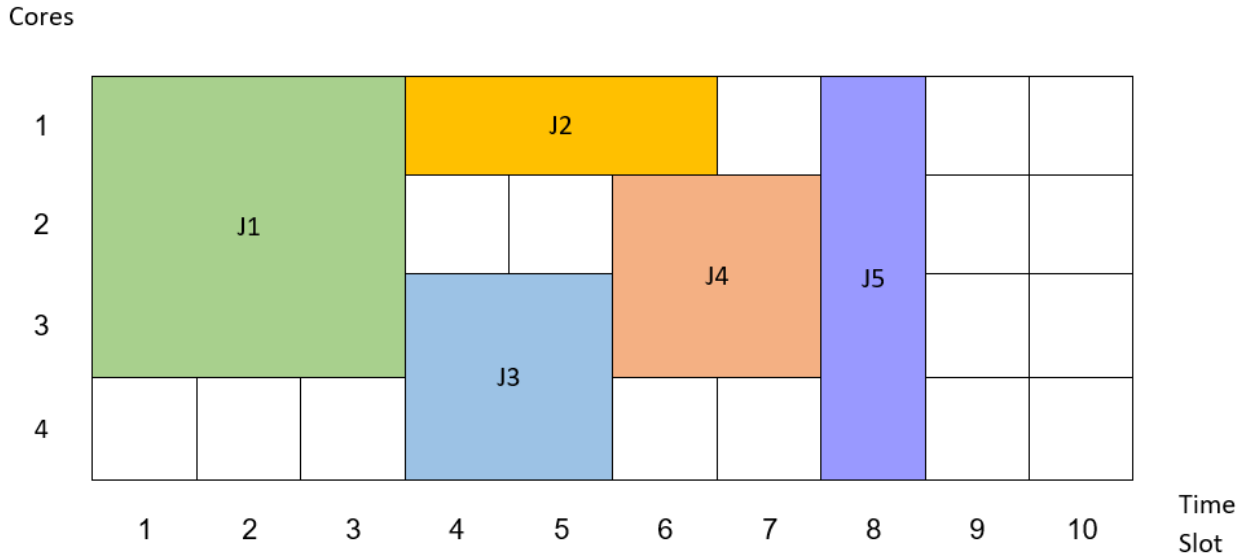
מערכות הפעלה 234123 תרגיל בית – 2, יבש

מגישים:
ilya.r@campus – איליה רוזנצוייג – 320517550
ariel.weiss@campus - אריאל וייס – 204361315

(מתנצלים ☹, זה לא הצליח להוריד בדיוק שורה אחת מבלי לרדת בעמוד נוסף)

שאלה 1 - זימון תהליכים (50 נק')

שאלה זו עוסקת בנושא batch-scheduling כפי שנלמד בהרצאות. שימו לב שבכל טבלאות התזמון הבאות המספר המייצג time-slot כלשהו מייצג את הזמן שבו החלון הנתון **מסתיים** (כלומר - אם time slot = 3 אזי החלון **מתחיל** ב-t=2 ומסתיים ב- t=3).



1. (18 נק') בהינתן התזמון הנתון למעלה, חשבו את המדדים הבאים (הראו את דרך החישוב). הניחו שכל התהליכים בנתונים למעלה הגיעו באותו הזמן (t=0):
   1. (6 נק') מדד זמן ההמתנה הממוצע (average wait-time):

נציין עבור כל משימה את הזמן הממוצע שלה ולאחר מכן נבצע ממוצע של כל הזמנים הללו:

J1=0 ; J2 = 3 ; J3=3; J4=5; J5=7

Average = $\frac{0+3+3+5+7}{5} = \frac{18}{5} time\ slots$

   2. (6 נק') מדד זמן התגובה הממוצע (average response-time):

כעת נוסיף לכל משימה את זמן הריצה שלה ונחשב שוב את הממוצע:

J1=3; J2=3+3; J3=3+2; J4=5+2; J5=7+1

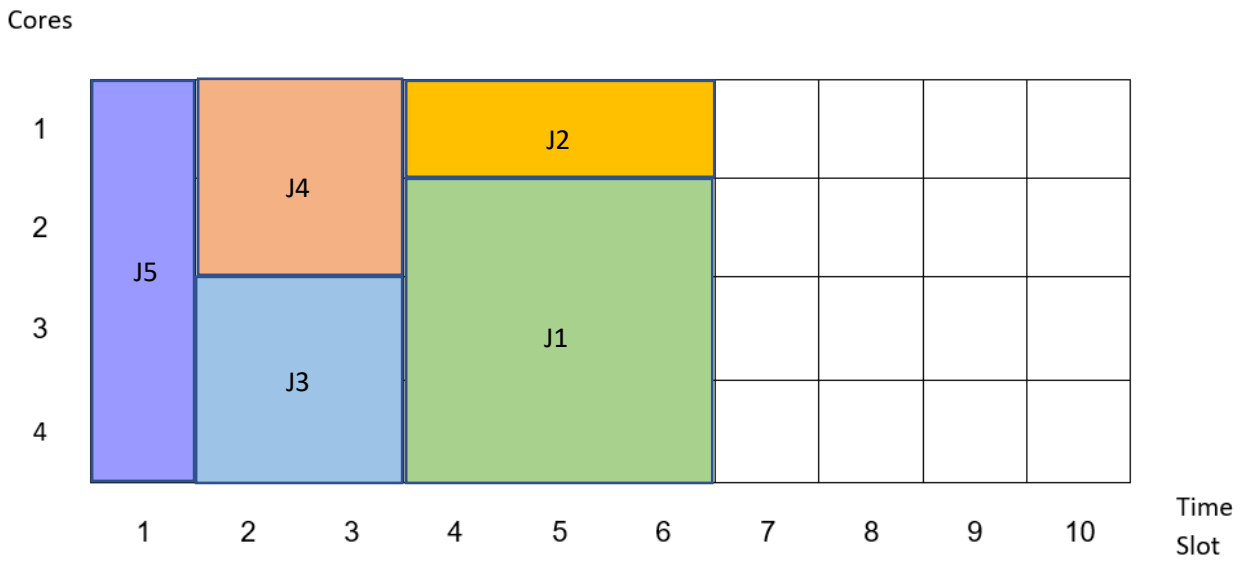
Average=$\frac{3+6+5+7+8}{5} = \frac{29}{5} time\ slots$

   3. (6 נק') מדד הניצולת (utilization):

Utilization=$100 \cdot \frac{(8 \cdot 4 - 8)}{8 \cdot 4} = 75\%$

2. (8 נק') בהנחה שכל התהליכים למעלה מגיעים בזמן 0. איזו מדיניות זימון יכולה לשפר את הניצולת כפי שחושבה בסעיף הקודם? **נמקו** (ראו מצורפת טבלת זימון ריקה על מנת להמחיש את הזימון החדש לפי האלגוריתם, **עליכם למלא את הטבלה בהתאם**)

<u>נשתמש באלגוריתם SJBF מכיוון שהוא נותן עדיפות לתהליכים אשר רצים מעט יותר זמן. אנחנו יודעים כמה זמן כל תהליך רץ בדוגמה, לכן התהליכים הקצרים ירוצו במקביל, כיוון שבמקרה שלנו הם מתחלקים שווה בכמות הליבות ואילו התהליך הארוך יותר ירוץ בסוף. כך נוכל להריץ יותר תהליכים קצרים (קיימים כאלו יותר בדוגמה) ולנצל יותר את המערכת. SJBS מאפשר BACKFILLING ולכן הנצילות תהיה מקסימלית.</u>



3. (8 נק') מנו 2 יתרונות ו-2 חסרונות של מדיניות זימון מסוג batch-scheduling על פני מדיניות Round-robin. נמקו.

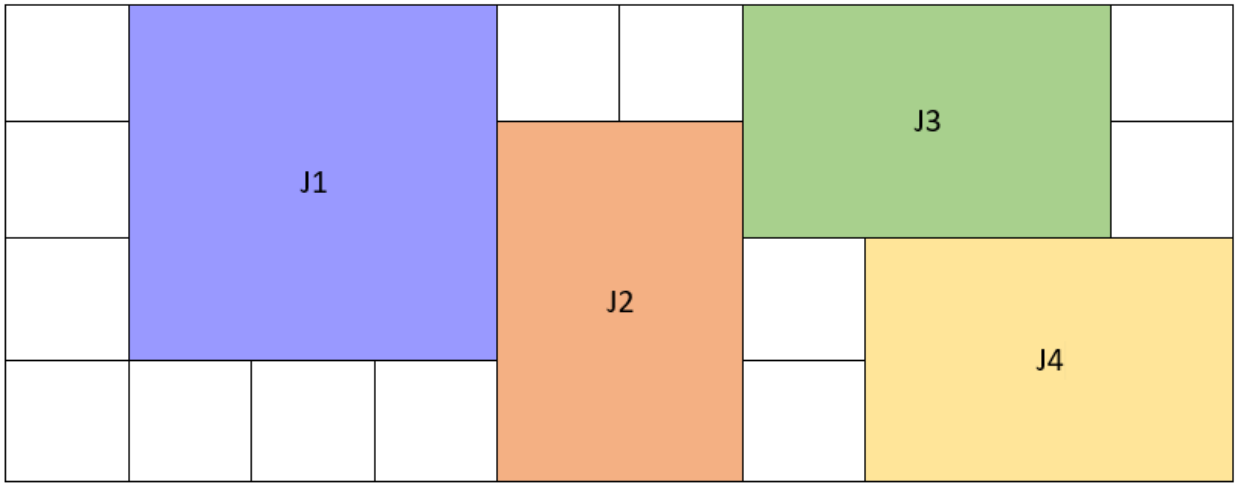
יתרונות:

1. כל משימה יכולה להתפרש על כל הליבות וכך לנצל את המערכת העומדת לרשותה באופן הכי יעיל.
2. ההפרש בין זמן המתנה לזמן תגובה הינו קבוע – הוכחנו זאת בהרצאה,כלומר זמן תגובה ממוצע נמוך יותר.

חסרונות:

1. הרעבת תהליכים. מכיוון שלתהליכים ניתנת עדיפות לפי כלל מסויים, יהיו תהליכים אשר לא יזכו לרוץ באותה כמות, אם בכלל, כמו תהליכים אחרים. כלומר המדיניות של BS אינה תמיד "הוגנת".
2. BS צריך לדעת מבעוד ומועד כמה זמן משימות ירוצו, בRR אנחנו מקצים לכל תהליך timeslice מראש.

נתונה מדיניות זימון חדשה BSAF (Biggest Surface-Area First) לפיה בהינתן 2 תהליכים זה שנבחר לרוץ קודם הוא זה **שהשטח שלו הגדול יותר** (שטח = זמן * מספר-מעבדים). שימו לב שמדיניות זימון זו תומכת ב-backfilling - כלומר שאם התהליך העדיף ביותר לפי BSAF לא יכול להיות מתוזמן בחלון כלשהו, יתוזמן התהליך הטוב ביותר שמתאים לחלון אחריו (זה שמתאים לחלון הפנוי והוא הטוב ביותר לפי BSAF). בהינתן 2 תהליכים בעלי אותו שטח נבחר בזה שיש לו את זמן הריצה הקצר ביותר. בהינתן 2 תהליכים בעלי שטח זהה וזמן ריצה זהה, נבחר בזה עם ה-ID (מספר) הנמוך יותר.
למשל, בהינתן 4 התהליכים הבאים (האיור להמחשה של מימדי התהליכים בלבד):

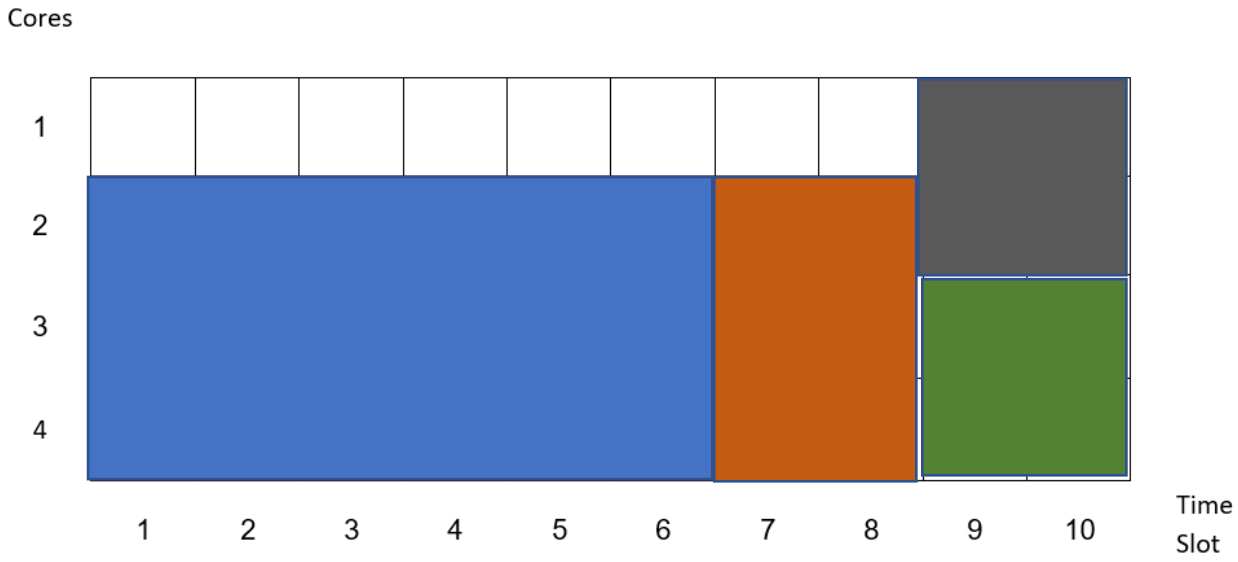


האלגוריתם BSAF יתעדף אותם בסדר הבא:

J1 (3*3=9 surface area)
J2 (3*2=6, shorter runtime than J3 and J4)
J3 (2*3=6, same as J4, but lower ID)
J4

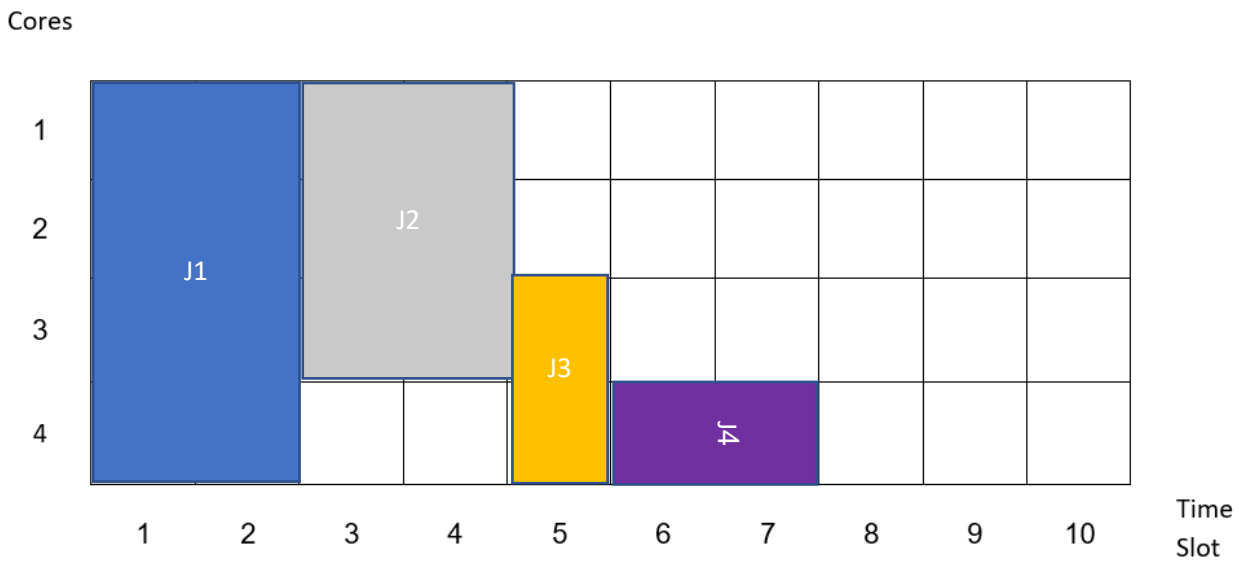
4. (8 נק') האם BSAF סובל מ-convoy effect? נמקו (עליכם להמחיש בעזרת הטבלה הנתונה).

<u>בהחלט. אם יהיו לנו יותר תהליכים בעלי שטח גדול יותר (בהכרח גם בעלי זמן ריצה גבוה), אזי תהליכים בעלי שטח קטן ייאלצו לרוץ רק בתום ריצתם של הגדולים. במצב כזה, נקבל Convoy effect .</u>



5. (8 נק') האם אלגוריתם הזימון BSAF הינו אופטימלי מבחינת מדד הניצולת? הוכיחו או הפריכו (עליכם להמחיש בעזרת הטבלה הנתונה)

לא. בדוגמה למטה, מכיוון שלJ3 ולJ4 יש אותו שטח, לJ3 יש זמן ריצה קצר יותר, הוא יתוזמן קודם לכן. נשים לב כי אם היינו מכניסים קודם כל את J4 (מתחת לJ2), הנצילות שלנו הייתה משתפרת בהרבה.



שאלה 2 - החלפת הקשר (20 נקודות)

1. נתון הקטע הבא מתוך קוד הגרעין להחלפת הקשר:

```
01. movl prev, %eax
02. movl next, %edx
03. pushl %esi
04. pushl %edi
05. pushl %ebp
06. movl %esp, prev->thread.esp
07. movl next->thread.esp, %esp
08. movl $1f, prev->thread.eip
09. pushl next->thread.eip
10. jmp __switch_to
11. 1:
12. popl %ebp
13. popl %edi
14. popl %esi
```

א. מה יקרה אם נחליף את שורות 8-10 בשורה **call __switch_to** ?

כלום אם התהליך כבר זכה לרוץ. מכיוון שבסוף switch to יש פקודת return, בלי הcall אנחנו נקפוץ תמיד בחזרה לתווית $1 שדחפנו בשורה 09. אם נשנה את השורות 8-10 לקריאת פונקציה, התווית $1 תשמש גם במקרה זה לכתובת חזרה. אנחנו משתמשים בקפיצה במקום קריאה לפונ' בקטע קוד זה בעיקר לאופטימיזציה. **עם זאת**, במידה שהתהליך רק נוצר, אנחנו נרצה לקפוץ

לכתובת אחרת שהינה הכתובת שממנה תהליך האב שיצר את הבן בFork עצר, לכן ההתנהגות תהיה לא צפויה.

ב. נניח כי ביצענו קריאה למאקרו current לפני שורה 1 ,לאחר איזה שורה קריאה נוספת למאקרו תחזיר תשובה שונה?

לאחר שורה מס' 7, שכן בשורה זאת אנחנו מעדכנים את המצביע למחסנית. המאקרו משתמש במצביע זה כדי לשלוף את כתובת תחילת הpcb .

ג. מתרגל בקורס רצה לשפר את זמן הביצוע של החלפת ההקשר. מכיוון ששם לב כי המאקרו **switch_to** אינו עושה כלל שימוש ברגיסטר ecx ומאחר וידוע כי פעולות על רגיסטרים מהירות משמעותית מפקודות המערבות את הזיכרון, הציע להחליף את שורה **05** בשורה **mov %ebp, %ecx** ואת שורה 12 בשורה **mov %ecx, %ebp** האם הקוד יעבוד כראוי לאחר השינוי המוצע?

הקוד לא יעבוד שכן האוגר ecx מתעדכן בערכים אחרים לאורך כל הקודם עד שורה מס' 12. כאשר נגיע לשורה זאת, אף אחד לא מבטיח כי הערך שנדחף בשורה 5 יהיה שמור בecx, לכן הקוד לא בהכרח יעבוד.

2. כפי שראינו בתרגול מצביע לבסיס מחסנית הגרעין של התהליך הנוכחי נשמר גם ב **thread->esp0** וגם ב– **tss->esp0** האם ניתן לוותר על אחד מהם? אם לא הסבר מדוע ואם כן פרט על מי וכיצד נשיג את המידע הדרוש.

לא ניתן לוותר על המצביע tss->esp0 שכן אחרת אין לנו דרך לדעת מה הכתובת של המחסנית בגרעין של התהליך אשר ביצע syscall. כמו כן, ללא thread->esp0, לא נדע לגשת לבסיס המחסנית שלנו בעת החלפת ההקשר, כך שלא נוכל להחליף את tss->esp0. עם זאת, מכיוון שהpcb של התהליך נגיש בגרעין, נוכל באמצעות הoffset המתאים לגשת לtss->esp0, לכן לא בהכרח נצטרך להשתמש בthread->esp0.

שאלה 3 – זימון תהליכים (30 נקודות)

שאלה זו עוסקת במדיניות זימון התהליכים של לינוקס כפי שנלמדה בתרגולים.

לנוחיותכם מצורף חלק מהקוד כפי שנלמד בתרגולים:

```
#define MAX_PRIO 140
#define MIN_TIMESLICE (10 * HZ / 1000) /* 10 msec */
#define MAX_TIMESLICE (300 * HZ / 1000) /* 300 msec */
#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
if (prio < MAX_RT_PRIO)
prio = MAX_RT_PRIO;
if (prio > MAX_PRIO - 1)
prio = MAX_PRIO - 1;
BONUS(p) = 10*(SleepAvg/MaxSleepAvg-1/2)
DELTA(p) = 5*TaskNice(p)/20+2
```

א) (5 נק') האם ייתכן מצב בו תהליך אינטראקטיבי **A** יהיה באותה העדיפות הדינאמית כמו תהליך חישובי **B** ? אם כן תארו מצב כזה (תארו 2 תהליכים, את עדיפותם הסטטית, ואת עדיפותם הדינאמית - והוכיחו מספרית בעזרת חישובי **bonus** ו **DELTA**-שהתהליכים אכן מקיימים את הדרישות), אם לא נמקו מדוע.

לדוגמה: תהליך חישובי עם הנתונים הבאים:

Nice-20;Bonus=-2;Delta=-3;Prio=102

תהליך אינטרקטיבי:

Nice=-16;Bonus=2;Delta=-2;Prio=102

נשים לב כי המשתמש יכול בעצם לעשות מניפולציה באמצעות ערך הnice (כל עוד לא גורם לחריגת התנאי delta<=bonus) ובכך לגרום לתאריך אינטרקטיבי לקבל עדיפות דינאמית זהה לתאריך חישובי.

ב) (2 נק') בתרגול מצוינות **4** סיבות בגינן עשוי תהליך להגיע לפונקציה schedule. בחנו את הקוד של **sys_sched_yield** ו- **interruptible_sleep_on**. בכל אחת מהפונקציות הנ"ל ישנה קריאה ישירה לפונקציה **schedule** . אם היינו משנים קריאה זו ל-

set_tsk_need_resched(current) כיצד זה היה משפיע אם השינוי היה מתבצע רק ב-

sys_sched_yield? כיצד זה היה משפיע אם השינוי היה מתבצע רק ב-

interruptible_sleep_on ? נמקו (בתשובתכם התייחסו לנכונות הביצוע - כלומר האם התהליך אכן מוצא מהקשר? האם הקרנל ימשיך לעבוד בצורה תקינה? וכו')

עבור sched_yield, לא יקרה שום דבר, שכן בכל מקרה לפני החזרה לuser-space אנחנו נקפוץ ל()schedule בגלל הדגל. עבור interruptible_sleep_on, מכיוון שלאחר ()schedule מופיע מאקרו אשר מוציא את התהליך מרשימת ההמתנה (כאשר התהליך יתעורר), למעשה קטע הקוד לא יפעל כראוי כי התהליך ישוחרר באותה הפונ' מרשימת ההמתנה ולכן המטרה שלה לא תתבצע.

ג) לצורך שאלה זו נזכיר כיצד מחושב **sleep_avg** של תהליך בגרעין:

```
#define MAX_SLEEP_AVG (2*HZ)
sleep_time = jiffies - p->sleep_timestamp;
```

```
p->sleep_avg += sleep_time;
if (p->sleep_avg > MAX_SLEEP_AVG)
p->sleep_avg = MAX_SLEEP_AVG;

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

1. נניח שקיים תהליך אינטראקטיבי **A** בעדיפות סטטית 100 ובעל **sleep_avg=MAX_SLEEP_AVG**.

בנקודת זמן t=0 התהליך החל לבצע משימה חישובית ארוכה. מהו הזמן המקסימלי (במילישניות) שהתהליך ירוץ לפני שיעבו ר ל **expired** בהנחה שהוא התהליך היחיד ב runqueue?

כל עוד שהתהליך שלנו מסווג כאינטרקטיבי, הוא מקבל time_slice מחודש. תהליך נחשב לאינטרקטיבי כל עוד הנוסחה הבאה מתקיימת:

dynamic priority<=3*static_prio/4+28

או במילים אחרות, כאשר לתהליך שלנו יש יותר מms200 (שקול לבונוס הגבוהה מ2 ואנחנו יודעים כי ה**בונוס תלוי בזמן השינה**) של זמן שינה ממוצע, אזי הוא נחשב לאינטקטיבי. לפי הנתון, זמן השינה הממוצע שלנו שווה לHZ2 שזה 2000ms או 2 שניות. זה הרבה זמן. אם כן, על מנת שנרד מms200, נוכל לרוץ 2000-200=1800ms או 1.8 שניות.

2. כיצד תשובתכם לסעיף הקודם הייתה משתנה אם נתון שקיים תהליך **B** ב **expired** והחל מהרגע t=0 שתואר קודם, נותרו 1000 מילישניות עד כילוי ה**timestamp_expired** ? כלומר עד שהמאקרו **EXPIRED_STARVING** מתחיל להחזיר **TRUE**).

לפי הנתון של השאלה, לאחר 1000 מילישניות, המאקרו EXPIRED_STARVING יחזיר true ותתבצע הכנסת התהליך לתור הexpired. לפני כן, התהליך צריך לסיים את כל הtime_slice שלו, שהוא המקסימלי (ms300). כלומר, כשנגיע ל1000, כבר בזבזנו 100 מילי שניות, לכן קודם נסיים את ה200 שניות ולאחר מכן התהליך יכנס לexpired. בסה"כ: 1200 מילי-שניות

ד) מהו פרק הזמן המינימלי שיכול לעבור מהרגע שהודלק הדגל **need_resched** בתהליך שרץ ועד שהוא מגלה את הצורך בהחלפת הקשר (במערכת מרובת מעבדים)?

-נאמר במפורש בpiazza שסעיף זה לא רלוונטי

ה) האם אפשר לתת דוגמה בה תהליך יכול להתחיל לרוץ קצת אחרי פסיקת שעון ולעזוב את ה CPU קצת לפני פסיקת השעון הבאה (פחות מtick)? אם הדבר בלתי אפשרי הסבירו למה, אחרת תנו דוגמה מפורטת.

ראינו בסעיפים קודמים כי sched_yield מבקשת לוותר על זמן מעבד. ניקח לדוגמה תוכנית מאוד אינטרקטיבית המחכה לפקודה של משתמש כדי ללכת לישון. נניח כי לאחר פסיקת השעון, התוכנית חזרה לרוץ החל מהפקודה של נעילת המנעול ב()schedule. בהכרח היא תעזוב את המעבד לפני פסיקת השעון הבאה.