Operating Systems – 234123

Homework Exercise 2 – Dry

{ **Name: Roni Roitbord , ID: 313575599, email: roniro@campus.technion.ac.il**

Name: Amit Gabay , ID: 206040768, email: amitg@campus.technion.ac.il }

חלק 1 - שאלות בנושא התרגיל הרטוב (50 נק')

מומלץ לקרוא את הסעיפים בחלק זה לפני העבודה על התרגיל הרטוב, ולענות עליהם בהדרגה תוך כדי פתרון התרגיל הרטוב.

1. (6 נק') מה עושה פקודת yes בלינוקס? מה הארגומנטים שהיא מקבלת?
היעזרו ב-man page, ולאחר מכן השתמשו בפקודה ב-shell שלכן כדי לבדוק.
הפקודה מקבלת כקלט מחרוזת ומדפיסה אותה יחד עם ירידת שורה לפלט הסטנדרטי באופן חזרתי, שוב ושוב כל עוד התהליך לא קיבל SIGKILL.
במידה והפקודה נקראה ללא ארגומנטים, היא תפעל כאילו קיבלה את המחרוזת היחידה "y" כערך דיפולטיבי.

2. (6 נק') מדוע השתמשנו בפקודת yes עם מחרוזת ריקה במהלך הפקודה הבאה?

```
>> yes '' | make oldconfig
```

נסו להריץ את הפקודה make oldconfig לבדה והסבירו מה הבעיה בכך.
הפקודה make oldconfig דורשת (פעמים רבות יחסית) קבלת קלט נוסף מהמשתמש עבור התאמת הקוניפגורציה הרצויה לכל יוזר ויוזר.
לכן, בלינוקס קיימת האפשרות להקיש על מקש ה-ENTER ובכך להגיד לפקודה שתיקח את הערך הדיפולטיבי עבור השאילתא במקום קלט מהמשתמש, אבל עבור מספר רב של פעמים, גם זה יכול לדרוש הרבה לחיצות ENTER מיותרות מהמשתמש.
לכן, כשאנחנו קוראים ל-yes עם מחרוזת ריקה ומבצעים עליה pipeline לפקודה make oldconfig, התוצאה היא שימוש בערך הדיפולטיבי ENTER, כלומר הכנסת מחרוזות ריקות אינסופיות עד ש-make oldconfig תסיים את פעולתה, ללא הצרכת התערבות נוספת של המשתמש.

3. (6 נק') מה משמעות הפרמטר GRUB_TIMEOUT בקובץ ההגדרות של GRUB?

```
GRUB_TIMEOUT=5
```

הסבירו מה היתרונות ומה החסרונות בהגדלת הפרמטר GRUB_TIMEOUT.
הפרמטר GRUB_TIMEOUT מציין משך הזמן שבו ה-bootloader יחכה שהיזור יכניס כקלט את בחירתו עבור רצף האתחול, לפני שהמערכת תאותחל למערכת ההפעלה הדיפולטיבית. פרמטר זה נמדד בשניות.
יתרונות השימוש ב-GRUB_TIEMEOUT:

1. אינטראקציה עם ה-bootloade וגמישות: פרמטר זה מאפשר אינטראקציה של המשתמש עם ה-bootloader על ידי מתן חלון זמן לבחירת מערכת הפעלה או תצורה אחרת, מה שנותן למשתמש את היכולת לבחור מבין מערכות הפעלה מרובות המותקנות במחשב שלהם או לגשת לאפשרויות מתקדמות כגון מצב שחזור, ולהגדיר את הקונפיגורציות הנ"ל באופן מותאם אישית.
גמישות זו שימושית במיוחד כאשר היוזר צריך לעבור ממערכת הפעלה אחת לאחרת לעיתים קרובות.
2. התאמה אישית ויעילות: הפרמטר`GRUB_TIMEOUT` יכול להיות מועיל למטרות פתרון בעיות. אם המערכת נתקלת בבעיות אתחול, הגדרת ערך זמן ארוך יותר יכולה לספק זמן רב יותר לקרוא את הודעות השגיאה שמוצגות על ידי GRUB ולאבחן ולפתור אותן, או באופן כללי אם היוזר מעוניין בזמן ארוך יותר לצפייה בכל האפשרויות שיש למערכת להציע.
לעומת זאת, כשאין שגיאות או רצון לשהייה ארוכה יותר, ניתן להגדיר זמן קצר יותר, ובכך לשמור על יעילות ומהירות האתחול.
התאמת אינטרוול השניות לפי הפעולות שהמשתמש רוצה לבצע במהלך האתחול שומרת על אתחול אולטימטיבי מבחינת זמן, זאת לעומת הגדרת אינטרוול זמן יחיד וקבוע בו המשתמש

יוכל לתקשר עם ה-bootloader, שמן הסתם עבור חלק מהמשתמשים יהיה ארוך מדיי שלא לצורך, ועבור חלק קצר מדיי שלא לצורך.
חסרונות השימוש ב-GRUB_TIMEOUT:
בעוד שהפרמטר 'GRUB_TIMEOUT' בקובץ התצורה של GRUB מציע יתרונות, יש גם כמה חסרונות פוטנציאליים שיש לקחת בחשבון:

1. זמן אתחול ארוך יותר: אם נגדיר ערך זמן קצוב ארוך יחסית עבור `GRUB_TIMEOUT`, זה אומר שה-bootloader ימתין תקופה ארוכה יותר לפני אתחול אוטומטי של מערכת ההפעלה המוגדרת כברירת מחדל. אם משתמש שכח כיצד להגדיר זמן קצר יותר, או שהמקלדת שלו התקלקלה, זמן ההמתנה עד להתחלת רצף האתחול יכול להיות מייגע.
2. סיכון מוגבר לגישה לא מורשית: אם המחשב נגיש לאנשים אחרים, בין אם פיזית או מרחוק, ערך זמן קצוב ארוך יותר מגדיל את חלון ההזדמנויות של פעילות לא מורשת להפריע להתחלת רצף האתחול.
3. מורכבות למשתמשים לא מנוסים: קובץ התצורה של GRUB יכול להיות מורכב, ושינוי שגוי בפרמטר `GRUB_TIMEOUT` עלול להוביל לבעיות אתחול או אפילו להפוך את המערכת לבלתי ניתנת לאתחול. משתמשים לא מנוסים שאינם מכירים את תצורת GRUB צריכים לנקוט משנה זהירות בעת ביצוע שינויים כדי למנוע הגדרות שגויות לא מכוונות.

4. (6 נק') מדוע הפונקציה `run_init_process()` אשר נמצאת בקובץ `init/main.c` בקוד הגרעין קוראת לפונקציה `do_execve()` במקום לקריאת המערכת `execve()`?

```
static int run_init_process(const char *init_filename)
{
        argv_init[0] = init_filename;
        return do_execve(getname_kernel(init_filename),
                (const char __user *const __user *)argv_init,
                (const char __user *const __user *)envp_init);
}
```

נסו להחליף את הפונקציות זו בזו ובדקו האם הגרעין מתקמפל.
לאחר שהחלפנו את הקריאה לפונקציה `do_execve()` בקריאת המערכת `execve()`, הגרעין לא התקמפל והחזיר error:implicit decaleration עבור הקריאה ל-execve , כלומר לא קיימת הגדרה עבור פונקציה זו.
הסיבה היא ש-execve היא פונקציה שנועדה לשימוש מקוד משתמש, על מנת לאפשר ביצוע של קריאת מערכת שקוראת לגרעין.
אבל, הקובץ main.c הנ"ל נמצא בקוד הגרעין, כלומר, אנו רוצים לבצע את מה ש-execve() עושה, אבל מתוך הגרעין ולא על ידי קריאה חיצונית.
הדבר מתבצע מבוצע ע"י do_execve().

5. (6 נק') מה עושה קריאת המערכת `syscall()`? כמה ארגומנטים היא מקבלת ומה תפקידם? באיזו ספריה ממומשת קריאת המערכת `syscall()`? היעזרו ב-man page בתשובתכן.
קריאת המערכת syscall() היא פונקציית מעטפת המסופקת על ידי שפת C על מנת לאפשר הפעלה של קריאות מערכת בלינוקס ובמערכות הפעלה דמויות יוניקס.
פונקציה זו מאפשר לתוכנית ליצור אינטראקציה ישירה עם ליבת מערכת ההפעלה כדי לבקש שירותים או פעולות ספציפיות שאינן זמינות דרך קוד משתמש.
syscall() מבצעת קריאת מערכת לפי הפרמטר הראשון שהיא מקבלת, כלומר – אם קריאת המערכת דורשת x פרמטרים, syscall() תקבל $x+1$ פרמטרים, כשהראשון מבניהם מציין את מספר קריאת המערכת שיש להפעיל.
מספר הארגומנטים המדויק משתנה בהתאם לקריאת המערכת הספציפית המופעלת, והארגומנטים תואמים לפרמטרים הנדרשים על ידי קריאת המערכת.
הפונקציה syscall() ממומשת בספריית glibc.

6. (10 נק') מה מדפיס הקוד הבא? האם תוכלו לכתוב קוד ברור יותר השקול לקוד הבא?

```
int main() {
    long r = syscall(39);
    printf("sys_hello returned %ld\n", r);
    return 0;
}
```

רמז: התבוננו בקובץ `arch/x86/entry/syscalls/syscall_64.tbl` בקוד הגרעין.

לכל קריאת מערכת יש מספר ייחודי המשויך אליה, מספר 39 משוייך לקריאת המערכת getpid, שמחזירה את מזהה התהליך (PID) של התהליך הקורא.

לכן, הקוד הנ"ל מדפיס: sys_hello returned <PID>, כאשר <PID> הוא ה-pid של התהליך הקורא.

נוכל לכתוב את אותו הקוד ולהחליף את שורת הקוד syscall(39) בפונקציית המעטפת הייעודית getpid(), המחזירה בדיוק את אותו הפלט, אך הופכת את הקוד לקל יותר להבנה.

7. (10 נק') התבוננו בתוכנית הבדיקה `test1.c` שסופקה לכן והסבירו במילים פשוטות מה היא בודקת:

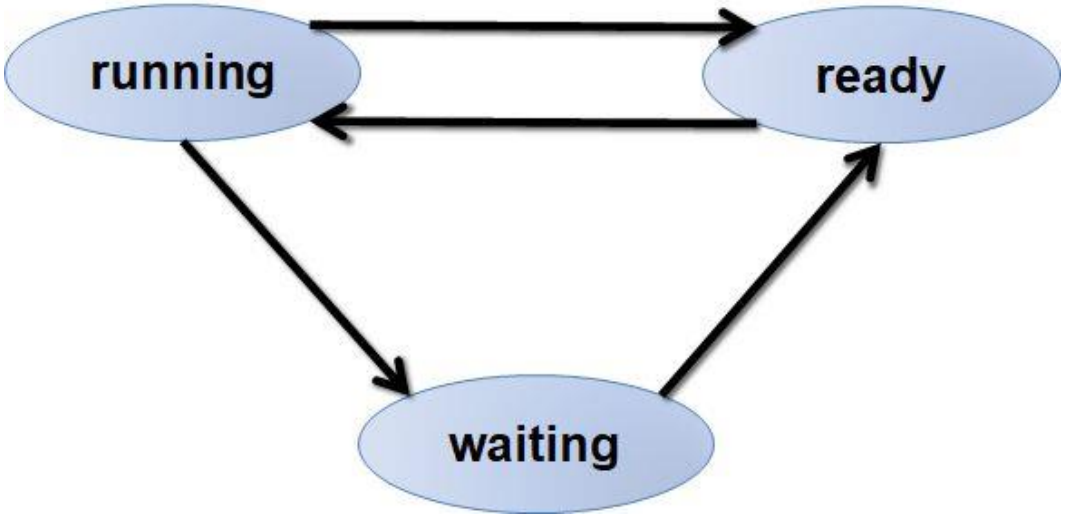
```
int main() {
    int x = get_weight();
    cout << "weight: " << x << endl;
    assert(x == 0);
    x = set_weight(5);
    cout << "set_weight returns: " << x << endl;
    assert(x == 0);
    x = get_weight();
    cout << "new weight: " << x << endl;
    assert(x == 5);
    cout << "===== SUCCESS =====" << endl;
    return 0;
}
```

התוכנית בודקת שמשקל התהליך הנבדק אכן התחיל עם המשקל הדיפולטיבי 0, לאחר מכן משנה את המשקל של התוכנית להיות 5, מדפיסה את ערך החזרה של פעולת שינוי המשקל, מוודאה שהמשקל השתנה ל-5 והפעולה אכן הצליחה ומדפיסה SUCCESS.

חלק 2 - זימון תהליכים (50 נק')

נא לנמק את תשובותיכם לכל הסעיפים

1. נתון התרשים המופשט של מצבי התהליך:



עבור כל מעבר תנו תרחיש המוביל לאותו מעבר:

a. running→ready: החלפת הקשר- במקרים מסוימים מערכת ההפעלה יכולה להתערב ולעצור ריצה של תהליך באמצע, למשל בביצוע הפקעה (הגיע תהליך יותר קצר או בעדיפות גבוה יותר), או שהתהליך סיים את פיסת הזמן שלו שהוגדרה מראש (quantum באלגוריתם RR למשל) - מערכת ההפעלה תיכנס לפעולה ותעצור את ריצת התהליך, כשהיא מעבירה אותו ממצב running למצב ready.

b. ready→running: למשל אם מעבד התפנה והתהליך הבא להרצה בתור התהליכים הוא התהליך במקום הראשון שנמצא במצב ready, אלגוריתם הזימון יכניס אותו לריצה ויהפוך אותו ממצב ready למצב running.

c. running→waiting: כאשר תהליך נמצא במצב running ומבקש I/O – אלגוריתם הזימון יעביר אותו למצב waiting ויזמן תהליך אחר במקומו.

d. waiting→ready: כאשר תהליך נמצא במצב waiting כי הוא מחכה לפלט מאיזשהו התקן (I/O למשל), כאשר ההתקן מסיים את פעולתו הוא שולח פסיקה למעבד שהדאטה שהתהליך ביקש מוכן ואז אלגוריתם הזימון יחזיר את התהליך למצב מוכן לריצה - ready (לא בהכרח יריץ אותו).

2. נתון שהמערכת עובדת עם זמן תהליך מסוג RR (round robin):

a. מה היתרון בשימוש ב quantum גדול?
בסופו של כל קוונטום מתבצעת החלפת הקשר על מנת להחליף בין התהליך שסיים את פיסת הזמן שהוקצתה לו לרוץ לבין התהליך הבא.
ככל שהקוונטום גבוה יותר, כל תהליך ירוץ זמן רב יותר לפני שתתרחש החלפת הקשר, ולכן נקבל פחות החלפות הקשר (שלוקחות זמן ומשאבים) בסך הכל, ולכן נקבל ביצועים טובים יותר.

b. מה היתרון בשימוש ב quantum קטן?
ככל שהקוונטום יותר קטן, זמן ההמתנה למעבר בין תהליך לתהליך נמוך יותר, מה שיוצר תחושת מקבול טובה יותר אצל המשתמש ומגביר את האינטראקטיביות.

c. במידה והמערכת עמוסה (מכילה הרבה תהליכים מוכנים לריצה), מדוע עדיף להוסיף תהליכים חדשים בסוף התור?
באלגוריתם RR הוספת תהליכים חדשים בסוף התור כשהמערכת עמוסה עדיפה מהסיבות הבאות:

1. הגינות – שמירה על סדר הגעה: הוספת תהליכים חדשים בסוף התור שומרת על הסדר המקורי בו הגיעו, מה שעוזר לשמור על תחושת הגינות ומונע מכל תהליך ספציפי לקבל באופן עקבי יחס מועדף. כל תהליך יתוזמן בצורה סיבובית, ויבוצע לפיסת זמן קבועה לפני המעבר לחלק האחורי של התור ויאפשר לתהליך הבא לפעול. סדר ההגעה נשמר לאורך כל התהליך הנ"ל.
2. מניעת הרעבה: הוספת תהליכים חדשים בסוף התור מבטיחה שאף תהליך לא ימתין ללא הגבלת זמן לקבלת

משאבי CPU. תהליכים חדשים היו מתווספים בקדמת התור, במערכת עמוסה תהליכים קיימים עלולים להידחק שוב ושוב לסוף התור ולעולם לא יקבלו הזדמנות לרוץ.

3. הגינות - תיעדוף תהליכים ממתינים: במערכת עמוסה עם מספר תהליכים מוכנים, הוספת תהליכים חדשים בסוף התור פירושה שלתהליכים שחיכו זמן רב יותר תהיה עדיפות גבוהה יותר לביצוע. זה מונע מתהליך שזה עתה הגיע לקפוץ מיידית לראש התור לפני אחרים שחיכו לתורם תקופה ארוכה יותר.

3. בזמן תהליכים (completely fair scheduler) CFS, איזו בעיה פותרת ה min_granularity?

אלגוריתם CFS קובע טווח זמן שבמהלכו הוא ינסה להריץ את כל התהליכים $sched\_latency = 48\,ms$, ומקצה לכל תהליך פיסת זמן שבה הוא מקבל את המעבד. כלומר, אם יש N תהליכים במערכת וכולם באותה עדיפות, הקוונטום של כל תהליך הוא: $Qi = \frac{sched\_latency}{N}$.

הבעיה היא, שבמערכת עמוסה, שבה מספר התהליכים N במערכת גבוה, המערכת עלולה לסבול מהחלפות הקשר תכופות ופגיעה בביצועים. לכן מוגדר גם זמן מינימום על הקוונטום:

$$Qi \geq min\_granularity = 6\,ms$$

לדוגמה, אם יש 20 תהליכים במערכת, אז הקוונטום של כל אחד אמור להיות: $Qi = \frac{48}{20}\,ms = 2.4\,ms$, בפועל, כל תהליך יקבל $min\_granularity = 6\,ms$, ולכן משך הסיבוב שבו כל התהליכים ירוצו יהיה: $epoch = 120\,ms \geq sched\_latency$.

4. במערכת עם ליבה אחת, בה כל התהליכים מגיעים יחד וזמני הריצה שלהם ידועים מראש. איזה אלגוריתם batch scheduling (כלומר בלי הפקעות תהליכים) ימזער את ה- average response time (זמן התגובה הממוצע)?

a. RR (round robin) algorithm
b. FCFS (first come first serve) algorithm
c. SJF (shortest job first) algorithm
d. EASY (FCFS + back-filling) algorithm

הוכחה: תהי מערכת עם ליבה אחת, בה כל התהליכים מגיעים באותו זמן וזמן הריצה של כל תהליך ידוע מראש, ונניח שקיים אלגוריתם A שאינו SJF, שממזער את זמן התגובה הממוצע.
נשים לב שאלגוריתם SJF מסדר את התהליכים בצורה ממוינת מבחינת זמן ריצה (=Shortest Job First).
עבור כל סידור אחר, התהליכים לא יהיו ממוינים, ולכן קיימים שני תהליכים צמודים: P_i, P_j , כך שהאחד שרץ לפני רץ זמן ארוך יותר מזה שירוץ אחריו, כלומר, סדר ההרצה הוא P_i, P_j ומתקיים:

$$Runtime(P_j) > Runtime(P_i)$$

כעת, נחליף בין סדר ההרצה של התהליכים, כך שתהליך P_j ירוץ קודם, ותהליך P_i ירוץ אחריו. נקבל:

- זמן התגובה של תהליכים 1 עד $i-1$ נשאר אותו דבר.
- זמן התגובה של תהליך i הוא זמן התגובה כפי שהיה קודם ועוד זמן הריצה של תהליך j.
- זמן התגובה של תהליך j הוא זמן התגובה הקודם שלו פחות זמן הריצה של i.

לכן זמן התגובה הכולל תחת הסידור החדשה שווה לזמן התגובה הכולל תחת הסידור הישן בתוספת הביטוי

$$Runtime(P_j) - Runtime(P_i)$$

מההנחה שלנו מתקיים כי זהו ביטוי שלילי, שכן זמן הריצה של P_j ארוך יותר, ולכן זמן התגובה החדש קצר יותר.
כעת, נוכל לחזור על פעולה זו עבור כל זוג התהליכים שהתהליך המגיע אחרי בסידור רץ זמן ארוך יותר מהתהליך המגיע לפניו (לכל היותר n^2 פעמים), עד שלא יהיה יותר מה להחליף ונקבל

רשימה ממוינת של תהליכים.
כל פעולה כזאת מקטינה את הזמן הריצה של הסידור ובסופו של דבר מתכנסים לסידור הממוין - SJF.
לכן, התחלנו מאלגוריתם טוב יותר מ-SJF, מבחינת זמן ריצה ממוצע, שיפרנו אותו לכל היותר n^2 פעמים והגענו ל-SJF .
כלומר, זמן הריצה הממוצע של SJF גדול יותר מעצמו.
וזו סתירה.
לכן, אלגוריתם SJF אכן ממזער את זמן ההמתנה הממוצע.

5. נגדיר מערכת בעלת 3 ליבות (המסוגלות להריץ תהליכים במקביל בהתאם לדרישות של התהליכים), בה ישנם אך ורק 4 תהליכים המעוניינים לרוץ:

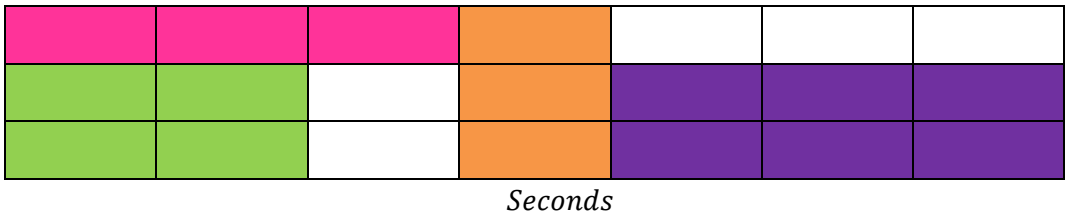
תהליך 1 דורש 2 ליבות וירוץ למשך 2 שניות עד לסיום.
תהליך 2 דורש 1 ליבות וירוץ למשך 3 שניות עד לסיום.
תהליך 3 דורש 3 ליבות וירוץ למשך 1 שניות עד לסיום.
תהליך 4 דורש 2 ליבות וירוץ למשך 3 שניות עד לסיום.

התהליכים נשלחים למעבד בסדר זה. איזה אלגוריתם יגרום לסיום כל התהליכים ראשון? נמקו.

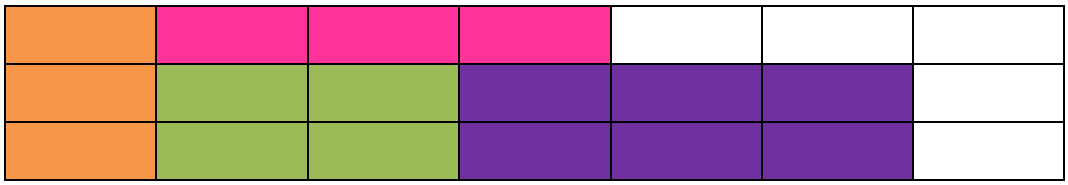
a. FCFS
b. SJF
c. EASY
d. תשובות b ו c נכונות.

נסמן את תהליך 1 בירוק, את תהליך 2 בוורוד, את תהליך 3 בכתום ואת תהליך 4 בסגול:

FCFS and EASY:



SJF:



$Seconds$

לכן קיבלנו שב-SJF כל התהליכים יסתיימו שניה אחת מוקדם יותר מב-FCFS, EASY.

6. במערכת בה תהליכים מגיעים בזמנים שרירותיים, באיזה אלגוריתם תזמון נעדיף להשתמש – SRTF או SJF כדי לקבל את ה- average response time (זמן התגובה הממוצע) הקטן ביותר? מדוע?

נעדיף להשתמש באלגוריתם $SRTF$ מפני ש שאלגוריתם של SJF הוא אלגוריתם מסוג Batch, ולכן, ברגע שהוא התחיל להריץ משהו הוא לא עושה שינויים.

לעומת זאת, אלגוריתם $SRTF$ טומן בחובו את האפשרות לארגן מחדש את סדר הריצה בכל פעם שתהליך חדש מצטרף ובכך לשנות החלטות קודמות.

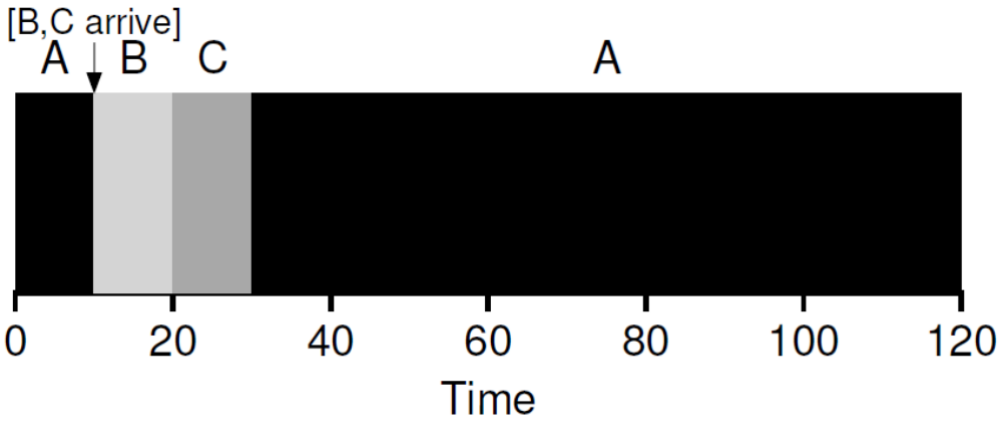
לכן, במערכת בה צפויים להגיע תהליכים בזמנים לא ידועים, נרצה אלגוריתם שיחשב בכל פעם מחדש כמה לכל תהליך נשאר עוד לרוץ ויריץ תמיד את הקטן ביותר – $SRTF$

לדוגמה, עבור תהליך A המגיע בזמן $t = 0$ ורץ למשך 100 שניות. ותהליכים B, C המגיעים בזמן t=10 ומבקשים לרוץ 10 שניות כל אחד מתקיים:

$$avg\_response\_time_{SJF}(A,B,C) = \frac{100 + (90 + 10) + (90 + 10 + 10)}{3} = \frac{310}{3}\ secs$$

$$avg\_response\_time_{SRTF}(A,B,C) = \frac{(10 + 110) + (10) + (20)}{3} = \frac{150}{3}\ secs < avg\_response\_time_{SJF}(A,B,C)$$

ואכן, זמן התגובה הממוצע קטן מפני שאלגוריתם $SRTF$ חישב למי מבין התהליכים (כולל 2 התהליכים החדשים) נותר הכי פחות זמן לרוץ, ובחר את התהליך הזה לריצה ברגע הגעתו (**בכל פעם שמגיע תהליך חדש למערכת**):



* יש לשים לב שהנ"ל נכון תחת ההנחה כי זמן החלפת ההקשר הוא יחסית זניח, שכן אם הזמן של החלפות ההקשר גבוה יחסית, אלגוריתם $SRTF$ מבצע 3 החלפות, בעוד שאלגוריתם SJF לא מבצע כלל, מה שעלול לגרור זמן תגובה ממוצע קצר יותר של SJF ולא של $SRTF$.