבד עליו הוא כרגע רץ. זה על מנת להשתמש ב-cache באופן היעיל ביותר. מערכת ההפעלה תומכת ב-affinity בשתי צורות:

1. Soft affinity – אין הבטחה שאותו המעבד יהיה בשימוש.

2. Hard affinity – אם התהליך מבקש, הוא לא יועבר למעבד אחר.

מערכות Hard real-time – מבקש לסיים משימה קריטית בתוך פרק זמן מסויים. המתזמן יבטיח את קיום הבקשה או שידחה אותה מראש.

מערכות Soft real time – תהליכים קריטיים יקבלו עדיפות, אך לא מובטח שהבקשה תתקיים בזמן המבוקש.

שערוך אלגוריתמים

ניתן להעריך את המערכת היעילה ביותר לאלגוריתם נתון בכמה דרכים:

1. לוקח סביבת עבודה נתונה מראש ומגדיר את היעילות של כל אלגוריתם על סביבת העבודה הזו.
2. תורת התורים – תחום שעוסק בחישוב ביצועים של מערכת ניהול תורים.
3. סימולציה – נריץ סימולציה של המתזמן וכך ניתן לחשב את זמן ההמתנה.
4. אימפלמנטציה – נבדוק את כל המתזמנים ונבדוק עם מה היה הכי טוב.

הרצאה 5

מנגנון סינכרוניזציה

אם למשל נותנים לשני תהליכים לגשת בו זמנית למשתתף מסוים, יכולות להיות בעיות (מידע לא מסונכרן וכו'). למצב כזה, שבו כמה תהליכים משנים את אותו ה-data נקרא race condition.

Critical section – קטע קוד שבו התהליך משנה משתנים משותפים עם תהליכים אחרים.

בהינתן מערכת עם n תהליכים $\{p_0, p_1, \dots, p_{n-1}\}$, שלכל אחד יש critical section, נרצה שאם תהליך נמצא ב-critical section שלו, אף תהליך אחר לא יריץ את ה-critical section שלו.

נבנה פרוטוקול, לפיו כל פעם שאנחנו נכנסים וכל פעם שנצא מ-critical section, נריץ קוד כלשהו, שינהל את הכניסה/יציאה.

פתרונות לבעיה

פיטרסון:

פתרון מבוסס קוד, מבוסס על שני משתנים משותפים:

```
Int turn;  
Boolean flag[2];
```

המשתנה turn קובע תור איזה תהליך.

המשתנה flag מסמל מי מבין שני התהליכים רוצים להיכנס ל-critical section.

```
Flag[i] = true;  
Turn = j;  
While(flag[j] && turn==j);  
    // Critical section  
Flag[i] = false;
```

כל הפתרונות מבוססים על lock כלשהו. כמו רמזור שלא נותן למכוניות מכמה כיוונים להיכנס ביחד.

Lock

נעל לפני ונפתח אחרי ה-critical condition. בעיה: יכול להיות שניאלץ לחכות זמן לא מוגבל.

הגדרה: פקודה אטומית היא פקודה שלא ניתן לעשות לה interrupt.

דוגמאות לפקודות אטומיות:

1. Test_and_set – מחזיר את הערך של המשתנה המקורי, מעדכן את הערך שלו ל-true.
2. Swap – מחליף בין ערכם של שני משתנים בוליאניים.

Bounded waiting

התהליך מחכה שיתפנה הקטע ומתחיל להריץ אותו. לאחר מכן התהליך בודק בצורה ציקלית אם יש תהליך אחר שמחכה לרוץ. אם כן הוא נותן לו את הגישה והקטע עובר לתהליך הזה.

הפרוטוקול הזה מקיים את כל 3 התנאים:

1. Mutual exclusion – רק אחד בכל פעם עם גישה.
2. Progress – בוודאות אחד מהתהליכים הממתינים יוכל להתקדם.
3. Bounded waiting – במקרה הכי גרוע מחכים $O(n - 1)$.

Semaphore

משתנה שניתן לבצע עליו פעולות indivisible. משמש לנכסרון.

בעיית הפילוסופים האוכלים

ישנם 5 פילוסופים מסביב לשולחן, שבכל רגע הם הוגים או אוכלים. על השולחן 5 מקלות אכילה ובמרכז קערת אורז. כאשר פילוסוף נהיה רעב הוא מנסה להרים את שני מקלות האכילה הקרובים אליו.

אלגוריתם:

נחכה לשני מקלות האכילה, ואז נאכל. נחזיר אותם ואז נחזור לחשוב.

בעיה: עלול לגרום ל-starvation.

פיטרסון עם יותר משני תהליכים

כל תהליך מתקדם בשלבים לעבר הכניסה ל-critical section.

כל פעם התהליך יסמן באיזה שלב הוא ב-flag (1,2 עד $n - 1$). בנוסף ב-turn נסמן את התהליך באינדקס של השלב. (אם אני תהליך i בשלב k , אזי $turn[k] = i$.)

ברגע שכל התהליכים בשלבים שהם נמוכים משל התהליך, או שאחד התהליכים האחרים "חשב" שהוא לפני התהליך, זה סימן שניתן לקפוץ לראש התור ולהריץ את ה-critical section. לאחר מכן נסמן את השלב שלנו ב-1 ונחזור על התהליך.

הרצאה 7 – Deadlock

Deadlock – מצב בו כל תהליך מחכה לתהליך אחר שיסיים (כדי שמשאב כלשהו ישתחרר). יוצא מצב שכל התהליכים מחכים בלי להתקדם.

מבנה המערכת

מערכת מורכבת מתהליכים ומשאבים. משאבים יכולים להיות consumable או reusable. את המשאבים אנחנו מחלקים לקבוצות שנקראות resource types: R_1, R_2, \dots, R_m . לכל משאב R_i יש W_i מופעים.

אנחנו נשתמש במשאבים באופן של request-use-release.

תנאים לדד-לוק

נגיד שאנחנו במצב של דד-לוק, אם מתקיים:

1. Mutual exclusion – בכל רגע נתון רק תהליך אחד יכול להשתמש במשאב מסוים.
2. Hold and wait – תהליך שמחזיק לפחות משאב אחד מחכה לקבל משאבים נוספים שמוחזקים על ידי תהליכים אחרים.
3. No preemption – משאב יכול להיות משוחרר רק כשהתהליך שמחזיק אותו שחרר אותו מרצונו.
4. Circular wait – קיים סט תהליכים ממתינים $\{P_0, \dots, P_n\}$ כך ש- P_0 מחכה למשאב שמוחזק על ידי P_1, P_1 מחכה למשאב שמוחזק על ידי P_2, P_2, \dots, P_{n-1} מחכה למשאב שמוחזק על ידי P_n, P_n מחכה למשאב שמוחזק על ידי P_0 .

גרף הקצאת משאבים

נניח תהליכים באמצעות קודקודים, ואת המשאבים באמצעות ריבועים. כמות הנקודות בתוך הריבועים מייצגים את כמות המופעים של אותו המשאב.

כעת חץ מהתהליך למשאב מסמל שהתהליך מבקש להשתמש במשאב, וחץ מהמשאב לתהליך מסמל שהמשאב מוקצה לתהליך.

נשים לב: אם הגרף לא מכיל מעגל \Leftarrow אין דד-לוק.

אם הגרף מכיל מעגל \Leftarrow אם לכל משאב יש מופע אחד, זה בהכרח דד-לוק. אחרת, יש רק אפשרות לדד-לוק.

התמודדות עם Deadlock

יש לנו שלוש שיטות להתמודד עם דד-לוק:

1. נוודא שאנחנו לא נכנסים אף פעם לדד-לוק, באמצעות תקיפת ארבע התנאים לדד-לוק.
2. נמצא דרך להתאושש מדד-לוק במצב שכן נכנסו אל אחד.
3. התעלמות – נתעלם מהעובדה שהמערכת יכולה להיכנס לדד-לוק.

איך נוודא שלא ניכנס לדד-לוק? איך נתמודד עם כל אחד מהתנאים?

1. Mutual exclusion – לא נדרש עבור משאבים שיתופיים (למשל קבצי read-only). עבור משאבים אחרים נצטרך לוודא שתקף.
2. Hold and wait – ננסה לוודא שכאשר תהליך מבקש משאב, הוא לא מחזיק אף משאב אחר. למשל נוכל להחליט שתהליך יבקש את כל המשאבים שהוא צריך מהתחלה, ואז נקצה לו הכל בבת אחת. אופציה נוספת היא לאפשר לו לבקש משאבים רק כאשר המשאב לא משומש על ידי אף תהליך אחר. בעיה: הפתרונות האלה עלולים לגרום ל-starvation.
3. No preemption – אם תהליך שמחזיק משאבים מבקש משאב נוסף שלא ניתן להקצות לו מייד, נשחרר את כל שאר המשאבים שלו. משאבים שנקבעו מראש נוספים לרשימת המשאבים שהתהליך ממתין להם. תהליך יאותחל רק כשהוא יכול לקבל מחדש את המשאבים שהוקצו לו, כולל החדשים שהוא ביקש.
4. Circular waiting – קודם נסדר את כל המשאבים. לאחר מכן נדרוש שכל תהליך יבקש משאבים בסדר עולה של ספירה.

גישה שנייה – התחמקות מדד-לוק בזמן הריצה

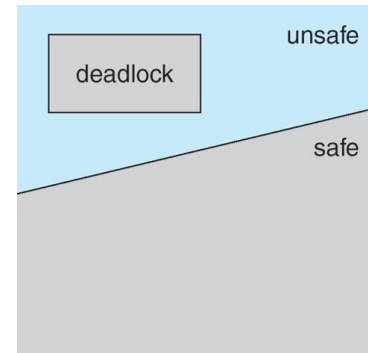
הגישה הפשוטה ביותר דורשת שכל תהליך יצהיר על כמות מקסימלית של משאבים שהוא עלול להזדקק להם. לאחר מכן נשתמש ב deadlock-avoidance algorithm כדי לוודא שלא מתקיים ה-circular wait condition. מצב הקצאות משאבים מוגדר על ידי: 1. כמות המשאבים הזמינים, 2. כמות המשאבים המוקצים, 3. הדרישה המקסימלית של התהליכים.

Safe state

נגיד שהמערכת נמצאת ב-safe state, אם קיים רצף של כל התהליכים במערכת (P_1, P_2, \dots, P_n) כך שלכל P_i , כל המשאבים שהוא עדיין יכול לבקש הם פנויים או שמוחזקים על ידי P_j כאשר $j < i$.

כך, כל פעם שתהליך P_i מסיים, התהליך P_{i+1} יכול לרוץ.

נשים לב: safe state גורר שבהכרח אין דדלוק, אבל אם אנחנו לא ב-safe state זה לא בהכרח אומר שיהיה דד-לוק.



כיצד נשמור על המערכת ב-safe state?

- אם לכל משאב יש מופע אחד, נשתמש בגרף הקצאת משאבים.
- אחרת, נשתמש באלגוריתם Banker.

בגרף נסמן $R_j \rightarrow P_i$ עם חץ מקוקו, אם ייתכן ש- P_i יבקש את המשאב R_j .

כשתהליך מבקש את המשאב בפועל, החץ יהפוך לחץ רגיל (לא מקוקו). כאשר הוא מקבל את המשאב החץ יהפוך מחץ בקשה לחץ הקצאה (בפועל הוא ישנה כיוון בגרף).

אלגוריתם לגרף הקצאת משאבים: בקשה $P_i \rightarrow R_j$ תתקבל רק אם אחרי שהתקבלה לא נוצר מעגל.

אלגוריתם Banker

ניצור את המשתנים הבאים:

Available – רשימה של m מספרים. אם $Available[i] = k$, קיימים k מופעים פנויים של R_i .
 Max – מטריצה $n \times m$. אם $Max[i, j] = k$, תהליך P_i יכול לבקש לכל היותר k מופעים של R_j .
 Allocation – מטריצה $n \times m$. אם $Alloc[i, j] = k$, אז P_i כרגע מקצה k מופעים של R_j .
 Need – מטריצה $n \times m$. אם $Need[i, j] = k$, אז P_i צריך עוד k מופעים של R_j על מנת לסיים את המשימה.

האלגוריתם:

1. נגדיר וקטורים $work = Available$ ו- $finish[i] = false$ ($i = 1, \dots, n$).
2. נמצא i עבורו:
 $finish[i] = false$
 $Need_i \leq work$
 אם לא קיים i כזה, דלג לשלב 4.
3. $work += Allocation$
 $finish[i] = true$
 לך לשלב 2.
4. אם $finish[i] == true$ לכל i , המערכת נמצאת ב-safe state.

הרצאה 6

כאשר שני CPU מחלקים ביניהם זיכרון, כל אחד שומר את הזיכרון ששלו בשני אופנים – כתובת פיזית וכתובת לוגית.

כתובת פיזית: הכתובת של משהו במיקום הממשי שלו על ה-RAM.

כתובת לוגית: הכתובת של משהו ביחס לזיכרון המוקצה לאותו CPU או תהליך.

על מנת להשתמש בכתובות לוגיות, שהן נוחות בהרבה לשימוש, אנחנו נצטרך להצמיד לכל כתובת פיזית כתובת לוגית, או לפחות למצוא דרך "לתרגם" ביניהם. נוכל לעשות זאת או בזמן הקימפול (זה יחייב קומפילציה מחדש במקרה ורוצים להריץ במקום אחר בזיכרון), או בזמן ה-loading (זה יחייב יצירה של relocatable code), או בזמן ה-execution, כלומר הצימוד יתבצע בזמן הריצה. ביצוע התהליך הזמן הריצה מאפשר הזה של תהליך בזיכרון תוך כדי ביצוע. בעיה: זה דורש כל פעם לתרגם מחדש לכתובת פיזית.

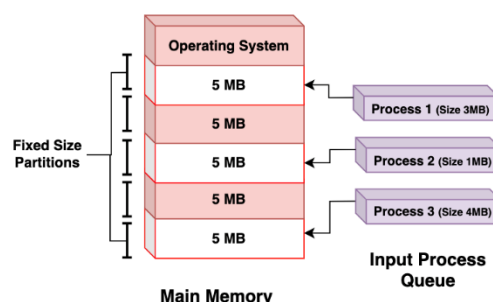
MMU

מטרת ה-MMU שדיברנו עליו בהרצאה הראשונה היא לתרגם בין כתובת לוגית לפיזית. זה נעשה על ידי ה-relocation register (מוסיפים אותו לכתובת הלוגית).

הערה: כל זה נכון לגבי זיכרון לוגי ששמור בצורה רציפה, לגבי זיכרון שאינו שמור בצורה רציפה נדבר בהמשך.

Contiguous Allocation

נחלק את הזיכרון לשני חלקים רציפים. באחד נשים את מערכת ההפעלה ובשני את התהליכים שמריצים המשתמשים. נשתמש ב-relocation register על מנת להגן על התהליכים של המשתמש זה מזה. משתמשים בשני רגיסטרים: אחד base שאומר מה הכתובת הכי נמוכה, ואחד limit ששומר את הכתובת הכי גבוהה שהתהליך אמור לרוץ בו. אם חורגים מטווח הזיכרון שהוקצה, נפעיל trap.



בכל פעם שמגיע process חדש נחפש לו חור בזיכרון שמספיק גדול עבורו.

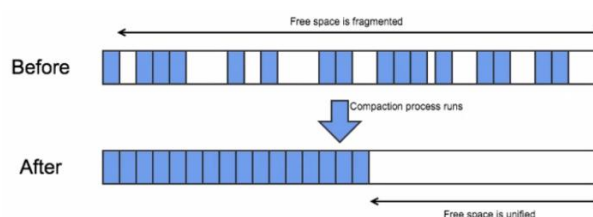
נוכל להקצות או לפי החור הראשון שאנחנו מוצאים שמתאים, או לפי החור הכי קטן שמתאים, או לפי החור הכי גדול שמתאים.

Fragmentation

פרגמנטציה חיצונית – יש זיכרון פנוי ב-RAM שמספיק ע"מ להכניס תהליך חדש, אבל הוא מלא בחורים, כלומר הוא אינו רציף.

פרגמנטציה פנימית – ניתן לתהליך יותר זיכרון ממה שהם צריכים, למקרה שהתהליך יגדל. כעת אם מגיע תהליך חדש זה נראה כאילו אין לנו מספיק זיכרון עבורו אבל בפועל לא כל הזיכרון משומש.

נפתור את הבעיות האלה באמצעות compaction.



Paging

הרעיון: אחסון תהליכים בזיכרון בצורה לא רציפה. ניקח את הזיכרון הפיזי ונחלק אותו לבלוקים בגודל קבוע, להם נקרא frames. גודל הפריים הוא חזקה כלשהי של 2. גם את מרחב הכתובות הלוגי נחלק לבלוקים, שלהם נקרא pages. לכל page נתאים frame. למיפוי הזה נקרא page table.

TLB

טבלת הדפים נשמרת בזיכרון מהיר TLB, בדומה ל-cache. נחפש בו את התרגום לדף כלשהו. אם מצאנו את התרגום מצב כזה ייקרא TLB hit, אחרת TLB miss.

מה תוחלת הזמן לגישה לזיכרון ב-TLB? נקרא לו בקיצור EAT. נסמן את זמן החיפוש הוא ε יחידות זמן. בנוסף נניח שזמן הגישה לזיכרון הוא 1. נסמן את יחס הפגיעות ב- α . זהו אחוז הפעמים שקיבלנו hit בחיפוש.

אזי:

$$EAT = (1 + \varepsilon)\alpha + (2 + \varepsilon)(1 - \alpha) = 2 + \varepsilon - \alpha$$

מכיוון ש- α הוא אחוז הפגיעות, ו- $(1 - \alpha)$ הוא אחוז הפספוסים.

Memory Protection

כדי למנוע מתהליך לגשת לדפים שהוא לא אמור לגשת אליהם, נוכל להשתמש ב-valid bit שיסמן האם דף מסוים הוא valid.

באמצעות טבלת הדפים נוכל גם לחלוק דף בין שני תהליכים. נרצה לעשות זאת ב-shared memory או כאשר תהליכים נוספים צריכים לגשת למידע מתהליך כלשהו.

Two level paging

במקום לשמור טבלה אחת עם מלא דפים, נשמור סוג של "טבלה חיצונית" שתעזור לנו לנווט אל העמוד. אנחנו נבצע paging לטבלה עצמה, ונשתמש בטבלה החיצונית כדי לנווט לחלק בטבלה שאנחנו צריכים, ומשם אל העמוד שאנחנו צריכים. ניתן באופן דומה לבצע Three level paging וכן הלאה.

נוסחה כללית ל-EAT של n-level paging:

$$EAT = \alpha(\varepsilon + m) + (\alpha - 1)(\varepsilon + (n + 1)m)$$

כאשר:

α – hit rate – אחוז הפגיעה ב-TLB.

ε – זמן הגישה ל-TLB.

m – זמן הגישה לזיכרון הפיזי.

הרבה פעמים ניתן להניח $m = 1$.

הרצאה 8 – זיכרון וירטואלי

תזכורת: דפים מאפשרים לנו לשמור את הזיכרון כאשר הוא לא שמור כולו ברציפות, אך בכל דף בנפרד מתקיימת רציפות.

זיכרון וירטואלי מהווה הפרדה חזקה יותר בין ה-logical memory וה-physical memory. הוא בעצם מאפשר לטעון רק חלק מהתוכנית לזיכרון לשם הרצה – כלומר ניתן להריץ תוכנית גדולה יותר מהזיכרון עצמו.

Demand Paging

הרעיון – נביא לזיכרון את תוכן ה-page רק כשבאמת נזדקק לו – כלומר רק כאשר הוא מכיל instruction שצריכים להריץ או שהוא מכיל data שאנחנו צריכים.

Lazy swapper – נביא לזיכרון דף רק אם הוא באמת נדרש.

swapper שעוסק בדפים נקרא pager.

אופן פעולה: אם דף נדרש, נבצע reference אליו.

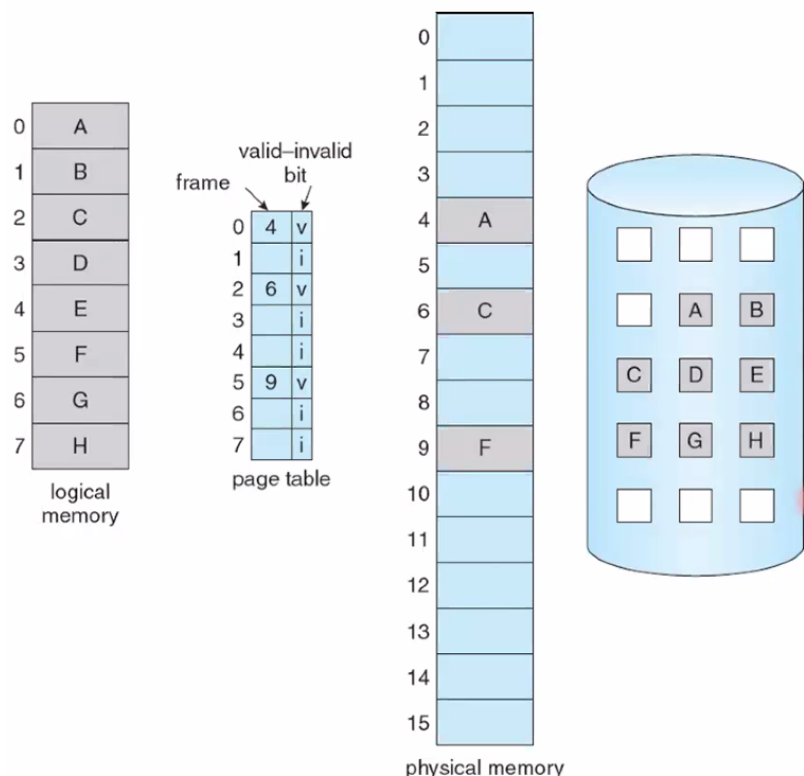
אם ה-reference לא קיים -> abort. אחרת, אם הדף חוקי אך לא נמצא בזיכרון -> נביא אותו לזיכרון.

יתרונות של ה-lazy swapper:

- פחות I/O ביצירת התהליך.
- נדרש פחות זיכרון להרצת התהליך.
- מהירות תגובה טובה יותר ביצירת התהליך (אך לא בהרצה עצמה).
- ניתן לשרת יותר משתמשים על בסיס אותה כמות זיכרון.

נוכל להשתמש ב-page table על מנת לקבוע האם page כלשהו נמצא או לא נמצא כרגע בזיכרון בעזרת ה-valid bit.

כך זה נראה כאשר חלק מהדפים לא נמצאים בזיכרון עצמו, אלא בדיסק:



Logical memory – מתרגם בין מספר פריים לעמוד.

Page table – מראה איפה כל פריים ממוקם בזיכרון, והאם הוא valid/invalid.

Physical memory – הזיכרון – RAM.

Page Fault

איך נטפל ב-page fault?

נניח שרצינו לגשת לעמוד כלשהו וקיבלנו שהוא invalid. אנחנו נוציא trap למערכת ההפעלה, ומערכת ההפעלה תקבע האם מדובר ב-invalid reference (ואז נבצע abort) או שפשוט הדף לא נמצא בזיכרון. אם מדובר פשוט בדף שאינו בזיכרון, מערכת ההפעלה תלך למקום המתאים בדיסק, תכניס אותו לזיכרון, ותעדכן את ה-page table. לבסוף נבצע restart ל-instruction, וכעת נוכל לבצע אותה.

מהו ה-EAT? נסמן ב- p את אחוז ה-page fault שלנו. אזי:

$$EAT = (1 - p) * (memory\ access) + p(page\ fault\ overhead + swap\ page\ out + swap\ page\ in + restart\ overhead)$$

Copy-On-Write

מאפשר לתהליך אב ובן לחלוק מלכתחילה את כל הדפים השמורים בזיכרון. רעיון: בעקבות שינוי באחד הדפים שנוצר כתוצאה מפקודה באחד התהליכים, נוצר עותק חדש.

בנוסף הוא מאפשר יצירה מהירה יותר של תהליך הבן מכיוון שרק דפים שבהם מתבצע שינוי מועתקים.

Page Replacement

מה נעשה במצב שבו אנחנו רוצים להביא דף לזיכרון ואין מקום עבורו? נמצא דף שכרגע בזיכרון אך אינו בשימוש, ונחליף אותו בדף שאנחנו רוצים להכניס. (לדף שנוציא קוראים ה-victim frame).

לאחר ההחלפה נעדכן את ה-page table (גם של הדף שהוצאנו וגם של הדף שהכנסנו), ונבצע restart ל-instruction.

אלגוריתמים ל-page replacement:

FIFO

נוציא את הדף שהגיע לזיכרון הכי מוקדם מבין כל הדפים.

בעיה: האנומליה של בלדי (Bélády's anomaly). יכול להיווצר מצב בו יותר פריימים \Leftarrow יותר page faults.

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

1	1	4	5
2	2	1	3
3	3	2	4

9 page faults

- 4 frames

1	1	5	4
2	2	1	5
3	3	2	
4	4	3	

10 page faults

Optimal Algorithm

דומה לאלגוריתם האסטרונאוטים שראינו באלגו. כאשר נרצה להוציא דף כלשהו, נוציא את הדף שהשימוש הבא בו יהיה המאוחר ביותר. זהו אלגוריתם חמדני שתמיד יביא לכמות ה-page fault הקטנה ביותר האפשרית.

LRU – least recently used

נוציא את הפריים שהשימוש האחרון בו היה לפני הכי הרבה זמן.

בעיה: כל פעם נצטרך למצוא עבור איזה פריים השימוש האחרון בו היה לפני הכי הרבה זמן. נצטרך לשמור לכל דף counter של מתי הייתה הפעם האחרונה שנגעו בו, וכל פעם שניגע בו נעדכן את ה-counter.

לזמן הנוכחי. הצעה נוספת למימוש: נשמור stack, וכל פעם שנגענו בדף נעביר אותו מהמיקום הנוכחי שלו ב-stack להכי למעלה. רעיון: במקום למצוא ממש את ה-LRU, נמצא אפרוקסימציה. אולי לא תמיד נמצא את ה-LRU עצמו, אבל נמצא משהו קרוב אליו. נאתחל מערך שכולו אפסים, כאשר כל ביט במערך מייצג דף. כל פעם שניגע בדף, נשנה את ערך הביט המיוחס אליו ל-1. כאשר כל המערך הוא 1, נאפס את כולו.

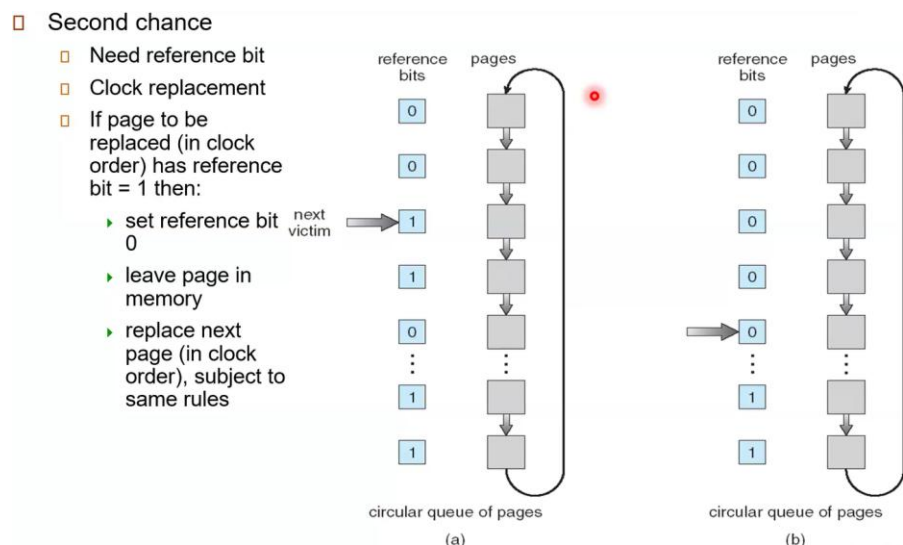
נרצה בעצם להוציא דף כלשהו שערכו במערך הוא 0.

יש צורך במערך בגודל כמות הדפים שנמצאים בזיכרון.

Second-Chance

גם כאן לכל דף נשמור reference bit כמו באלגוריתם הקודם, יחד עם pointer על ביט כלשהו במערך. כל פעם שנוגעים בדף נשים ב-reference bit שלו ערך של 1.

אם קורה page fault, נתחיל לעבור על הביטים בצורה ציקלית. כל עוד הביט עליו מצביע ה-pointer הוא 1, נשנה אותו לאפס ונשאיר אותו בזיכרון (כאן בא לידי ביטוי ה"הזדמנות השנייה"). ברגע שאנחנו מגיעים לביט שערכו 0, נבחר אותו להיות הקורבן, ונחליף אותו.



Counting Algorithms

אלו אלגוריתמים שמקצים לכל דף counter משלו, ואז לפי זה בוחרים את מי להוציא.

LFU – מוציא את הדף עם ה-counter הכי נמוך.

MFU – מוציא את הדף עם ה-counter הכי גבוה. (הגיון – הדף עם ה-counter הכי נמוך בטח הוכנס לא מזמן).

Frame Allocation

כמה דפים צריכים על מנת להריץ תוכנית?

תשובה: למשל ב-IBM 370 דרושים 6 דפים כדי לטפל בפקודת MOVE: גודל הפקודה יכול להיות עד 2 עמודים, יחד עם 2 עמודים לטפל ב-from ושני עמודים לטפל ב-to.

שיטות ל-Allocation

Fixed Allocation

ניתן כמות קבועה של פריימים לתוכנית. ניתן למשל לחלק באופן שווה את כל המשאבים בין כל התוכניות, או לקבוע מראש שמחלקים את המשאבים ביחס לגודל התוכנית.

Priority Allocation

נשתמש בחלוקה פרופורציונאלית, לפי עדיפות במקום לפי גודל. אם בתהליך P_i מתרחש page fault, יש לנו שתי אפשרויות:

1. נחליף את הדף בדף שכבר נמצא בזיכרון של P_i .
2. נחליף את הדף עם דף שנמצא בזיכרון של process עם עדיפות נמוכה יותר.

יכולים להיות לנו שני סוגים של מדיניות החלפה:

1. החלפה גלובלית: נחליף פריים מתוך כל הפריימים בזיכרון.
2. החלפה מקומית: נחליף פריים מתוך כל הפריימים בזיכרון של ה-process הנוכחי.

Thrashing – מצב בו לתהליך כלשהו לא מוקצים מספיק פריימים. זה גורם ל-page-fault גבוה מאוד, וגם גורם לניצולת CPU נמוכה ולכן גורם למערכת ההפעלה לחשוב שהיא צריכה להכניס עוד תהליך. זה יגרום למצב שבו ניצולת ה-CPU תהיה אפילו יותר נמוכה, כי התהליך מבלה את רוב הזמן שלו בהוצאה והכנסה של דפים, ובפועל בקושי מריץ פקודות.

עקרון הלוכליות

הסיבה ש-page demand עובד היא בגלל עקרון הלוכליות. כלומר אם הייתי צריך עמוד כלשהו לא מזמן, יש סיכוי גבוה שאני אדקק לאותו עמוד בקרוב.

גודל הלוכליות: כמות הדפים שהתהליך צריך בקטע הזמן עד לשינוי הלוכליות הבא.

אם לתהליך מסויים מוקצים פחות פריימים מגודל הלוכליות, הוא נמצא ב-thrashing.

איך נמדוד את גודל הלוכליות? נסמן Δ בתור חלון זמן כלשהו אחורה (למשל, 10,000 פעולות אחורה). כעת נשמור ערך WSS_i עבור התהליך P_i שיבדוק בכמה דפים השתמשנו בחלון הזמן Δ .

אם נסמן $D = \sum_i WSS_i$, נוכל למצוא את כמות הדפים הכוללת שנמצאו בשימוש בפרק הזמן האחרון. זוהי בעצם הדרישה הנוכחית לפריימים.

אם $D > m$ (כאשר m היא כמות הפריימים שיש לנו ב-RAM), זה סימן שיש לנו Thrashing.

לבדוק כל הזמן מהו ה-working set דורש המון עבודה, ולכן נרצה להשתמש בקירוב. למשל נשמור מערך של ביטים, שכל פעם שנגע בעמוד נהפוך את הביט המתאים ל-1. כאשר נגיע ל-10,000 נגיעות (כלומר כשנמצא את Δ), נאפס את כל המערך. אם נרצה ברגע נתון לדעת את WSS , פשוט נספור את כמות הביטים הדולקים.

שיפור: נשתמש בשני מערכים, כל אחד מהם לחצי אחר של Δ . נעדכן את כל הערכים במערך העליון, וכשעברנו חצי נחליף בין המערך העליון והתחתון, ונאפס את המערך שעלה למעלה. ואז נעדכן את הערכים של העליון וכשעבר עוד חצי שוב נחליף ונאפס. כעת ה- WSS שלנו נכון בין ה-5,000 ל-10,000 נגיעות האחרונות.

Page-Fault frequency Scheme

שיטה להקצאת פריימים לתוכנית. בכל שלב נחשב את ה-page-fault של תוכנית מסויימת, אם הוא נמוך מדי נוריד פריימים מהתוכנית, ואם הוא גבוה מדי נוסיף פריימים לתוכנית.

Memory mapped files

ניתן להרחיב את הרעיון של page faults ושמירה בדיסק לקבצים שאנחנו רוצים לפתוח. נרצה לפתוח קובץ, אז נחפש אותו בזיכרון. אם הוא לא בזיכרון נביא אותו מהדיסק ואז הקריאות וכתובות אליו יהיו מהירות בהרבה.

הערות:

1. לא חייבים להשתמש ב-lazy swap. ניתן למשל להיעזר בעיקרון הלוכליות ולהביא יחד עם הדף שהיינו צריכים עוד כמה דפים שנמצאים לידו בזיכרון. יכול להיות שנרוויח מזה ויכול להיות גם שנפסיד מזה, תלוי סיטואציה.
2. צריכים לחשוב מראש באיזה גודל עמוד נרצה להשתמש. ייתכנו יתרונות/חסרונות לגדלים שונים.
3. TLB REACH מסמל את גודל ה-TLB כפול גודל הדפים. זוהי כמות המידע שניתן להשיג מה-TLB.

הרצאה 10 – File System Interface

קובץ – רצף של זיכרון שמאוחסן בזיכרון המשני.

מאפיינים של קובץ :

- שם – החלק היחיד שמאוחסן בשפה קריאה לבני אדם.
- סוג – על מנת שנדע איך לקרוא אותו.
- מאפיין ייחודי (ID).
- פוינטר למידע עצמו בזיכרון.
- גודל הקובץ.
- סיווג הקובץ – מי יכול לגשת אליו ואיזה פעולות הוא יכול לבצע.
- זמן, תאריך וזיהוי משתמש – נשמר למטרות אבטחה. (מי יצר? מתי שינה? וכו').

פעולות system calls על קבצים :

Create – מחפש מקום, יוצר כניסה במיקום הקובץ.

Write – מחפש את מיקום הקובץ, מחזיק פוינטר לאיפה שצריך לכתוב (כתיבה עצמה מבוצעת על ידי device controller).

Read – מחפש את מיקום הקובץ, מחזיק פוינטר לאיפה שצריך לקרוא (קריאה עצמה מבוצעת על ידי device controller).

File Seek – מחפש את מיקום הקובץ, משנה את מיקום הקובץ לערך כלשהו (לא דורש באמת I/O).

Delete – מצא את הקובץ, שחרר את כל הזיכרון שלו, מחק את הכניסה לקובץ.

Turncate – כמו delete אבל שומר על תכונות הקובץ קיימות.

על מנת לנהל קובץ פתוח, דרושים כמה דברים כמו פוינטר למקום read/write, סופר לכמות הפעמים שפתחנו את הקובץ, מיקום הקובץ בדיסק והרשאות גישה.

כאשר נפתח קובץ, חלק ממנו נשמר ב-cache לקבצים, על מנת לזרז תהליכים בעתיד אם נרצה לפתוח אותו שוב.

Open file locking

כאשר מספר תהליכים רוצים לגשת לקובץ בבת אחת, נשתמש בנעילה על מנת שלא יהיו בעיות כמו שני תהליכים שכותבים יחד לקובץ, או תהליך שקורא בזמן שאחר כותב. יש כמה סוגי נעילה :

Shared Lock – כאשר shared lock מוצב על קובץ, הוא מותר רק לקריאה על ידי התהליכים הניגשים.

Exclusive Lock – כאשר תהליך צריך לכתוב לקובץ, הוא יכול להציב עליו exclusive lock. בכך, אף תהליך אחר לא יכול להציב עליו נעילה בזמן שהוא כותב. תהליכים אחרים לא יכולים לקרוא או לכתוב לקובץ בזמן הזה.

Access Methods

הדרך שבה הקומפיילר ניגש לקבצים שונה משל בני אדם שרוצים לקרוא את הקובץ. בדרך כלל הקומפיילר ישתמש ב read/write בצורה כזאת :

read next → write next → read next → ... → reset

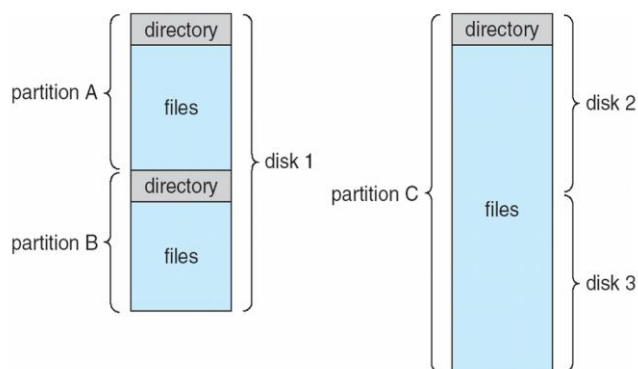
לעומת זאת, דרך שבה תהליכים או בני אדם ניגשים לקובץ היא בדרך כלל :

*read n
write n
position to n
read next
write next*

שתי השורות האחרונות מקדמות את המצביע, בשביל הקריאה/כתיבה הבאים.

Partitions

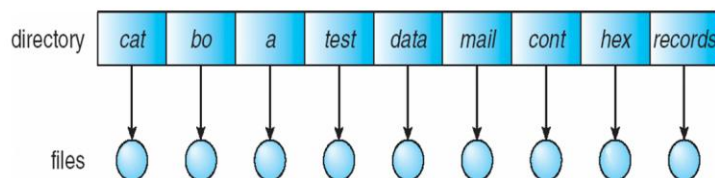
מחלקים את הדיסק לחלקים. ניתן ליצור לכל קובץ partition משלו, או לחלק קובץ אחד לכמה partitions.



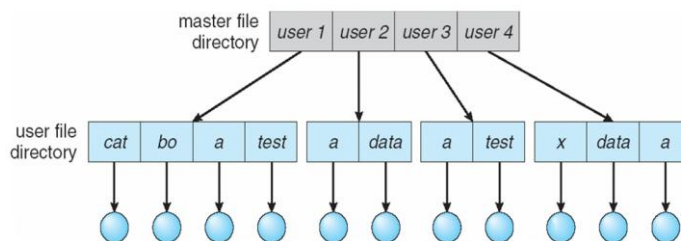
מבנה הקבצים במערכת

ניתן לסדר את מבנה הקבצים בכמה צורות:

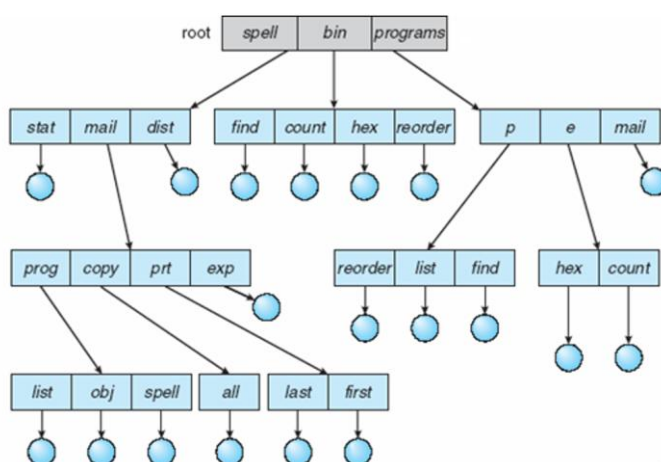
1. Single-level



2. Two-level

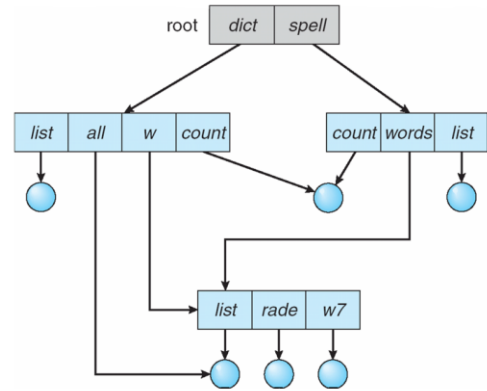


3. Tree-structured – זהו המבנה שאנחנו רגילים אליו.



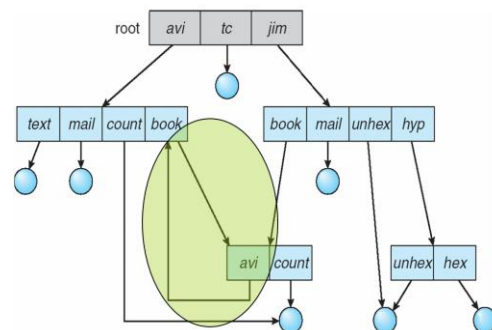
ישנם דרכים נוספות לארגן את מבנה הקבצים, אבל יש איתן גם בעיות.

מבנה גרף א-ציקלי:



בעיה : אם dict מוחק את list, תהיה לנו בעיה של dangling pointer.
פתרון : נחזיק גם פוינטרים אחורנית.

מבנה גרף כללי :



Mounting

התהליך של לעלות מערכת קבצים הנמצאת בדיסק כלשהו, לעץ הקבצים נקרא mounting.

Protection

כאשר נרצה לאפשר לכמה משתמשים לגשת לקובץ במשותף, נצטרך להגן עליו איכשהו. ישנן כמה דרכים לעשות זאת :

לפי user ID – נסתכל על כל משתמש ונחליט על הרשאות והגנות לפי כל משתמש.

לפי group ID – ניתן לקבוצות מסוימות לגשת, ונחליט על הראשות והגנה לפי כל קבוצה.

בעל הקובץ מחליט מי יכול לקרוא, לכתוב ולהריץ את הקובץ.

הרשאת הגישה מאוחסנת ב-3 ביטים, לפי הסדר *RWX*. למשל 100 – יכול לקרוא אך לא לכתוב או להריץ. 101 – יכול לקרוא ולהריץ, אך לא לכתוב.

הרצאה 11 – ניהול זיכרון דיסק

הגדרות:

1. סקטור – היחידה הקטנה ביותר של זיכרון שאנחנו עובדים איתה בדיסק. יכול להיות 512 בתים או 4kb במערכות חדשות יותר.
 2. בלוק – רצף של כמה סקטורים. בפועל כאשר נרצה לכתוב, לקרוא או להקצות זיכרון אנחנו ניגש לבלוק שלם ולא לסקטור.
 3. קלאסטר – רצף של כמה בלוקים. כאשר אנחנו מקצים זיכרון אנחנו יכולים להקצות קלאסטר שלם במקום בלוק כדי להשתמש בעקרון הבלוקיות.
- דיסקים מיוצגים בתור מערך חד ממדי של בלוקים לוגיים.

הקצאת זיכרון בדיסק

Contiguous Allocation

כל קובץ מאכלס כמה בלוקים ברצף. גישה פשוטה, שמאפשרת random access. בעיה: קבצים לא יכולים לגדול. פתרון: Extent based system - כל הקצאה תהיה במקבץ של כמה בלוקים. כאשר יוצרים קובץ משאירים לו extent אחד או יותר למקרה שהוא יגדל.

Linked allocation

כל בלוק מיוצג על ידי רשימה מקושרת של בלוקים. בלוקים יכולים להיות מפוזרים בכל מקום בדיסק.

יתרונות: פשטות – נדרש רק כתובת התחלה וסוף, אין בזבוז זיכרון.

חסרון: אין random access.

FAT (file-allocation table) – דרך לגשת רק פעם אחת לדיסק על מנת להגיע לבלוק m, במקום לגשת m פעמים. נעלה את כל המצביעים לזיכרון, ואז כשנרצה לגשת לבלוק כלשהו נטייל על המצביעים כמו שהיינו עושים בדיסק, אבל אנחנו עושים את זה בזיכרון אז זה לא נחשב בזמן ריצה.

Indexed allocation

נשתמש בבלוק אחד שיצביע לכל שאר הבלוקים של הקובץ.

יתרון: יש random access (אבל צריכים קודם לקרוא בלוק אחד בשביל זה, לא גישה ממש ישירה). בלי external fragmentation.

הערה: ביצירת הקובץ, כל המצביעים יהיו null.

דומה מאוד ל-paging. (מצביעים לבלוקים שממופים לזיכרון ומצביעים לנתונים).

ניתן גם לשלב בין השיטות. עבור קבצים קטנים ניגש אל המידע מיידי. ככל שהקובץ גדול יותר נשתמש ביותר "שכבות" וניצור מבנה דומה לעץ עבור מציאת הבלוקים.

Free space management

ניצור מערך בגודל כמות הבלוקים שיש לנו. בכל מקום במערך נשים 1 אם הבלוק המתאים תפוס, ו-0 אם הוא פנוי. על מנת למצוא בלוק פנוי נצטרך פשוט לחפש 0.

נוכל גם להשתמש ברשימה מקושרת של כל הבלוקים הפנויים.

Counting – בבלוק ההתחלתי נשמור כמה רצף של בלוקים פנויים יש אחריו, ומי הבלוק הבא שאינו ברצף.

Grouping – ניצור קובץ שכולו בלוקים ריקים בשיטת ה-indexed allocation.

Disk scheduling

נושא זה מדבר רק על דיסק מגנטי.

Access time – מורכב מהזמן שלוקח להזיז את הראש על פני הטבעות השונות (Seek time), ועוד הזמן שלוקח לסובב את הדיסק.

יש לנו רשימה של מקומות לגשת אליהם. איך נבחר את הסדר?

הרצאה 12 – Threads

עד כה הנחנו כי כל תהליך מריץ תכנית באמצעות thread אחד בכל פעם (בכל רגע מבוצעת פקודה אחת). ניתן להשתמש ב-multithreading כדי למקבל תהליכים. למשל שרת יכול להשתמש בו על מנת לטפל בכמה בקשות במקביל.

ה-thread מורכב מ:

1. Thread id
2. Program counter
3. Register set
4. Stack

הוא חולק עם שאר ה-threads של התהליך את: Code section, Data section, Other OS resources such as open files
לעומת זאת כל אחד בנפרד שומר רגיסטרים ו-stack משלו.

יתרונות ל-threads:

- Responsiveness – מאפשר לתכנית לרוץ גם כאשר חלק ממנה ב-blockes או שמבצע פעולה ארוכה.
- Resource Sharing – ה-threads חולקים את הזיכרון והמשאבים של התהליך.
- Economy – יותר כלכלי ליצור ולבצע context switch ל-threads (מכיוון שהם חולקים את משאבי התהליך).
- Scalability – אותו תהליך יכול לרוץ על מספר מעבדים במקביל באמצעות threads.

User/Kernel Threads

Kernel threads – נוצרים ומשובצים על ידי ה-kernel.

User Threads – נוצרים על ידי ה-threading library ומשובצים ומנוהלים על ידי הספרייה עצמה:

- כל ה-user threads שייכים לתהליך שיצר אותם.
 - ה-kernel לא מודע אליהם.
- ה-kernel threads "יקרים" משמעותית מה-user threads.

Many-to-one

מספר user threads שממופים ל-kernel thread אחד.
כל ניהול ה-threads מבוצע למעשה בספרייה היושבת ב-user space – יעיל ביותר משיקולי overhead.
חסרונות:

- התהליך כולו ייכנס למצב blocked אם אחד מה-threads קורא ל-system call שהוא blocking.
- אין כאן באמת הרצה במקביל כאשר למעבד יש מספר ליבות (או שיש מספר מעבדים).

One-to-one -חסר-

Many-to-many -חסר-

Two-level Mode

דומה ל-M:M, אבל מאפשר ל-user threads להיות מחוברים ל-kernel threads.