סיכום קורס
מערכות הפעלה
על בסיס השקופיות של זילברשץ וגלבין

תוכן עניינים

פרק 1 – הקדמה 7

מערכת הפעלה - הגדרה 7
מהי מערכת הפעלה? 7
תפקידי מערכת ההפעלה: 7
מבט מופשט על מערכת ההפעלה 7
הגדרות שונות למערכת ההפעלה: (עמ' 3) 7
מצבי מערכת ההפעלה 7
הגדרות שונות 7
מערכות אצווה בסיסיות (SIMPLE BATCH SYSTEMS) (עמ' 4) 8
מערכות מרובות תוכניות - MULTIPROGRAMMING SYSTEMS 8
דרישות ממערכת ההפעלה לצורך MULTIPROGRAMMING 8
SPOOLING - שימוש בדיסק לאכסון מידע של ה CPU (עמ' 1.10) 8
סוגים שונים של מערכות מרובות תוכניות 9
1) שיתוף זמן של מעבד אחד - TIME-SHARING SYSTEMS 9
2) מחשב אישי - DESKTOP SYSTEMS 9
3) עיבוד מקבילי - PARALLEL SYSTEMS 9
4) מערכות מבוזרות - DISTRIBUTED SYSTEMS 10
5) מערכות זמן אמת - REAL TIME SYSTEMS 10
6) מערכות כף יד - HANDHELD SYSTEMS 10

פרק 2 – מבנה מערכות מחשב 11

תרשים מופשט: כיצד מדבר ה-PROCESS עם החומרה 11
פעולת מערכת המחשב - COMPUTER SYSTEM OPERATION 11
פסיקה – INTERRUPT 11
פונקציות נפוצות של פסיקות - COMMON FUNCTIONS OF INTERRUPTS 12
תהליך הטיפול בפסיקה 12
גישה המיידית לזיכרון -DIRECT MEMORY ACCESS (DMA) 12
מבנה אמצעי האחסון - STORAGE STRUCTURE 12
מבנה הדיסק 13
היררכית האכסון - STORAGE HIERARCHY 13
הגנה על חומרה - HARDWARE PROTECTION 13

פרק 3 – מבנה מערכת ההפעלה 15

רכיבי מערכת ההפעלה השונים - SYSTEMS COMPONENTS 15
פניות למערכת ההפעלה ע"י תוכנה - SYSTEM CALLS 15

פרק 4 – תהליכים 16

תפיסת רעיון ה"תהליך" - Process Concept 16
יצירת תהליכים - PROCCESS CREATION 16

מצבי תהליך : (שקף 4.5) 16
תרשים זרימה של חיי התהליך 16
מצבי תהליך בלינוקס 17
בלוק שליטה על "תהליך" - PROCESS CONTROL BLOCK - PCB 17
תזמון תהליכים - PROCESS SCHEDULING 17
סיכום התורים הקיימים במערכת (בהקשר לתהליכים) : 18
SCHEDULERS 18
החלפת תוכן - CONTEXT SWITCH 18
תהליכים שמשתפים פעולה - COOPERATING PROCESSES (4.15) 18
בעיית היצרן-צרכן (4.16) 19
שיטת העברת הודעות - PROCESS COMMUNICATION 19
שיטת העברת הודעות באמצעות BUFFERING 19

פרק 5 – Threads **21**

"חוטים" – THREADS 21
סוגי THREADS 21
חוטים ב SOLARIS 2 21

פרק 6 – תזמון CPU **22**

רעיון כללי - Basic Concepts 22
מעגל ה"התפרצויות" של המעבד - CPU-I/O BURST CYCLE 22
מתזמן המעבד - CPU SCHEDULER 22
המשגר - DISPATCHER 22
קריטריונים לתזמון - Scheduling Criteria (6.5) 23
אלגוריתמים שונים לתזמון - SCHEDULING ALGORITHMS 23
1) ראשון בא, ראשון ישורת - FCFS - FIRST COME, FIRST SERVED (FIFO) 23
2) הקצר ביותר ראשון - SJF - SHORTEST JOB FIRST 24
3) תזמון ע"פ עדיפות - PRIORITY SCHEDULING 25
4) תזמון סרט נע - ROUND-ROBIN SCHEDULING (6.16) 25
5) תזמון מרובה תורים עם רמות - MULTILEVEL QUEUE 26
6) תזמון מרובה תורים עם מעבר ע"פ מאפייני פרץ - MULTILEVEL FEEDBACK QUEUE 26
7) MULTIPLE-PROCESSOR SCHEDULING 27
8) BACKFILLING 27
9) GANG SCHEDULING 27

פרק 7 – סנכרון תהליכים **28**

רקע - Background 28
בעיית קטע הקוד הקריטי - The Critical-Section Problem 28
פתרונות לשני תהליכים 29
אלגוריתם המאפייה - BAKERY ALGORITHM (7.13) 31
חומרת סנכרון - Synchronization Hardware 32

Semaphores 33
דוגמא לשימוש ב-SEMAPHORE - EXAMPLE: CRITICAL SECTION FOR *N* PROCESSES 33
מימוש של SEMAPHORE - SEMAPHORE IMPLEMENTATION 33
קִפָּאוֹן והרעבה - DEADLOCKS AND STARVATION 34
נעילת קטע קריטי בינארית - BINARY SEMAPHORES 35
בעיות סנכרון שונות - Classical Problems Of Synchronization 36
בעיית ה BUFFER החסום - BOUNDED-BUFFER PROBLEM 36
בעיות של קריאה וכתיבה במקביל - READERS-WRITERS PROBLEM (עמ' 94 קוד מפורט) 37
בעיית הפילוסופים הסועדים - DINING-PHILOSOPHERS PROBLEM (עמ' 95,דוג' של קוד בעמ' 309) 37

פרק 8 – קִפָּאוֹן - Deadlocks 39

מודל מערכת (בהקשר הקִפָּאוֹן) System Model 39
תנאים למצב הקִפָּאוֹן - Deadlock Characterization 39
שיטות לטיפול בקִפָּאוֹן - Methods for Handling Deadlocks 39

פרק 9 – ניהול זיכרון 40

רקע - Background 40
כתובת לוגית מול פיסית - Logical Versus Physical Address Space 40
רכיב חומרה לניהול זיכרון (מיפוי לוגי/פיסי) - MEMORY MANAGEMENT UNIT - MMU 40
החלפות זיכרון שמבצעת מערכת ההפעלה - Swapping 40
הקצאת רצף זיכרון - Contiguous Allocation 41
הקצאה שרק לתהליך אחד מותר לרוץ - SINGLE-PARTITION ALLOCATION 41
הקצאה כשכמה תהליכים יכולים לרוץ במקביל - MULTIPLE-PARTITION ALLOCATION 41
הבעיה עם הקצאת זיכרון באופן דינאמי - DYNAMIC STORAGE-ALLOCATION PROBLEM 42
קִטוּעַ – שיברור – זיכרון התהליך מחולק להרבה חלקים - Fragmentation 42
פתרון קִטוּעַ ע"י שימוש בדפים - PAGING 43
מימוש טבלת הדפים 43
טבלת דפים בעלת שתי רמות - TWO-LEVEL PAGE-TABLE SCHEME 44
HASHED PAGE TABLE 44
ADDRESS TRANSLATION SCHEME 44
שיתוף דפים – שיתוף של קוד שכיח בין תהליכים - SHARED PAGES 44
סֶגְמֶנְטַצְיָה- פִּלּוּחַ – הסתרת הזיכרון הפיסי - Segmentation 44
שילוב דפים וסגמנטים ביחד - Segmentation With Paging 45

פרק 10 – זיכרון וירטואלי 46

רקע - Background 46
החלפת דפים ע"פ דרישה -Demand Paging 46
שגיאת דף – פניה לדף שכרגע לא בזיכרון - PAGE FAULT 46
כיצד ומתי נדפדף - Page Replacement 47

הפרדה אחרי שינוי - Copy On Write 47
מיפוי קבצים לזיכרון - Memory Mapped Files 47
אלגוריתמים לדפדוף - Page Replacement Algorithms 47
1) אלג' ראשון בא, ראשון יצא - FIRST IN FIRST OUT (FIFO) ALGORITHM 48
2) אלג' הדף האופטימלי - OPTIMAL ALGORITHM 48
3) אלג' הוצא את הלא פופולארי - LEAST RECENTLY USED (LRU) ALGORITHM 48
4) אלג' קירוב ל LRU - LRU APPROXIMATION ALGORITHM 48
5) אלג' שמשתמשים במונה - COUNTING ALGORITHM 48
הַקְצָאָה של מסגרות - Allocation of Frames 48
בעיית דפדוף מהיר מדי - Thrashing 49
אלג' קבוצת העבודה - WORKING-SET MODEL 49
שיקולים לבחירת גודל הדף 50
שיקולים נוספים בבחירת אלג' דפדוף - OTHER CONSIDERATION 50
DEMAND SEGMENTATION 50

פרק 12 – מימוש מערכת הקבצים 51

מבנה מערכת הקבצים - File-System Structure 51
שיטות הקצאה של מקום בדיסק לקבצים - Allocation Methods 52
1) הקצאה רציפה – CONTIGUOUS ALLOCATION 52
2) הקצאה משורשרת – LINKED ALLOCATION 52
3) INDEXED ALLOCATION 53
TWO-LEVEL INDEX 53
מערכת הקבצים של UNIX - UNIX FILE SYSTEM 53
סיכום מושגים 54
במה נבחר? 54
ניהול השטח הריק בדיסק - Free-Space Management 54
מאגר זמני - Buffer Cache 54
Recovery – אמינות 54
חיבור מערכות קבצים - File System Mounting 55
ניהול שטח החלפת זיכרון בדיסק - Swap-Space Management 55
מערכת קבצים רשתית – UNIX Network File System 55
שרות ספריות – LDAP 55
DHCP 55
SAMBA 55

פרק 14 – מבנה הזיכרון המשני (דיסק) 56

מבנה הדיסק - Disk Structure 56
תזמון הדיסק – שזמן הגישה יהיה מהיר - Disk Scheduling 56
אלג' שונים לקריאת מידע מדיסק 56
במה נבחר? 57

ניהול/מבנה הדיסק - Disk Management 57
פיזור מידע – Data Striping 57
RAID - קבוצת שיטות ארגון דיסק להגדלת אמינות דיסקים 58
ניהול הכפילות - RAID Management 58
SCSY vs. IDE 58
SCSY 59
IDE 59
יתרונות ל – SCSY 59
אינדקס 60

פרק 1 – הקדמה

מישהו שלח לי את הקובץ הזה , ואני שיניתי אותו קצת כדי שיתאים לדברים שוייסמן אמר בשיעור . בתכלס', רוב החומר נשאר זהה. אני לא יודעת למי הוא היה שייך במקור, אבל כל הכבוד על ההשקעה....

מערכת הפעלה - הגדרה

מהי מערכת הפעלה?

תוכנית שמתווכת בין חומרת המחשב לבין המשתמש ומייצרת סביבת עבודה בה יכול המשתמש להריץ תוכניות. מערכת ההפעלה מייצרת נוחות שימוש וניצול מקסימלי של חומרת המחשב.

תפקידי מערכת ההפעלה:

1. להריץ את תוכניות המשתמש.
2. להפוך את בעיות המשתמש לקלות יותר, להפוך את המחשב לנוח לשימוש.
3. שימוש בחומרת המחשב בדרך יעילה.

מבט מופשט על מערכת ההפעלה

מערכת המחשב מורכבת מ-4 רכיבים (תרשים בשקף 1.4):

1. חומרה – נותנת את משאבי המחשוב הבסיסיים. (CPU, זיכרון, מכשירי IO)
2. מערכת הפעלה – שולטת ומתאמת את השימוש בחומרה בין אפליקציות שונות בשביל משתמשים שונים, תוך ניצול יעיל של מערכת המחשב.
3. אפליקציות – מגדירות את הדרכים שבהן יעשה שימוש במשאבי המערכת לפתירת בעיות מחשוב של משתמשים. (קומפיילרים, מערכות בסיסי נתונים, משחקים, תוכניות עסקיות).
4. משתמשים – אנשים, מכונות או מחשבים אחרים(משתמשי קצה).

הגדרות שונות למערכת ההפעלה: (עמ' 3)

1. Resource allocator – ניהול והקצאת משאבים.
2. Control Program – שליטה והרצה של תוכניות המשתמש, והפעלת חומרת I/O.
3. Kernel – גרעין – התוכנית האחת שרצה כל הזמן (כל שאר התוכניות הן אפליקציות).

מצבי מערכת ההפעלה

מערכת ההפעלה יכולה להימצא באחד משני המצבים:

1. Kernel mode או monitor mode – מצב בו מערכת ההפעלה עובדת והמשתמש לא יכול לבצע דבר.
2. User Mode – ה-CPU נמצא במצב שבו process-ים שהתקבלו מהמשתמש רצים , והמערכת לא יכולה לגשת לחומרה.

הגדרות שונות

- Program – תוכניות שנמצאת בדיסק (אפליקציה היא מקרה פרטי של program).
- Job = Process – כאשר תוכנית שנמצאת על הדיסק נכנסת לריצה היא הופכת ל-Process.
- Multiprogramming – מחשב שבו רצות מספר תוכניות על אותה מערכת . התוכניות לא רצות במקביל, אלא זמן ה-CPU מתחלק בין כל התוכניות.
- Multiprocessing – מקרה פרטי של multiprogramming – מערכת עם מספר מעבדים . הדבר מאפשר לתוכניות לרוץ במקביל.

- Multitasking – מערכות של Time-sharing. המחשב יודע לחלק את הזמן בין התהליכים בצורה מהירה מאוד. בנוסף הוא יודע לחלק כל תהליך למשימות. הדבר מאפשר לעבוד עם מספר משתמשים במקביל. זהו מקרה פרטי של multiprogramming.

מערכות אצווה בסיסיות (Simple Batch Systems) (עמ' 4)

היו פעם.

מערכות מרובות תוכניות - Multiprogramming Systems

בכל שלב רק process אחד יכול לרוץ ב-CPU. המטרה שלנו היא שה-CPU תמיד יעבוד על משימה כלשהי.

בזיכרון נשמרים מספר process-ים, ומערכת ההפעלה דואגת בכל שלב לתת את משאב ה- CPU לאחד מהם. ברגע ש-process מסוים מתחיל לרוץ הוא עשוי להזדקק להתקן I/O. לצורך כך הוא יבצע System Call – קריאה של process למערכת ההפעלה. במצב כזה ה- CPU מפסיק לעבוד על ה- process, ומערכת ההפעלה פונה אל רכיבי החומרה הרלוונטיים. בינתיים, מעבירה מערכת ההפעלה את הפיקוח ל-process אחר וה-CPU עובד עליו.

כאשר רכיב ה- I/O מסיים, הוא שולח interrupt למערכת ההפעלה. ה-process שנמצא ב- CPU מופסק ומערכת ההפעלה מטפלת במידע שהוחזר. כעת לא ידוע מי מה-process-ים יקבל את ה-CPU.

תרשים בשקף 1.8. – אינו מאוד ברור – כנראה יורדים בזמן מלמעלה למטה, עמודת *U1* תייצג תוכנית אחת, עמודה *U2* תייצג תוכנית שנייה ו *OS* תייצג את מערכת ההפעלה. אנו רואים איך מתחלק זמן העיבוד על ה *CPU*.

דרישות ממערכת ההפעלה לצורך multiprogramming

1. רוטינות קלט/פלט המסופקות ע"י המערכת – כדי שפניה להתקני קלט/פלט תעבור דרך מעה"פ.
2. ניהול זיכרון כך שיוקצה מקום למספר משימות.
3. CPU scheduling – בחירה מי מבין המשימות תרוץ כעת.
4. הקצאת משאבים.

Spooling - שימוש בדיסק לאכסון מידע של ה CPU (עמ' 1.10)

פירוש ראשי התיבות: Simultaneous Peripheral Operation On Line

שימוש בחלק מהדיסק כ-buffer גדול, לצורך אכסון מידע זמני שישמש את ה CPU בזמן הקרוב.

כלומר כאשר ה-CPU תפוס שָמִים את המידע על הדיסק, ובינתיים ה-CPU עובד על תהליכים אחרים.

דוגמאות לשימוש זה:

1. הפלט של תוכניות שהורצו קודם יהיה מהדיסק למדפסת. כלומר, הפלט ישמר על הדיסק אם המדפסת עסוקה.
2. Spooling משמש גם לעיבוד נתונים באתר מרוחק. ה-CPU שולח את הנתונים דרך מסלול תקשורות למדפסת המרוחקת. העיבוד המרוחק נעשה בקצב שלו, ללא התערבות של ה- CPU. רק כאשר התהליך מסתיים, יש להודיע ל-CPU, כדי שהוא יוכל לעבור לקובץ הבא.

שוב במילים אחרות:

Spooling גורם לחפיפה בין חישוב משימה אחת לשימוש I/O עבור משימה אחרת.

סוגים שונים של מערכות מרובות תוכניות

1) שיתוף זמן של מעבד אחד - Time-Sharing systems

ה-CPU עובד על מספר משימות הנמצאות בזיכרון.

מעה"פ מחלקת את הזמן בין המשתמשים שעובדים על המחשב, ולמשתמש זה נראה כאילו הוא היחיד שעובד.

קיימת תקשורת on-line בין המשתמש למערכת . כאשר מערכת ההפעלה מסיימת הרצת משימה אחת היא מקבלת את הפקודה הבאה מהמשתמש (מקלדת).

חייבת להיות מערכת קבצים עם גישה מיידית (on-line) בשביל שהמשתמש יוכל לגשת למידע(נתונים) וקוד.

המעבר בין process למשתמש נעשה דרך מערכת ההפעלה.

2) מחשב אישי - Desktop systems

מחשב אישי - מחשב המיועד למשתמש אחד.

גם כאן יש טכנולוגיות שפותחו בשביל מערכות הפעלה גדולות יותר – כמו Scheduler למשל.

לעיתים ל- CPU שימוש בודד ואין צורך לנהל את השימוש וההגנה שלו כמו במערכות גדולות.

3) עיבוד מקבילי - Parallel Systems

מערכות שמשתמשות במספר מעבדים.

למעבדים יש משאבים משותפים כגון: זיכרון, bus, שעון ועוד. (sunshine – 4 CPU, זיכרון 1, מעבד 1) התקשרות מתבצעת דרך הזיכרון. מדוע?

1. כדי שמעה"פ תוכל לבחור עם איזה מעבד לעבוד (מאחר שלא צריך לשלוח את המידע לזיכרון מסוים)
2. אין צורך לחכות במקרה שה- CPU המבוקש עסוק.
3. חיסכון בקישורים – אין צורך לחבר בין כל מעבד ומעבד, יש חיבור רק בין כל מעבד לזיכרון.

יתרונות:

4. מגדיל Throughput - האפשרות להריץ דברים חופפים.
5. יתרון כלכלי – במקום להפעיל מספר עמדות מחשב למספר משתמשים מחברים את כולם לאותו מחשב. (אתה צריך להוסיף רק עוד CPU)
6. אמינות(יעילות) ה-CPU. הפסקת מעבד אחד אינה גורמת להפסקת עבודת המערכת.

משפר כאשר צוואר הבקבוק הוא המעבד.

סוגי עיבוד מקבילי: (multiprocessing)

1. עיבוד סימטרי - Symmetric multiprocessing - מספר מעבדים, כך שעל כל מעבד רצה אותה מערכת הפעלה. מערכות הפעלה אלו מקושרות ביניהן ע"י bus – יש צורך להבטיח שהקלט יגיע למעבד הנכון . מספר תהליכים יכולים לרוץ בו"ז בלי לפגוע בביצועים
2. עיבוד אסימטרי - Asymmetric multiprocessing - שיטת Master-Slave. ישנו מעבד ראשי ש מריץ את מעה"פ, ותפקידו להקצות משימות למספר מעבדים משניים. (מעבד זה רוב הזמן לא עושה כלום) לכל מעבד יש את המשימות שמוקצות רק לו . כמובן בד"כ גיבוי של המסטר – אם הוא נופל אז אחד הslaveים הופך למסטר.

במערכות שיש בהן הרבה CPU, נפוץ יותר השימוש בעיבוד אסימטרי כי אפשר לוותר על מעבד אחד בשביל ניהול האחרים. בנוסף, בעיבוד סימטרי יש overhead על גישות לזיכרון ויכולות להיות התנגשויות של מעבדים שונים. ככל שיש לך יותר מעבדים ככה זה יותר בעייתי.

4) מערכות מבוזרות - Distributed systems

מדובר מל מס' מחשבים שונים שמחלקים את העבודה ביניהם.

אין שיתוף של משאבים בין המעבדים השונים , משום שכל מעבד שייך למחשב אחר . הקשר בין המעבדים נעשה ע"י bus מהיר או קווי תקשורת מהירים . החישוב בד"כ יותר איטי – לא בגלל החומרה, אלא בגלל שהמידע צריך לעבור דרך קווי תקשורת מזיכרון לזיכרון (במקום להגיע ישירות). היתרון הוא שהמחשב הרבה יותר עצמאי – ניתן להשתמש במחשבים שונים, עם מעבדים שונים, מהירות שונה וכו'.

סיבות לשימושים במערכות מבוזרות:

1. שיתוף מידע/משאבים – שיתוף קבצים, שיתוף מדפסות.
2. חישוב מהיר יותר – פירוק חישובים לחישובי משנה וחלוקת העומס.
3. אמינות – אם אתר אחד משתבש ניתן להפעיל את העבודה מאתר מרוחק.

ויסמן אמר ש: יכולה להיות מערכת מבוזרת אסימטרית או סימטרית וי כולה להיות מערכת מקבילית אסימטרית או סימטרית, אבל בסיכום אחר כתוב שמערכות מבוזרות תמיד אסימטריות (לא ברור)

5) מערכות זמן אמת - Real Time Systems

לרוב משתמשים במערכות אלו כאשר ישנו זמן מוגבל לביצוע פעולה . כאשר מגיע זרם הנתונים י ש לעבד אותו במהירות . למערכת כזאת יש מערכת תזמון מאורגנת . (בד"כ מערכת embedded). לא מקבלות interrupt.

במערכות כאלו משתמשים בדרך כלל למעקב אחרי ניסויים מדעיים , מערכות הדמיה רפואיות , מערכות בקרה תעשייתיות וכו'.

מגבלות הזמן מוגדרות טוב.

שני סוגים עיקריים:

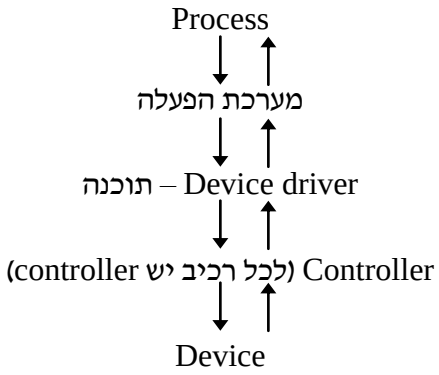
1. נוקשה – hard real-time system – צריך לקבל תשובה תוך פרק זמן קצר , אחרת המידע לא שווה כלום, למשל חישוב מסלול חץ . אין אמצעי אכסון משניים , המידע מאוכסן בזיכרון לתווך קצר, או זיכרון לקריאה בלבד. עובד מול מערכות מסוג "חלוקת זמן – time shearing". לא תומך במערכות הפעלה לשימוש כללי (כלומר הנפוצות).
2. מרוכך – soft real-time system – התשובה חייבת להגיע בזמן (למשל – צפייה בסרט) אבל אם לא, לא נורא . כלי המוגבל לשליטה תעשייתית או רובוטיקה , אפשר להשתמש במערכת זו באפליקציות מולטימדיה או מציאות מדומה – במידה ומערכת ההפעלה מכילה א פיונים מתקדמים.

6) מערכות כף יד - handheld systems

פאלמים, שעונים, פלפונים...

פרק 2 – מבנה מערכות מחשב

תרשים מופשט: כיצד מדבר ה-process עם החומרה



Controller – בקר. כל בקר אחראי להתקן ספציפי (כונן דיסקים, כונן אודיו ועוד), והמעבד מתקשר איתו דרך ה- bus. נתונים שמגיעים מהתקני I/O, עוברים דרך ה- bus לזיכרון. הזיכרון עובד ישירות עם ה- CPU, הוא יושב ישר על לוח האם של המחשב.

ה-CPU והבקרים רצים בו-זמנית.

לכל בקר (זיכרון/חומרה וכו') יש buffer מקומי.

ה-CPU מנתב מידע בין הזיכרון לבין ה-buffer-ים השונים.

מה קורה כאשר מבקשים מידע: ההתקן מעביר את הנתונים לזיכרון של הבקר שלו, וכאשר הוא מסיים, הוא מודיע שהוא סיים - מעלה פסיקה, והנתונים מועברים לזיכרון. (ע"י ה- Direct Memory Access – DMA, לא ע"י ה – CPU).

פעולת מערכת המחשב - Computer system operation

כדי שמחשב יתחיל לרוץ, הוא זקוק לתוכנית התחלתית (bootstrap program) שתרוץ. תוכנית זו מאתחלת את כל הרגיסטרים של ה- CPU והבקרים (מיקום, כתובת ועוד). בנוסף התוכנית מאתרת את גרעין מערכת ההפעלה וטוענת אותו לזיכרון. במצב זה מערכת ההפעלה מתחילה לפעול ומחכה לאירוע שיתרחש מתוכנה או חומרה – פסיקה.

פסיקה – Interrupt

אירוע שדורש את מעה"פ. קיימים סוגים שונים של אירועים העשויים לגרום לפסיקות – לדוגמא, השלמה של פעולת I/O, חלוקה באפס, גישה שגויה לזיכרון ובקשות לשירותי מערכת ההפעלה.

פסיקה מחומרה יכולה לקרות בכל שלב, ע"י שליחת signal ל-CPU, שיכול להיות מסוגים שונים. פסיקה מהתוכנה מתבצעת ע"י הוראה מיוחדת הנקראת System Call.

Interrupt Service Routine – ISR. לכל פסיקה קיימת פונקצית שירות פסיקה שאחראית לטפל בפסיקה.

פסיקה היא מספר. לשם כך למעה"פ יש טבלת כתובות התחלה של ISR. לפסיקה מס' X צריך להפעיל את הפונקציה שנמצאת במקום X.

כאשר ה-CPU מקבל פסיקה, הוא מפסיק את פעולתו ומיד עובר לרוץ במקום קבוע. מוצא את רוטינת השירות של הפסיקה ועובר לשם. ובסיום, ה-CPU מחדש את הפעולה שהופסקה.

פונקציות נפוצות של פסיקות - Common functions of interrupts

כל מחשב מעצב לעצמו מנגנון טיפול בפסיקות, אבל קיימות מספר פונקציות משותפות:

ה-Interrupt חייב להעביר את הבקרה ל- service routine המתאימה. מכיוון שיש לטפל ב- interrupt במהירות, נעשה שימוש בטבלת מצביעים לרוטינות . רוטינת ה-interrupt נקראת בעקיפין דרך הטבלה . לרוב, נשמרת טבלה זו בזיכרון נמוך . טבלה זו נקראת Interrupt Vector. (אוסף של הכתובות של הרוטינות לכל פסיקה. 256 מתודות הנמצאות ב - kernel) הטבלה עצמה מסודרת לפי אינדקסים של ההתקנים המעוררים את אותם interrupt-ים.

כאשר מפעילים interrupt מערכת ההפעלה שומרת על כתובת החזרה וכן את מצב הרגיסטרים במערכת . בסיום פעולת ה-interrupt יש להחזיר את המצב לקודמתו.

כאשר ה- kernel עובד – שאר הפסיקות חסומות כדי לא לגרום ל- lost interrupt. לא ניתן לקבל פסיקה בזמן פעולת פסיקה אחרת. קיים "תור פסיקות" – פסיקות מטופלות אחת אחרי השנייה.

מערכת ההפעלה היא מונחת interrupt (Interrupt driven) – כלומר היא מחכה לאירועים שיתעוררו במערכת.

Trap – פסיקה, נגרמת ע"י בקשת תוכנה או טעות בתוכנה. (בקשת i/o או פנייה לכתובת לא חוקית לדוג').

SCSI -בקר חכם שלא צריך שמעה "פ תנהל לו תור, הוא מנהל אותם בעצמו (ניסיון להקל על ה-CPU). לשאר ההתקנים מעה"פ מנהלת את התור.

Device status table – מערכת ההפעלה שומרת טבלה לניהול פניות להתקנים - כל רשומה בטבלה מציינת device ומידע עבורו . הבקשות עבור כל device נשמרות בטבלה . כאשר מתעורר interrupt I/O מערכת ההפעלה בודקת מי ההתקן שגרם לכך ומעדכנת את הסטאטוס שלו בטבלה . לצורך כך דואגת המערכת גם לנהל תור בקשות עבור כל device.

היתרון בשיטה זו הוא היעילות. בזמן שה-I/O עובד ה-CPU ממשיך לעבוד על דברים אחרים.

תהליך הטיפול בפסיקה

1. החומרה מעבירה את הבקרה למערכת ההפעלה.
2. שמירת מצב ה-CPU ע"י שמירת ערכי הרגיסטרים וה-program counter.
3. קביעת סוג ה- interrupt שהתקבל בעזרת ה-Vectored interrupt system (היתרון בשיטה זו היא שלא מבזבזים זמן לשאול מי צריך).
4. הפעלת סגמנט הקוד המתאים לביצוע ה-interrupt.

גישה המיידית לזיכרון - Direct Memory Access (DMA)

רכיב חומרה שמעביר מידע מהזי כרון להתקן מסוים ולהפך . ה-device controller של רכיב I/O יעביר את המידע ל - buffer ומשם ה-DMA יעביר את המידע לזיכרון (לא דרך ה-CPU).

המטרה היא לאפשר ל-CPU לעבוד בינתיים על process-ים אחרים בזמן שהמידע עובר. ברגע שה-DMA נוגע פיזית בזיכרון, ה-CPU מפסיק לעבוד עם הזיכרון . התזמון בין השניים נעשה בעזרת מערכת ההפעלה (DMA מודיע למערכת ההפעלה שהוא צריך את הזיכרון , היא מעבירה לו את השליטה ובסיום מחזירה את השליטה ל-CPU).

משמש devices בעלי i/o מהיר.

מעביר דפים של מידע – לא ביט ביט – כך פחות פניות, וגם יש את עיקרון המקומיות.

מידע נוסף בשקף 12.6 (עמ' 216).

מבנה אמצעי האחסון - Storage Structure

1. Main Memory – זיכרון ראשי. המקום שבו תוכניות רצות. הזיכרון היחיד איתו ה-CPU יכול לעבוד בצורה ישירה. זהו זיכרון חשמלי (כמו ה-CPU, וזוהי הסיבה שה-CPU יכול לעבוד רק עליו).

זיכרון זה מתחלק ל-2 סוגים:

ROM - Read Only Memory – זמין לקריאה בלבד . מכיל מידע הדרוש למערכת ההפעלה על מנת להעלות את המחשב. לא נמחק כאשר המחשב נכבה. מידע צרוב.

RAM - Read Access Memory – זיכרון חשמלי, נמחק עם כיבוי המחשב.

2. Secondary Storage – זיכרון משני . הרחבות של הזיכרון הראשי המספקות יכולת אחסון מאוד גבוהה. זיכרון מכני. גישה אליו נעשית בעזרת interrupt.
3. Magnetic disks – דיסקטים.

מבנה הדיסק

הדיסק בנוי מאוסף של פלטות שבקצה של הן מוט (מסרק) עם ראש קורא כותב לכל פלטה . הפלטה מורכבת מ-Track-ים. כאשר כל Track בנוי ממספר סקטורים. כל ה-Track-ים באותו רדיוס נקראים צילינדר (לרוב 4 Track-ים בצילינדר). אוסף הצילינדרים מרכיב את הדיסק.

הפלטות בדיסק כל הזמן מסתובבות.

Disk device – הדיסק עצמו.

Device driver – התוכניות שנמצאות במערכת ההפעלה ומאפשרות לעבוד מול ההתקן.

לכל יחידה פיזית בדיסק מתאימה יחידה לוגית שנקראת בלוק . בלוק היא היחידה הקטנה ביותר אותה מערכת ההפעלה כותבת וקוראת. הבלוקים נמדדים בבתים, כאשר גודל הבלוק משתנה ממערכת הפעלה אחת לשנייה (Windows במערכת הקבצים של DOS - בלוק = 4K).

מה משפיע על מהירות המחשב בהקשר החומרה ?

1. Transfer Rate – מעבר של מידע בין הדיסק למחשב.
2. Positioning time – זמן מיקום הזרוע (הראש קורא/כותב) במקום הנכון, בסקטור המתאים. בנוי מ-
   א. Seek time – הזמן שלוקח להזיז את זרוע הדיסק לצילינדר הנכון.
   ב. Rotational latency – הזמן שלוקח לסקטור המבוקש להיות מסובב עד לראש הקורא (סיבוב של הדיסק).

היררכית האכסון - Storage Hierarchy

ההיררכיה מלמטה למעלה (שקף 2.9)

1. טייפ – משמשים לגיבוי בעיקר. שמירת כמויות עצומות של מידע.
2. דיסק אופטי – מהיר יותר.
3. דיסק מגנטי (H.D. הוא לרוב דיסק מגנטי).
4. דיסק אלקטרוני – כמעט חשמלי. גישה מאוד מהירה. (ההבדלים בין הדיסקים הם בעיקר במהירות).
5. RAM – הזיכרון הראשי
6. Cache – זיכרון זמני מהיר המשמש לשמירת המידע שהמעבד משתמש בו תדיר.
7. Registers - אוגרי המעבד עצמם.

Caching – העתקת מידע אל רכיב זיכרון יותר מהיר. (לאו דווקא ל cache עצמו).

הגנה על חומרה - Hardware Protection

מערכת ההפעלה מספקת מספר הגנות על משאבים קריטיים במערכת:

1. Dual-Mode operation – הגנה על מערכת ההפעלה . בשל השימוש במשאבי מערכת משותפים של מערכת ההפעלה יש להבטיח שטעות בתוכנית אחת לא תגרור לטעויות בתוכניות אחרות . לצורך כך למערכת ההפעלה יש שני מצבי עבודה:

א. User Mode – רק תוכנית המשתמש יכולה לרוץ. כאן ה-CPU מוגבל – לא יכול לגלוש ממרחב הכתובות המותר לתוכנית.

ב. Kernel mode – רק מערכת ההפעלה יכולה לרוץ.

ישנו ביט המתווסף לחומרת המחשב - mode bit – המציין מהו המצב הנוכחי , כאשר 1 זה – user mode ו-0 זה Kernel mode.

מעבר ל- Kernel mode מתבצע ע"י פסיקות – עוברים לכתובת מסוימת (בד"כ כתובת נמוכה שתמיד שייכת למעה"פ), ולכן המשתמש לא יכול להעביר את התוכנית שלו ל - Kernel mode. מעה"פ מעבירה חזרה ל - User Mode כאשר היא מסיימת טיפול בפסיקה.

2. I/O protection – כל פעולות ה- I/O מוגנות. גישה ל-I/O עוברת דרך מערכת ההפעלה , כלומר ב - Kernel mode בלבד . יש למנוע מהמשתמש לקבל בקרה על המחשב בזמן שהמערכת ב- Kernel mode.

המשתמש יכול לגשת ל-I/O ע"י System Calls

3. Memory Protection – הגנה על הזיכרון. למעשה, ברגע שניתן לגעת בזיכרון הרי שניתן לעשות הכול. (לדוגמא - ניתן יהיה לשנות את מערך ה- interrupt-ים). לכן לכל process מוקצה שטח בזיכרון וזהו השטח היחיד שה-process יכול לגשת אליו. גישה לכתובת לא חוקית תגרור שגיאה.

כדי להגן על הזיכרון משתמשים בשני רגיסטרים:

Base register – מציין עבור התוכנית מאיזה מקום היא תטען.

Limit register – מכיל את הגודל המקסימאלי של הזיכרון שמוקצה לתוכנית.

התוכנית יכולה לרוץ רק בגבולות הכתובות של הרגיסטר (איור שקף 2.16). קצת בעייתי – מה עושים כשהזיכרון יושב לא רציף? (למשל – הקצאות זיכרון כמו new/malloc) במקרה הזה מה שיושב רצוף זה הכתובות של נתונים. הרחבה בפרקים הבאים.

כיצד מתבצע התהליך? (שקף 2.17):

א. ה-CPU מקבל כתובת לפענוח.

ב. ה-CPU בודק האם הכתובת קטנה מה-base או גדולה מ-(base+limit). אם כן – שגיאה. יישלח Trap – ניסיון של process לגשת למקום בזיכרון שלא מוקצה לו (Segmentation fault).

ג. אחרת – תתאפשר גישה לזיכרון.

4. CPU Protection – יש צורך להגן על ה- CPU כדי למנוע מ- process-ים להשתלט על ה- CPU (לדוגמא בלולאה אין סופית).

ההגנה מתבצעת בעזרת חומרה ייעודית – Timer (שעון). זהו רכיב תוכנה שסופר כמה clocks עברו. מעה"פ נותנת לכל תהליך פלח זמן , כאשר הזמן נגמר , השעון מגיע ל- 0, ויש פסיקה (Time Slice Exceeded). כאשר מתקבלת פסיקה השליטה עוברת למעה"פ ואפשר לתת את הזמן לתהליך אחר.

רק ה - kernel יכול לשנות את השעון. (ב - kernel mode). פעולה זו נקראת CPU Scheduling.

פרק 3 – מבנה מערכת ההפעלה

אנחנו (התלמידים בבר אילן) דילגנו על הפרק הזה. אז בקצרה:

רכיבי מערכת ההפעלה השונים - Systems Components

1. ניהול תהליכים:
   - יצירה ומחיקה של תהליכים.
   - השעיה או הפעלה מחדש של תהליכים.
   - מנגנוני סנכרון ותקשורת בין התהליכים.
2. ניהול הזיכרון הראשי:
   - מעקב אחר אילו חלקים של הזיכרון נמצאים בשימוש וע"י מי.
   - להחליט איזה תהליך יוטען לזיכרון במקום תהליך שהסתיים.
   - הקצאה ופינוי מרחב הזיכרון כשצריך.
3. ניהול הדיסק:
   - ניהול השטח הפנוי.
   - הקצאות.
   - תזמון הדיסק.
4. ניהול קבצים:
   - יצירה ומחיקה של קבצים וספריות.
   - תמיכה בפעולות שונות האפשריות על קבצים וספריות.
   - מיפוי קבצים לאחסון משני.
   - גיבוי.

ממשק משתמש - Command Interpreter System – ממשק בין המשתמש למערכת ההפעלה (Shell). חלק ממערכות ההפעלה מחזיקות תוכניות זו בגרעין , וחלק כשהמחשב עולה או כאשר המשתמש נכנס לראשונה למערכת. תוכנית זו צריכה להיות ידידותית למשתמש . פקודות התוכנית עוסקות ב : יצירת תהליכים וניהולם, גישה למערכת הקבצים, הגנה ותקשורת.

פניות למערכת ההפעלה ע"י תוכנה - System Calls

פנייה ישירה למערכת ההפעלה לביצוע עבודה מסוימת . מספקת ממשק בין תוכניות רצות לב ין מערכת ההפעלה. לרוב בקשות אלו הן בשפת אסמבלר.

ישנן 5 קטגוריות של System Call:

1. Process Control – בקרה על תהליכים . System calls הקשורים לשליטה על תהליכים : אפשרות טעינת תהליך, הרצת תהליך, יצירת, המתנה, שחרור משאבים ועוד. לדוגמא: fork, exec, wait.
2. File manipulation – עבודה עם קבצים וספריות: יצירה, מחיקה, פתיחה, סגירה וחיפוש.
3. Device manipulation – ניהול ההתקנים.
4. תחזוקת המידע – שינוי התאריך, שינוי/ קבלת מידע על מאפייני תהליך/קובץ.
5. תקשורת – מכל סוג עם "העולם החיצוני" (למחשב).

פרק 4 – תהליכים

תפיסת רעיון ה"תהליך" - Process Concept

הגדרה: Job = Process – תוכנית בזמן ריצה.

יש 2 סוגי תהליכים:

Job – תהליך במערכת אצווה (Batch). תהליך שפועל ברקע.

User Program/Task – תוכניות שיש להם קשר עם המשתמש. Time Shared Systems.

Process כולל בתוכו:

1. Program counter - איפה אנחנו בקוד
2. מחסנית – מכילה משתנים מקומיים, כתובות חזרה, פרמטרים וכו'
3. Data section - נתונים שלא במחסנית כגון משתנים גלובליים, סטטיים, קבצים שנפתחו

יצירת תהליכים - Proccess creation

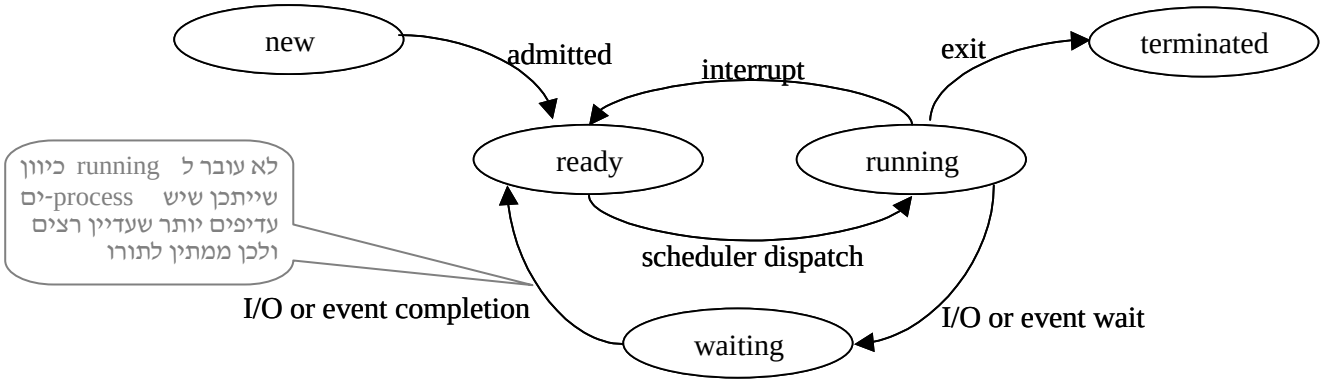
כל תוכנית מתחילה עם תהליך אחד. תהליך אב יכול ליצור תהליך בן (ע"י הפקודה Fork()), וכך נוצר בעצם עץ תהליכים. לעץ אין שורש אחד. יש תהליך Unit ממנו נוצרים כל התהליכים. יש תהליכים שנוצרים כשהמחשב עולה, לפני שתהליכים נוצרים ע"י מעה"פ, למשל Demons.

בנים חולקים את המשאבים של האב . הבנים והאב רצים במקביל , אך האב צריך לחקות שהבן יסיים ע "מ לקבל את הערך החזר. מתי מסתיים תהליך? נגמר הקוד ; בקשה ליציאה של התהליך ; בקשה להפסקת תהליך ממקום אחר.

מצבי תהליך: (שקף 4.5)

1. New – ברגע שה-process נוצר.
2. Running – ה-process מריץ פקודות. **בזמן נתון רק process אחד יכול להימצא במצב זה**.
3. Waiting – ה-process ממתין ל-event (לרוב I/O). ממצב זה לא ניתן לעבור לריצה.
4. Ready – ה-process ממתין לקבל את ה-CPU. מוכן לעבודה.
5. Terminated – ה-process הסתיים.

תרשים זרימה של חיי התהליך



המצב הראשוני של ה-process הוא New, ומיד הוא עובר למצב של ready. ברגע שמגיע תורו של התהליך הוא עובר למצב של running. כעת יכול לקרוא אחד מהמקרים הבאים:

1. התקבלה פסיקה – התהליך יחזור למצב של ready.(כיוון שאין לו cpu כרגע וייתכן שבסיום הפסיקה עדיין לא יקבל cpu כיוון שייתכן שיש process-ים עדיפים ממנו).

2. התהליך זקוק ל-event מסוים או פעולת I/O (או system call)– התהליך יעבור למצב waiting.(עובר ל waiting ולא ל ready כיוון שאם יעבור ל ready אז יש מצב ששוב יועבר לריצה אבל עדיין לא קיבל את מה שביקש ולכן עדיין לא מוכן)
3. התהליך מסתיים – יעבור למצב terminated.

משלב של waiting התהליך יחזור להיות ready ברגע שהמידע שהוא המתין לו חזר.

מצבי תהליך בלינוקס

1. R - בתור, או מוכן לריצה
2. S - ישן או מחכה (Waiting)
3. T - נעצר. תהליך שלא מקבל CPU בגלל המשתמש
4. Z - זומבי. סיים אבל אין אף אחד לא קרא את הערך החזר עדיין
5. D - מחכה להתקן חיצוני

מה ההבדל בין מצב D למצב S?

1. יש תהליכים שעובדים על התקנים שונים לא דרך מעה "פ. במקרה זו משהים את כל הסיגנלים ואי אפשר להרוג אותו עד שהוא יסיים את הפעולה על ההתקן.
2. תהליך שכרגע מחזיק חלקים של הזיכרון בדיסק. לא ניתן להרוג אותו עד שהוא יסיים לשמור את הזיכרון שלו בדיסק.

scheduler dispatcher – קיים רכיב שנקרא scheduler dispatcher שהינו מתזמן מערכת ההפעלה – משגר את ה process-ים לריצה לפי הזמן שהוא קובע והתהליך שכרגע מחכה בתור ל- CPU עובר ל- running.

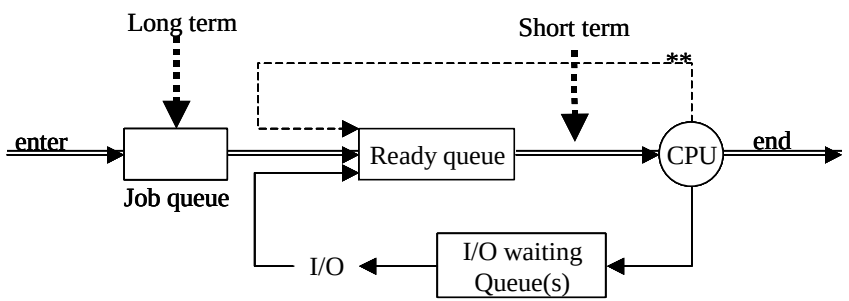
בלוק שליטה על "תהליך" - Process Control Block - PCB

כל תהליך מיוצג במערכת ההפעלה ע"י PCB, המכיל מידע המשויך לתהליך מסוים. בין היתר כולל ה-PCB:

1. מצב התהליך.(ready, new …)
2. Program Counter – כתובת הפעולה הבאה שצריכה להתבצע ע"י התהליך.
3. CPU registers – כאשר מתרחש interrupt יש צורך לשמור את מצב הרגיסטרים במערכת , במטרה לאפשר לתהליך לחזור למצבו הקודם, לפני התרחשות ה-Interrupt.
4. CPU scheduling information – עדיפות התהליך , מצביע לתור התזמונים ופרמטרים נו ספים של תזמון. Unix מבטא עדיפות ע"י הפרמטר nice, שמציין כמה אתה מוכן לוותר על ה- CPU לתהליכים אחרים.
5. מידע על ניהול זיכרון – כולל בין היתר ערך ה-base register ו-limit register, טבלת הדפים ועוד.
6. Accounting information – בכמה CPU הוא משתמש, גבולות זמנים, מספרי תהליכים ועוד.
7. מצב I/O – רשימת התקנים בהם משתמש התהליך, רשימת קבצים פתוחים.

תזמון תהליכים - Process Scheduling

ברגע שתהליך נכנס למערכת, הוא נכנס לתוך ה-job queue. תור זה מכיל את כל התהליכים במערכת(עדיין לא process-ים שמוכנים לרוץ). תהליכים שנמצאים בזיכרון הראשי ומוכנים לריצה או ממתינים לריצה נשמרים ברשימה הנקראת ready queue. רשימה זו נשמרת לרוב במבנה נתונים של רשימה מקושרת . ראש הרשימה



יכיל מצביעים ל-PCB הראשון והאחרון ברשימה (כל PCB יכיל בנוסף מצביע ל-PCB הבא אחריו).

** תהליך חוזר לתור בגלל החלטה של מערכת ההפעלה (interrupt או סיבה אחרת).

סיכום התורים הקיימים במערכת (בהקשר לתהליכים):

Job queue – תור של process-ים שרוצים להיכנס לריצה.

Ready queue – התור של הזיכרון. תהליכים שיקבלו הרשאה לריצה. מכאן התהליכים נכנסים לזיכרון והם זמינים לריצה.

I/O waiting queue – תהליכים שמחכים. ברגע ש מתקבלת פעולת ה- I/O הדרושה התהליך חוזר לתור של ready.

schedulers

כל זמן שהתהליך קיים , הוא נודד בין מספר תורי- תזמון. מערכת ההפעלה חייבת לבחור תהליכים מתוך התורים, ומשתמשת לצורך כך בשני מתזמנים:

1. Long-term scheduler – נקרא גם job scheduler. המתזמן של התור job queue. מחליט אילו תהליכים יעברו למצב ready. מקור השם הוא בכך שהדגימות בתור זה נעשות בדחיפות נמוכה. גישה בטווחים של שניות או אפילו דקות.

   שולט בכמות התהליכים שמערכת ההפעלה מוכנה להריץ במקביל(נעביר מ job queue כמות תהליכים לפי המקום הפנוי ב ready).

   חריג – לא קיים ב - unix – רק במערכות הפעלה ליישומים כבדים.

2. Short term scheduler – נקרא גם CPU scheduler. המתזמן של התור ready queue. מחליט מי מהתהליכים יקבל את ה- CPU. הדגימות מאוד מהירות. מידע יוצא ונכנס מהזיכרון מצורה מהירה מאוד(חשוב ליעילות המערכת).

ההבדל העיקרי בין שני המתזמנים הוא תדירות הגישה עליהם.

באופן כללי, רוב התהליכים מוגדרים כ-I/O bound או CPU bound:

1. I/O bound process – תהליך שכל הזמן זקוק ל-I/O. תהליך מסוג זה נמצא הרבה במצב waiting.
2. CPU bound process – תהליך שצריך כל הזמן פעולות CPU.

דרגת ה multiprogramming – כמה תוכניות שממתינות לרוץ יש בזיכרון, וזה תלוי בגודל ה ready queue

החלפת תוכן - Context switch

החלפת ה- CPU לתהליך אחר דורשת שמירת המצב הקיים של התהליך הישן , וטעינת המצב של התהליך החדש. פעולה זו נקראת context switch. מה קורה בתהליך זה?

לכל תהליך יש מספר טבלאות ששומרות את סביבת העבודה שלו ב- PCB (process control block – שם לוגי ולא ספציפי למערכת הפעלה).

תהליך שמופעל כרגע נמצא ב- user mode, ברגע שהוא צריך להיעצר (פסיקה או system call), הוא מחזיר את הבקרה למערכת ההפעלה (kernel mode) ואז מערכת ההפעלה טוענת את הטבלאות ומחזירה את הבקרה למשתמש.

פעולה זו גורמת ל- overhead – המערכת לא עובדת בזמן תהליך זה.

תהליכים שמשתפים פעולה - Cooperating Processes (4.15)

קיימים 2 סוגי תהליכים:

1. Independent process – תהליכים לא תלויים ולכן הם לא מושפעים מתהליכים אחרים. הכוונה היא שהם לא נמצאים באותו מקום בזיכרון, לא נוגעים באותו I/O ועוד.(לא מעבירים מידע אחד לשני, יכול להיות שתהליך אחד כותב אל הדיסק והשני קורא מהדיסק מאותו מקום, ועדיין נחשבים עצמאיים – תלויים במערכת ההפעלה, אשר נועלת את הקובץ בזמן קריאה/כתיבה עבור התהליך השני
2. Cooperating process – תהליכים שיתופיים. תהליך המושפע מריצה של תהליך אחר (ישירות ללא מתווך). לדוגמא קובץ משותף. האזור המשותף מנוהל ע"י I/O. אפשרות אחרת היא שהמידע המשותף יהיה בזיכרון (למשל buffer), ולמערכת ההפעלה אין שליטה על מה שקורה באזור זיכרון זה – אין הגנה על אזור הזיכרון הזה.

מתי יש צורך בשיתוף תהליכים:

א. שיתוף מידע.
ב. זירוז חישוביות. במקום שתהליך אחד יבצע הכול, כל תהליך מבצע פעולה קטנה.
ג. מודולאריות.
ד. נוחות.

בעיית היצרן-צרכן (4.16)

דוגמא לשיתופיות בין תהליכים: תהליך הצרכן צורך מידע שמיוצר ע"י תהליך היצרן. קיימות 2 אפשרויות:

1. Unbounded buffer – מאגר לא מוגבל. היצרן ממשיך לייצר מידע.
2. Bounded buffer – מאגר מוגבל. כאשר הוא מתמלא היצרן צריך להמתין שהצרכן ירוקן אותו.

שקף 4.17/18 – אלגוריתם ל Bounded buffer

שיטת העברת הודעות - Process Communication

איך תהליכים מדברים ביניהם?

1. Message passing – תהליך א' כותב לתוך אזור במערכת ההפעלה ותהליך ב' קורא מאותו אזור. כלומר התהליכים מעבירים ביניהם מידע, אבל הוא לא עובר ישירות ביניהם אלא דרך מערכת ההפעלה. שימוש באופרטור "|" ב-Unix הוא דוגמא לשיטה זו.
2. ניהול עצמאי - לתהליכים יש אזור משותף בזיכרון, הם החליטו שהם רוצים גישה ישירה בלי התערבות של מערכת ההפעלה, והם אחראים לבד על המידע (שנמצא במקום ששניהם הגדירו).

שיטת העברת הודעות מסוג - Inter-Process Communication - IPC

שיטה נוספת עבור תהליכים לדבר ביניהם בעזרת הודעות. הפעולות הבסיסיות הן send ו-receive.

כאשר שני תהליכים רוצים לדבר נוצר communication link ביניהם והם מחליפים הודעות בעזרת הפקודות הבסיסיות. קיימים 2 סוגי תקשורת:

1. תקשורת ישירה – התהליכים מציינים את שם התהליך אליו הם שולחים הודעה; הקשר נוצר באופן אוטומטי והוא משויך לשני תהליכים בלבד; בין שני תהליכים יכול להיווצר קשר אחד בלבד; הקשר יכול להיות חד-סטרי או דו-סטרי.
2. תקשורת עקיפה – שליחת ההודעות נעשית באמצעות משאב משותף. לדוגמא, קובץ או סמאפור. הקשר נוצר ברגע שהתהליכים משתפים ביניהם את המשאב; מספר התהליכים השותפים אינו מוגבל. בעיה בשיטה זו – מה קורה כאשר תהליך אחד שולח הודעה ושני תהליכים אחרים מנסים לקבל?

פתרונות אפשריים: 1) לאפשר קשר רק בין שני תהליכים; 2) בכל שלב רק תהליך אחד יכול לבצע פעולת receive; 3) המערכת תבחר את המקבל בצורה אקראית, והשולח יקבל הודעה מי קיבל את המידע.

שיטת העברת הודעות באמצעות Buffering

עבודה עם תור של הודעות. ניתן לממש בדרכים הבאות:

1. Zero capacity – מקסימום 0 הודעות. כלומר אין בעצם buffer ששומר הודעות. השולח חייב לחכות שמקבל יקרא את ההודעה לפני שהוא ממשיך הלאה.

2. Bounded capacity – מקסימום n הודעות בתור . כאשר התור מלא השולח צריך לחכות שהתרו יתרוקן.
3. Unbounded capacity – התור אינסופי. השולח אף פעם לא צריך לחכות.

פרק 5 – Threads

"חוטים" – Threads

Thread = lightweight process – היחידה הקטנה ביותר שיכולה לרוץ ולהשתמש ב-CPU. מכילה:

1. Program counter - איפה אנחנו בקוד
2. מחסנית – מכילה משתנים מקומיים, כתובות חזרה, פרמטרים וכו'
3. PCB

בכל תהליך יש מספר thread-ים והם חולקים יחד:

1. Code section.
2. Data section - נתונים שלא במחסנית כגון משתנים גלובליים, סטטיים, קבצים שנפתחו
3. Operating system resources.

ברגע ש process רץ לבד, והוא לא מייצר process-ים אחרים, אז הוא thread אחד.

fork לא מייצרת thread-ים , היא מייצרת process חדש (מייצרת code section ו – data section חדשים), קיימת אפשרות לייצר עוד thread ואז יהיה להם data משותף.

סוגי Threads

1. Kernel threads - החלפת ה - thread נעשית ע"י מערכת ההפעלה . אם התהליך הבא הוא thread מאותו תהליך, לא יהיה צורך ב- Context switch מלא.
2. User threads - צריך רק מחסנית ו- program counter. מערכת ההפעלה אינה מעורבת בתהליך ולכן אין כאן צורך ב- Context switch. בזמן ש- thread אחד נחסם (כלומר הוא מחכה ולא רץ כרגע), thread אחר באותה משימה יכול לרוץ. שיטה זו יעילה יותר ובעלת ביצועים טובים. חסרונות:
   א. אם thread אחד נחסם (צריך I/O למשל) כל השאר נחסמים גם.
   ב. כאשר יש כמה CPU לא ניתן לנצל זאת
   ג. יחס בין מס' החוטים - התהליך עם 100 חוטים יקבל אותו זמן כמו תהליך עם 10 חוטים
3. Anderson threads - "רמת התהליך". התהליך אומר למעה"פ שיש לו X חוטים. וכך מעה"פ תהיה מסוגלת להתחשב בזה . ברגע שיהיה אירוע שדורש החלפת תהליך (למשל I/O), מעה"פ תלך לתהליך ותגיד לו להפעיל חוט אחר (וכך לא יהיה צורך ב- Context Switch מלא). בנוסף, מעה"פ תדע כמה חוטים יש לתהליך הזה ותוכל לתת לו זמן בהתאם.

חוטים ב solaris 2

מעין מודל של many to many. נקרא גם LWP – דרגת ביניים בין user level thread ל- kernel thread supported. כן צריך PBC בשביל להחליף בין החוטים הללו, וה - Switching איטי.

פרק 6 – תזמון CPU

Basic Concepts - רעיון כללי

המטרה של multiprogramming היא להחזיק מספר תהליכים רצים במשך כל הזמן, בכדי לנצל את ה-CPU באופן מקסימלי. במערכות uni-processor (מעבד אחד) רק תהליך אחד יכול לרוץ. יתר התהליכים יצטרכו להמתין עד שה-CPU יתפנה.

הרעיון של multiprogramming ביסודו פשוט. תהליך רץ עד שהוא צריך להמתין. ההמתנה היא בדרך כלל לסיום של בקשות I/O. במצב כזה במערכת מחשב פשוטה, ה-CPU יעבור למצב ש ל idle (סרק). כל זמן ההמתנה הזה מבוזבז – כי אפשר היה לנצל את המעבד בזמן זה לביצוע מטלות אחרות.

ב-multiprogramming אנחנו מנסים להשתמש בזמן הזה בצורה מועילה. מספר תהליכים נשמרים בזיכרון באותו זמן, וכאשר תהליך אחד נמצא במצב המתנה, מערכת ההפעלה לוקחת את ה- CPU מהתהליך ונותנת אותו לתהליך אחר.

תזמון הנו אחד מהפעולות העיקריות של מערכת ההפעלה. כמעט כל משאבי המחשב מתוזמנים לפני שימוש.

מעגל ה"התפרצויות" של המעבד - CPU-I/O Burst Cycle

ריצת תהליך מורכבת ממעגל של ריצה ב-CPU והמתנה ל-I/O, כאשר כל הזמן התהליך עובר בין השניים. כמו כן, התהליך מתחיל ב-CPU ומסתיים בו. תוכנית המבוססת בעיקר על I/O סביר שיהיו לה פרצי CPU (burst) קצרים, ואילו תוכנית המבוססת על CPU תכיל פרצים ארוכים. הצלחת מתזמן ה-CPU תלויה למעשה בריצת התהליך.

מתזמן המעבד - CPU Scheduler

ברגע שה-CPU נכנס למצב של סרק, מערכת ההפעלה חייבת לבחור תהליך מתוך ה- ready queue ולתת לו לרוץ על ה-CPU. החלטות המתזמן עשויות לקרות באחד מארבעת הנסיבות הבאות:

1. כאשר תהליך עובר ממצב ריצה למצב המתנה.
2. כאשר תהליך עובר ממצב ריצה למצב ready.
3. כאשר תהליך עובר ממצב המתנה למצב ready.
4. כאשר תהליך מסתיים.

**מעבר מ ready ל run לא נמצא פה כיוון שמתבצע ע"י ה – dispatcher (למטה)

במקרים 1,4 אין אפשרות בחירה. התהליך מסתיים מיוזמתו, ותהליך חדש חייב להיבחר מתוך התור. מקרים מסוג זה נקראים non-preemptive (מדיניות ללא-הפקעה). המשמעות היא שברגע שה-CPU הוקצה לתהליך, התהליך שומר על ה- CPU עד שהוא משחרר אותו, ולא ייתכן מצב שמערכת ההפעלה תפריע לו באמצע, או תעצור אותו ותיתן עדיפות לתהליך אחר. זוהי מערכת שאדישה לשינויים. תהליך בעל עדיפות גבוהה יותר לא יכול להפריע לתהליך שכבר רץ.

עבור מקרים 2 ו-3 קיימת אפשרות בחירה. מקרים אלו נקראים preemptive (מדיניות עם הפקעה) – קיימת עדיפות לתהליך. מערכת מסוג זה אינה א דישה לשינויים, והיא בודקת כל הזמן את מצבי התהליכים ועשויה להפסיק תהליך אחד עבור תהליך בעל עדיפות גבוהה יותר. תזמון מסוג זה גורם לעלויות – זמן ה-context switch. בעיה נוספת במערכת מסוג זה היא הסנכרון.

המשגר - Dispatcher

module זה מעביר את הבקרה של ה-CPU לתהליך שנבחר ע"י ה-short-term scheduler. פעולה זו כוללת:

1. Switching context.
2. מעבר ל-user mode.
3. קפיצה למקום הנכון בתוכנית המשתמש במטרה להתחיל להריץ אותה.

module זה צריך להיות מהיר ככל האפשר , שכן הוא נקרא בכל פעם שמחליפים תהליך . הזמן שלוקח ל-dispatcher להפסיק את התהליך הרץ ולהתחיל בהרצת תהליך חדש נקרא dispatch latency.

קריטריונים לתזמון - Scheduling Criteria (6.5)

כאשר אנו מחליטים באיזה אלגוריתם להשתמש במצב מסוים , יש לקחת בחשבון את המאפיינים של כל אלגוריתם. הקריטריונים בהם משתמשים להשוואות אלגוריתמים של מתזמני CPU הם:

1. ניצול CPU – נרצה לוודא שה-CPU עסוק כל הזמן. ניצול ה-CPU יכול לנוע מ-0 אחוז ל-100 אחוז. במערכת אמיתית הטווח צריך להיות בין 40 אחוז ל-90 אחוז.
2. תפוקה (Throughput) – יעילות הרצת תהליכים חופפים. אחת הדרכים לבדוק את עבודת האלגוריתם היא מספר התהליכים שסיימו ביחידת זמן אחת.
3. זמן סבב (Turnaround) – מנקודת מבט של תהליך ספציפי , הקריטריון החשוב הוא כמה זמן לוקח להריץ את התהליך. המרווח בין הזמן שהתהליך ביקש לרוץ לבין הזמן שהתהליך יסתיים נקרא זמן סבב. זמן זה הוא סכום משך הזמנים שהתהליך בזבז בהמתנה לזיכרון , המתנה בתור ready, ריצה ב-CPU והפעלת I/O.(מדידה מהרגע שנכנס ל memory queue)
4. זמן המתנה – האלגוריתם אינו משפיע על הזמן בו תהליך רץ או משתמש ב- I/O, אלא רק על הזמן שתהליך מעביר בהמתנה בתור ה- ready. זמן ההמתנה הוא סכום משך הזמנים הכולל שהתהליך העביר בתור ה-ready.(כשזמן ההמתנה אינסופי אז יש הרעבה)
5. זמן תגובה – במערכת אינטראקטיבית, זמן הסבב אינו יכול להיות קריטריון מתאים, משום שלעיתים קרובות, תהליך יכול לספק חלק מהפלט די מהר , ולהמשיך בחישובים אחרים בזמן שחלק מהפלט כבר מוצג למשתמש. לכן אמת מידה נוספת הי א הזמן שלוקח מרגע הבקשה ועד לתגובה הראשונה . זמן התגובה הנו משך הזמן שלוקח להתחיל להגיב, אבל לא כולל את זמן התגובה עצמו.
קובע את מידת ההגינות של המערכת – זמן תגובה דומה לכל התהליכים זה ממתי שנכנס ל ready queue ועד שקיבל cpu ועבר ל running.

המטרה היא להביא את ניצול ה- CPU ואת התפוקה למצב מקסימ אלי, ולמזער כמה שיותר את זמן הסבב , זמן ההמתנה וזמן התגובה. ברוב המקרים מנסים לייעל את הממוצע של כל הקריטריונים , אך קיימים מצבים בהם רצוי לייעל את ערכי המקסימום או המינימום ולא את הממוצע . לדוגמא, כאשר רוצים להבטיח שכל המשתמשים יקבלו שירות טוב, נרצה להקטין את זמן התגובה המקסימאלי.

אלגוריתמים שונים לתזמון - Scheduling Algorithms

1) ראשון בא, ראשון ישורת - FCFS - First Come, First Served (FIFO)

ה-CPU מוקצה ראשון לתהליך הראשון שביקש אותו. הדרך הפשוטה ביותר לממש אלגוריתם זה היא בעזרת תור FIFO. ברגע שתהליך נכנס ל- ready queue, ה-PCB שלו מקושר לזנב התור . כאשר ה-CPU פנוי, הוא מוקצה לתהליך שנמצא בראש התור (התהליכים יתבצעו לפי סדר ההגעה שלהם).

זמן ההמתנה הממוצע במקרה זה הוא לרוב ארוך , והוא תלוי בזמן ההגעה של התהליכים . נסתכל על הדוגמא הבאה:

Process	Burst time
P1	24
P2	3
P3	3

הזמן שלוקח לסיים

אם התהליכים הגיעו על פי הסדר הבא: P1, P2, P3, נקבל את המצב הבא:

זמן ההמתנה של P1 יהיה 0 (משום שהוא התחיל מיד), זמן ההמתנה של P2 יהיה 24, וזמן ההמתנה של P3 יהיה 27. זמן ההמתנה הממוצע שמתקבל הינו 17=3/(27+24+0).

לעומת זאת , אם התהליכים היו מגיעים בסדר הבא : P2, P3, P1, זמן הממוצע שהיה מתקבל הוא (כאשר P2 לא ממתין, זמן ההמתנה של P3 הוא 3 ובזמן ההמתנה של P1 הוא 6). (0+3+6)/3=3

עשוי להיווצר מצב של convoy effect (אפקט השיירה) כאשר כל התהליכים ממת ינים לתהליך אחד ארוך שיסיים עם ה-CPU.

אלגוריתם זה הוא non-preemptive. ברגע שה-CPU מוקצה לתהליך, התהליך מחזיק בו עד לרגע שב ו הוא משחרר אותו.

אלגוריתם זה בעיקר בעייתי במערכות time-sharing, בהן חשוב שכל משתמש יקבל את ה- CPU המשותף במרווח רגיל.

באלגוריתם מסוג זה תיתכן הרעבה, ברגע שתהליך שקיבל את ה-CPU ונכנס ללולאה אינסופית.

* פה לא מתחשבים ב context switch– הכוונה היא שזמן ההעברה הוא אפס.

2) הקצר ביותר ראשון - SJF - Shortest Job First

אלגוריתם זה מקשר לכל תהליך את אורך פרץ ה- CPU (CPU Burst) הבא שלו. ברגע שה-CPU נגיש, הוא מוקצה לתהליך בעל פרץ ה- CPU הבא הקצר ביותר. אם לשני תהליכים יש את אות ו אורך פרץ CPU, יעשה שימוש באלגוריתם FCFS לבחירת התהליך הראשון. נשים לב שהאלגוריתם לא מתייחס לזמן הכולל הנדרש לתהליך על ה-CPU, אלא רק לפרץ ה-CPU הבא.

זהו אלגוריתם אופטימאלי הנותן זמן המתנה מינימאלי.

נסתכל על הדוגמא הבא:

Process	Burst time
P1	6
P2	8
P3	7
P4	3

שימוש באלגוריתם זה יתזמן את התהליכים בסדר הבא : P4, P1, P3, P2. זמני ההמתנה יהיו : P4 לא ימתין, זמן ההמתנה של P1 יהיה 3, זמן ההמתנה של P3 יהיה 9 וזמן ההמתנה של P2 יהיה 16. זמן ההמתנה הממוצע יהיה (3+16+9+0)/4=7 (בעוד שאלגוריתם ה-FCFS נתן זמן המתנה של 10.25).

אלגוריתם זה יכול להיות non-preemptive או preemptive. הדוגמא שהוצגה לעיל הציגה אלגוריתם non-preemptive. אפשרות הבחירה מתעוררת כאשר מגיע תהליך חדש לתור ה- ready בזמן שתהליך אחד רץ . התהליך החדש יכול להיות עם פרץ CPU קצר יותר מזה שנשאר לתהליך שרץ. אלגוריתם preemptive יאפשר לתהליך החדש לתפוס את מקומו של זה שרץ , בעוד שאלגוריתם non-preemptive היה מאפשר לתהליך שרץ לסיים את ריצתו. אלגוריתם preemptive מסוג זה נקרא גם Shortest remaining time first (SRTF).

לדוגמא:

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

באלגוריתם SRTF התהליכים ירוצו בסדר הבא: P1, P2, P4, P1, P3.

P1 יתחיל לרוץ בזמן 0 מכיוון שהוא היחיד בתור . ברגע ש-P2 מגיע ל-P1 נשאר פרץ CPU בגודל 7, ולכן P2 יתחיל לרוץ במקומו. כאשר P3 ו-P4 מגיעים פרץ הזמן שלהם גדול יותר מפרץ הזמן שנותר ל-P2 ולכן אף אחד

מהם לא גורם ל- P2 לעצור. זמן ההמתנה של P2 הוא אפס, זמן ההמתנה של P4 הוא 2 (משום שהוא מתחיל ביחידת הזמן 5, אבל הוא הגיע בזמן 3), זמן ההמתנה של P3 הוא 15 וזמן ההמתנה של P1 הוא 9.

זמן ההמתנה הממוצע המתקבל הוא 6.5. זמן ההמתנה הממוצע באלגוריתם non-preemptive SJF היה 7.75.

אלגוריתם זה מניח שתי הנחות:

1. זמני ההחלפה בין התהליכים הם מהירים, ואינם משפיעים על זמני ההמתנה.
2. זמני הריצה בין ההחלפות הם ארוכים, כלומר לא ייתכן מצב שבו רק מחליפים כל הזמן.

לאור הנחות אלו ניתן לראות כי אלגוריתם זה אינו ישים, **והוא מתאר שיטה תיאורטית בלבד.**

סיכום של SJF - בוחרים את התהליך הקצר ביותר מתוך קבוצה נתונה ויכול להיות שלאחר שהתחלנו לרוץ הגיע תהליך קצר יותר , ולכן תתכן הרעבה של תהליך ארוך (SJF נחשב לאופטימ אלי כיוון שמתיימר לדעת את האורך של כולם אך כאן נעוץ החיסרון)

3) תזמון ע"פ עדיפות - Priority Scheduling

אלגוריתם זה נותן לכל תהליך עדיפות , וה-CPU מוקצה לתהליך בעל העדיפות הגבוהה ביותר . העדיפות נקבעת העדיפות נקבעת פעם אחת , בכניסה אל המערכת . אם לשני תהליכים יש את אותה עדיפות , יעשה שימוש באלגוריתם FCFS לבחירת התהליך הראשון.

אלגוריתם SJF הינו אלגוריתם עדיפות פשוט , בו העדיפות נקבעת על פי פרץ זמן ה- CPU הבא. בעל פרץ ה-CPU הגבוה ביותר הינו בעל העדיפות הנמוכה ביותר ולהפך.

אלגוריתם זה עשוי להיות preemptive או non-preemptive.

אחת הבעיות באלגוריתם מסוג זה היא הרעבה – מצב שבו תהליך שמוכן לרוץ אינו מקבל את ה-CPU (*מערכת לא מסוגלת לזהות הרעבה*). תהליך שנמצא במצב של המתנה נחשב לחסום – Blocked. אלגוריתם עדיפויות עשוי לגרום לתהליכים בעלי עדיפות נמוכה לחכות לנצח . במקרה של מערכות הנטע נות בכבדות , תהליכים בעלי עדיפות גבוהה עשויים למנוע מתהליכים בעלי עדיפות נמוכה גישה ל-CPU.

הפתרון לבעיית ההרעבה הוא Aging – שיטה בה מגדילים את העדיפות של התהליך ככל שזמן ההמתנה שלו גדל.

4) תזמון סרט נע - Round-Robin Scheduling (6.16)

אלגוריתם זה עוצב במיוחד למערכת time-sharing. האלגוריתם דומה לאלגוריתם FCFS, אלא שהפעם ייתכן והתהליכים יוחלפו ביניהם תוך כדי ריצה. מוגדרת יחידת זמן קטנה, הנקראת time quantum. מתייחסים אל תור ה-ready כאל תור מעגלי. מתזמן ה-CPU עובר על התור ומקצה את ה-CPU בכל פעם לתהליך אחר לזמן של time quantum.

בעת מימוש האלגוריתם , שומרים את תור ה- ready בתור FIFO. תהליך חדש מתווסף תמיד לסוף התור . מתזמן ה- CPU בוחר בכל פעם את התהליך הראשון בתור , מגדיר timer interrupt שיופעל אחרי time quantum אחד, ומפעיל את התהליך. כעת יכולים לקרות שני מקרים:

1. התהליך עשוי להזדקק לפרץ CPU הקטן מ-time quantum. במקרה כזה התהליך עצמו ישחרר את ה-CPU, והמתזמן ימשיך לתהליך הבא בתור.
2. אם פרץ ה- CPU לו זקוק התהליך גדול מ- time quantum אחד, ה-timer יגרום ל- interrupt. Context switch יופעל, התהליך יועבר לסוף התור , והמתזמן ייבחר את התהליך הנמצא בראש התור.

אחד היתרונות באלגוריתם זה הוא שנפתרים מהר מתהליכים קצרים . הבעיה היא בעיקר בתהליכים שגדולים במעט מה-time quantum, ואז עצירת התהליך לקרת סיומו מיותרת.

זמן ההמתנה הממוצע הינו בד"כ ארוך.

במצב שבו יש n תהליכים בתור ה- ready וה-time quantum הוא q, אזי כל תהליך יקבל 1/n מזמן ה-CPU בכפולות של q. כלומר זמן תגובה אחיד. כל תהליך יחכה לא יותר מ- (n-1)*q יחידות זמן עד לפעם הבאה בו יקבל את ה-CPU.

הביצועים של האלגוריתם תלויים בצורה משמעותית בגודל ה-time quantum:

- אם ה-time quantum מאוד גדול (מספר אינסופי), נקבל אלגוריתם שמתנהג כמו אלגוריתם FCFS. כלומר, זמן תגובה ארוך.
- אם ה-time quantum הוא מאוד קטן, מתקבלת תפוקה נמוכה, משום שמערכת ההפעלה עסוקה ב-context switching, מצב שבו ה-CPU נמצא במצב סרק.

לכן, נרצה שה-time quantum יהיה גדול תוך כדי התחשבות בזמן ה-context switch. אם זמן ה-context switch הוא בערך 10 אחוז מזמן ה-time quantum, אזי עשר אחוז מהזמן ה-CPU יבזבז על context switch.

אחד מחוקי האצבע לבחירת גודל ה-time quantum דורש בחירת quantum גדול מספיק, כך שלפחות 80 אחוז מפרצי ה-CPU הדרושים יהיו קטנים ממנו (כלומר כ-80 אחוז מהתהליכים יכלו לרוץ מבלי שייאלצו להפסיק את פעולתם באמצע בגלל שה-quantum שלהם הסתיימה).

בשיטה זו לא תיתכן הרעבה.

5) תזמון מרובה תורים עם רמות - Multilevel Queue

קבוצה נוספת של אלגוריתמים לתזמון נוצרה עבור מצבים בהם ניתן לחלק בקלות את התהליכים לקבוצות. לדוגמא, חלוקה מוכרת נעשתה בין foreground process לבין background process. לשתי הקבוצות הנ"ל יש זמני תגובה שונים, דרישות שונות וייתכן ויזדקקו למתזמנים שונים. בנוסף, foreground process עשויים לקבל עדיפות על background process.

אלגוריתם multilevel queue-scheduling מחלק את תור ה-ready למספר תורים נפרדים. התהליכים מוקצים באופן קבוע לאחד התורים, בד"כ על פי תכונות התהליך (לדוגמא גודל זיכרון, עדיפות או סוג התהליך). לכל תור יש אלגוריתם תזמון משלו.

בנוסף קיים מתזמן בין התורים עצמם. ברוב המקרים מתזמן זה ממומש ע"י אלגוריתם preemptive של עדיפויות קבועות. תור בעל עדיפות נמוכה לא יתחיל לרוץ לפני שכל התורים שבעדיפות גבוהה ריקים. אפשרות נוספת היא לחלק "פרוסות" זמן (time slice) בין התורים. כל תור מקבל זמן CPU מוקצב אותו הוא מחלק (עפ"י אלגוריתם התזמון בו עובד) בין כל התהליכים בתור.

היתרונות במקרה זה הוא ניצול ה-CPU.

החיסרון בשיטה זו היא שתיתכן הרעבה, משום שאם התורים בעלי העדיפות הגבוהה יותר לא מתרוקנים, לא נגיע לתורים בעלי העדיפות הנמוכה.

6) תזמון מרובה תורים עם מעבר ע"פ מאפייני פרץ - Multilevel Feedback Queue

באלגוריתם multilevel queue תהליכים מוקצים באופן קבוע לאחד התורים. התהליכים לא עוברים בין התורים. הדבר מתאים למקרים כמו תורים נפרדים ל-foreground process ו-background process, שכן התהליכים לא משנים את סוג הריצה שלהם (ברקע או לא). שיטה זו זוכה ל-overhead נמוך אבל היא אינה גמישה.

לעומת זאת, אלגוריתם Multilevel feedback queue מאפשר לתהליכים לעבור בין התורים. הרעיון הוא להפריד תהליכים עפ"י מאפייני פרץ ה-CPU שלהם. אם תהליך דורש זמן רב של CPU, הוא יעבור לתור בעל עדיפות נמוכה יותר. שיטה זו משאירה תהליכים מונחי I/O ותהליכים אינטראקטיביים בתורים בעלי עדיפות גבוהה. באופן דומה, תהליך שממתין יותר מידי זמן בתור בעל עדיפות נמוכה יעבור לתור בעל עדיפות גבוהה. שיטה זו של aging מונעת הרעבה.

באופן כללי, אלגוריתם זה מוגדר עפ"י הפרמטרים הבאים:

- מספר התורים.
- אלגוריתם התזמון עבור כל תור.
- השיטה על פיה מעלים תהליך לתור בעל עדיפות גבוהה/ מורידים תהליך לתור בעל עדיפות נמוכה.
- השיטה על פיה קובעים לאיזה תור ייכנס תהליך ברגע שהוא צריך שירות.

באלג' זה התהליכים רצים בין התורים, ככל שהתור גבוה יותר אז הוא בעדיפות גבוהה יותר => כלומר מקבל זמן cpu גדול יותר.

תור	זמן cpu
1	80%
2	15%
3	5%

בתור שהתהליכים בו לוקחים הרבה זמן (תור 3) נבחר את תזמון FIFO , אם נבחר אלג ' "הוגן" (round robin למשל) אז כולם יסיימו תוך המון זמן. אנו מעדיפים שכל תהליך שהתחיל יסתיים בזמן סביר.

בתור הראשון נשתמש באלג ' round robin כיוון שאנו רוצים שוויון בתהליכים אינטראקטיביים , אם הזמן שהוקצה לתהליך לא הספיק לו, הוא עובר לתור נמוך יותר.

(דוגמא ב – 6.23)

7) Multiple-Processor Scheduling

במערכת הפעלה סימטרית יש מעבדים זהים שבכל אחד מהם עותק שומה שלמעה "פ, כלומר- לכל אחד מתזמן נפרד. כולם עובדים ע"י אותו אלגוריתם ולכן, התור משותף לכל המעבדים.

במערכות הפעלה אסימטריות מעבד אחד עושה את התזמונים וכך הוא יכול להתחשב ביכולות המעבד שברשותו כדי לבחור איזה תהליך רץ על איזה מעבד.

8) Backfilling

9) Gang Scheduling

תזמון שנועד ל בערכות הפעלה אסימטריות. המתזמן עובד על פלחי זמן . כל המעבדים כולם עובדים על אותה משימה (בהנחה שלמשימה יש כמה תהליכים). ברגע שמשימה לא צריכה את כל המעבדים, אפשר לשים בשאר המעבדים משימה קטנה, וכך המעבדים יהיו תפוסים כמה שיותר.

פרק 7 – סנכרון תהליכים

Background - רקע

תהליכים שיתופיים הם תהליכים המושפעים מריצה של תהליכים אחרים במערכת . תהליכים אלו יכולים ישירות לחלוק מרחב כתובות לוגי, או בעקיפין לחלוק מידע דרך קבצים. גישה בו-זמנית לנתונים עלולה ליצור מצב של חוסר עקביות במידע (תהליך אחד מקבל ת וצאה אחת ואילו תהליך אחר שניגש לאותו נתון מקבל תוצאה אחרת). שמירה על עקביות הנתונים דורשת מנגנון להבטחת סדר הרצת התהליכים.

בפרק 4 הצגנו פתרון של זיכרון משותף לבעיית ה- bound buffer. הפתרון מאפשר שיהיו ב-buffer מקסימום n-1 פריטים. נניח שנרצה לשנות את האלגורי תם הנ"ל בכדי לתקן את הליקוי הנ"ל. אפשרות אחת היא להוסיף מונה, המאותחל ל-0. בכל פעם שנוסיף פריט חדש ל-buffer נגדיל את המונה ב-1, וכן נקטין אותו ב-1 בכל פעם שנוציא פריט מה-buffer.

פירוט מידע משותף ליצרן וצרכן בעמ' 81

קוד **היצרן** יהיה מעתה:

```
repeat
        …
        produce an item in nextp
        …
        while counter = n do no-op;
        buffer[in] := nextp;
        in := (in + 1) mod n;
        counter := counter + 1
until false;
```

משתנה גלובלי

קוד צרכן יהיה מעתה:

```
repeat
        while counter = 0 do no-op;
        nextp := buffer[in];
        out := (out + 1) mod n;
        counter := counter - 1
        …
        consume the item in nextc
until false;
```

למרות שרוטינות היצרן והצרכן נכונות, הן לא יעבדו נכון כאשר ירוצו במקביל. הבעיה נעוצה בשורות הקוד: *counter* := *counter* + 1 ו- *counter* := *counter* - 1. מכיוון שהרוטינות רצות במקביל, לא יודעים מה יתרחש קודם, ולכן ערכי ה-counter עלולים להיות שגויים.

The Critical-Section Problem - בעיית קטע הקוד הקריטי

לא נרצה שתוכנית תפיק בכל פעם פלט אחר כתוצאה מכניסות שונות לקטע הקריטי – נרצה חד ערכיות. נסתכל על מערכת המ כילה n תהליכים. לכל תהליך יש סגמנט קוד הנקרא critical section (או CS) בו התהליך עשוי לשנות משתנים משותפים, לעדכן טבלאות, לכתוב לקובץ ועוד. המאפיין החשוב של המערכת הוא לדאוג, שכאשר תהליך נמצא ב- critical section שלו, אף תהליך אחר לא יוכל להיכנס לקטע הקריטי שלו. בצורה כזאת, ריצת התהליכים בקטעים הקריטיים שלהם היא mutually exclusive (בלעדית) בזמן. בעיית הקטע הקריטי היא ליצור פרוטוקול שהתהליכים יוכלו להשתמש בו יחד. כל תהליך חייב לבקש רשות להיכנס לקטע הקריטי.

פתרון לבעיית הקטע הקריטי חייב לספק את 3 הדרישות הבאות:

1. Mutual Exclusion (מניעה הדדית)– אם תהליך P_i נמצא בקטע הקריטי שלו, אזי כל יתר התהליכים לא יוכלו לרוץ בקטע הקריטי - לא יהיה יותר מתהליך אחד שמטפל במשאב המשותף (בזמן מסוים).
2. Progress - יש התקדמות. אין מצב של קיפאון (dead lock) - תמיד יהיה מישהו שמשתמש במשאב. התהליכים כל הזמן מתקדמים ובפרט התהליך שנכנס לקטע הקריטי . אם אף תהליך לא רץ בקטע הקריטי שלו, וקיימים תהליכים המעוניינים להיכנס לקטע הקריטי שלהם , אז תתבצע בחירה איזה תהליך נכנס - אין לדחות החלטה זו לאין סוף.
3. Bounded Waiting – קיים חסם עליון למספר התהליכים שיכולים להיכנס לקטע הקריטי שלהם , מהרגע שתהליך מסוים ביקש להיכנס לקטע הקריטי שלו . כלומר לא י יתכן שתהליך מסוים יחכה לנצח.יש הגינות בהמתנה – המתנה סופית לכל תהליך. **לא יתכן מצב הרעבה.**

נציג כעת אלגוריתמים לפתרון בעיית הקטע הקריטי , אשר מספקים את כל הדרישות הנ "ל. כאשר נציג אלגוריתם, נגדיר אך ורק את המשתנים בהם נשתמש למטרת סנכרון , ונתאר תהליך Pi אופייני אשר המבנה שלו:

repeat

פה יתבצעו ההגנות ויאותחלו המשתנים של ההגנות

entry section

critical section

נוהל יציאה מהקטע הקריטי

exit section

remainder section

until *false*;

פתרונות לשני תהליכים

נסתכל תחילה על אלגוריתם המתאימים לשני תהליכים בכל שלב.

אלגוריתם 1 (שקף 7.9)

נאפשר לתהליכים לחלוק משתנה משותף אחד בשם turn. כאשר turn=i תהליך Pi רשאי להיכנס לקטע הקריטי. באתחול turn=0

P_i: Repeat

While *turn* $\neq$ *i* ***do*** *no-op;*

Critical section

turn := j;

Remainder section

Until *false*;

P_j: Repeat

While *turn* $\neq$ *j* ***do*** *no-op;*

Critical section

turn :=i;

Remainder section

Until *false*;

הפתרון מבטיח שבכל שלב רק תהליך אחד יהיה בקטע הקריטי שלו . אבל, פתרון זה לא מספק את הדרישה ל-progresses, משום שהוא דורש /מחייב ריצות לסירוגין של התהליכים בקטע הקריטי שלהם . כלומר אם עכשיו התור של תהליך P_i, אזי גם אם התהליך לא זקוק לקטע הקריטי , התהליך השני (j) לא יוכל לקבל את התור. ואז אם P_i לא נכנס אל הקטע הקריטי אז הוא גם לא יסיים ויאפשר ל j להיכנס (לא ישחרר), ולכן ימתין עד ש P_i יזדקק לקטע הקריטי ולכן יכולה להיות הרעבה של תהליך j.

אלגוריתם 2

הבעיה באלגוריתם 1 היא שהוא לא מחזיק מידע על הסטאטוס של כל תהליך . הוא רק זוכר איזה תהליך תורו להיכנס לקטע הקריטי. כדי לתקן בעיה זו, ניתן להחליף את המשתנה *turn* במערך הבא:

var *flag*: **array** [0..1] **Of** *Boolean;*

נאתחל את כל איברי המערך ל- *false*. Flag[i] = true מציין כי התהליך P_i מוכן להיכנס לקטע הקריטי. קוד התהליך יראה כך:

P_i: Repeat

i רוצה להיכנס אבל אם גם j רוצה אז אל תעשה כלום

flag[i] := true;

***while** Flag [j] **do** no-op;*

Critical section

flag [i] := false;

Remainder section

until *false*;

באלגוריתם זה התהליך P_i מציב תחילה את הערך true ב-flag[i], בכדי לציין שהוא מוכן להיכנס לקטע הקריטי שלו. כעת התהליך בודק האם P_j מעוניין בקטע הקריטי או לא . אם כן, אזי P_i ממתין עד ש-P_j ייצא מהקטע הקריטי, ורק אז התהליך P_i יכנס לקטע הקריטי שלו. כאשר Pi מסיים, הוא מעדכן את הערך flag[i] ל-false, ומאפשר לתהליך אחר (אם זה מחכה) להיכנס לקטע הקריטי.

באלגוריתם זה מסופקת הדרישה של mutual-exclusion. אבל עדיין לא מסופקת דרישת ה- progress. במצב שבו שני התהליכים עדכנו את ערכי ה- flag שלהם ל-true, כלומר ציינו שהם רוצים להיכנס לקטע הקריטי, כל אחד מהם ימתין לנצח שהשני ייכנס ויצא מהקטע הקריטי - deadlock.

אלגוריתם זה תלוי בצורה מכרעת בתזמון של התהליכים.

<u>אלגוריתם 3</u>

ע"י שילוב של הרעיונות העיקריים של שני האלגוריתמים הקודמים , נקבל פתרון נכון לבעיית הקטע הקריטי , בו כל שלושת הבקשות מסופקות. התהליכים חולקים שני משתנים:

var *flag*: **array** [0..1] **of** *Boolean;*

Turn: 0..1;

בתחילה, נאתחל את איברי המערך ל-false, ו-turn יקבל 0 או 1. מבנה התהליך Pi יהיה:

Repeat

flag[i] := true;

turn := *j*;

while (*flag*[*j*] and *turn=j*) **do** *no-op;*

critical section

flag[i] := false;

remainder section

until *false*;

כיוון ש turn גלובלי אז יהיה לו ערך חד משמעי בכל פעם

אם תהליך j גם רוצה וגם יכול , אז תהליך I לא ייכנס וימתין ל j שיסיים

פתרון זה מנ יח שתמיד תהיה תעופה מאחת הלולאות אל התהליך השני אשר ייכנס אל CS וכשיסיים נחזור אל הראשון

כדי להיכנס לקטע הקריטי , תהליך P_i מעדכן תחילה את הערך flag[i] ל-true, ולאחר מכן מצהיר שזה תורו של התהליך השני. אם שני התהליכים ינסו להיכנס לקטע הקריטי באותו זמן, turn יקבל את הערך של i ושל j בערך באותו זמן . רק אחת מההשמות הנ "ל תהיה תקפה ; ההשמה האחרת אומנם תתרחש אבל היא מיד תידרס. בסופו של דבר ערכו של turn יחליט מי מהתהליכים רשאי להיכנס לקטע הקריטי.

נראה כעת כי פתרון זה מספק את 3 הדרישות.

1. Mutual exclusion – נשים לב כי תהליך P_i יכול להיכנס לקטע הקריטי שלו רק אם flag[j]=false (כלומר תהליך j לא רוצה להיכנס לקטע הקריטי) או turn=i (כלומר תורו של i להיכנס לקטע הקריטי). כמו כן , אם שני תהליכים היו יכולים לרוץ בו זמנית בקטע הקריטי שלהם , אזי flag[i]=flag[j]=true. על פני שתי הבחנות אלו , ניתן לראות כי התהליכים לא יכלו לעבור את הוראת ה-while שלהם בהצלחה בו זמנית, משום שהערך של turn יהיה או i או j. לכן, אחד התהליכים, נניח P_j, יצליח לעבור את הוראת ה- while בהצלחה, ואילו התהליך השני יהיה חייב לבצע turn=j, ולכן להמתין בתוך לולאת ה-while עד שהתהליך P_j יסיים.

2. Bounded-waiting + Progress – תהליך לא יוכל להיכנס לקטע הקריטי שלו רק אם הוא נתקע בלולאת ה-while. כלומר flag[j]=true וגם turn=j. אם P_j לא מוכן (כלומר turn=j), אזי התהליך יוכל להיכנס לקטע הקריטי , גם אם זה לא תורו , אם התהליך מוכן , אזי P_i יחכה. אבל, ברגע ש-P_j יוצא מהקטע הקריטי שלו , הוא יאפס את flag[j] ל-false, וכך התהליך הראשון יוכל להיכנס לקטע הקריטי. אם כעת P_j יחליט שוב שהוא מעוניין בקטע הקריטי , הרי שהוא יעדכן flag[j]=true, אבל הוא גם יצטרך לעדכן turn=i. כך, מכיוון ש-P_i לא שינה את הערך של turn הוא ייכנס לקטע הקריטי (כלומר יש התקדמות), וכן הוא חיכה רק לכניסה אחת של P_j לקטע הקריטי (כלומר הושגה bounded waiting).

אלגוריתם המאפייה - Bakery Algorithm (7.13)

אלגוריתם זה פותר את בעיית הקטע הקריטי עבור n תהליכים (כמובן שיש cpu יחיד).

כאשר תהליך מבקש להיכנס לקטע הקריטי, הוא מקבל מספר. התהליך בעל המספר הנמוך ביותר יקבל אישור להיכנס לקטע הקריטי . לצערנו, האלגוריתם לא יכול להבטיח ששני תהליכים לא יקבלו את אותו מספר . במקרה כזה התהליך בעל השם הקטן ביותר יקבל אישור . כלומר, אם P_i ו-P_j קיבלו את אותו מספר , ו-i קטן מ-j, אזי P_i יקבל אישור ראשון . האלגי אף פעם לא מ ייצר מספר קטן מהקודם , אך לפעמים מייצר כמה מספרים זהים. **מבני הנתונים המשותפים יהיו:**

var *choosing*: **array** [0..n-1] **of** *Boolean ; //choosing[I]=true התהליך נמצא בתהליך של קבלת מספר*

number: **array** [0..n-1] **of** *integer; //number[I] = thread[I] מה המספר שקיבל*

בתחילה, מבני הנתונים הנ"ל מאותחלים ל-false ול-0 בהתאמה. לצורך הנוחות נגדיר את הסימונים הבאים:

$(a,b) < (c,d)$ if $a < c$ or if $a = c$ and $b < d$

$max(a_0, \ldots, a_{n-1})$ is a number, k such that $k >= a_i$ for $i = 0, \ldots n-1$.

**בעיקרון כל nתהליכים יכולים להיות במצב של choosing[i] = true בו זמנית כאשר ה cpu רץ בינהם

מבנה התהליך Pi יהיה:

בשלב הראשון P_i מקבל מספר – את המקסימום מבין כל המספרים שניתנו עד עכשיו +1. וכיוון שיש הרבה thread-ים שרצים, וייתכן שנרוץ בינהם , וכיוון ששורה זו מפוצלת באסמבלר לכמה ש ורות, **אז** אם תתבצע קפיצה לפני ההשמה ל thread אחר. אז שני thread-ים יקבלו את אותו מספר

```
repeat
   choosing[i] := true;
   number[i] := max(number[0], number[1], ..., number[n-1]) + 1;
   choosing[i] := false;
   for j := 0 to n – 1  //עכשיו נבדוק אם תורו להיכנס לכן נרוץ על כל התהליכים במערכת
        do begin
             while choosing[j] do no-op;  // no-op אם true אז עדיין לא קיבל מספר ולכן
             while number[j] ≠ 0  and (number[j] , j) < (number[i] , i)   do no-op;
           end;
        critical section
   number [i] := 0;//סיים עם הקטע הקריטי
        remainder section
until false;
```

יש חשש שהתהליך יקבל את אותו מספר כמוני (number[i]) האינדקס שלו קטן משלי ואז הוא חייב להיכנס לפני . החשש הוא ש choosing[j] = false ובאותו זמן התהליך שלי מקבל גישה אבל j צריך להיכנס לפניי. לכן נוסיף את הלולאה הנוספת בה נבדוק לפי הסימון לעיל את מספרי ה thread-ים (מערך number) ואם שווה אז נבדוק את מספר ה – thread (אם $j < i$)

אם מספרו אפס אז הוא עדיין לא קיבל מספר או שהוא כבר קיבל שירות ולכן לא מקבל שירות כאן

כדי להוכיח שהאלגוריתם נכון , יש להראות תחילה כי אם תהליך P_i בנמצא בקטע הקריטי שלו , ותהליך P_k (שונה) בחר לעצמו מספר השונה מאפס , אזי $(number[i] , i) < (number[k] , k)$. מסיבה מאוד נחמדה , הוכחה של זה לא מופיעה בספר ??? אבל נניח שהצלחנו להוכיח...

כעת, ניתן להראות שדרישת ה- mutual exclusion מסופקת. נניח שתהליך P_i נמצא בקטע הקריטי שלו , ותהליך P_k מנסה לה יכנס לקטע הקריטי שלו . כאשר P_k יגיע להוראת ה- while השנייה (כאשר j=i) המצב יהיה:

$number[i] \neq 0$

$(number[i], i) < (number[k], k)$

ולכן הוא יישאר בלולאה עד ש-P_i יעזוב את הקטע הקריטי שלו.

חומרת סנכרון - Synchronization Hardware

בעיית הקטע הקריטי יכולה להיפתר בקלות במערכת עם מעבד אחד , אם לא נרשה לקבל interrupt ברגע שמשתנה משותף מעודכן. בעניין זה, נוכל להיות בטוחים שסידרת ההוראות תבוצע בסדר הנכון ללא הפרעות . אף הוראה אחרת לא תרוץ, וכך לא יתבצעו שינויים לא צפויים במשתנה המשותף.

הבעיה היא שפתרון זה לא אפשרי במערכת multiprocessor. מניעת interrupt-ים במערכת multiprocessor צורכת זמן , כאשר מעבירים את ההודעה בכל המעבדים . העברת ההודעה בכל כניסה לקטע קריטי תגרום להפחתת יעילות המערכת . בנוסף, יש לקחת בחשבון את ההשפעה שתהייה על שעות ה מערכת, אם השעון מעודכן בעזרת interrupt-ים.

לכן מכונות רבות מספקות הוראות חומרה מיוחדות המאפשרות לנו או לבדוק ולעדכן תוכן של מילה , או להחליף תוכן שתי מילים בצורה אטומית. נוכל להשתמש בהוראות מיוחדות אלו בכדי לפתור את בעיית הקטע הקריטי בצורה פשוטה מאוד.

נגדיר את ההוראה Test-and-Set באופן הבא:

function *Test-and-Set (**var** target: boolean): boolean:*

begin

Test-and-Set := target;
target := true;

end;

מקבלים ערך מסוים של target, בלי קשר למה שקיבלה, target משתנה ל – 1 והערך המוחזר הוא מה שנשלח בפרמטר target

מימוש בשפת C

```
int TaS(int & target){
int save = target;
target = 1;
return save;}
```

התכונה החשובה של הוראה זו היא שהיא מתבצעת באופן אטומי . כלומר, ההוראה מתבצעת כיחידה אחת שלא ניתנת להפרעה ע"י interrupt-ים. לכן, אם שתי הוראות Test-and-Set יופעלו, הן ירוצו אחת אחרי השנייה בסדר כלשהו.

Test-and-Set זה תהליך של בדיקה שבודק את הערך של הפו', וכיוון שבוליאני אז מימושו בחומרה ולכן אין חשש מאיבוד ה cpu (שיקפוץ ל thread אחר בזמן החישוב).

אם המכונה תומכת בהוראה Test-and-Set, אזי ניתן לממש mutual exclusion ע"י הצהרת משתנה בוליאני בשם lock אשר יאותחל ל-false. קוד התהליך יראה בצורה הבאה: אתחול: lock:=false

repeat

while *Test-and-Set(lock)* ***do*** *no-op;*

critical section

lock := false;

remainder section

until *false*;

ה thread הראשון שנכנס הופך את lock ל-true ונכנס אל CS, ולכן אף thread לא יוכל להיכנס עד שהוא לא יסיים עם CS ויבצע lock:=false וברגע שתתבצע שורה זו אז ה- thread הראשון שיגיע אל הלולאה כאשר lock==false ייכנס אל CS.

יתרון: אין את החלק של do no-op

חסרון: אין סדר, לא ידוע מי יגיע לפני מי אל CS

Semaphores

Semaphore s הוא משתנה מסוג integer, אשר פרט לשלב בו הוא מאותחל , ניתן לגשת אליו רק דרך שתי פעולות אטומיות: wait ו-signal. ההגדרה הקלאסית של פקודות אלו היא:

wait(S): ***while*** *S <= 0* ***do*** *no-op; // המתנה לכניסה אל הקטע הקריטי*

S := *S* – 1;

signal(*S*): S := *S* + 1; *// יציאה מהקטע הקריטי ואיתות על כך*

עדכון ערכו של ה- semaphore בהוראות ה-wait ו-signal חייב להתבצע באופן שאינו ניתן לחלוקה . כלומר, כאשר תהליך אחד מעדכן את ערך ה- semaphore, אף תהליך אחר לא יכול לעדכן בו- זמנית את אותו semaphore. בנוסף, במקרה של *wait*(*S*), הבדיקה האם S<=0 ולאחר מכן העדכון *S* – 1 =: S, חייבים גם כן לרוץ ללא הפרעות . נראה כיצד ניתן לממש פעולות אלו , תחילה נראה כיצד ניתן להשתמש ב-semaphores.

**ב windows יש הבדל בין סמפור ל-mutex

דוגמא לשימוש ב-semaphore - Example: Critical Section for *n* Processes

ניתן להשתמש ב- semaphores כדי להתמודד עם בעיית הקטע הקריטי עבור n תהליכים. n התהליכים חולקים semaphore בשם mutex המאותחל ל-1. כל תהליך מאורגן בצורה הבאה:

repeat //**mutex** פה הסמפור הוא מסוג

wait(mutex);

critical section

signal(mutex);

remainder section

until *false*;

מימוש של semaphore - Semaphore Implementation

אחד החסרונות בלולאות ה-while שהגדרנו עד כה הוא השימוש ב-busy waiting. כאשר תהליך רוצה להיכנס לקטע הקריטי והוא צריך להמתין הוא מבצע לולאה בתור ה- entry code. ללולאה זו יוצרת בעיה במערכות multiprogramming, בהן מעבד אחד מתחלק בין מספר תהליכים . Busy waiting מבזבז משאבים של ה-CPU, בזמן שתהליכים אחרים היו יכולים להשתמש ב-CPU בצורה יעילה יותר.

כדי להתגבר על הצורך של busy waiting נעדכן את הגדרת ההוראות wait ו-signal. כאשר תהליך מריץ את הוראת ה- wait ומגלה שערך ה- semaphore שלילי, הוא יהיה חייב לחכות . אבל, במקום להשתמש ב-busy waiting, התהליך **יחסום (block)** את עצמו. פעולת החסימה ממקמת את התהליך בתור ה- waiting המקושר ל-semaphore, וסטאטוס התהליך הופך ל-waiting.

תהליך שחסום יותחל מחדש כאשר תהליך אחר יריץ הוראת signal. התהליך יותחל מחדש בעזרת הוראת wakeup, אשר משנה את סטאטוס התהליך מ-waiting ל-ready, וכך ממוקם התהליך שוב בתור ה-ready.

כדי לממש semaphore תחת הגדרות אלו, נגדיר את ה-semaphore בתור רשומה:

type *semaphore* = **record**

value: *integer*;

L: **list of** *process*;

end;

לכל semaphore יש ערך מספרי ורשימת תהליכים . כאשר תהליך חייב לחכות ל-semaphore הוא מתווסף לרשימת התהליכים. הוראת signal מסירה תהליך אחד מרשימת התהליכים הממתינים, ומעירה אותו.

הוראות ה-semaphore יוגדרו כעת כך : אתחול : S.value:=1

```
wait(S):    S.value := S.value – 1;
            if S.value < 0 then
                begin
                    add this process to S.L;
                    block;
                end;

signal(S):  S.value := S.value + 1;
            if S.value <= 0 then
                begin
                    remove a process P from S.L;
                    wakeup(P); // העברת התהליך ל-ready
                end;
```

אם ערך הסמפור שלילי אז נוסיף אותו אל הרשימה ונחסום אותו (block)

אם עדיין ≥ 0 למרות שהעלנו ב 1, זה אומר שעדיין יש ממתינים ולכן נסיר process מרשימת ההמתנה

<u>בעיה</u> – אם באתחול S.value = 10 אז ייכנסו 10 thread-ים אל cs .

הוראת block משהה את התהליך . הוראת wakeup מחדשת את ריצת התהליך שנחסם . שתי הוראות אלו מסופקות ע"י מערכת ההפעלה כ-system call בסיסיות.

** על הפקודות wait ו- signal ב windows ראה דוגמא בעמוד 300...

<u>ביצוע סנכרון ע"י סמפור</u> (6.20 עמ' 89 פירוט מלא)

הסמפור יכול לשמש גם לתזמון נניח שיש שני thread-ים, נרצה לבצע את A ב Pi לפני B ב Pj

נגדיר סמפור בשם flag שיאותחל ל 0 <u>והקוד ייראה כך :</u>

Pi	Pj
.	.
.	.
A	wait(flag)
signal(flag)	B

אם Pj מתבצע, נוכל לבצע את B רק אם flag יהיה דלוק, וזה אומר ש A כבר התבצע. אם A לא התבצע אנחנו נחכה (wait).

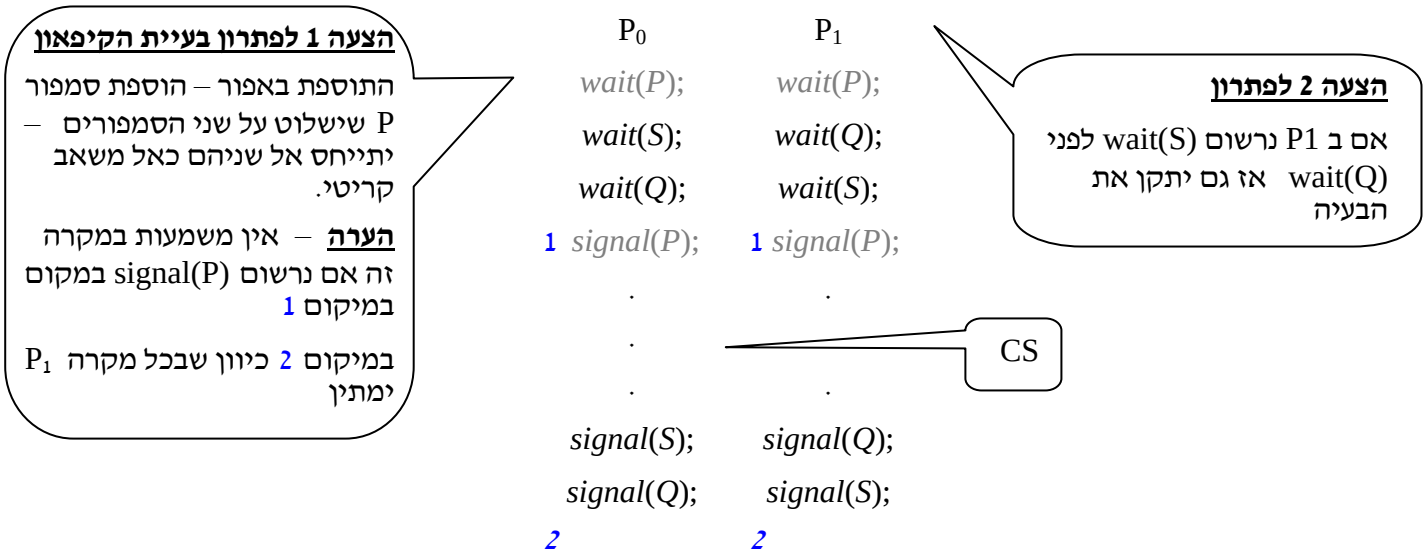
אם Pi מתבצע ראשון, רק אחרי ש Pi יתבצע נעלה את ערך הדגל/סמפור flag ל - 1

<u>קִפָּאוֹן והרעבה - Deadlocks and Starvation</u>

Deadlock – מצב שבו שני תהליכים או יותר מחכים לאירוע שצריך להתרחש , אך הוא תלוי בתהליכים שמחכים. כלומר, אף אחד לא נכנס לקטע הקריטי.

מימוש של semaphore עם תור ממתינים עשוי לגרום למצב שבו שני תהליכים או יותר ממתינים לנצח לאירוע שעשוי להתרחש רק ע"י אחד התהליכים שנמצא גם כן בתור . במצב כזה התהליכים נמצאים בקיפאון (deadlock).

כדי להדגים זאת נסתכל על מערכת המכילה 2 תהליכים, P_0 ו-P_1, שניהם ניגשים לשני semaphore-ים, S ו-Q, המכילים את הערך 1 (Q=S=1).



נניח ש-P_0 מריץ את *wait*(*S*), ו-P_1 את *wait*(*Q*). P_0 חייב לחכות עד ש-P_1 יריץ *signal*(*Q*), בעוד ש-P_1 חייב לחכות עד ש-P_0 יריץ *signal*(*S*). מכיוון ששתי פעולות ה-signal לא יכולות לרוץ, התהליכים במצב קיפאון.

נאמר שקבוצה של תהליכים נמצאת בקיפאון כאשר כל תהליך בקבוצה מחכה לאירוע שצריך להתרחש ע"י תהליך אחד בקבוצה . בעיה נוספת הקשורה לקיפאון היא הרעבה – מצב שבו תהליך מחכה לנצח בתוך semaphore.

יש שני סוגים של סמפורים: 1. סמפור מונה/סופר

2. סמפור בינארי

נעילת קטע קריטי בינארית - Binary Semaphores

תבנית ה-semaphore שתוארה עד עכשיו מוכרת בשם counting semaphore, משום שהערך המספרי שלה יכול להגיע לטווח לתחום לא מוגבל . ב-Binary semaphore טווח הערכים הוא 0 או 1. Binary semaphore יכול לפשט את המימוש של counting semaphore, תלוי בארכיטקטורת ה חומרה. נראה כעת כיצד ניתן לממש counting semaphore בעזרת binary semaphore.

נגדיר את S בתור ה-counting semaphore. כדי לממש זאת במונחים של binary semaphore נגדיר את מבנה הנתונים הבא:

```
var S1: binary-semaphore;
    S2: binary-semaphore;
    C: integer;
```

בתחילה S1=1, S2=0 וערך המסר C יקבל את הערך הראשוני של ה-S counting semaphore.

הוראת ה-wait תמומש בצורה הבאה:

```
wait(S1);
  6. C := C – 1;
If C < 0 then
begin
   signal(S1);
   wait(S2);
end
signal(S1):
```

הוראת ה-signal תמומש בצורה הבא:

wait(S1);

7. C := C + 1;

If C <= 0

then *signal*(*S*2);
else *signal*(*S*1);

בעיות סנכרון שונות - Classical Problems Of Synchronization

נציג כעת מספר בעיות שונות של סנכרון, אשר הפתרון שלהם הוא שימוש ב-semaphores. (6.25 עמ' 92)

בעיית ה buffer החסום - Bounded-Buffer Problem

בעיית היצרן צרכן.

נניח שקיימים n buffers, כל אחד מסוגל להחזיק פריט אחד. נגדיר 3 semaphore-ים:

1. mutex – מספק mutual exclusion לגישה ל-buffer, ומאותחל ל-1.
2. empty – לוקח בחשבון את ה-buffer-ים הריקים. מאותחל לערך n. (empty=0 – יש 0 תאים ריקים)
3. full – לוקח בחשבון את ה-buffer-ים המלאים. מאותחל ל-0. (full=n – כל התאים מלאים)

קוד היצרן יהיה מעתה:

מטיפוס nextp,nextc // item
אתחול:
full:=0, empty:=n, mutex:=1

```
repeat
        …
    produce an item in nextp
        …
    wait(empty); // נחכה שיהיה לפחות אחד פנוי
    wait(mutex); //נבקש אישור להיכנס אל הקטע הקריטי
        …
    add nextp to buffer
        …
    signal(mutex);
    signal(full);
until false;
```

נוריד את מספר הריקים ב 1

נעלה את מספר המלאים

קוד צרכן יהיה מעתה:

```
repeat
    wait(full); // נחכה שיהיה לפחות אחד מלא
    wait(mutex);//נחכה לאישור
        …
    remove an item from buffer to nextc
        …
    signal(mutex);
    signal(empty);
        …
    consume the item in nextc
        …
until false;
```

נוריד את מספר המלאים ב 1

נעלה את מספר הריקים

בעיות של קריאה וכתיבה במקביל - Readers-Writers Problem (עמ' 94 קוד מפורט)

קוראים וכותבים יחדיו יוצרים בעיה . אם ניתן עדיפות לקוראים , העבודה תהייה מהירה יותר . אם ניתן עדיפות לכותבים, המידע יהיה עדכני בכל רגע נתון.

אסור שיוצר מצב שבו מספר כותבים או מספר כותבים וקוראים. המצב היחיד האפשרי לעבודה בו זמנית הוא מספר קוראים. (אפשר שיהיו כמה כותבים ביחד על אותו קובץ רק אם ננעל חלקים מסוימים מהקובץ , חלק עבור כל כותב)

עדיפות לכותבים

אם יש כותבים בפנים וכותב נוסף בתור , נכניס אותו ברגע שהקודם יסיים . אם יש קורא בפנים וקורא נוסף בתור אזי נכניס אותו, אבל אם יש כותבים נוספים בתור נכניס אותם קודם.

עדיפות לקוראים

נעדיף להכניס את הקוראים , ואולי אף מספר קוראים ביחד , על חשבון כותבים . במצב זה תיתכן הרעבה לכותבים, וקריאת מידע לא מעודכן.

לכן במקרה זה אין פתרון אופטימלי.

אלגוריתם החדר

הראשון שנכנס לחדק מדליק את האור . כאשר נכנסים קוראים לחדר , הראשון מדליק את האור , וכל היתר נכנסים. הכותבים חייבים לחכות שהאור יכבה על מנת להיכנס.

אלגוריתם בשקף 6.30 עמוד 94 בחוברת (בדוגמא זו תתכן הרעבה של הכותבים).

בעיית הפילוסופים הסועדים - Dining-Philosophers Problem (עמ' 95,דוג' של קוד בעמ' 309)

תיאור הבעיה : 5 פילוסופים יושבים סביב שולחן עגול . בין כל 2 צלחות יש מקל אכילה אחד . על מנת לאכול, כל פילוסוף זקוק לשני מקלות אכילה. כל פילוסוף רוצה לאכול ולדבר.

אם כל פילוסוף ייקח תחילה את המקל שמימינו , ורק לאחר מכן את המקל שמשמאלו , המקל כבר יילקח ע"י הפילוסוף האחר.

פתרון 1

נחזיק מערך של 5 semaphores. כל פילוסוף שרוצה לאכול יבצע wait לימינו ולשמאלו. פתרון זה עלול ליצור הרעבה – כל אחד תופס מקל ימני ומחכה לנצח לשמאלי.

```
shared data:
        var chopstick: array[0,…4] of semaphore (=1 אתחול)
philosopher i:
        repeat:
                wait(P)
                wait(chopstich[i]); //פילוסוף i מרים מקל מימין
                wait(chopstick[i+1 mod 5]); // המתנה להרמת מקל משמאל
                signal(P)
                …..
                eat
                …..
                signal(chopstick[i]);
                signal(chopstick[i+1 mod 5);
                ……
```

פתרון באפור – נוסיף semaphore עבור תפיסת שני מקלות . פתרון זה צולע כיוון שלמרות ששניים יכולים לאכול (במידה ולדוגמא 2 ו 4 אוכלים), אבל תתכן גם הרעבה אם אחד אוכל ושניים תופסים מקל מימינם

think

…..

until false

<u>פתרון 2</u>

נפתח את המעגל, ונוסיף מקל וירטואלי. בקצוות קיים פילוסוף אחד עם מקל אחד בוודאות.

| 5 | 4 | 3 | 2 | 1 | כעת כשיש שישה מקלות , לא ייווצר מצב של deadlock בוודאות, כיוון שהפילוסופים מסיימים לאכול בשלב מסוים ולכן המקל האפור יגיע בוודאות גם ל 5 וגם ל 4 (למשל) .

<u>פתרון 3</u>

wait(chopstick[min(i, ((i+1) mod 5)]);

wait(chopstick[max(i, ((i+1) mod 5)]);

מסתכלים לכל פילוסוף על min ו-max. עבור פילוסוף 1 לדוג' מסתכלים רק על מקלות 1,2, פילוסוף 2 – 2,3, פילוסוף 4 – 0,4, פילוסוף 0 – 0,1.

פילוסופים 0 ו-4 מתחרים על מקל 0 כאשר רק אחד מהם יכול לקבלו, ושניהם קודם כל רוצים את 0 (מינימום 0,4 ו- 4,0), ואז המעגל נפתח.

האלג' – כל פילוסוף לוקח קודם את המינימלי ורק אז את המקסימלי. נניח ש-1 לקח את מקל 0, ואז מקל 4 פנוי, אם פילוסוף 3 לקח את מקל 4 אז בהכרח כבר יש לו את מקל 3, ואם פילוסוף 3 לא לקח את מקל 3 אז אם פילוסוף 2 לקח את מקל 3 אז בהכרח לקח את מקל 2.

באלג' זה תתבצע נעילה של אחד בלבד ולא של 3 .

פרק 8 – קִפָּאוֹן - Deadlocks

מודל מערכת (בהקשר הקִפָּאוֹן) System Model

תהליך חייב לבקש משאב לפני שהוא משתמש בו , וכן חייב לשחרר אותו לאחר שסיים להשתמש בו . תהליך אינו מוגבל במספר המשאבים אותם הוא יכול לבקש. לכן ברוב המקרים מספר המשאבים המבוקש יהיה גדול ממספר המשאבים בפועל.

במערכת רגילה, תהליך עשוי לנצל משאב באחת מהדרכים הבאות בלבד:

1. Request – אם לא ניתן לבצע את הבקשה באופן מיידי , הרי שהתהליך המבקש יאלץ להמתין עד שיוכל "לרכוש" את המשאב.
2. Use – התהליך יכול לפעול על המשאב.
3. Release – התהליך משחרר את המשאב.

בקשה ושחרור משאב נעשים ע"י system call.

בכל שימוש במשאב , מערכת ההפעלה מוודא כי לתהליך יש בקשה וכן הוקצה לו משאב . המערכת שומרת טבלה עבור משאבי המער כת בה היא שומרת את סטאטוס המשאבים בכל מצב נתון , ובמידת הצורך לאיזה תהליך מוקצה המשאב. אם תהליך מבקש משאב שמוקצה כרגע לתהליך אחר, הוא ממתין למשאב זה.

מצב קיפאון (deadlock) הינו מצב שבו קבוצה של תהליכים (חלקם תופסים משאבים וחלקם לא) הממתינים למשאב שתפוס ע"י אחד מהתהליכים בקבוצה. המשאבים עשויים להיות משאבים פיזיים (מדפסת, כונן טייפ, מרחב זיכרון, CPU ועוד) או משאב לוגי (קובץ, semaphores ו-monitors).

תנאים למצב הקִפָּאוֹן - Deadlock Characterization

לא ניתן להימנע ממצב של קיפאון . במצב של קיפאון , תהליכים אף פעם לא מסתיימים ומערכת המשאבים "קשורה" ואינה מאפשרת לתהליכים חדשים להתחיל לרוץ.

מצב של קיפאון עשוי להתעורר **אם כל ארבעת התנאים הבאים** מתקיימים במקביל במערכת:

1. Mutual exclusion – לפחות משאב אחד מוחזק באופן שלא ניתן לחלוק בו . כלומר, בכל פעם רק תהליך אחד יכול להשתמש במשאב . כאשר תהליך אחד יבקש את המשאב הנ "ל, הוא יאלץ להמתין עד שהמשאב יתפנה.
2. Hold and Wait – קיים תהליך שמחזיק לפחות משאב אחד , והוא ממתין למשאבים נוספים שכרגע מוחזקים ע"י תהליכים אחרים.
3. No preemption – שחרור המשאבים נעשה אך ורק ע "י התהליך עצמו , ברגע שהתהליך סיים את משימתו. לא ייתכן מצב בו "לוקחים" לתהליך את המשאב.
4. Circular wait – קיימת קבוצת $\{P_0, P_1, ..., P_n\}$ של תהליכים שממתינים למשאבים, כאשר P_0 ממתין למשאב שתפוס ע"י P_1, P_1 ממתין למשאב שתפוס ע"י P_2 וכך הלאה, כאשר P_n ממתין לתהליך שתפוס ע"י P_0.

שיטות לטיפול בקִפָּאוֹן - Methods for Handling Deadlocks

קיימות שלוש דרכים שונות לטפל בבעיית קיפאון:

1. מניעה – שימוש בפרוטוקול אשר מבטיח שמהערכת לעולם לא תיכנס למצב קיפאון.
2. נאפשר למערכת להיכנס למצב של קיפאון, ואז להתאושש ממנו.(windows – recovery פועל כך)
3. נתעלם מהבעיה, ונעמיד פנים שלא ייתכן מצב של קיפאון. שיטה זו נפוצה ברוב מערכות ההפעלה כולל Unix.

פרק 9 – ניהול זיכרון

Background - רקע

ברוב המקרים, תוכנית נמצאת בדיסק בתור קובץ בינארי הניתן להרצה . בכדי להריץ את התוכנית יש להביא תחילה את התוכנית לזיכרון ולמקם אותה בתוך תהליך . בהתאם לצורת ניהול הזיכרון , התהליך עשוי לעבור בין הדיסק לזיכרון במשך הריצה שלו . אוסף הת הליכים בדיסק אשר ממתינים להילקח לזיכרון מהווים input queue.

התהליך הרגיל הוא בחירת תהליך אחד מתור ה- input וטעינתו לזיכרון . ברגע שהתהליך רץ , הוא ניגש לפקודות ונתונים היושבים בזיכרון. בסופו של דבר, התהליך מסתיים, וכל מרחב הזיכרון שלו מוכרז כפנוי.

רוב המערכות מאפשרות לתהליכי משתמש להיות בכל חלק מהזיכרון הפיזי . לכן, למרות שמרחב הכתובות במחשב מתחיל ב- 00000, הכתובת הראשונה של תהליך משתמש לאו דווקא תהייה שם . סידור זה משפיע על הכתובות בהם יכולה תוכנית משתמש להשתמש . ברוב המקרים, תוכנית משתמש תעבו ר מספר שלבים לפני הרצתה. לאורך השלבים האלו , הכתובות יכולות להופיע בדרכים שונות . כתובות בתוכנית המקור הן לרוב כתובות סימבוליות . המחשב יקשר (bind) את הכתובות הנ ״ל לכתובות relocatable (לדוגמא 14 בתים מתחילת המודול). ה-linkage editor או ה-loader יהפוך כתובות אלו לכתובות מוחלטות (לדוגמא 74014). כל קישור כזה (binding) הוא מיפוי של מרחב זיכרון אחד לאחר.

Logical Versus Physical Address Space - כתובת לוגית מול פיסית

כתובת לוגית – כתובת שנוצרת ע״י ה-CPU. מכונה גם virtual address. כתובת לוגית **מתחילה ב-0** ומוגבלת ב-2^{32}-1. מכיוון שכתובת היא בגודל 32 ביט מס׳ הכתובות המקסימלי יהיה 2^{32}-1.

כתובת פיזית – כתובות ממשית בזיכרון. מוגבלת באורך הזיכרון הקיים. כתובת ב RAM

אוסף כל הכתובות הלוגיות שנוצרת ע ״י התוכנית נקראות מרחב כתובות לוגי . אוסף כל הכתובות הפיזיות המתאימות לכתובות לוגיות אלו נקראות מרחב כתובות פיזי.

בשיטת קישור בזמן קומפילציה או בזמן טעינה נוצרת סביבה שבה הכתובות הפיזיות והלוגיות זהות . אולם בשיטת קישור בזמן ריצה נוצרת סביבה בה הכתובות שונות.

רכיב חומרה לניהול זיכרון (מיפוי לוגי/פיסי) - Memory Management Unit - MMU

MMU הינו רכיב חומרה אשר מבצע את מיפוי הכתובות הלוגיות לכתובות פיזיות.

אחת הדרכים לממש MMU הוא להשתמש ב-relocation register. למעשה מתייחסים ל-base register בשם החדש relocation register. מוסיפים את ערך הרגיסטר לכל כתובת המיוצר ע״י תהליך המשתמש ברגע שזאת נשלחת לזיכרון . לדוגמא, אם ערך הרגיסטר הוא 14,000, אזי ניסיון של המשתמש לגשת לכתובת 0 יוקצה באופן דינאמי לכתובת 14,000.

נשים לב שתוכנית משתמש לא רואה את הכתובת הפיזית . התוכנית מבצעת את כל הפעולות שלה לפי כתובת לוגית. רכיב ה-MMU הופך כתובות אלו לכתובות פיזיות.

למעשה יש לנו שני סוגי כתובות : כתובות לוגיות בתחום מ- 0 עד max, וכתובות פיזיות בתחום מ- (0+R) עד (max+R).

Swapping - החלפות זיכרון שמבצעת מערכת ההפעלה

הרעיון הוא להעביר תהליכים מהדיסק אל הזיכרון וחזרה מהזיכרון אל הדיסק.

תהליך צריך להיות בזיכרון כדי לרוץ . אבל, ניתן להוציא (swapped) באופן זמני את התהליך מחוץ לזיכרון ל-backing store, ולהביא אותו בחזרה לזי כרון להמשך ריצה . לדוגמא, נניח שיש לי סביבת multiprogramming שעובדת בעזרת מתזמן round-robin. ברגע שיחידת הזמן שהוקצתה לתהליך פגה, מנהל

הזיכרון יתחיל להוציא החוצה (swap out) את התהליך שהסתיים , ויכניס חזרה (swap in) תהליך אחר מהזיכרון. בינתיים, מתזמן ה-CPU יקצה יחידת זמן חדשה לתהליך אחר בזיכרון.

שיטה נוספת של פעולת swapping, הנקראת roll out, roll in, משמשת לצורך אלגוריתם priority-based scheduling. כאשר תהליך בעדיפות גבוהה מגיע , מנהל הזיכרון מוציא החוצה תהליך בעל עדיפות נמוכה . ברגע שהתהליך בעל העדיפות הגבוהה מסיים , התהליך בעל העדיפות הנמוכה יכול להיות מוחלף חזרה לזיכרון.

במידה והקישור נעשה בזמן ריצה ניתן יהיה להעביר את התהליך למקום חדש.

תהליך ה-swap דורש backing store. ברוב המקרים זה יהיה דיסק מהיר . גדול מספיק להכיל את כל מופעי הזיכרון של כל המשתמשים , וכן חייב לספק גישה ישירה לאותם מופעי זיכרון . המערכת שומרת תור ready של כל התהליכים שמופעי הזיכרון שלהם נמצאים ב-backing store, או שהתהליכים מצויים בזיכרון ומוכנים לריצה. ברגע שמתזמן ה- CPU מחליט להריץ תהליך , הוא קורא ל- dispatcher. ה-dispatcher בודק האם התהליך הבא נמצא בזיכרון. במידה ולא, וכן אין מספיק מקום בזיכרון , ה-dispatcher מוציא החוצה תהליך שנמצא בזיכרון ומכניס חזרה את התהליך הדרוש.

זמן ה-context-switch במקרים כאלה הוא גבוה מאוד . כדי ליעל את תפוקת ה-CPU, נרצה שזמן הריצה של תהליך יהיה ארוך באופן יחסי לזמן ה- swap. נשים לב כי חלק עיקרי מזמן ההחלפה הוא זמן העברה . סך כל זמן העברה הוא פרופורציונאלי ביחס ישר לגודל הזיכרון שהוחלף.

הקצאת רצף זיכרון - Contiguous Allocation

שיטה ראשונה של הקצאה שומרת את כל התוכנית ברצף בזיכרון.

הזיכרון הראשי מכיל הן את מערכת ההפעלה והן סוגים שונים של תוכניות משתמש . ברוב המקרים, הזיכרון מחולק לשני חלקים, בכדי להפריד בין השניים. את מערכת ההפעלה ניתן לאחסן באזור זיכרון נמוך או גבוה . השיקול המרכזי בהחלטה זו תלוי במיקום ה- interrupt vector. מכיוון שווקטור זה נמצא לרוב באזור זיכרון נמוך, מקובל למקם גם שם את מערכת ההפעלה.

הקצאה שרק לתהליך אחד מותר לרוץ - Single-Partition Allocation

אם מערכת ההפעלה יושבת באזור זיכרון נמוך , ותוכניות משתמש רצות באזור זיכרון גבוה , יש צורך להגן על הקוד והנתונים של מערכת הפעלה מפני שינויים שהתבצעו ע"י תהליכי משתמש . בנוסף יש להגן על תהליך משתמש אחד מפני השני. ניתן לספק הגנות אלו תוך כדי שימוש ב-relocation register ו-limit register.

ה-relocation register מכיל ערך של הכתובות הפיזית הקטנה ביותר.

ה-limit register מכיל את טווח הכתובות הלוגיות.

בעזרת שני רגיסטרים אלו , כל כתובת לוגית תהייה קטנה יותר מה- Limit register. רכיב ה-MMU ממפה באופן דינאמי את הכתובת הלוגית ע"י הוספת ערך ה-relocation register. הכתובת הממופה נשלחת לזיכרון.

כאשר מתזמן ה- CPU בוחר תהליך , ה-dispatcher טוען את שני הרגיסטרים כחלק מתהליך ה- context switch. – פה לא צריך swapping כיוון שיש partition אחד

הקצאה כשכמה תהליכים יכולים לרוץ במקביל - Multiple-Partition Allocation

בזיכרון מצויים מספר תהליכי משתמש בו זמנית , ולכן עלינו להתחשב בבעיה של הקצאת זיכרון נגיש למגוון התהליכים הנמצאים בתור ה- input וממתנים להגיע לזיכרון . אחת השיטות הפשוטות יותר להקצאת זיכרון היא חלוקת הזיכרון למספר קבוע של partition-ים, כל partition יכול להכיל תהליך אחד בלבד. בצורה כזאת רמת ה-multiprogramming תלויה במספר ה-partition-ים. כאשר partition פנוי, נבחר תהליך מתוך תור ה-Input והוא נטען אל תוך ה-partition. כיום לא משתמשים יותר בשיטה זו , אך נסתכל על וריאציה של שיטה זו.

מערכת ההפעלה מחזיקה טבלה המורה אילו חלקים בזיכרון פנויים ואילו חלקים תפוסים . בתחילה כל הזיכרון זמין לתהליכי משתמש, ונחשב כבלוק אחד ארוך של זיכרון זמין . בלוק זה נקרא Hole. כאשר תהליך

מגיע וזקוק לזיכרון , מחפשים אחר hole גדול מספיק לתהליך . במידה ונמצא , מקצים זיכרון רק בכמות הדרושה, ושומרים על יתר הזיכרון זמין לבקשות עתידיות.

באופן הכללי התהליך מתואר כך : כאשר מגיע תהליך למערכת , הוא מאוחסן בתוך ה- input תור. כאשר מערכת ההפעלה מחליטה להקצות זיכרון לתהליך , היא צריכה לקחת בחשבון את כמות הזיכרון הדרושה לכל תהליך וכן את כמות הזיכרון הזמינה.

בכל זמן נתון ק יימת רשימה של גדלי בלוקים זמינים וכן את תור ה- input. זיכרון מוקצה לתהליכים , עד שלבסוף לא ניתן לספק את בקשת הזיכרון של תהליך הבא . כלומר אין בלוק זיכרון זמין הגדול מספיק בשביל אותו תהליך . במקרה כזה מערכת ההפעלה יכולה לחכות עד שיהיה בלוק גדול מספיק , או להמשיך הלאה לתהליכים הבאים בתור, ולמצוא תהליך שדרוש כמות זיכרון קטנה יותר.

הבעיה עם הקצאת זיכרון באופן דינאמי - Dynamic Storage-Allocation Problem

באופן כללי, בכל זמן נתון קיי מת קבוצה של hole-ים, בגדלים שונים מפוזרים על פני הזיכרון . כאשר מגיע תהליך וזקוק לזיכרון , מחפשים בקבוצה hole גדול מספיק לתהליך . אם ה-hole גדול מידי הוא מחולק לשני חלקים : חלק אחד מוקצה לתהליך והחלק השני חוזר לקבוצת ה- hole-ים. כאשר התהליך מסיים , הוא משחרר את בלוק הזיכרון שלו, אשר חוזר לקבוצת ה-hole. במידה והבלוק ששוחרר צמוד פיזית ל- hole אחד בקבוצה, מאחדים את השניים ביחד. בשלב זה יהיה עלינו לבדוק האם האיחוד של שני הבלוקים יצר לנו hole גדול מספיק עבור תהליך שממתין לזיכרון.

תהליך זה הינו דוגמא לבעיית ההקצאה הדינא מית. הבעיה היא למעשה איך לבחור hole מתוך הרשימה . קיימות 3 שיטות נפוצות :

1. First Fit – הקצאת ה-hole הראשון ברשימה אשר גדול מספיק בשביל התהליך . החיפוש יכול להתחיל מתחילת הרשימה או מהנקודה האחרונה שבה עצרנו.
2. Best Fit – הקצאת ה-hole הקטן ביותר אשר גדול מספיק בשביל התהליך . יש לעבור על כל הרשימה.
3. Worst Fit – הקצאת ה-hole הגדול ביותר ברשימה. שוב יש לעבור על כל הרשימה.

First-fit ו-best-fit עדיפות על פני worst-fit מבחינת מהירות ואחסון.

קטע – שיברור – זיכרון התהליך מחולק להרבה חלקים - Fragmentation

האלגוריתם שתואר לעיל (שימוש ב-hole) יוצר בעיות שיברור (fragmentation). מכיוון שהתהליכים נטענים ומוצאים מהזיכרון , הזיכרון הפנוי "נשבר" לחתיכות קטנות . External Fragmentation קיים כאשר יש מספיק זיכרון פנוי עבור הבקשה, אבל זיכרון זה אינו רציף. במקרה הגרוע, אנחנו יכולים לקבל בלוקים פנויים של זיכרון בין כל שני תהליכים. אם כל הבלוקים האלו היו יושבים באופן רציף, היינו עשויים להריץ תהליכים נוספים.

בעיה נוספת - נתון hole בגודל 18,464 בתים, והתהליך עצמו זקוק ל-18,462 בתים. אם נקצה בלוק בגודל הבקשה, נקבל hole בגדול 2 בתים. ה-overhead הדרוש לעקוב אחר hole זה הוא לרוב גדול יותר באופן ממשי מה-hole עצמו. לכן הגישה היא להקצות hole-ים קטנים מאוד כחלק מהבקשה עצמה . בצורה כזאת, הזיכרון שמוקצה גדול במעט מהזיכרון הדרוש . ההבדל בין שני ערכים אלו נקרא Internal Fragmentation – זיכרון שנמצא בתוך partition אבל אין בו כל שימוש.

אחת הדרכים לפתור את בעיית ה- External Fragmentation נקראת Compaction. המטרה היא לערבב את תוכן הזיכרון ולמקם את כל הזיכרון הפנוי בבלוק אחד גדול (איור 8.10 עמוד 142 בחוברת).

פעולת הדחיסה אפשרית רק כאשר מדובר בהקצאה דינאמית, והיא נעשית בזמן ריצה.

הדרך הפשוטה ביותר לבצע דחיסה (Compaction) היא ע"י העברת כל התהליכים לצד אחד של הזיכרון , ואת כל החללים הפנויים לצד שני של הזיכרון , וכך נוצר חלל אחד גדול של זיכרון נ גיש. תהליך זה עלול להיות יקר מידי.

פתרון קטוע ע"י שימוש בדפים - Paging

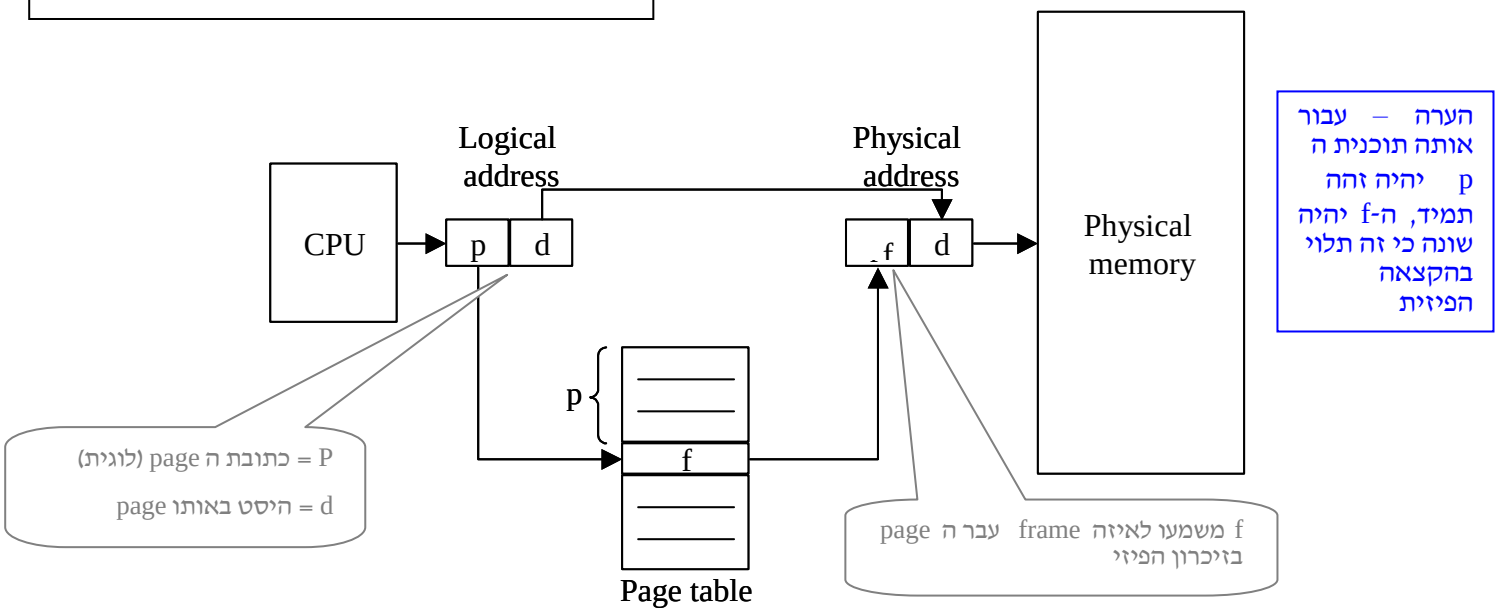
פתרון נוסף לבעיית ה-External Fragmentation הוא לאפשר לתהליך לעבוד עם מרחב כתובות לוגי לא רציף, וכך התהליך מוקצה לתהליך זיכרון פיזי היכן שהוא פנוי . שיטה אחת למימוש פתרון זה נעשית ע"י שימוש ב-paging.

הזיכרון הפיזי (RAM) מחולק לבלוקים בגודל קבוע הנקראים frames. הזיכרון הלוגי מחולק גם הוא לבלוקים באותו גודל הנקרא ים pages. כאשר תהליך אמור לרוץ הדפים שלו נטענים מתוך ה-backing storage.

ה-backing storage עצמו מחולק גם כן לבלוקים באותו גודל.

ביאור – page מגיע מהדיסק (היה בלוק שם) אל הזיכרון הפיזי(RAM) והופך ל frame. תהליך זה נקרא paging

החומרה הדרושה לתמיכה ב-paging מוצגת באיור הבא:



הערה – עבור אותה תוכנית ה p יהיה זהה תמיד, ה-f יהיה שונה כי זה תלוי בהקצאה הפיזית

כל כתובת שנוצרת ע"י ה-CPU מחולקת לשני חלקים: page number (p) ו-page offset (d). מספר העמוד משמש כאינדקס לטבלת הדפים (page table) אשר מכילה את כתובת הבסיס של כל דף בזיכרון הפיזי. כתובת הבסיס מאוחדת עם ההיסט של הדף כדי להגדיר את הכתובת הפיזית שנשלחת לזיכרון.

דוגמא שקף 8.18 עמוד 143 בחוברת.

גודל הדף נקבע ע"י החומרה. ברוב המקרים מדבר בחזקה של 2, בין 512 בתים ל- 16 מגה בית, תלוי בארכיטקטורת המחשב.

נשים לב ש-paging הוא סוג של הקצאה דינאמית. כל כתובת לוגית מקושרת ע"י דף החומרה לכתובת פיזית.

שימוש בשיטת ה-paging מונע external fragmentation. ניתן להקצות לת הליך כל frame פנוי. אבל, אנחנו עשויים לקבל internal fragmentation. Frame-ים מוקצים כיחידה. במידה והזיכרון הדרוש אינו נופל בגבולות ה-page, הרי ש-frame האחרון שמוקצה עשוי להיות לא מלא לחלוטין. במקרה הגרוע, לתהליך שזקוק ל-n דפים ועוד בית אחד, יוקצו (n+1) דפים, וכתוצאה מכך יתקבל internal fragmentation של כמעט frame שלם.

גודל טבלת הדפים הינו: (גודל התוכנית / גודל דף (4k)) * 4 (גודל כתובת)

** paging – הוצאת page מסוים והכנסת אחר.

swapping – הוצאת כל ה – pages של תהליך מסוים

מימוש טבלת הדפים

טבלת הדפים נשמרת בזיכרון הראשי. ב-PCB נשמר רגיסטר page table base register (PTBR) המכיל מצביע לטבלת הדפים, וכן רגיסטר page table length register (PTLR) המכיל את גודל טבלת הדפים.

הבעיה בשיטה זו היא זמן הגישה לזיכרון , משום שדרושות שתי גישות לזיכרון (גישה אחת לטבלת הדפים וגישה אחת לנתון עצמו). הפתרון לבעיה זו הוא שימוש ב- cache, הנקרא associative register או translation look-aside buffers (TLBS). נבנה אוסף של רגיסטרים בזיכרון מהיר במיוחד . הרגיסטרים מכילים מספר מסוים של כניסות מטבלת הדפים . כאשר כתובת לוגית מיוצרת ע"י המעבד, מספר הדף שלה משווה מול סט הרגיסטרים . במידה ומספר הדף נ מצא, ניתן לקבל מהרגיסטר את מספר ה- frame המתאים ולפנות ישירות לזיכרון . במידה והמספר לא מופיע , יש לחפש את ה- frame בטבלת הדפים, ולעדכן את אחד הרגיסטרים בערך זה.

ה- TLB הוא בעצם cache שצמוד לזיכרון, והרעיון הוא שכאשר מוציאים דפים מהזיכרון מתקיים:

1. אם אלה דפים ששינינו אז נשמור אותם.
2. אחרת פשוט נכתוב עליהם.

כאשר מבקשים דף , מבקשים אותו מהדיסק , כדי לקבל אותו יש תהליך שלוקח זמן , הרעיון של TLB הוא שדפים שיוצאים נשמרים בו ע"י אלג' מסוים(יותר מהיר מה memory) ולכן כאשר רוצים דף חדש , קודם בודקים אם הוא מופיע ב TLB ולכן זה חוסך את הגישה אל הדיסק.

טבלת דפים בעלת שתי רמות - Two-Level Page-Table Scheme

רוב מערכות המחשב המודרניות תומכות במרחב כתובות גדול מאוד . בסביבה כזו טבלת הדפים עצמה גדולה בצורה מוגזמת. לא נרצה לשמור את טבלת הדפים כגוש אחד בזיכרון. פתרון אחד הוא לחלק את טבלת הדפים לחתיכות קטנות.

דרך אחת ליישם זאת היא שיטת ה- two level page table (שקף 8.23 עמוד 148 בחוברת) שבה הטבלה עצמה מורכבת מדפים. כתובת לוגית מורכבת מ-20 ביטים שיכילו את מספר העמוד ו-12 ביטים המכילים את היסט העמוד. מכיוון שהפכנו את טבלת הדפים לעמודים , נחלק את 20 הביטים של מספר העמוד לשני חלקים : 10 ביטים שיכילו את מספר העמוד ו-10 ביטים שיכילו את ההיסט.

כלומר קיבלנו דף שמכיל k כניסות. כל כניסה מצביעה על חלק מטבלת הדפים.

Hashed Page Table

Address Translation Scheme

הפעם מחלקים ל –3 כל כתובת לוגית:

P1 – ההיסט בטבלת הטבלאות (כניסה בטבלה הראשונה) => איפה נמצאת הטבלה הרלוונטית.

P2 – ההיסט בטבלת הדפים => בהיסט של P2 נמצאת הכתובת של ה – page האמיתי

d – הפקודה נמצאת בהיסט d ב-page הנכון

שיתוף דפים – שיתוף של קוד שכיח בין תהליכים - Shared Pages

אחד היתרונות בשימוש ב-paging היא היכולת לשתף קוד שכיח.

הגישה אומרת – לא נקצה את הזיכרון מחדש לכל תהליך , אלא כאשר נטען מידע לזיכרון עבור תהליך מסוים, נבדוק אולי המידע הזה כבר קיים בזיכרון מתהליך אחר.

קוד משותף חייב להופיע באותן כתובות לוגיות בכל התהליכים שמשתמשים בו.

בדוגמא משתפים את קוד התוכנית ב- 3 תהליכים וה data שונה לכן לשלושת התהליכים הפניה לאותו מקום אבל data שונה.

סֶגְמֶנְטַצְיָה- פִּלּוּחַ – הסתרת הזיכרון הפיסי - Segmentation

אחד האספקטים החשובים בניהול זיכרון אשר נעשה בלתי- נמנע בעבודה עם paging הוא ההפרדה בין איך שהמשתמש רואה את הזיכרון לבין איך שהזיכרון באמת בנוי פיזית.

Segmentation היא שיטת ניהול זיכרון התומכת בשיטה זו . מרחב הכתובת הלוגי הינו אוסף של segment-ים. **לכל segment יש שם ואורך.** הכתובות מורכבת גם משם הסגמנט וכן מההיסט בתוך הסגמנט . המשתמש אם כן צריך לציין את שני הגדלים . בכדי להקל על המימוש , הסגמנטים הם ממוספרים ופונים אליהם בעזרת המספר ולא השם.

למרות שהמשתמש יכול לפנות כעת אל אובייקט בתוכנית בעזרת כתובת בעל 2 ממדים, הכתובות הפיזית היא עדיין סידרה בממד אחד של בתים . לכן, יש צורך להגדיר כיצד יתבצע המיפוי מהכתובת המוגדרת ע"י המשתמש לכתובת הפיזית . תהליך המיפוי מתבצע בעזרת segment table. כל כניסה בטבלת הסגמנטים מכילה שני ערכים: סגמנט base וסגמנט limit. סגמנט הבסיס מכיל את הכתובת הפיזית בזיכרון בה מתחיל הסגמנט, וסגמנט ה-limit מציין את אורך הסגמנט.

טבלת הסגמנטים, כמו טבלת הדפים , יכולה להישמר או בזיכרון או ברגיסטרים מהירים (fast register). במצב שבו תוכנית כוללת מספר גדול של סגמנטים , לא ניתן לשמור את הטבלה ברגיסטרים , וחייבים לשמור אותה בזיכרון . לצורך כך שומרים segment table base register (STBR) המצביע לטבלת הסגמנטים. בנוסף, בגלל שמספר הסגמנטים של התוכנית גדול , נעשה שימוש ב- segment table length register (STLR), המציין את מספר הסגמנטים בתוכנית . עבור כתובת לוגית (s,d) נבדוק תחילה שמספר הסגמנט s חוקי (כלומר s צריך להיות קטן מ-STLR). לאחר מכן מוסיפים למספר זה את ה-STBR.

זה בעצם בין פיזור התוכנית לבין תוכנית בגוש אחד, החלקים הם לפי נושאים.
יתרון – לא כל התוכנית ברצף, מקל על ה external fragmentation
חסרון – עדיין external fragmentation כיוון שאם אין מקום לכל הגושים אז תהיה בעייה כאשר ייתכן שאם נפזר את התוכנית אז יהיה מקום.

הפתרון – נעשה segmentation ונעבוד בו עם frames (ה segment יהיה בנוי מ frame-ים). להלן…

שילוב דפים וסגמנטים ביחד - Segmentation With Paging

גם ל-paging וגם ל-segmentation יש את היתרונות והחסרונות שלהם . קיימת אפשרות לשלב את השניים במטרה לשפר את שניהם. שילוב זה מוצג בשתי ארכיטקטורות שונות:

1. מערכת MULTICS – מערכות אלו פותרות את בעיית ה- external fragmentation וזמן החיפוש ע"י חלוקת הסגמנטים לדפים . כל הזיכרון מחולק לדפים , וגודל הסגמנטים הוא כפולות של דפים . שקף 8.38 עמוד 155 בחוברת.
2. מעבדי Intel 386 – לכל סגמנט יש טבלת דפים מפוצלת. עובדים עם סגמנטציה ודפדוף.

פרק 10 – זיכרון וירטואלי

Background - רקע

<u>הנחה</u> – לא כל התוכנית בזיכרון ואם כולה בזיכרון אז היא לא ברצף.

זיכרון וירטואלי הוא טכניקה המאפשרת הרצת תהליכים שאינם נמצאים בשלמותם בז יכרון, ומה שנמצא בזיכרון אינו נשמר באופן רציף . היתרון העיקרי שבולט הוא שתוכניות יכולות להיות גדולות יותר מהזיכרון הפיזי.

היכולת להריץ תוכנית שרק חלקה נמצא בזיכרון יוצרת מספר יתרונות:

1. תוכנית יותר לא תהייה מוגבלת בכמות הזיכרון הפיזי הפנוי.
2. משתמשים יוכלו לכתוב תוכניות למרחב כתובות לוגי גדול, מה שעושה את העבודה לקלה יותר.
3. יותר תוכניות יוכלו לרוץ בו זמנית, משום שהן תופסות פחות מקום.
4. פחות I/O יהיה דרוש לטעינה והחלפה של כל תוכנית משתמש, כל התוכניות ירוצו מהר יותר.

זיכרון וירטואלי הוא הפרדה בין זיכרון לוגי לזיכרון פיזי.

ניתן ליישם זיכרון וירטואלי בשתי דרכים:

1. Demand paging
2. Demand segmentation

Demand Paging - החלפת דפים ע"פ דרישה

מערכת demand paging דומה למערכת paging הכוללת swapping. תהליכים נשמרים בזיכרון המשני . כאשר נרצה להריץ תהליך נכניס (swap) אותו לזיכרון. בנוסף לכך נשתמש ב-lazy swapper. Lazy swapper לעולם לא יכניס לזיכרון תהליך אלא אם כן יש בו צורך . מכיוון שאנו מתייחסים כעת לתהליך כ על סידרה של דפים, ולא מרחב כתובות רציף , ולכן המונח swap אינו נכון מבחינה טכנית . Swapper מופעל על תהליכים שלמים, בעוד ש-pager עוסק בדפים עצמאיים של תהליך . ולכן, נשתמש במונח pager בהקשר של demand paging.

כאשר יש צורך להביא תהליך לזיכרון, ה-pager מנחש אילו דפים יהיו בשימוש לפני שהתהליך יוצא שוב מחוץ לזיכרון. במקום להכניס את כל הזיכרון , ה-pager מביא רק את אותם דפים דרושים . כך, נמנעים מקריאת דפים לזיכרון שבמילא לא יעשה בהם שימוש , ובכך מפחיתים את זמני ההחלפות ואת כמות הזיכרון הפיזי הנדרשת.

בשיטה זו, יש צורך לתמוך בחומרה היודעת להבחין בין דפים שנמצאים בזיכרון לבין אלו שבדיסק . לצורך כך מוסיפים לכל דף ביט valid-invalid. ערך valid מציין שהדף הוא חוקי ונמצא בזיכרון . ערך invalid מציין שהדף או לא חוקי או לא נמצא בזיכרון . כאשר הערך הוא ב- valid הכניסה המתאימה לדף בטבלת הדפ ים מעודכנת כרגיל, אבל כאשר הערך ה- invalid הכניסה עצמה מסומנת כ- invalid או מכילה את כתובת הדף בדיסק.

<u>שגיאת דף – פניה לדף שכרגע לא בזיכרון - Page Fault</u>

גישה לדף המסומן כ- invalid גורמת לשליחת page fault trap למערכת ההפעלה . Trap זה הוא תוצאה מניסיון כושל של מערכת ההפעלה להביא דף רצוי לזיכרון , בניגוד לשגיאת כתובת לא נכונה שהיא תוצאה של ניסיון גישה לכתובת לא חוקית. יש צורך להבין כיצד נוצרה הטעות האם בגלל שהדף לא נמצא או הכתובת לא חוקית. <u>תהליך הטיפול ב-Page Fault</u> יהיה אם כן:

1. בודקים בטבלה פנימית של התהליך (בד"כ נמצאת ב- PCB) כדי לקבוע האם ההפניה לזיכרון חוקית או לא.
2. אם ההפניה לא חוקית, נפסיק את התהליך (באסה). אם ההפניה חוקית והדף לא נמצא , נדאג להביא אותו כעת.

3. נמצא frame פנוי.
4. נתזמן פעולת דיסק אשר תקרא את הדף המתאים אל ה-frame החדש.
5. כאשר פעולת הקריאה הושלמה , נעדכן את הטבלה הפנימית של התהליך וכן את טבלת הדפים כך שתצביע על כך שהדף נמצא כעת בזיכרון.
6. נאתחל את ההוראה שהופרעה ע"י ה-trap. התהליך יוכל כעת לגשת לדף המתאים כאילו הוא תמיד היה בזיכרון (והם חיו באושר ובעושר עד עצם היום הזה).

כיצד ומתי נדפדף - Page Replacement

גישת החלפת הדפים אומרת שבמידה ואין שום frame פנוי, נחפש frame שלא נמצא כרגע בשימוש ונשחרר אותו. הבעיה היא שאם אני משחררת דף ששיניתי, לא ניתן לשחרר אותו סתם, אלא יש צורך לכתוב אותו שוב לזיכרון. סדר הפעולות המתקבל הינו:

1. מצא frame פנוי.
2. במידה ויש כזה השתמש בו.
3. אחרת, השתמש באלגוריתם להחלפת דפים כדי להוציא את הדף החוצה.
4. כתוב לדיסק את הדף שהוצאת ועדכן את הטבלאות.
5. קרא את הדף הרצי ל-frame הפנוי ושנה את הטבלאות בהתאם.
6. התחל את התהליך מחדש.

נשים לב כי הכפלנו את זמן ה- page fault משום שהוספנו כתיבה לדיסק . בחלק מהמקרים , ייתכן ובדף שהחזרנו כלל לא נעשו שינויים , ולכן אין צורך לכתוב אותו חזרה לדיסק . לכן נוסיף ביט בשם **dirty**. ערך הביט יהיה 1 ברגע שנעשה שינוי בדף הנ מצא בזיכרון. כאשר נחזיר את הדף לדיסק , נכתוב אותו רק במידה והביט יצביע על 1.

הפרדה אחרי שינוי - Copy On Write

Copy on Write מאפשר לתהליך אב ותהליך בן לחלוק דפים בזיכרון . כאשר יוצרים תהליך בן , האב והבן משתמשים באותם דפים (הבן מקבל רק העתק של טבלת הדפים). ברגע שאחד מהם משנה דף מסוים , הדף משוכפל והוא מקבל הצבעה עצמאית לדף זה . כך חוסכים בזמן יצירת תהליך חדש . בנוסף, חוסכים במקום - לפחות הקוד של התהליכים יהיה משותף.

מיפוי קבצים לזיכרון - Memory Mapped Files

כאשר תהליך רוצה לקרוא מקובץ מעה "פ טוענת חלק מהקובץ (בגודל דף) לזיכרון. כך ניתן להתייחס לקובץ כמו אל פנייה רגילה לזיכרון - ללא System Calls. הרבה יותר מהיר . מקבלים מצביע לאזור זה בזיכרון ואפשר לגשת לכל מקום בדף ישירות. אם חורגים מהגבולות - מקבלים פסיקה רגילה.

בנוסף, כך תהליכים יכולים לשתף קבצים - יכולים כולם לגשת לאותו אזור.

אלגוריתמים לדפדוף - Page Replacement Algorithms

קיימים אלגוריתם רבים להחלפת דפים . באופן כללי נרצה אלגוריתם שבו שיעור דפדוף הדפים הוא הנמוך ביותר (page-fault rate). נדע להעריך את האלגוריתם ע"י הרצה יבשה של הפניות זיכרון , כאשר תוך כדי נספור את מספר הפעמים בהם ביצענו החלפה.

אלג' הדפדוף –

- קובעת את <u>מדיניות החלפת הדפים</u>: איזו מסגרת בזיכרון הראשי תפונה על- מנת לפנות מקום לדף החדש.
- <u>דפדוף ע"פ דרישה</u> – paging by demand – מביא דף לזיכרון רק אם הגישה אליו גרמה ל page fault, להבדיל מ<u>הבאה מוקדמת</u> – המנסה לחזות איזה דפים ידרשו.
- אלג' דפדוף <u>לוקלי</u> יפנה מהזיכרון הפיסי רק דפים של התהליך שגרם ל page fault. משמר את מספר המסגרות שהוקצא לתהליך. להבדיל מאלג' דפדוף <u>גלובלי</u>.

1) אלג' ראשון בא, ראשון יצא - First In First Out (FIFO) Algorithm

ה-frame הראשון שנכנס לזיכרון יהיה גם הראשון שישוחרר. נשים לב שהגדלת מספר ה-frame-ים לא תקטין את מספר ההחלפות.

דוגמא בשקף 10.16

2) אלג' הדף האופטימלי - Optimal Algorithm

באלגוריתם זה קיימת הנחה שאנחנו יודעים מראש מה הולך לקרות. כלומר נדע מראש מתי נזדקק לכל דף (דבר שכמובן לא קורה במציאות). במקרה כזה נוציא תמיד מהזיכרון את הד ף שלא יהיה בשימוש הכי הרבה זמן.

3) אלג' הוצא את הלא פופולארי - Least Recently Used (LRU) Algorithm

נוציא מהזיכרון את הדף שלא היה בשימוש הכי הרבה זמן. ניתן לממש אלגוריתם זה בשתי דרכים:

1. מחסנית – בכל פעם שיש הפנייה לדף , נכניס את הדף לראש המחסנית . כאשר הדף שנמצא בתחתית המחסנית הוא הדף שלא השתמשנו בו הכי הרבה זמן . מכיוון שנרצה להוציא דפים מאמצע המחסנית , נממש את המחסנית בעזרת רשימה דו מקושרת עם מצב יע לראש ולזנב . במקרה הגרוע נצטרך לעדכן שישה פוינטרים (בעת הוצאה מהאמצע). כלומר, כל שיני כאן הוא יקר, אבל אין צורך לרוץ על כל המחסנית כדי למצוא את הדף הישן ביותר.
2. אפשר להוסיף שעון ולבדוק אותו
3. תור דו כיווני – בכל פעם שהשתמשנו בדף , נשים אותו בתחילת התור , וכאשר נרצה לזרוק דף , נזרוק את זה שבסוף התור.

**מימוש יקר ולכן קשה לממש

4) אלג' קירוב ל LRU - LRU Approximation Algorithm

מעט מאוד מערכות הפעלה משתמשות בחומרה התומכת בהחלפת דפים בעזרת LRU. אולם רוב המערכות מספקות עזרה בצור ה של reference bit. לכל דף מוסיפים ביט המותחל ל- 0. ברגע שיש הפניה לדף הביט נדלק. כאשר יש צורך להוציא דף, נבחר דף שהביט שלו עדיין ב-0.

אין אפשרות לדעת את סדר השימוש בדפים.

5) אלג' שמשתמשים במונה - Counting Algorithm

מונה כניסות - נוסיף מונה לכל דף . בכל פעם שמ שתמשים בדף, מגדילים את המונה של ו באחד. קיימים שני אלגוריתמים המשתמשים במונה זה:

1. Least Frequently Used (LFU) – אלגוריתם זה דורש שהדף בעל המונה הנמוך ביותר יוחלף . הסיבה לבחירה זו היא שדף שכבר השתמשנו בו הרבה פעמים יהיה בעל מונה הגישות הגבוה ביותר. החיסרון של האלגוריתם הוא מצב שבו בתהליך האתחול השתמשנו בדף מסוים פעמים רבות, אבל עכשיו כבר אין לנו שום צורך בו . הבעיה היא שדף זה לא יוחלף ו יישאר בזיכרון . הפתרון יעשה ע "י שמירת ה- counter בבסיס בינארי ולא נתייחס אליו כאל מספר . בכל פעם שניגשים לדף, נבצע הזזה ימינה במונה של כל דף ודף, כאשר רק בדף שנגענו נכניס 1 ולא 0.
2. MFU – אלגוריתם יעבוד באותה שיטה, אלא שהפעם נוציא מהזיכרון את הדף בעל המונה הגובה ביותר. אלגוריתם זה מתבסס על ההנחה שדף בעל מונה נמוך רק הובא לזיכרון ועדיין לא הספקנו להשתמש בו.

אף אחת מהשיטות הנ"ל אינה נפוצה מכיוון והן די יקרות.

הקצאה של מסגרות - Allocation of Frames

הבעיה מתעוררת כאשר יש דרישה לדפדף תוך כדי שילוב עם multiprogramming. במקרה כזה יש יותר מתהליך אחד בזיכרון, והשאלה היא כמה frame-ים נקצה לכל תהליך.

לא ניתן להקצות יותר frame-ים ממה שיש . קיים מספר מינימאלי של frame-ים שניתן להקצאת . ככל שמספר ה-frame-ים המוקצה לכל תהליך קטן , שיעור הדפדוף גדל ומאט את זמן הריצה של התהליך . בנוסף,

קיים מספר מינימלי של frame-ים שחייבים להקצות . מספר מינימלי זה מוגדר ע״י ה- instruction-set architecture. יש לזכור שברגע שמתרחש page-fault תוך כדי הוראה, ההוראה תאתחל מחדש. כתוצאה מכך, עלינו להחזיק מספיק fame-ים כך שיחזיקו את הדפים השונים אליהם יכולה כל הוראה לפנות.

המספר המינימלי של ה-frame-ים מוגדר ע״י הארכיטקטורה של המחשב.

המקרה הגרוע ביותר הוא כאשר יש רמות רבות של מיעון עקיף. כלומר כל frame מכיל בתוכו כתובת ל-frame אחר, ובכך מבזבזים מספר רב של frame-ים על כתובות ולא על מידע.

קיימות שתי שיטות הקצאה עיקריות:

1. Fixed allocation – הקצאה שווה. כל תהליך מקבל מספר קבוע של Frame-ים. . קיימת הקצאה לפי פרופורציה, בה כל תהליך מקבל frame-ים ביחס מתאים לגודל שלו.
2. Priority allocation – הקצאה על פי עדיפות . הקצאה על פי עדיפות עובדת באותה שיטה , אלא שהפעם מקצים לתהליך Frame-ים על פי העדיפות שלו . אם תהליך מסוים מייצר page fault, אזי בוחרים דף מאוסף הדפים שמוקצים לו . במידה ואין דפים פנויים , אזי מקצים לו דף מתהליך בעל עדיפות נמוכה יותר.

Global vs Local Allocation –

הקצאה לוקלית פרושה שלתהליך יש מס ׳ קבוע של מסגרות . כאשר תהליך צריך דף , רק דפים של התהליך שגרם ל page fault יוצאו מהזיכרון. להבדיל מהקצאה גלובלית, שם לתהליכים אין מספר קבוע של מסגרות , וזה משתנה בהתאם לבקשות התהליך.

בעיית דפדוף מהיר מדי - Thrashing

אם מספר ה- frame-ים שהוקצו לתהליך קטן מהמספר המינימ אלי שהוגדר ע ״י ארכיטקטורת המחשב , יש להשעות את ריצת התהליך.

למעשה, כל תהליך שאין לו מספיק דפים יגרום מהר מאד ל- page fault. בשלב זה הוא יהיה חייב להחליף כמה מהדפים . אבל, מאחר וכל הדפים שלו כרגע בשימוש , הוא יהיה חייב להחליף דף שהוא יהיה זקוק לו בהמשך. בצורה כזאת מהר מאוד יגרם page fault נוסף. כלומר נוצר מצ ב שבו שיעור דפדוף הדפים גבוה מאוד. מצב זה נקרא Thrashing. תהליך שב-thrashing מבזבז יותר זמן על החלפות מאשר על ריצה.

התוצאה של Trashing יוצר בעיות ביצוע רבות . מערכת ההפעלה עוקבת אחר הניצולת של ה- CPU. במידה וניצולת זו נמוכה מידי מגבירים את הרמה של ה- multiprogramming ע״י הכנסת תהליך חדש למערכת . החלפת הדפים נעשית ע ״י אלגוריתם גלובלי , בלי התחשבות בתהליכים אליהם שייכים הדפים . נניח כעת כי תהליך נכנס למצב שבו הוא זקוק לדפים נוספים . הוא מתחיל לייצר page fault, וכתוצאה לוקח דפים מתהליכים אחרים. תהליכים אלו זקוקים גם כן לדפים , ולכן גורמים ל- page fault גם כן , ולוקחים דפים מתהליכים אחרים. כל התהליכים הנ״ל שיצרו page fault משתמשים ברכיב ה-paging בכדי להוציא ולהכניס דפים. מכיוון שכל התהליכים זקוקים לרכיב , התהליכים יוצאים מתור ה- ready ועוברים לתור ה- waiting. תור ה-ready מתרוקן, וניצולת ה-CPU יורדת. מתזמן ה-CPU רואה את הירידה בניצלות ה- CPU ומכניס תהליך נוספים... לכן אסור לבחון מערכת רק ע״י ניצולת cpu.

כדי למנוע trashing עלינו לספק לתהליך את כמות ה- frame-ים לה הא זקוק . Windows ו- Solaris לא מטפלו ת במצב זה , מתוך הנחה שהוא לא אמור לקרות . Linux מעיפה תהליכים אחד אחד . טכניקה נוספת למניעה היא מודל קבוצת העבודה המתחיל בכך שהוא בודק כמה frame-ים תהליך צורך.

אלג׳ קבוצת העבודה - Working-Set Model

נגדיר לכל תהליך את מספר הדפים המקסימאלי שנשמור עבורו בזיכרון . מספר זה נקרא קבוצת עבודה ומיוצג ע״י Δ.

עבור Δ קטן, לתהליך יש מספר דפים קטן ולכן סיכוי גבוה שהוא ידפדף הרבה.

כאשר Δ שואף לאינסוף, הרי שכל התהליך בזיכרון ובעצם הרסנו את רעיון הדפדוף.

נסמן ב-D את סך כל ה-frame-ים בזיכרון. כאשר מספר זה גדול מהזיכרון האמיתי ייווצר trashing. לכן ברגע ש-D גדול מהזיכרון יש לבצע swap out לתהליך שלם.

זהו אלג' דפדוף גלובאלי.

היחס בין מספר המסגרות שהוקצו לדפדוף - ככל שמספר ה-frame-ים המוקצה לתהליך עולה, שיעור הדפדוף יורד. המטרה להימצא תמיד בשיעור דפדוף סביר הנמצא בין גבולות ה-upper bound לבין ה-lower bound.

שיקולים לבחירת גודל הדף

1. אם הדף גדול מדי, נגביר את הסיכוי ל- Internal Fragmentation, לא כל האפליקציות צריכות דפים גדולים.
2. גודל הטבלה. TLB Coverage - גודל הזיכרון שנגיש מה - TLB. זה בעצם גודל הטבלה כפול גודל הדף. באופן אידיאלי, קבוצת העבודה של כל תהליך נשמרת ב- TLB ע"מ לחסוך בגישות לזיכרון. לכן ככל שהדף גדול יותר, ניתן לכסות יותר נתונים.
3. תקורה של O/I. הגודל שמביאים מהזיכרון קבוע (נקבע ע"י הבקר). ככל הדף גדול יותר , צריך יותר גישות לזיכרון. לכן עדיף דפים קטנים.
4. נעילה של דפים. יש דפים שאי אפשר להוריד - למשל, של מעה"פ, או דפים שמשתתפים ב - O/I. עדיף דפים קטנים.
5. מקומיות - דפים גדולים מכסים יותר נתונים שקשורים אחד לשני , סביר להניח שנצטרך מספר קטן יותר של דפים חדשים. עדיף דפים גדולים.

שיקולים נוספים בבחירת אלג' דפדוף - Other Consideration

בחירת אלגוריתם הדפדוף ושיטת ההקצאה הם גורמים חשובים בהחלטה איך לממש דפדוף במערכת . אבל, קיימים שיקולים נוספים שצריך לקחת בחשבון.

שקף 9.27 עמוד 180 בחוברת מציג דוגמא לכך שיש לקחת בחשבון גם את מבנה התוכנית.

התוכנית מגדירה מטריצה בגודל kXk. דרך אחת לסרוק את המטריצה היא לרוץ לפי שורות בעמודה (הדרך שבה תמיד משתמשים...). אם נתייחס אל כל שורה כאל דף נפרד, הרי שסה"כ קראנו k דפים.

השיטה השנייה בוחרת לרוץ לפי עמודות . כלומר בכל פעם קוראים איבר מדף , ומתקבל מצב שבו קוראים את הדפים k^2 פעמים.

שיקולים נוספים : הגדרת כמות דפים מראש , בחירת גודל הדף , מקטוע, גודל הטבלה , משקלן של פעולות הקלט/פלט, סביבה.

Demand Segmentation

שיטת הסגמנטים אומרת שחלוקת מרחב הזיכרון לא נעשית בחלקים בגודל קבוע . גם כאן ניתן להעלות סגמנטים לפי הצורך. OS/2 מקצה זיכרון בסגמנטים. UNIX משתמשת בשיטה שמשלבת סגמנטים ודפים.

פרק 12 – מימוש מערכת הקבצים

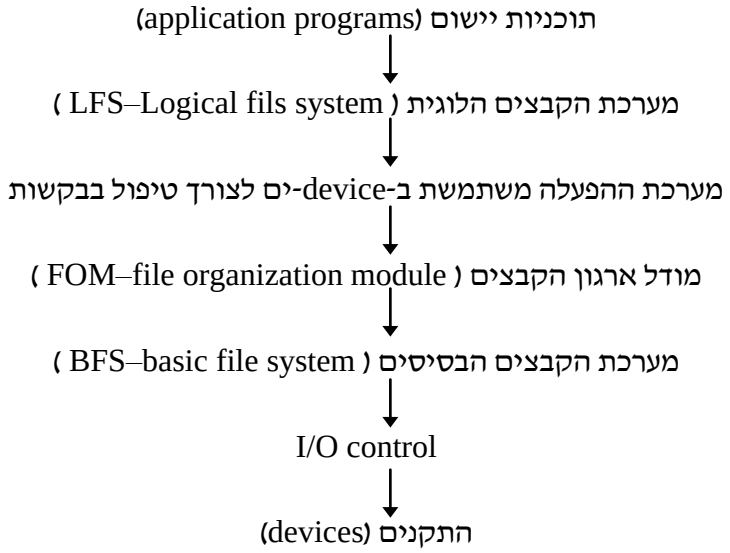
מבנה מערכת הקבצים - File-System Structure

העברת מידע בין הזיכרון לדיסק נעשית בבלוקים , במטרה לשפר את יעילות ה- I/O. כל בלוק מכיל סקטור אחד או יותר.

כדי לספק גישה נוחה ויעילה לדיסק, מערכת ההפעלה יוצרת מערכת קבצים המאפשר שמירה, מיקום ומציאה של נתונים בקלות. מערכת קבצים יוצרת שתי בעיות עיצוב:

1. כיצד צריכה להיראות מערכת הקבצים למשתמש . בעיה זו כוללת את ההגדרה של קובץ והתכונות שלו, פעולות המותרות על קובץ, ומבנה הספריות לצורך ארגון הקבצים.
2. חייבים ליצור אלגוריתמים ומבני נתונים בכדי למפות מערכות קבצים לוגיות לתוך אמצעי אחסון משני פיזי.

מערכת הקבצים מחולקת לוגית למספר שכבות:



כל רמה משתמשת ב-feature-ים של רמות נמוכות יותר ומייצרת feature -ים חדשים לרמות הגבוהות.

- I/O control – מורכבת מ- device drivers ו-interrupt handler להעברת מי דע בין הזיכרון למערכת הדיסקים. Device driver יכול להיחשב כמתרגם. הקלט שלו מורכב מפקודות בשפה גבוהה , ואילו הפלט שלו נתון ברמה נמוכה – הוראות חומרה מדויקת, בהם משתמש הבקר החומרה.

Basic file system – מוציא פקודות להתקנים לקרוא ולכתוב בלוקים (רמה פיזית).

File-organization module – מכיר גם רמה לוגית ויכול לתרגמה לרמה הפיזית עבור ה-BFS. נעזר גם ב- free space manager כדי למצוא מקומות ריקים בדיסק.

Logical file system – משתמש ב-directory structure כדי לספק ל-FOM את המידע לו הוא זקוק , בהינתן שם קובץ. אחראי גם להגן על הקבצים.

תהליך יצירת קובץ חדש:

כדי ליצור קובץ חדש , תוכנית היישום קוראת ל- LFS. הנ"ל קורא את ה- directory המתאים לזיכרון, מעדכן אותו ורושם אותו חזרה לדיסק . מערכות הפעלה מסוימות (לדוגמא Unix) מתייחסות לספרייה כמו אל קובץ . באחרות קיימים system calls נפרדים לקבצים וספריות.

קובץ צריך להיות פתוח בכדי שנוכל לבצע עליו פעולות I/O. כשקובץ נפתח מחפשים ב- directory structure את הרשומה שלו . חלקים מה- directory structure מועברים ל- cache כדי להאיץ את המהירות . כשקובץ

מאותר מעבירים מידע לגביו לטבלת הקבצים הפתוחים שנמצאת בזיכרון. האינדקס בטבלה מוחזר לתוכנית, ומאז כל ההתייחסות נעשית דרך הטבלה בעזרת האינדקס.

האינדקס מכונה ב- file descriptor Unix וב-NT: file handle. במערכות אחרות הוא מכונה file control block.

כשקובץ נסגר המידע לגביו מועתק מהטבלה ל-directory structure שבדיסק.

כאשר טוענים קובץ לזיכרון לא טוענים את כל הקובץ אלא רק בלוקים בכל שלב.

מערכות הפעלה שונות מאפשרות גדלים שונים של קבצים . ייתכן שמערכת הפעלה תאפשר קובץ הגודל שהינו גודל מהדיסק.

שיטות הקצאה של מקום בדיסק לקבצים - Allocation Methods

1) הקצאה רציפה – Contiguous Allocation

הקבצים מוקצים ברצף בדיסק. לכל קובץ מקצים מספר בלוקים. הנתונים נשמרים בבלוקים עוקבים. בשימוש בד"כ בטייפים, CD וכו'. בכל קובץ ישנו Header המכיל את גודל הקובץ , וכן באיזה סקטור פי זית הקובץ יושב. בצורה כזאת מאוד פשוט לאתר את הקובץ וכן בלוקים בתוך הקובץ.

יתרונות השיטה:

- פשוט ביותר למימוש ומהיר מאוד.
- Random access - ניתן לגשת למקום באמצע הקובץ ישירות

חסרונות השיטה:

- יש לדעת את גודל הקובץ מראש.
- ניצולת גרועה של הדיסק - כאשר רוצים להגדיל את הקובץ, ייתכן שנצטרך להעביר את הקובץ למקום חדש. כתוצאה מכך עשויים להיווצר חללים בדיסק (External Fragmentation). הפתרון הוא ע"י הקצאת מקום גדול יותר מהנדרש – כלומר מבזבזים מקום (Internal Fragmentation).

איך מוצאים שטח בדיסק

- Best Fit – מציאת החלל הקטן ביותר שיכול להתאים לגודל הקובץ.
- Worst Fit – מציאת החלל הגדול ביותר בדיסק . נשתמש בשטח זה כאשר נצפה לגידול משמעותי בקובץ (כשמו כן הוא - נותן תוצאות גרועות לטווח ארוך).
- First Fit – מציאת החלל הראשון בדיסק שיכול להתאים.

2) הקצאה משורשרת – Linked Allocation

כל קובץ הוא רשימה מקושרת של בלוקים , כאשר הבלוקים אינם יושבים בצורה רציפה בדיסק . כל בלוק מצביע לבלוק הבא (בהנחה שכל כתובת היא 4 בתים, אזי הבלוק מכיל 4 בתים ככתובת לבלוק הבא).

ה-directory מחזיק את הבלוק הראשון והאחרון של הקובץ.

יתרונות השיטה:

- הגדלת הקובץ לא דורשת הזזות.
- אין חללים בדיסק.

חסרונות השיטה:

- בזבוז – בכל בלוק יש 4 בתים שלא קשורים לנתוני הקובץ (שומרים את הכתובת הבאה).
- זמן גישה עצום: (כאשר כל בלוק במקום אחר, הראש של הדיסק כל הזמן זז).
- אין Random access.

- כאשר בלוק אחד הולך לאיבוד מאבדים את כל הקובץ.
- בבלוק האחרון עלול להיות Internal fragmentation.

שיפור של השיטה – מחזיקים את שרשרת המצביעים בנפרד – **File Allocation Table – FAT**

הטבלה מחזיקה צילום של הדיסק. לכל בלוק בדיסק תהייה כניסה בטבלה, והכניסה תכיל את כתובת הבלוק הבא בתור. כניסות שלא שייכות לאף קובץ יצביעו על אפס או -1. Eof יציין בלוק אחרון בקובץ. הטבלה נשמרת כמערך בזיכרון.

מה השיפור? חוסכים בגישה ישירה – מחפשים את הבלוק בטבלה ואז מבצעים גישה ישירה אל הבלוק הדרוש.

3) Indexed Allocation

מחזיקים אינדקסים שאומרים לי איפה נמצא הקובץ. הבלוק הראשון מחולק לקבוצות של 4 בתים, כאשר כל 4 בתים הם הפנייה לבלוק מסוים בדיסק. כלומר, הבלוק הראשון מצביע לכל יתר הבלוקים. בשימוש ב – NTFS, UNIX.

יתרונות השיטה:

- תמיכה בגישה ישירה (כאשר הדבר לא היה ניתן בהקצאה משורשרת כאשר אין לי FAT).

חסרונות השיטה:

- בזבזני עבור קבצים קטנים (עדיין יש צורך בבלוק לטבלה)
- ניגשים פעמיים – קודם לטבלה ואח״כ לנתונים
- מה קורה אם הקובץ גדול והטבלה לא מספיקה?

Two-level index

כל בלוק הוא 4K, לכן הבלוק מכיל K כניסות. כל כניסה מצביעה על בלוק נוסף של 4K ולכן גודל הקובץ המקסימאלי במקרה זה הוא 4MB (K*4K).

פתרון – הכתובת האחרונה בבלוק הראשון תצביע לבלוק אינדקסים נוסף, וכעת נקבל קובץ של 8MB.

פתרון נוסף – הבלוק הראשון מפנה ל- K בלוקים שכל אחד מהם הוא בלוק אינדקסים, כלומר מפנה ל- K בלוקים אחרים. גודל מקסימאלי 4G.

מערכת הקבצים של UNIX - UNIX File System

1. מקצים את בלוקי ההפניה במרכז הדיסק, כדי שהגישה לכל כיוון תהיה מהירה.
2. (שקף 12.10) – שימוש ב-inode. קבוצת הכתובות הראשונה בבלוק מפנות לבלוק אחד בלבד המצביע ישירות לנתונים (direct block). אם הקובץ הוא עד 40k הגישה אליו מיידית.
   הכניסה הבאה מיועדת ל- single indirect והיא מצביעה לבלוק הפניות נוסף, כאשר בלוק זה מצביע ישירות לנתונים.

   הכניסה הבאה נקראת double indirect והיא מצביעה לבלוק הפניות נוסף בו כל כניסה מצביעה לבלוק הפניות ורק משם מגיעים לנתונים. מתאים לקובץ בגודל 4G.

   INode – מבנה המכיל מידע על הקובץ כמו – סוג הקובץ (file, directory, character device, block device, pipe, socket) הרשאות ועוד. לא מכיל את שם הקובץ.

יתרונות:

- יעיל גם בקבצים קטנים.
- ניתן להגדיל את הקבצים בלי בעיה.

חסרונות:

- הרבה תנועות ראש בקבצים גדולים.
- קיים חסם עליון לגודל מקסימלי של קובץ, כי מספר הפוינטרים הוא סופי.

סיכום מושגים

- **Fragmentation - חללים שנוצרים בדיסק עקב כתיבה של קבצים בעזרת הקצאות רציפות.**
  - External Fragmentation - חללים בדיסק בין הקבצים שהם קטנים מידי לקבצים חדשים.
  - Internal Fragmentation - חללים שנמצאים בבלוק האחרון שהוקצה לקובץ.
- **Defrag – "ביטול" החללים שנוצרו בדיסק. מעבירים את כל הבלוקים ברצף אחד אחרי השני.**

במה נבחר?

אם ידוע שהגישה לקובץ תהיה ישירה /אקראית עדיף להשתמש בהקצאה רציפה או משורשרת עם אינדקסים . אחרת, נשתמש בהקצאה משורשרת.

ניהול השטח הריק בדיסק - Free-Space Management

כיצד נמצא מקומות פנויים בדיסק:

1. מערך ביטים (שקף 12.11) – נחזיק ווקטור בלוקים פנויים מ- 0 עד n-1. כל בלוק מיוצג ע"י ביט אחד בלבד. אם הבלוק תפוס יופיע הביט 1, אחרת 0. כאשר רוצים להגדיל קובץ , מחפשים בלוק שהביט שלו 0 ומוסיפים לקובץ שלי.

   חסרון: בזבוז – מתקבל ווקטור ענק שתמיד נשמר בזיכרון, גם כאשר אין אף בלוק פנוי.

   יתרון: אם רוצים להקצות בלוקים רצופים (cluster) ז"א קבצים ברצף, ואם נניח שרוצים 5 בלוקים אז נחפש 5 אפסים רצופים במערך ושם נקצה.

2. רשימה מקושרת (שקף 11.12) – רשימה מקושרת של בלוקים פנויים . בלוק שמתפנה מתווסף לרשימה ולהפך. אפשרות יעילה יותר היא שכל בלוק פנוי מצביע לבלוק הפנוי הבא מיד אחריו, ואז אם צריכים X בלוקים פנויים ניגשים לאחד ומוצאים את היתר. שיטה זו לא תתאים כאשר עובדים על cluster-ים.

   חסרון: לא יעיל – איטי (הרבה תנועות ראש).
   יתרון: חסכון במקום.

מאגר זמני - Buffer Cache

אזור בזיכרון בו נמצאים הבלוקים האחרונים שהשתמשנו בהם (לפי תדירות), מתוך הנחה ש נשוב ונשתמש בהם. בצורה כזאת אין צורך לשלוף אותם שוב מהדיסק.

זהו רכיב חשמלי, ולכן מהירות הגישה גבוהה בהרבה.

מעה"פ לא כותבת ישירות לדיסק . היא כותבת קודם למאגר ורק אח"כ, כשהוא מתמלא, היא מעבירה את ה-buffer לדיסק. לכן, אם המחשב נכבה בצורה לא מסודרת יכול להיות שהדיסק לא יהיה אמין בגלל שהנתונים לא נכתבו.

Recovery – אמינות

מושג זה מתייחס לעד כמה מערכת ההפעלה מסוגלת להתאושש במצב של נפילה.

ברוב המקרים הזיכרון יותר מעודכן (כי עליו עבדנו).

1. Consistency checker – השוואה בין המידע בזיכרון לבין המי דע בדיסק (ב-unix נעשה ע״י הפקודה fsck).
2. גיבויים – ע״י שימוש ב-system programs והתקנים פיזיים אחרים.

אמינות באמצעות Logging

מחזיקים קובץ log שבו נכתבים כל העדכונים לפני שהם מבוצעים בדיסק . יתרון: במקרה של נפילה ניתן לדעת מה נשמר לדיסק ומה לא . חסרון: הכפלת מספר הכתיבות .(עדכונים שבוצעו נמחקים מה – log, כך שמכיל רק את מה שהיה צריך להיות מבוצע ועדיין לא בוצע).

חיבור מערכות קבצים - File System Mounting

ניהול שטח החלפת זיכרון בדיסק - Swap-Space Management

כאשר עובדים במערכת עובדים תמיד מול הזיכרון והדיסק . לעיתים קורה שכמות הזיכרון לא מספיקה למערכת ההפעלה . הפתרון הוא להעביר חלקים שלמים מהזיכרון החוצה (כלומר לדיסק) ולהפך. רצוי שהאזורים בדיסק אליהם כותבים יהיו באותו מקום , ואם אפשר אף בדיסק נפרד (דיסק swap) בכדי שהפעולה תהיה מהירה יותר.

אזור ההחלפה יכול להיות כחלק כק ובץ במערכת הקבצים הרגילה או כמו יותר שכיח כ- partition מופרד מיוחד לפעילות זו.

UNIX מסוג 4.3BSD מוצא ומארגן מקום לאזור ההחלפה כשהתהליכים מתחילים – כלומר כשהמערכת עולה.

הגרעין משתמש במפות מיוחדות – swap maps – לעקוב אחרי השימוש בשטח המדובר.

Solaris 2 – מוצא ומארגן את שטח ההחלפה רק כשדף נדחף החוצה מהזיכרון הפיסי . לא כשדף הזיכרון הווירטואלי נוצר.

מערכת קבצים רשתית – UNIX Network File System

שרות ספריות – LDAP

מסד נתונים היררכי ששומר מידע ברשת . בד״כ שומרים בו מידע על משתמשים , מחשבים מדפסות וכו׳. יש Demon מיוחד שאחראי על הפניות ל - LDAP. כל פעם שיש שינוי ה - Demon מקמפל את הנתונים מחדש.

DHCP

פרוטוקול שקובע דינאמית כתובות IP למחשבי הארגון. ברגע שמחברים מחשב לרשת צריך לתת לו IP. לנהל רשימות פנימיות של ה - IP's כדי לדעת מה פנוי די מסובך וגורם בד ״כ להתנגשויו ת. DHCP זה שרת IP שמחזיק רשימה של כתובות ה - IP לפי ה- Mac Address של מחשבי הארגון . כאשר מוסיפים מחשב חדש , מוקצת לו כתובת IP באופן אוטומטי.

SAMBA

פרוטוקול (שיטה סטנדרטית) לתקשורת בין מערכות . כל מערכת קבצים עובדת בשיטה שונה , לכל אחד בלוק בגודל שונה, נתונים שונים על הבלוק וכו׳, ולכן יש צורך בשיטה אחידה. בעזרת פרוטוקול זה יכולים משתמשי Windows לראות קבצים ומדפסות של משתמשי UNIX ולהפך.

פרק 14 – מבנה הזיכרון המשני (דיסק)

מבנה הדיסק - Disk Structure

- פונים לכונני דיסקטים כמערכי ם גדולים של logical blocks, כשה-logical block הוא היחידה הקטנה ביותר שתועבר.
- המערך החד מימדי של logical blocks ממופה אל הסקטורים של הדיסק באופן רציף.
- סקטור 0 הוא הסקטור הראשון של ה-track הראשון על הצילינדר החיצוני ביותר.
- המיפוי מתקדם בסדר מה- track הזה, ואז לשאר ה track -ים בצילינדר הזה , ואז דרך שאר הצילינדרים מחיצוני ביותר לפנימי ביותר.

תזמון הדיסק – שזמן הגישה יהיה מהיר - Disk Scheduling

הדיסק מכני, ולכן הוא פועל באיטיות יחסית. אנחנו מחפשים את זמן הגישה המהיר ביותר לדיסק. זמן הגישה מושפע משני גורמים:

1. Seek Time – הזמן שלוקח לזרוע הדיסק להזיז את הראש קורא/כותב בין הצילינדרים.
2. Rotational latency – זמן סיבוב הדיסק עד שהסקטור המבוקש נמצא מול הראש קורא/כותב.

מסקנה: על מנת לחסוך זמן seek וזמן רוטציה, כדאי שהסקטור הבא ימצא על אותו track או לפחות על אותו צילינדר. לכן, עדיף לשרת בקשות מאזור אחד ורק אח"כ לעבור לאזור אחר.

Disk bandwidth – רוחב הפס של הדיסק = מספר הבתים שעובדים לחלק לזמן שעבר מרגע שביקשנו את המידע ועד לרגע שקיבלנו אותו. (זהו מספר הבתים שהדיסק משדר ביחידת זמן, נרצה למקסם אותו)

אלג' שונים לקריאת מידע מדיסק

1. FCFS - First come First Served - (שקף 14.4) קבלת מידע סדרתית. מטפלים בבקשות לפי הסדר שהם הגיעו . עפ"י הדוגמא ב שקף נקבל 640 תזוזות של הראש קורא /כותב (כדי להעריך את זמן העבודה של הראש, נספור כמה צילינדרים הוא עבר בדרך מהבקשה הראשונה ועד לאחרונה).
2. SSTF - Shortest Seek Time First – (שקף 14.5/6) מטפלים בבקשה עם seek time מינימאלי. עפ"י הדוגמא בשקף נקבל 236 תזוזות של הראש קורא/כותב.

חסרונות:

- על כל בקשה שמגיעה צריך לה כניס אותה לרשימה בצורה ממוינת . **אבל**, מיון הרשימה מהיר יחסית לתנועת הראש.
- הרעבה – Starvation – הרעבה תיתכן כאשר יש process שדורש מידע מצילינדר מאוד מרוחק, וכל הזמן נכנסות לי בקשות לצילינדרים קרובים . אז לא ניתן להעריך מתי התהליך יקבל את המידע.

למרות אפשרות ההרעבה , זו השיטה שבה כמעט כל המערכות עובדות (Windows, Unix ועוד). מדוע? מבחינה הסתברותית הסיכוי שתתרחש הרעבה הוא נמוך מאוד , בד"כ הדיסק לא מבצע עבודה (אפשר לראות זאת במחשב הביתי – רוב הזמן הנורה של הדיסק כבויה).

3. SCAN (שקף 14.7/8) – זרוע הדיסק נעה מהנקודה בה היא נמצאת לכיוון ההתחלה ומשם אל הסוף . בכל פעם שעוברים דרך צילינדר בודקים האם צריך לקרוא אותו ואם כן קוראים . פתרנו את בעיית ההרעבה. בדוגמא שבספר מתקבלות 208 תזוזות.

בעיות:

1. ייקח יותר זמן לקבל מידע שנמצא בקצוות - נניח שאני בצילינדר 1 ומתחילה לנוע הלאה ובדיוק מגיעה בקשה לצילינדר 1. אומנם לא תהיה הרעבה אבל יהיה צורך לעבור הלוך חזור כדי להגיע אליו חזרה. השיטה דוגלת באחידות אבל לא ביעילות.

2. הראש נע מהצילינדר ה-0 ועד הצילינדר 199, כאשר ייתכן והמידע הדרוש מצוי בין צילינדרים 50-100 (אין טעם להגיע לקצה, תזוזה מיותרת).

4. C-SCAN (שקף 14.9/10) – שיפור השיטה SCAN: הראש נע מהמיקום הנוכחי אל הצילינדר האחרון, וכאשר מגיעים אליו , מזיזים את הראש ישירות לצילינדר ה- 0, ומשם שוב הולכים לסוף . כלומר מבצעים קריאה מצילי נדר רק כאשר מתקדמים קדימה . אלגוריתם זה פותר את בעיה מס ' 1 של השיטה SCAN.
5. C-LOOK – שיטה זו דומה ל- C-SCAN, אלא שהתנועה היא בין הצילינדר הימני ביותר המבוקש (ולא המינימאלי בדיסק) לבין השמאלי ביותר. פתרון בעיה 2 של השיטה SCAN.

במה נבחר?

כאמור, SSTF היא השכיחה ביותר . אבל SCAN ו C-SCAN – מתפקדות טוב יותר למערכות שמעמיסות הרבה על הדיסק. (כשהוא כן עובד רוב הזמן)

הביצועים של האלגוריתמים תלויים במספר וטיב הבקשות . בנוסף, בקשות לשירותי הדיסק יכולות להיות מושפעות משיטת ארגון הקבצים.

(אלג' תזמון הדיסק צריך להיות כתוב כמודל נפרד ממערכת ההפעלה כך שיהיה אפשר להחליפו במידה וזה יהיה נחוץ)

ניהול/מבנה הדיסק - Disk Management (13.15)

1. Low level formatting – חלוקת הדיסק לסקטורים שבקר הדיסק יכול לקרוא או לכתוב.
2. High level formatting – קביעת מערכת הקבצים של הדיסק:
   - חלוקת הדיסק למחיצות - Partition – קבוצה אחת או יותר של צילינדרים.
   - פירמוט לוגי – "הכנת מערכת קבצים"
3. מערכת אתחול ה boot block, שתי אופציות:
   - ה- bootstrap מאוכסן ב rom
   - תוכנת bootstrap loader –

פיזור מידע – Data Striping

הדיסק הוא יחידה עם מנגנונים מכאניים, ועם חלקים נעים. כלומר ייתכנו הרבה נפילות. נפילת דיסק גורמת לאובדן מידע רב. השחזור לוקח זמן רב ולא תמיד אפשרי בשלמות . בד"כ אין חלוקה שווה של העומסים בין הדיסקים במחשבים ארגוניים (יש אחד שטוחנים אותו ואחד שפונים אליו לעיתים רחוקות).

הרעיון של data stripping הוא לחלק את הקובץ לכמה דיסקים פיזיים (מתייחסים לקבוצה של דיסקים כאל יחידה אחת). ככה יש איזון בעבודה בין הדיסקים , ואין אחד שהופך להיות צוואר הבקבוק . הזמן הנדרש להעברת בלוק לזיכרון משתפר בצורה משמעותית, משום שכל הדיסקים מעבירים את הבלוקים שלהם בצורה מקבילית.

יתרון:
הרבה דיסקים קטנים וזולים במקום דיסק אחד גדול ויקר.

חסרון:
העברת הרבה יחידות קטנות במקביל – גישה איטית יותר . פתרון עשוי להיות ע "י העברת יחידה גדולה במקביל לזיכרון אם יש דיסקים מסונכרנים).

RAID - קבוצת שיטות ארגון דיסק להגדלת אמינות דיסקים

רוצים שכאשר דיסק ייפול , יהיה אפשר להמשיך לעבוד . עושים זאת ע״י עבודה עם כמה דיסקים במקביל . ארגון מסוג זה נקרא בד״כ Redundant array of inexpensive disks. בנוסף, שיטות RAID רבות שיפרו את האמינות ע״י כפילות (redundancy) נתונים.

RAID level 0 – מחלקים את המידע ל מערך של דיסקים (data stripping). אין גיבויים, נניח 4 דיסקים, על כל אחד מידע שונה , המידע נכתב על כולם במקביל . כאן אין כפל מידע , ואי אפשר לשחזר את הנתונים אוטומטית. אבל משפר את האמינות של הדיסקים.

RAID level 1 – נקרא גם mirroring אן shadowing. תמיד יהיו 2 עותקים מכל דבר . כל מידע נכתב גם לעותק. במקרה של נפילה תמיד ניתן לשח זר מהעותק. לשיטה זו אמינות גבוהה, אבל צורכת מספר כפול של דיסקים. בנוסף, פעולת כתיבה לוקחת זמן כפול.

RAID level 2 – memory style error correcting code. שימוש בקוד המתקן שגיאות ברמת הבתים.

יתרונות: מאוד אמין, פחות דיסקים מהגישה הקודמת

חסרונות: עבור I/O גדול זה כמו level 1 ועבור I/O קטן מאוד גרוע, צריך לקרוא את כל הדיסקים.

RAID level 3 – Byte-interleaved parity. קיים דיסק parity יחיד המתקן שגיא ות. כל ביט בדיסק מיוע ד לכל יתר הדיסקים. בזמן הכתיבה מחשבים את הזוגיות של הביטים הנמצאים בכל הדיסקים באותה הכתובת, וכותבים את bit הזוגיות באותה כתובת בדיסק הנוסף. ברגע שדיסק אחד הלך או בלוק אחד הלך ניתן לשחזור אותו עפ״י ה-parity bit (כמובן בהתייחסות לשאר הדיסקים בנוגע לאותו ביט).

יתרונות: אמינות גבוהה, דיסק בדיקה יחיד (פחות מ level 2)

חסרונות: כתיבה דורשת גישה לכל הדיסקים.

RAID level 4 – Block-interleaved parity. הפעם מקצים **בלוק** לכל דיסק **ולא ביט**. בכל כתיבה מחשבים בלוק parity ולא ביט בודד.

יתרונות: אמינות עדיין גבוהה , דיסק בדיקה י חיד (פחות מ level 2), כתיבות גדולות שניגשות לכל הדיסקים פועלות כראוי . כתיבות קטנות יותר בסדר כי ניתן לבצע I/O במקביל (וכן לקרוא רק את הערך הקודם מהדיסק אליו קוראים).

חסרונות: דיסק הבדיקה הינו צוואר הבקבוק.

RAID level 5 – Block-interleaved distributed parity. בלוק ה-parity נשמר בכל ה דיסקים ולא בדיסק בודד. פותר את הבעיה של level 4 - דיסק הבדיקה אינו צוואר הבקבוק יותר.

ניהול הכפילות - RAID Management

ניתן לנהל RAID דרך מעה״פ או דרך חומרה . היום RAID זה סטנדרט של חומרה , ולא עובדים עם תוכנה ייעודית. מדוע לא לנהל את ה- RAID דרך מעה״פ?

1. ירידה בביצועים – עוד מישהו יתחרה על ה – CPU
2. Fault tolerance – כאשר יש כשל בחומרה אפשר פשוט להחליף אותה . כאשר מעה״פ מנהלת את ה – RAID, יש צורך לשמור חלק בדיסק שלא ינוהל כ – RAID ובו תשב מעה״פ, כדי שנוכל להפעיל אותה גם במקרה של כשל.
3. Hot Swapping – אם ה – RAID מנוהל ע״י החומרה ניתן פשוט להוציא דיסק ולשים אחר תוך כדי עבודה.

SCSY vs. IDE

אלו הם 2 דרכים בהם המחשב מתקשר עם ה – devices שלו.

SCSY

צריך בקר לעצמו . בד״כ בא כאשר קונים את הכרטיס . הכרטיס מסוגל לדבר עם כמה התקנים באותו זמן . מנהל תור פניות בעצמו. (למשל – בזמן שיש עבודת הדפסה ממתינה, הוא יכול לקבל פניות חדשות).

היום מדובר על 32bit רוחב פס

IDE

הרבה יותר פשטני ומקובל ב- PC היום. אין חוכמה – אין כרטיס שיודע להפעיל אלגוריתם . מעה״פ צריכה לטפל בנושאים כמו – מציאת המקום הקרוב ביותר וכו׳.

היום מדובר על 16bit רוחב פס

יתרונות ל – SCSY

1. מהיר יותר (גם בגלל רוחב פס וגם בגלל עיבוד מקבילי)
2. אין צורך בחישובים של מעה״פ ולכן פחות עומס על ה- CPU
3. SCSY מסוגל לטפל ב- 256 בקשות, IDE משאיר את הטיפול בתור למעה״פ
4. SCSY תוכנן לעמוד בטמפרטורות גבוהות יותר

חסרון:

SCSY יותר יקר.

אינדקס

Directory Structure 54
Disk Reliability 62
disk stripping 62
dispatch latency 24
dispatcher 44 ,24
Dispatcher 23
DMA 12
Dual-Mode operation 13
Dynamic Linking 42
Dynamic Loading 42
Dynamic Storage-Allocation Problem 45

E

entry section 29
Execution time 42
External Fragmentation 46 ,45

F

FCB 54
FCFS 60 ,27 ,26 ,25 ,24
File Concept 54
File-System Structure 55
First Fit 56 ,45
Fragmentation 58 ,56 ,46 ,45
frames 46

H

Hardware Protection 13
Hold and Wait 40
Hole 45

I

I/O bound process 17
I/O waiting queue 17
Independent process 19
input queue 42
Internal Fragmentation 56 ,45
Interrupt 17 ,12 ,11
Interrupt Vector 12
Inverted Page Table 47

A

Access Method 54
Aging 26

B

backing store 44
Bakery Algorithm 32
Best Fit 56 ,45
Binary Semaphores 36
Bounded Waiting 30
Burst Cycle 23
Burst time 25 ,24

C

Circular wait 40
Classical Problems Of Synchronization 37
C-LOO 61
Common functions of Interrupts 12
Compaction 46
Compile time 42
context switch 44 ,27 ,23 ,18
Context Switch 54
Contiguous Allocation 56 ,44
convoy effect 25
Cooperating process 19
Cooperating Processes 19
Counting Algorithm 51
counting semaphore 36
CPU bound process 18
CPU Scheduler 23
CPU-I/O Burst Cycle 23
critical section 34 ,33 ,32 ,31 ,30 ,29
C-SCAN 61

D

Deadlock Characterization 41 ,40
Deadlocks 41 ,40 ,35
Demand Paging 49
device controller 12
Device status table 12
Dining-Philosophers Problem 38
Direct Memory Access 12

Priority Scheduling 26
priority-based scheduling 44
Process Concept 16
Process Scheduling 17
Progress 30
Protection 14, 54
PTBR 47
PTLR 47

Q

quantum 26, 27

R

RAID 62
Ready queue 17
redundancy 62
relocation register 43, 44
remainder section 29
roll out, roll in 44
Rotational latency 13
Round-Robin Scheduling 26

S

SCAN 61
Scheduling Algorithms 24
Scheduling Criteria 24
Seek time 13
segment table 48
segment table base register 48
segment table length register 48
Segmentation 48
Segmentation With Paging 48
Semaphores 33
Shared Pages 48
Short term scheduler 17
Shortest Seek Time Fi 60
short-term scheduler 23
Simple Batch Systems 8
Single-Partition Allocation 44
SJF 25, 26
Spooling 8
SRTF 25
SSTF 60
STBR 48
STLR 48
stub 42
swap in 44

J

Job queue 17

L

limit register 17, 44
Load time 42
Logical Versus Physical Address Space 43
Long term scheduler 17
LOOK 61
LRU Approximation Algorithm 51

M

Memory Management Unit 43
MMU 43, 44
Multilevel Feedback Queue 27
Multilevel Queue 27
Multiple-Partition Allocation 45
Multiprocessing 8
Multiprogramming 8
Multiprogramming Batch Systems 8
Multitasking 8
Mutual exclusion 30, 40
mutually exclusive 29

N

No preemption 40
non-preemptive 23, 25, 26

O

Operation On Processes 18
Other Consideration 53
Overlays 43

P

Page Fault 49, 53
Page Replacement 50
Page Replacement Algorithms 50
page table base register 47
page table length register 47
pages 46
Paging 46, 48, 49
PCB 16, 17, 24, 47, 49
Positioning time 13
preemptive 23, 25, 26, 27

swap out ..... 52 ,44
Swapping ..... 44 ,18
system call ..... 40 ,19 ,18
System Calls 15
Systems Components 15

T

The Critical-Section Problem 29
Thrashing 52
Thread 20
Throughput 24
TLBS 47
Transfer Rate 13
translation look-aside buffers 47
Trap ..... 14 ,12
Turnaround 24
Two-Level Page-Table Scheme 47

U

Unix ..... 57 ,56 ,55 ,54 ,41 ,19
user mode ..... 23 ,18 ,14

W

Working-Set Model 52
Worst Fit ..... 56 ,45

א

אבטחה של קבצים 54
אלגוריתם ההזדמנות השנייה או אלגוריתם השעון 51
אלגוריתמים לדפדוף 50
אלגוריתמים שונים לתזמון 24

ב

בלוק שליטה על 16
בעיות סנכרון שונות 37
בעיות של קריאה וכתיבה במקביל 37
בעיית היצרן-צרכן 19
בעיית הפילוסופים הסועדים 38
בעיית קטע הקוד הקריטי 29

ה

האלגוריתם של בארקלי 32
הבעיה עם הקצאת זיכרון באופן דינאמי 45
החלפות זיכרון ..... 44 ,18
החלפת דפים ע 49
החלפת תוכן 18
היררכית האכסון 13

הַלְקָאָה 52
המשגר 23
הקצאה רציפה ..... 56 ,44
הַקְצָאָה של מסגרות 51
הקצאת רצף זיכרון 44
הרעבה ..... 61 ,60 ,38 ,36 ,30 ,28 ,27 ,26 ,25
התפרצויות 23

ז

זיכרון וירטואלי 49
זמן המתנה ..... 25 ,24
זמן תגובה ..... 27 ,24

ח

חומרת סנכרון 33

ט

טבלת דפים בעלת שתי רמות 47
טעינת רוטינות דינאמית 42

י

יצירת תהליך 18

כ

כיצד ומתי נדפדף 50
כתובת לוגית ..... 48 ,47 ,46 ,44 ,43
כתובת לוגית מול פיסית 43
כתובת פיזית 43

מ

מבט מופשט על מערכת ההפעלה 7
מבנה אמצעי האחסון 12
מבנה הגישה המיידית לזיכרון 12
מבנה הדיסק ..... 61 ,60 ,13
מבנה הזיכרון המשני 60
מבנה מערכות מחשב 11
מבנה מערכת ההפעלה 15
מבנה מערכת הקבצים 55
מבנה ספריות הקבצים - 54
מה משפיע על מהירות המחשב 13
מימוש טבלת הדפים 47
מימוש מערכת הקבצים 55
ממשק מערכת הקבצים 54
מניעה 41
מערכות אצווה בסיסיות 8
מערכות אצווה מרובות תוכניות 8
מערכות מבוזרות 10
מערכת הפעלה ..... 44 ,12 ,11 ,9 ,7
מצבי מערכת ההפעלה 7
מתזמן המעבד 23

נ

ניהול זיכרון 8, 17, 42, 48

ס

סָגְמֶנְטַצְיָה 48
סוגים שונים של מערכות מרובות תוכניות 9
סיום תהליך 18, 21
סיכום מושגים 58
סנכרון תהליכים 29

פ

פונקציות נפוצות של פסיקות 12
פְּלִוּחַ 48
פניות למערכת ההפעלה ע 15
פעולות על תהליכים 18
פעולת מערכת המחשב 11
פתרון הַלָקָאָה ע 52
פתרון קִטּוּעַ ע 46
פתרונות לשני תהליכים 30

ק

קִטּוּעַ 45
קטע קריטי 36
קיפאון 30, 36, 40, 41
קישור רוטינות דינאמי 42
קְפָּאוֹן 35, 40
קְפָּאוֹן והרעבה 35
קריטריונים לתזמון - 24

ר

רכיב חומרה לניהול זיכרון 43
רכיבי מערכת ההפעלה השונים 15

ש

שגיאת דף 49
שיברור 45
שיטות לטיפול בקְפָּאוֹן 41
שיטות פנייה לקבצים 54
שיטות שונות לקריאות מידע מדיסק 60
שיטת העברת הודעות באמצעות Buffering 20
שיטת העברת הודעות מסוג 20
שיטת לשיפור הביצועים 59
שילוב דפים וסגמנטים ביחד 48
שימוש בטבלת דפים מאוחדת לכל התהליכים 47
שיקולים נוספים בבחירת אלג' דפדוף 53
שיתוף דפים 48

ת

תאור מופשט של קובץ 54
תהליך הטיפול ב-Interrupt 12
תהליך הטיפול בפסיקה 12
תהליכים 8, 15, 16, 17, 19, 20, 21, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 34, 35, 36, 40, 43, 45, 48, 49, 52
תהליכים שמשתפים פעולה 19
תזמון תהליכים 17
תכנות עם שימוש בשכבות 43
תנאים למצב הקְפָּאוֹן 40
תפיסת רעיון ה 16
תפקידי מערכת ההפעלה 7
תרשים זרימה של חיי התהליך 16