תרגיל 4 OS.

<u>חלק א'</u>

שאלה 1א :

P #	1	2	1	3	4	1	2	5	4	1	4	1	4	#
ST	0	2	4	6	8	10	12	13	14	16	18	20	22	23

סיום ריצה של תהליך מסומנת כך

נחשב את הפרמטרים הדרושים

Waiting : P1 = (22-10) = 12 P2 = (13 – 3 – 2) = 8 , P3 = (8 – 2 – 4) = 2 , P4 = (23-4-7) = 12

P5 = (13 – 7 – 1) = 5, ttl = 12+8+2+12+5= 39

Util time = 23/ (23+13cs)

AVG TurnAround = (39+23+13cs) / 5 = 12.4

AVG Wait = (39+13cs)/5 = 7.8

Throughput = 5 / (23+13cs)

נריץ FCFS.

P	1	2	3	4	5
ET	10	13	15	22	23

Util = 23 / (23+4cs)

Ttl Waiting = 0+8+9+11+15+4cs = 43

Avg TurnAround = (ttlWaiting + 23) / 5 = 13.2

Avg Waiting = (43 + 4cs) /5 = 8.6

Throughput = 5 / (23+4cs)

נריץ SRTF.

P	1	5	3	2	4
ET	10	11	13	16	23

Util = 23 / (23+4cs)

Ttl Waiting = 0+8+7+9+9+4cs = 33

Avg TurnAround = (ttlWaiting + 23) / 5 = 11.2

Avg Waiting = (33 + 4cs) /5 = 6.6

Throughput = 5 / (23+4cs)

נריץ Priority Schedueling without preemption.

P	1	2	4	3	5
ET	10	13	20	22	23

Util = 23 / (23+4cs)

Ttl Waiting = 0+8++9+16+154cs = 48

Avg TurnAround = (ttlWaiting + 23) / 5 = 14.2

Avg Waiting = (48+ 4cs) /5 = 9.6

Throughput = 5 / (23+4cs)

נריץ Priority Schedueling with preemption.

P	1	2	4	3	1	5
ST	0	2	5	12	14	22
ET	2	5	12	14	22	23

Util = 23 / (23+4cs)

Ttl Waiting = 12 + 0 + 1 + 8 + 15 = 36

Avg TurnAround = (ttlWaiting + 23) / 5 = 11.8

Avg Waiting = (36+ 4cs) /5 = 7.2

Throughput = 5 / (23+4cs)

2- לא בהכרח, התהליך שבו אנחנו "מתחזקים" את הcached blocks שלנו הוא תהליך שלוקח זמן מסויים, נוכל להסתכל על תוכנית שקוראת לכל בלוק מהזכרון פעם אחת (לדוגמא אחת כזו שמאחדת את כל מסמכי הטקסט שיש לנו במחשב. תוכנית שכזו לא תשתמש בכלל במערך הבלוקים השמורים ב cache בגלל שהיא תצטרך לקרוא כל בלוק רק פעם אחת ולעומת זאת היא תצטרך לתחזק את מערך ה cached block לאחר כל קריאה. (תחפש ברשימה שלה כדי לבדוק האם הבלוק באמת לא נמצא ובמידה וכן לבצע את כל תהליך ההקצאה ומחיקה מהרשימה שזה תהליך שלוקח זמן כלשהו)

נשים לב שבגלל שמערכת הcache לא נותנת לנו ערך מוסף (שכן כל בלוק יקרא רק פעם אחת) אז המערכת רק מאיטה את ביצועי התוכנית.

לכן נוכל לגרוס כי במקרים שצפויים misses רבים (לדוגמא כל בלוק נקרא פעמים בודדות אך לעומת זאת הרבה בלוקים נקראים) נוכל לומר שמערכת הcache לא משפרת זמני הריצה אלא להפך, מאיטה אותם.

3 – הרעיון בלהשתמש באלגוריתמים מסובכים נובע מהעובדה שאלגוריתם מסובך יכול להיות מתאים יותר ל"צריכת הבלוקים" של התוכנית מה שיתן ליותר hits במהלך הריצה. הגישה הזו מחזיקה בטענה כי עדיף לבצע "הרבה" פעולות מעבד מאשר "קצת" פעולות גישה לדיסק ולכן מעדיפה להריץ אלגוריתמים מסובכים יחסית אשר יתנו תוצאות טובות יותר באגירת בלוקים נכונה וכתוצאה מכך תיחסך ממערכת ההפעלה גישות רבות לדיסק דבר אשר יגרום לתוכנית בoverall להיות מהירה יותר.

4- מקרה בו LRU טוב יותר מLFU הוא מקרה שבו אנחנו מבצעים ניתוח קבצים ואנחנו רוצים לנתח קובץ מסויים, לקרוא אותו מספר כלשהו של פעמים ואז לעבור לקובץ הבא. במקרה הנ"ל אלגוריתם הLRU ישמור את הקובץ הנוכחי בCACHE (או לפחות את רובו) ו"ישכח" מהקבצים שהיו לפני.

לעומת זאת הLFU כל הזמן "יאבד" את הבלוקים של הקובץ הנוכחי לאור העובדה שכמות הגישות שלהם מתחילת העבודה עד כמעט לסוף העבודה על הקובץ לרוב תהייה קטנה מכמות הגישות לבלוקים של הקובץ הקודם דבר אשר יגרום שלכל קובץ שנקרא הבלוקים שלו לרוב לא ישמרו בcache דבר אשר "מייתר" את מנגנון הCACHE שלנו.

מקרה נגדי היא תוכנית אשר בודקת האם קובץ אחד מכיל מספר רב של קבצים.

במקרה הנ"ל תוכנית הLFU תשמור את קובץ ההשוואה בזכרון לאורך כל התוכנית (בגלל כמות הגישות אליו) ותוכל להשוות יחסית ב"חסכון" את כל שאר הקבצים אליו לאור העובדה שהוא שמור בCACHE

לעומת זאת מנגנון הLRU לרוב "יאבד" את קובץ ההשוואה ויצטרך לקרוא את כולו בכל השוואה מחדש (שכן הוא "יעלם" כאשר נקרא מספר קבצים אשר עליהם נרצה לבצע את ההשוואה)

מקרה שבו שתי המקרים לא טובים הוא כמו שציינתי בסעיף 2. מקרה שבו התוכנית ניגשת לכל קובץ מספר קטן של פעמים וכל ה"לוגיקה" והסיבוכיות של אלגוריתם הCACHE פוגעת בזמני הריצה שכן אין hits כמעט בכלל. ולעומת זאת צריך לתחזק את הרשימות באלגוריתם.

5 – הבעיה שנתפרת היא locals problem מצב שבו ניגש לבלוק ספציפי מספר רב של פעמים אך ב long run לא באמת נצטרך אותו. באלגוריתם הLFU הוא יקבל דירוג גבוה ויהפוך לuntouchable וסתם "יתפוס מקום" ולעומת זאת בFBR הבלוק לא יקבל עדיפות וישאר בעל דירוג נמוך.

<u>חלק ב'</u>

1 - מספר הבלוקים המאוכסן ב-single indirect block הוא Block-size/Pointer-size. אם כן באמצעות כזה נוכל להגיע ל-2KB*512>40000 בייטים.(עשרת המצביעים הראשונים מאפשרים גישה רק ל-20K בייטים) אם כן תתבצע גישה ל-single indirect block ולאחר מכן לאחד הבלוקים אליו הוא מצביע. לבסוף תתבצע גישה למקום הספציפי בזכרון אותו נרצה לשנות(40000). נוסף על אלה, המחשב ייגש לתיקייה "/", ואז למדריך שלה, וכן ל -"os/", ולמדריך שלה. בסה"כ נספור 7 גישות לדיסק.

2- תתקיים פסיקה לכל system-call שהמשתמש ביצע(2, אחת ל-seek ואחת ל-read), כמו כן תתבצע פסיקה נוספת ע"י המערכת בפקודה read, כאשר הראשונה לקריאת המידע מהזכרון, והשניה לאחר קבלת המידע. בסה"כ 3 פסיקות.

3- א':בהנחה שהזמן שלוקח context-switch הוא קטן ביחס ליחידת הריצה של משימות, נבחר באלגוריתם Round robin, בו בכל זמן נתון(למשל כל יחידת זמן אחת) תתבצע החלפה של המשימות לפי סדרן ברשימת ההמתנה. באופן כזה כל אחת מהמשימות תתחיל את פעולתה בתוך:

(פרק הזמן בין כל פעולת הפקעה של האלגוריתם + context-switch)*(מספר המשימות לפניה בתור)

את פרק הזמן בין פעולות הפקעה נוכל לבחור להיות קטן כרצוננו(עד כדי מגבלות חומרתיות) ולכן לכל אלגוריתם אחר נקבל זמן ריצה פחות טוב.

3-ב': לא, לאלגוריתם שהצענו בסעיף הקודם תקורה גבוהה מאד, למעשה כל הפקעה יוצרת תקורה, וכיוון שההפקעות באלגוריתם זה מתרחשות לעתים תכופות, גם התקורה גבוהה. אלגוריתם בעל תקורה נמוכה יהיה FIFO, כלומר הרצה רציפה של כל אחת מהמשימות לפי הסדר אחת אחרי השניה, כשההחלפות מתבצעות רק בסיומן. אלגוריתם זה בעל תקורה מינימלית, כיוון שמספר החילופים המתבצע בין משימות הוא כמספר המשימות. לא ייתכנו פחות חילופים כיוון שאז נקבל שמשימה כלשהי לא התבצעה.