מרצים: דר' לאוניד רסקין

מתרגלים: ארתור קייאנובסקי, אריה טל, אסף רוזנבאום, איגור סמוליאר, מתיאס בונה, סאהר עודה

מערכות הפעלה (234123)

סמסטר חורף תשע"ו

מועד א'

<u>הנחיות</u>

1. ענו על כל השאלות; מלאו מספר סטודנט בעמוד זה.
2. אסור שימוש בכל חומר עזר
3. משך המבחן: שלוש שעות.
4. במבחן זה 22 דפים, כולל דף זה. וודאו שכל הדפים נמצאים.
5. יש לענות על כל השאלות בטופס הבחינה (מחברת הבחינה משמשת לטיוטה בלבד).
6. **<u>תשובות לא מנומקות לא תתקבלנה</u>**

ת.ז.: _____

שאלה 1	30
שאלה 2	25
שאלה 3	25
שאלה 4	20
סה"כ	100

בהצלחה!

שאלה 1 - אלגוריתם הזימון של לינוקס (30 נקודות)

שאלה זו עוסקת במדיניות זימון התהליכים של לינוקס כפי שנלמדה בתרגולים. לנוחיותכם מצורפים מספר macros המשמשים את זמן התהליכים

```
#define MAX_PRIO 140
#define MIN_TIMESLICE   (10 * HZ / 1000)
#define MAX_TIMESLICE   (300 * HZ / 1000)
#define TASK_TIMESLICE(p)
        MIN_TIMESLICE + (MAX_TIMESLICE – MIN_TIMESLICE) * \
        (MAX_PRIO – 1 – (p)->static_prio)/39
#define TASK_INTERACTIVE(p) \
        ((p)->prio <= (p)->static_prio – DELTA(p))
```

```
prio = static_prio – bonus
```

```
#define EXPIRED_STARVING(rq) \
        ((rq)->expired_timestamp && \
        ((jiffies – (rq)->expired_timestamp) >=STARVATION_LIMIT * ((rq)->nr_running + 1))
```

$$BONUS(p) = 25\% \times 40 \times \left(\frac{SleepAvg}{MaxSleepAvg} - \frac{1}{2}\right)$$

$$DELTA(p) = 5 \times \frac{TaskNice(p)}{20} + 2$$

א.(5 נק') נניח כי תהליך A מסווג כחישובי על ידי אלגוריתם הזימון של לינוקס ובעל עדיפות סטטית x ותהליך B מסווג כאינטראקטיבי על ידי אלגוריתם הזימון של לינוקס ובעל אותה עדיפות סטטית x. האם יתכן כי העדיפות הדינמית של A טובה יותר משל B?

ב.(5 נק') נניח כי תהליכים A ו - B מסווגים כ**חישוביים** על ידי אלגוריתם הזימון של לינוקס ובעלי עדיפות **דינמית** שווה, אבל עדיפות הסטטית של A טובה יותר. איזה יתרון מקבל A על B?

ג.(5 נק') נניח כי תהליכים A ו - B מסווגים כ**אינטראקטיביים** על ידי אלגוריתם הזימון של לינוקס ובעלי עדיפות **דינמית** שווה. אבל העדיפות הסטטית של A טובה יותר. איזה יתרון מקבל A על B?

ד.(9 נק') לפניך קטע מתוך הקוד של פונקציית הגרעין sys_sched_yield אשר מממש את הטיפול בתהליכים עם מדיניות זימון OTHER.

```
1.  list_del(&current->run_list);
2.  if (!list_empty(array->queue + current->prio)) {
3.        list_add(&current->run_list, array->queue[current->prio].next);
4.        goto out_unlock;
5.  }
6.  __clear_bit(current->prio,array->bitmap);

7.  i = sched_find_first_bit(array->bitmap);

8.  if (i==MAX_PRIO || i<= current->prio)
9.        i = current->prio;
10. else
11.   current->prio = i;

12. list_add(&current->run_list, array->queue[i].next);
13. __set_bit(i,array->bitmap);

14. out_unlock:
15.   // release locks & call schedule
```

1.(5 נק') בהנחה שקיימים תהליכים נוספים שאינם expired ב - runqueue, האם יתכן כי ביצוע sched_yield על ידי תהליך עם מדיניות זימון OTHER לא יגרום להחלפת הקשר?

2.(2 נק') איזו בעיה הייתה נוצרת אם היינו מחליפים את שורה 8 בשורה:

```
if (i<= current->prio)
```

3.(2 נק') איזו בעיה הייתה נוצרת אם היינו מחליפים את שורה 8 בשורה:

```
if (i==MAX_PRIO)
```

ה.(6 נק') בפונקציה do_fork הכנסת תהליך הבן ל - runqueue והפיכתו למוכן לריצה נעשית על ידי הפונקציה wake_up_forked_process. להלן קטע מן הקוד שלה אשר מטפל בתהליכים עם מדיניות זימון OTHER:

```
p->state = TASK_RUNNING;
if (!rt_task(p)) {
  current->sleep_avg = current->sleep_avg * PARENT_PENALTY / 100;
  p->sleep_avg = p->sleep_avg * CHILD_PENALTY / 100;
  p->prio = effective_prio(p);
}
activate_task(p);
```

כאשר ערכו של PARENT_PENALTY הוא 100 וערכו של CHILD_PENALTY הוא 95.

מי ירוץ קודם לאחר קריאה ל-fork, האב או הבן? תארו מצב שבו המצב ההפוך יעיל יותר והסבירו מדוע הוא יעיל יותר.

שאלה 2 - זיכרון ווירטואלי (25 נקודות)

א.(4 נק') כזכור, ב-Power PC יש 2 רמות בזיכרון וירטואלי. ברמה העליונה משתמשים בדפים ענקיים (256MB). הסבירו מה היתרון בשימוש בדפים כה גדולים ברמה העליונה ב-Power PC.

ב.(4 נק') מה הם החסרונות בשימוש בדפים גדולים? הסבירו למה בארכיטקטורה של אינטל משתמשים בדפים קטנים בגודל של 4KB.

ג.(4 נק') ציינו לפחות 2 סיבות למה ברמה השנייה ב-Power PC משתמשים בדפים קטנים.

ד.(4 נק') בארכיטקטורה של אינטל (אפילו בארכיטקטורה של 64 ביט) TLB הוא קטן משמעותית מזה של Power PC. הסבירו את ההבדל הזה.

ה.(4 נק') מתברר שב-Power PC אין רגיסטר שמצביע לטבלאות הדפים (ל-PGD) כמו בארכיטקטורה של אינטל. הסבירו למה רגיסטר זה הכרחי לארכטקטורה של אינטל ואיך Power PC מתמודד עם זה.

ו.(5 נק') למה ב-HTAB ב-Power PC נשמרת לא רק כתובת פיזית אלא גם כתובת וירטואלית? האם זה קורה גם בטבלאות הדפים (של תהליך שרץ על Power PC)? נמקו תשובתכם.

שאלה 3 - קלט פלט (25 נקודות)

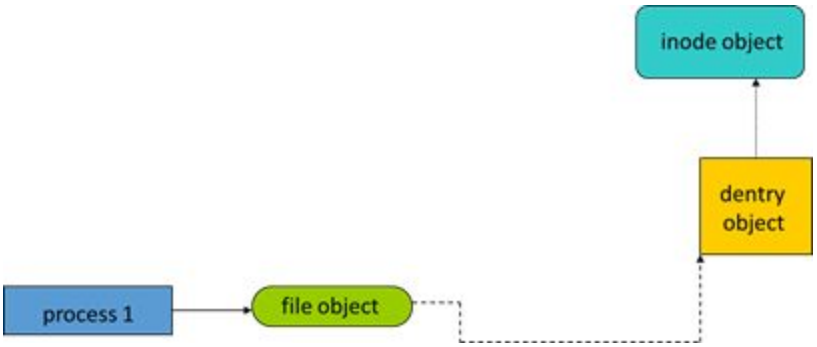
הנחיות לפתרון השאלה:

1. במקומות שבהם אתם צריכים path לקובץ כלשהו השתמשו בשבלונה מהצורה:./dir1/file1. אם אתם מתייחסים לקובץ שונה השתמשו בpath שונה, לדוגמא 2 הpaths הבאים מתארים 2 קבצים שונים: /dir1/file1, /dir1/file2
2. במקומות שבהם אתם צריכים מספר של file descriptor השתמשו במספר גדול מ-2 כלשהו והסבירו לאיזה קובץ הfile descriptor מתייחס.
3. בכל סעיף שבו מבקשים מכם לכתוב שורת קוד עליכם לכתוב קריאה לפונקציה/קריאת מערכת עם הפרמטרים הרלוונטיים שלה. אין צורך לזכור את כל הפרמטרים או את הסדר שלהם, אבל פרמטרים שחשובים לתשובה על השאלה צריכים להופיע – לדוגמא – עבור קריאת מערכת X שמקבלת path של קובץ ועושה איתו משהו, חשוב לזכור לכתוב איזה path זה באופן הבא X(path). אם הפרמטרים האחרים לא חשובים למענה על השאלה – אפשר להשמיט אותם.
4. בסעיפים שבהם יש שורה אחת בלבד עבור התשובה – יש לכתוב את השורה הנדרשת ללא הסברים נוספים.

בהתחלה מצב המערכת נראה כך:

מצב 1

process 1 קרא לקריאת מערכת כלשהי – והמצב השתנה למצב הבא:



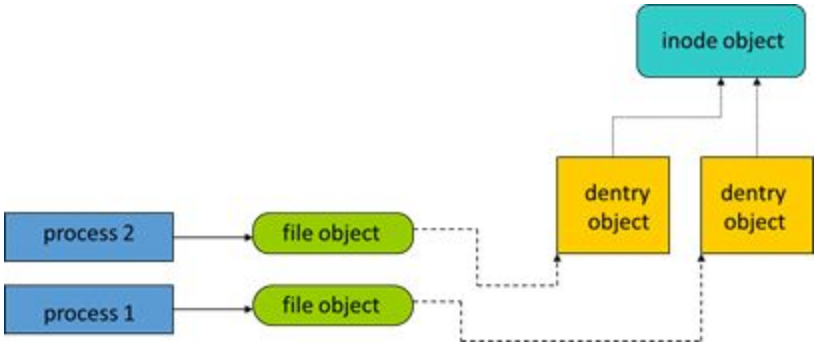
מצב 2

א.(1 נק') כתבו שורת קוד שגרמה למצב 2 להיווצר ממצב 1. הניחו שהpath לקובץ איתו עובדים הוא dir/file1./

ב.(4 נק') עבור כל אחד מהאובייקטים הבאים – היכן הוא יושב? (בדיסק? בRAM?), עבור כל אובייקט שיושב בRAM האם יש לו אובייקט מקביל בדיסק? נא למלא את הטבלה – אם התשובה בעמודה הראשונה היא שיושב בדיסק אז השאירו את התא באותה שורה בעמודה השניה ריק.

	יושב בRAM/דיסק?	אם בRAM האם קיים אובייקט מקביל בדיסק?
inode object		
file object		

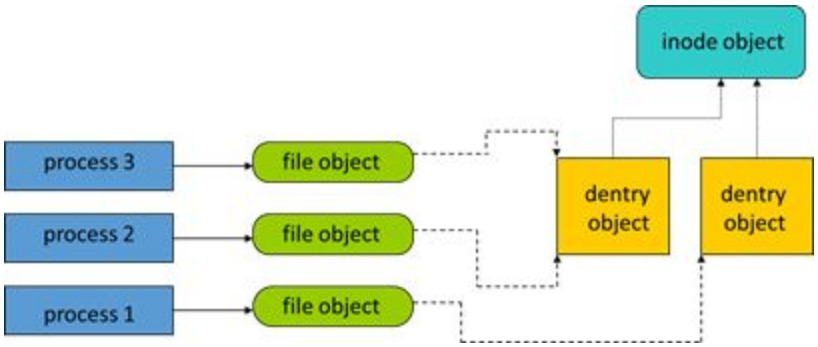
כעת process 2 מבצע את הקוד שלו – וזה גורם למצב המערכת להשתנות למצב הבא:



מצב 3

ג.(4 נק') כתבו שורת קוד שגרמה למצב 3 להיווצר ממצב 2. הסבירו איך יתכן שיש 2 dentry objects שונים אבל inode object יחיד. איך זה בא לידי ביטוי בשורת הקוד שכתבת יחסית לשורה בסעיף א?

כעת process 3 מבצע את הקוד שלו – וזה גורם למצב המערכת להשתנות למצב הבא:

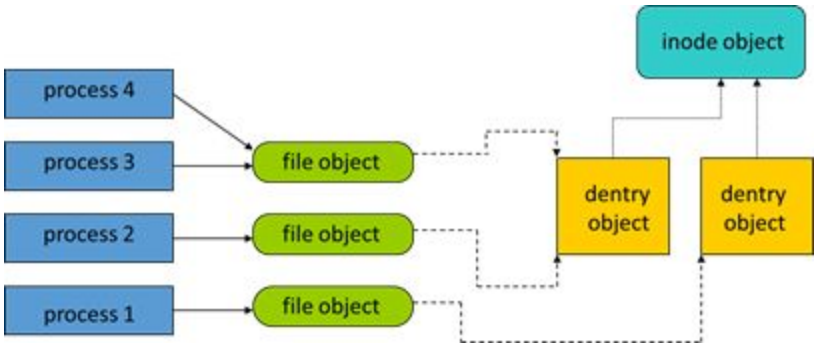


מצב 4

ד.(4 נק') כתבו שורת קוד שגרמה למצב 4 להיווצר ממצב 3. במה עוזר הקיום של file object שונים עבור אותו dentry object? לשם מענה על השאלה הניחו שלא היה file object, והמידע שיושב בfile objects היה יושב פעם אחת בdentry object (ולא בכל file object בנפרד) - תנו דוגמא לדבר אחד שלא היה ניתן לעשות במצב זה אבל ניתן לעשות במימוש הנוכחי עם file objects.

ה.(4 נק') באיזה מבנה נתונים בגרעין חיפשה (ומצאה) מערכת ההפעלה את הdentry object אליו מצביע הfile object של process 3? מה תפקידו של מבנה נתונים זה?

כעת נוצר המצב הבא:



מצב 5

ו.(2 נק') כתוב שורת קוד שגרמה למצב 5 להיווצר ממצב 4.

ז.(6 נק') נתון חלק מהקוד של process 5 (שלא מופיע באיורים):

```
printf("this should appear on screen\n");
```

סטודנט הריץ את process 5 ומצא את הטקסט "this should appear on screen" בתוך הקובץ tmp/file/. ישנן 2 דרכים שונות לקבל אפקט זה: בשורת הפקודה בterminal ובקוד של process 5. הניחו ששם הexecutable של process 5 הוא p5.

1.(2 נק') כתבו את שורת הפקודה בterminal שתגרום לטקסט להופיע בקובץ tmp/file/ במקום על המסך.

2.(4 נק') כתבו 2 שורות קוד שהופיעו לפני הprintf הנ"ל שיגרמו לטקסט להופיע ב tmp/file/ ולא על המסך.
הסבירו מדוע שורות אלה גורמות לפלט להגיע לקובץ במקום למסך.

שאלה 4 - סינכרוניזציה (20 נקודות)

(בשאלה זו 1K = 1024)

נתון רכיב חומרה שכולל זיכרון פנימי בגודל 8KB, שניתן לקרוא ממנו ולכתוב אליו נתונים. נתונות שתי הפונקציות הבאות:

```
void *device_malloc (size_t size);
void device_mfree (void *ptr);
```

הפונקציה `device_malloc()` **מחכה** עד שיתפנה מספיק מקום בזיכרון של הרכיב, ואז מקצה בו `size` בתים ומחזירה מצביע לזיכרון שהוקצה. הפונקציה `device_mfree()` משחררת זיכרון שהוקצה ב-`device_malloc()`.

מתכנת כתב את הפונקציה הבאה, שמקצה שני אזורים בזיכרון של הרכיב:

```
void func1()
{
     void *m1 = device_malloc(1024);
     void *m2 = device_malloc(1024);

     /* ... */

     device_mfree (m2);
     device_mfree (m1);
}
```

בסעיפים א'-ג' הניחו שבמצב ההתחלתי כל הזיכרון של הרכיב (8KB) אינו בשימוש, ומריצים במקביל N תהליכים, כך שכל אחד מהם מבצע קריאה אחת לפונקציה `func1()`.

א. (4 נק') הסבירו למה עבור N=8 עלול להיווצר deadlock בו כל 8 התהליכים משתתפים (כלומר: איך יכול להיווצר מצב שבו כל 8 התהליכים "תקועים" ולא יכולים להמשיך לרוץ).

ב. (4 נק') הסבירו למה עבור N=7 לא יכול להיווצר deadlock.

בניסיון לפתור את הבעיה, הוחלט לחלק את הזיכרון הפנימי של הרכיב ל־8 אזורים בגודל 1KB כל אחד, ולמספר אותם מ-1 עד 8. כמו־כן הוחלט לשנות את הגדרת הפונקציות כדלקמן:

```
void *device_malloc(int area, size_t size);
void device_mfree(int area, void *ptr);
```

כעת שתי הפונקציות מקבלות כפרמטר מספר של אזור בזיכרון של הרכיב. `device_malloc()` מחכה עד שיתפנה מספיק מקום באזור הנתון, ואז מקצה בו `size` בתים ומחזירה מצביע לזיכרון שהוקצה. הפונקציה `device_mfree()` משחררת זיכרון שהוקצה ב-`device_malloc()`.

בהתאם לשינויים בהגדרת הפונקציות, המתכנת שינה את הפונקציה שכתב כך שתשתמש בהגדרות החדשות:

```
void func1()
{
     void *m1 = device_malloc(1, 1024);
     void *m2 = device_malloc(2, 1024);

     /* ... */

     device_mfree (2, m2);
     device_mfree (1, m1);
}
```

ג. (4 נק') הסבירו למה עבור כל ערך של N, המימוש החדש לא יכול לגרום ל-deadlock.

מתכנת אחר כתב פונקציה אחרת שממומשת כך:

```
void func2()
{
     void *m2 = device_malloc(2, 1024);
     void *m1 = device_malloc(1, 1024);

     /* ... */

     device_mfree (1, m1);
     device_mfree (2, m2);
}
```

ד. (4 נק') מריצים במקביל שני תהליכים: תהליך A קורא ל-`func1()` ובאותו הזמן תהליך B קורא ל-`func2()`. הסבירו למה מצב כזה עלול לגרום ל-deadlock.

כדי למנוע בעיות מסוג זה, הגדירו את הפונקציה הבאה, שמקצה זיכרון בשני אזורים שונים מתוך הזיכרון הפנימי של הרכיב:

```
void device_double_alloc(void **p1, int area1, size_t size1,
                         void **p2, int area2, size_t size2);
```

הפוקנציה מקצה size1 בתים באזור area1 ו-size2 בתים באזור area2. כשהפונקציה חוזרת, p1* מצביע לזיכרון שהוקצה באזור הראשון ו-p2* מצביע לזיכרון שהוקצה באזור השני.

כעת, שני המתכנתים שינו את מימוש הפונקציות שכתבו:

```
void func1()
{
     void *m1, *m2;

     device_double_alloc(&m1, 1, 1024, &m2, 2, 1024);

     /* ... */

     device_mfree (2, m2);
     device_mfree (1, m1);
}

void func2()
{
     void *m2, *m1;

     device_double_alloc(&m2, 2, 1024, &m1, 1, 1024);

     /* ... */

     device_mfree (1, m1);
     device_mfree (2, m2);
}
```

ה. (4 נק') השלימו את מימוש הפונקציה `device_double_alloc()` כך שהתהליכים A ו-B שתוארו בסעיף הקודם יוכלו לרוץ במקביל מבלי להיכנס ל-deadlock:

```
void device_double_malloc (void **p1, int area1, size_t size1,
                void **p2, int area2, size_t size2)
{
     if (____________________________________)
     {
          *p1 = device_malloc(area1, size1);
          *p2 = device_malloc(area2, size2);
     }
     else
     {
          ------------------------------------------------
          ------------------------------------------------
     }
}
```

פתרונות

שאלה 1:

א. לא. לשני התהליכים עדיפות סטטית זהה ולכן $\Delta_A = \Delta_B$ (כי Δ תלויה רק ב - nice). תהליך הוא אינטראקטיבי אם ורק אם $Bonus \geq \Delta$ ומכאן ש - $Bonus_B \geq \Delta_B = \Delta_A > Bonus_A$ ולכן $Prio_A = x - Bonus_A > x - Bonus_B = Prio_B$.

סטודנטים רבים טעו ונתנו דוגמה כאשר לכל תהליך נקבע nice שונה, יש לשים לב כי הנתון על עדיפות סטטית זהה גורר שגם nice זהה לשני התהליכים ולכן הדוגמה אינה נכונה.

ב. ככל שתהליך עדיפות סטטית טובה יותר ה - time slice שלו יהיה גדול יותר

לא התקבלו תשובות שטענו כי יהיה לתהליך קל יותר להפוך לאינטראקטיבי מכיוון שתהליך הוא חישובי (cpu bound).

ג. ככל שהעדיפות הסטטית טובה יותר קל יותר לתהליך להיחשב אינטראקטיבי, כלומר הוא יכול לבצע יותר חישובים לפני שיהפוך לחישובי.

טעות נפוצה הייתה לענות גם פה שהיתרון הוא time slice יותר גדול, אך מכיוון שבכל מקרה תהליך אינטראקטיבי מקבל time slice חדש בכל פעם, זה אינו יתרון אמיתי. גם הטענה שבכל פעם שתהליך מסיים time slice הוא עובר לסוף התור אינה עוזרת מכיוון שתהליך אינטראקטיבי בכל מקרה יוצא מהתור פעמים רבות. היו שטענו כי במידה והגענו לסף ההערבה יש יתרון ל - time slice ארוך כי נוכל להמשיך לרוץ יותר זמן כאשר לא נקבל time slice חדש וזה גם כן לא נכון, כי לאו דווקא יש סנכרון בין הרגע בו התקבל סף ההרעבה ליתרת ה - time slice, כלומר יתכן לתהליך מסויים מקבל time slice ארוך יותר אבל דווקא בעת ההגעה לסף ההרעבה נותר לו time slice קצר בלבד.

ד.

1. לא. אם הרשימה ממנה הוצאנו את התהליך אינה ריקה הוא עובר למקום השני ברשימה ולכן לא יהיה הבא לרוץ ותהיה החלפת הקשר. אחרת, אם קיים תהליך בעדיפות טובה יותר התהליך שביצע sched_yield "נשאר במקום" ולא יהיה הבא לרוץ. אחרת משנים (לרעה) את העדיפות הדינמית של התהליך ומוסיפים אותו למקום השני ברשימה ולכן לא ירוץ.

כמעט כל הסטודנטים כתבו שתהליך מוכנס לסוף התור וזה לא נכון - הוא נכנס למקום השני ברשימה (לא הורדו על כך נקודות).

2. כאשר התהליך היה התהליך היחיד ב - active array הינו מבצעים גישה לא חוקית לזיכרון: array->queue[MAX_PRIO]

טעות נפוצה הייתה לטעון שתהליך יקבל את העדיפות של swapper וירוץ רק אחריו, זה לא נכון ראשית ל - swapper אין עדיפות 140 ושנית במערך יש 140 תאים ולכן התא במקום MAX_PRIO אינו קיים.

ג. היה ניתן להשתמש ב - sched_yield כדי לשפר את העדיפות ואף להפוך תהליך OTHER ל - RT

יש לשים לב שלא רק שנכניס את התהליך שמבצע yield לתור עדיף אלא ממש נשפר את העדיפות הדינמית שלו (לא הורדו על כך נקודות).

ה. הקוד הנ"ל גורם לאב לרוץ לפני הבן כי יש לו sleep_avg גדול יותר ולכן בונוס גדול יותר ועדיפות דינמית טובה יותר (לשניהם אותה עדיפות סטטית). במידה והבן מבצע exec בשלב מוקדם של ריצתו (מקרה נפוץ), אם ירוץ קודם נחסוך overhead של cow (מייתר את הצורך בהעתקות זיכרון).

כמובן מי שכתב "הבן ירוץ קודם כי ככה נלמד בכיתה" לא קיבל ניקוד, הקוד נתון ומצופה שטענו על פיו ובפרט שזה לא נכון במקרה הזה.

מי שטעה וחשב שהבן רץ קודם ותיאר בחלק השני של הסעיף מקרה בו עדיף שהאב ירוץ קודם קיבל ניקוד על חלק זה.

מי שנתן כדוגמה מקרה בו התכנון של הקוד אינו נכון לא קיבל ניקוד, למשל שהבן עושה busy-wait ולכן כדאי להריץ קודם את האב, במצב זה תכנון נכון לא כולל busy-wait.

שאלה 2:

א. אפשר לסתפק ב- SLB קטן ולקבל Hit Rate גבוה מאוד. תשובה שזה מקטין מספר פסיקות הדף בלי הסבר על SLB לא התקבלה.

ב. דפים גדולים גורמים לשיברור פנימי גבוה. ב-IA גודל של מסגרת זהה לגודל של דף. משמעות של הדבר הוא שהיינו פשוט מבזבזים המון זיכרון פיזי. שימו לב שטענה שזיכרון גדול דורש יותר זמן דיפדוף לא נכונה.

ג. 1. למנוע בזבוז זיכרון פיזי.

2. אפשר להשתמש ב-TLB גדול יותר כי הוא לא מתאפס בכל החלפת הקשר.

ד. ב-Power PC זיכרון וירטואלי הינו משותף לכל התהליכים ולכן אין צורך לרוקן TLB בכל החלפת הקשר.

ה. בINTEL חומרה משתמשת בPGD ולכן חייבת לדעת איפה היא יושבת. בPOWER PC החומרה לא מכירה את תבלאות תרגום ולכן אין צורך ברגיסטר שיצביע עליהן.

ו. כל כניסה בHTAB מכילה 8 תרגומים וצריך לדעת באיזה מהם לבחור.

שאלה 3:

א. open("/dir/file1");
ב. כל האובייקטים יושבים בRAM. רק לinode object יש אוביקט מקביל היושב בדיסק שקוראים לו inode.
ג. open("/dir/file2");
כאשר /dir/file2 הוא hardlink ל/dir/file1. זה בא לידי ביטוי בכך ששם הקובץ שמעבירים לopen שונה מזה שהעבירו בסעיף א.
ד. open("/dir/file2");

התקבלה גם התשובה שפותחים קובץ שלישי שהוא softlink לקובץ מסעיף ג (אבל לא לקובץ מסעיף א!!)- כי פתיחה של softlink בסופו של דבר פותחת את הקובץ שהנתיב שלו כתוב בו.
file object מתאר פתיחה של קובץ, אם 2 תהליכים פתחו את אותו הקובץ הם יכולים להימצא במקומות שונים בקובץ באותו הזמן כי בכל file object יש מחוון קובץ משלו. אם היה מחוון יחיד בdentry object זה לא היה אפשרי. דוגמא נוספת היא שאפשר לפתוח את הקובץ פעם אחת לקריאה בלבד ופעם אחרת לכתיבה – גם זה מצב שהיה בלתי אפשרי ללא הfile object.
חשוב להבין שfile object לא מנהל הרשאות - היו סטודנטים שכתבו שfile object אחד יכול לאפשר למשתמש כלשהו לפתוח קובץ לכתיבה ולמשתמש אחר לא לתת לפתוח את אותו הקובץ לכתיבה - הfile object לא מונע פתיחת קובץ לכתיבה (למשל) - אם למשתמש אין הרשאת כתיבה לקובץ, הopen של הקובץ יכשל ולא יווצר file object כלל!

ה. מערכת ההפעלה מצאה את הdentry object בdentry cache שתפקידו לקצר את איתור המסלול לקובץ בעת מתן path ובכך נחסכות גישות רבות לדיסק.

ו. fork();
סטודנטים רבים כתבו pthread_create() . תשובה זו לא נכונה מכיוון שחוטים בpthreads משתפים בינהם את הpdt ולכן אם יווצר חוט נוסף כמות ההצבעות על הfile object לא תגדל!

ז. 1. ./p5 > /tmp/file
2.
```
close(1);
open("/tmp/file");
```

שאלה 4:

א. ייתכן מצב שבו כל 8 התהליכים מבצעים את ההקצאה הראשונה במקביל, ורק אחר כך כולם מבצעים את ההקצאה השנייה. במצב כזה, אחרי 8 הקצאות של 1KB כל הזיכרון של הרכיב יהיה תפוס, ולכן כשהתהליכים ינסו להקצות זיכרון נוסף הם ייכנסו להמתנה שהם לעולם לא ייצאו ממנה (כי הזיכרון לעולם לא ישוחרר).

ב. כשיש פחות מ-8 תהליכים, תמיד אחד מהם יצליח לבצע את ההקצאה השנייה, ולאחר שיסיים ישחרר את שתי ההקצאות שביצע וכך ישחרר תהליכים אחרים מהמתנה. לכן לא ייווצר deadlock במצב הזה.

ג. המצב שקול לשני מיוטקסים שנעלים תמיד באותו הסדר, וכמו שראינו בהרצאה תבנית כזו לא יכולה לגרום ל-deadlock ללא קשר למספר התהליכים שמבצעים אותה.

הרבה סטודנטים כתבו שה-deadlock נמנע בגלל שסדר השחרורים הפוך לסדר הנעילות. שימו לב שאין שום חשיבות לסדר השחרורים (לא במקרה הזה ולא בשום מקרה אחר). קיום או אי-קיום deadlock מהסוג הזה תלוי רק בסדר הנעילות/הקצאות, ולא בסדר שבו משחררים אותן.

ד. כמו בסעיף הקודם, המצב שקול לשני מיוטקסים, אלא שכאן הם לא תמיד ננעלים באותו הסדר ולכן ייתכן deadlock. לדוגמה: אם שני התהליכים מבצעים במקביל את ההקצאה הראשונה, אז כל הזיכרון באזורים 1 ו-2 יהיה תפוס על-ידם, ולכן כשינסו לבצע את ההקצאה השנייה כל אחד מהם יחכה עד שהשני ישחרר את הזיכרון שהקצה, וכך שניהם ייכנסו ל-deadlock.

ה. הפתרון הכי פשוט וכללי הוא:

```
void device_double_malloc (void **p1, int area1, size_t size1,
                void **p2, int area2, size_t size2)
{
     if (area1 < area2)
     {
          *p1 = device_malloc(area1, size1);
          *p2 = device_malloc(area2, size2);
     }
     else
     {
          *p2 = device_malloc(area2, size2);
          *p1 = device_malloc(area1, size1);
     }
}
```

כמובן שכל וריאציה על הפתרון הזה התקבלה. הרבה סטודנטים השתמשו בתנאי ספציפי יותר שמתאים למקרה שתואר בשאלה (למשל `(area1 == 1)`) וקיבלו ניקוד מלא.

היו כמה סטודנטים שהחליפו בין p1 לבין p2, כלומר כתבו בבלוק ה-`else`:

```
          *p1 = device_malloc(area2, size2);
          *p2 = device_malloc(area1, size1);
```

פתרון כזה סותר את הגדרת הפונקציה `device_double_malloc()` שהובאה בשאלה, ולכן גרם להורדה של נקודה.

חלק מהסטודנטים ניסו לבדוק בתנאי כמה זיכרון מוקצה באזור מסויים, למשל על-ידי בדיקה מהצורה:

```
     if (!isAssigned(area1) && !isAssigned(area2))
     {
          *p1 = device_malloc(area1, size1);
          *p2 = device_malloc(area2, size2);
     }
     else
     {
          /* ... */
     }
```

הפתרון הזה לא תקין משתי סיבות. הסיבה הראשונה היא שלא הוגדרה בשאלה שום דרך לבדוק דבר כזה, ולכן לא יכולתם להניח שקיימת דרך כזו. הסיבה השנייה היא, שגם אם היתה דרך לבדוק את זה הפתרון לא היה עובד --- למשל, אם שני התהליכים נכנסים במקביל לפונקציה ובודקים במקביל את התנאי, שניהם יראו ששני האזורים פנויים ולכן ייכנסו לבלוק ה-`if`, במקום שאחד מהם ייכנס לבלוק ה-`else`. כך נקבל שתהליך A יקצה באזור 1 ואחר-כך באזור 2, ותהליך B יקצה באזור 2 ואחר-כך באזור 1, ונקבל את אותו deadlock שהיה בסעיף הקודם.