מערכות הפעלה - תרגיל בית יבש 2

Ido Tausi, *214008997,* ido.tausi@campus.technion.ac.il
Noam Bitton, *213745953*, bitton.noam@campus.technion.ac.il

חלק 1

1. הפקודה *yes* בלינוקס מדפיסה מחרוזת נתונה ואחריה ירידת שורה אינסוף פעמים עד הפסקת/ עצירת התהליך.
במידה והפקודה מקבלת ארגומנט אז המחרוזת המודפסת אינסוף פעמים היא המחרוזת שהתקבלה ובמידה שהפקודה לא מקבלת ארגומנט המחרוזת המודפסת אינסוף פעמים היא *'y'* וירידת שורה.

2. כאשר מבצעים *make oldconfig* תתבצע בנייה מחדש של גרעין הלינוקס, ותוך כדי הבנייה יהיה צורך בקלט מן המשתמש עבור ההגדרות גרעין הלינוקס שהוא בונה כעת. השימוש בפקודה *yes* עם מחרוזת ריקה שמחוברת בפייפ לערוץ הקלט של *make oldconfig*, תגרום לתוכנית הבונה לקרוא רק את תו השורה החדשה בכל פעם שהיא תבקש קלט מן המשתמש, דבר שיגרום לבחירה של הערך הדיפולטי בכל פעם, לכן פקודה זו תעזור למשתמשים שרוצים לבנות את הגרעין מחדש עם הערכים הדיפולטיביים. פקודה זו תאיץ את ההרצה מפני שלא תעצור את הבנייה ולא תבזבז זמן בהמתנה למשתמש שיזין את הקלט.
בהרצת *make oldconfig* לבדה נקבל שהתהליך ייחכה לקלט מהמשתמש ולא ייעשה דבר עד אשר ייקבל קלט.

3. הפרמטר *TIMEOUT_GRUB* בקובץ ההגדרות של *GRUB* מגדיר את מספר השניות שבהן יוצג למשתמש תפריט ה-*GRUB* שדרכו ניתן לבחור איזה מערכת הפעלה תיטען ובסופן במידה והמשתמש לא בחר, תיבחר מערכת ההפעלה שמוגדרת כדיפולטיבית.
היתרונות בהגדלת הפרמטר *TIMEOUT_GRUB*:

- עבור מקרים בהם יש מספר מערכות הפעלה שונות שמותקנות על המחשב, זמן המתנה ארוך יותר יכול לעזור למשתמש לראות את האפשרויות העומדות לרשותו ולבחור את המערכת אותה הוא רוצה לטעון, לפני שהזמן ייגמר והמערכת תבחר בשבילו את המערכת הדיפולטית.
- הגדלת הפרמטר יכולה לעזור גם במקרים בהם ישנה שגיאה בטעינת מערכת הפעלה, ורוצים יותר זמן בו ניתן לגשת להגדרות השחזור של מערכת ההפעלה ולהבין את הבעיה ולאחר מכן לנסות לתקן את התקלה.

החסרונות בהגדלת הפרמטר *TIMEOUT_GRUB*:

- המתנה ארוכה יותר של המשתמש לעליית מערכת ההפעלה, פוגעת בנוחות בעיקר של משתמשים שלרוב מעוניינים במצב ברירת המחדל.
- עשוי להגדיל את הסכנה שמשתמש זדוני ייגש למערכת במצב רגיש ובכך מגדיל את הסיכוי לבעיות אבטחה.

4. *execve()* היא פונקציית מעטפת לקריאת מערכת, הנמצאת בספרייה בצד של המשתמש (ולא נמצאת בגרעין) ונקראת על ידי קוד המשתמש. ואילו *do_execve()* היא פונקציית גרעין, אשר משתמשים בה בתוך הגרעין לאחר הקריאה לקריאת המערכת *execve()* מקוד המשתמש. למעשה, קריאת המערכת *execve()* מיועדת למשתמש אשר נמצא ברמת הרשאה של משתמש ולכן אין לו גישה לפונקציות גרעין כמו *do_execve()* ולעומת זאת כשמריצים את *process_init_run()*, מריצים אותו מתוך קוד הגרעין ולכן אין צורך בשימוש בקריאת המערכת ברמת הרשאת משתמש, כשניתן להשתמש בפונקציות גרעין מפני שרמת ההרשאה במצב זה היא כבר רמת הרשאת גרעין.
פונקציית *process_init_run()* מבוצעת בקוד הגרעין ולכן תשתמש בפונקציות גרעין ולא בפונקציית משתמש, ובנוסף פונקציות אלו לא קיימות בתוך קוד הגרעין,אלא רק בספרייה בצד של המשתמש, למשל *execve()* לא קיימת בקוד הגרעין. לכן הפונקצייה *process_init_run()* תקרא ל- *do_execv()* שהיא פונקציית גרעין.
בנוסף, אם נחליף בין הפונקציות הללו, הקוד לא יתקמפל מפני שבקוד הגרעין אין גישה לספריה libc ולכן קריאת המערכת *execve()* לא קיימת עבור הגרעין, ולכן תיווצר שגיאת קומפילציה.

5. קריאת המערכת *syscall()* היא פונקציית מעטפת ברמת הרשאת משתמש לביצוע פקודת ה-syscall בצורה ישירה עם מזהה פקודת המערכת הרצויה. הפונקצייה שומרת את הרגיסטרים, לפני ביצוע קריאת המערכת, ולאחר הקריאה היא משחזרת אותם.
הפונקציה *syscall()* מקבלת ארגומנט אחד מטיפוס *long* של קוד קריאת המערכת שברצונו של המשתמש לבצע, ועוד מספר לא מוגבל של ארגומנטים מפני שיש ... בחתימת הפונקציה. מצהירים על קריאה זו ב- *unistd.h* והיא ממומשת בספריה *glibc*.

6. הקוד הנ"ל מדפיס : " $sys\_hello\ returned\ \ < pid >$ " כאשר $< pid >$ הוא ה-pid של התהליך הנוכחי המתקבל מקריאה לקריאת המערכת 39 שהיא למעשה sys_getpid.
ניתן במקום לכתוב את הקוד:

```
int main() {
 printf("sys_hello returned %ld\n", getpid());
 return 0;
}
```

7. התכנית שומרת במשתנה x את המשקל של התהליך הנוכחי ומוודאת שערכו 0.
לאחר מכן, היא משנה את ערכו ל-5 ומוודאת שערך החזרה של הפונקצייה הוא 0, כלומר שקריאת המערכת הצליחה.
לבסוף, היא שומרת במשתנה x את המשקל החדש ומוודאת שהוא אכן 5.
אם הגענו לסוף התכנית, כלומר כל ה-$assert$ עברו כמצופה, יודפס " $=====\ \ SUCCESS\ \ =====$ " ונחזיר 0 מהפונקצייה.

חלק 2

.1

a) תהליך 1 רץ כאשר התקבלה פסיקת שעון והתבצעה החלפת הקשר, כעת הוא במצב $ready$ ומוכן לרוץ שוב.
b) זמן התהליכים מחליט שהגיע תורו של התהליך לרוץ, ולכן מתזמן אותו לרוץ על המעבד, לאחר פסיקת השעון הבאה.
c) תהליך רץ מבקש לבצע פעולות I/O, ויוצא להמתנה.
d) הגעת הנתונים לתהליך שהיה במצב $waiting$ כי ביקש לבצע פעולות I/O, יגרמו לכך שמערכת ההפעלה תעורר את התהליך ותעביר אותו למצב $ready$, להמתנה לתורו לרוץ.

.2

a) היתרון בשימוש ב-$quantum$ גדול הוא הקטנת כמות החלפות ההקשר, שמקטינה את התקורה הנגרמת מהרצת $context\ switch$ וכתוצאה מכך נקבל ניצול טוב יותר של המעבד.
b) היתרון בשימוש ב-quantum קטן הוא שזמן ההמתנה של כל תהליך הוא נמוך יותר, ולכן יש יותר אינטראקטיביות בעת הרצת התהליכים.
c) הוספת תהליכים בסוף התור שומרת על הוגנות מכיוון שתהליכים שהגיעו קודם יקבלו הזדמנות לרוץ ירוצו קודם. בנוסף, הוספת תהליכים בתחילת התור עלולה ליצור הרעבה אם תהליכים חדשים ממשיכים להגיע והתהליכים בסוף התור לא זוכים לרוץ.

.3 הבעיה שה-$min\_granularity$ היא שבמקרה שבו מספר התהליכים במערכת גבוה, המערכת עלולה לסבול מהחלפת הקשר תכופות ופגיעה בביצועים עקב זמן Q_i קטן מידי שעבורו התקורה של החלפת ההקשר אינה משתלמת. לכן, אם יוצא בחישוב $Q_i = \frac{sched\_latency}{N}$ שקטן מהמינימום, ניתן לאותו התהליך את $min\_granularity$.

4. תשובה c.

לפי הנלמד בהרצאה, תחת התנאים המוגדרים בשאלה, האלגוריתם שימזער את *average response time* הוא *SJF (shortest job first) algorithm*.

נציג דוגמה:

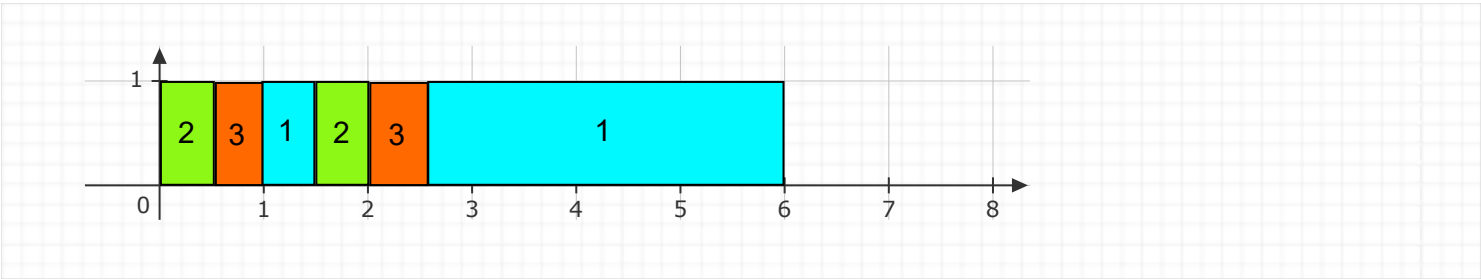
$$responseTime = terminationTime - arrivalTime$$

נתבונן בדוגמה הבאה- בהינתן שלושת התהליכים הבאים המגיעים באותו הזמן:

- תהליך 1 המבקש לרוץ 4 שניות.
- תהליך 2 המבקש לרוץ שנייה 1.
- תהליך 3 המבקש לרוץ שנייה 1.

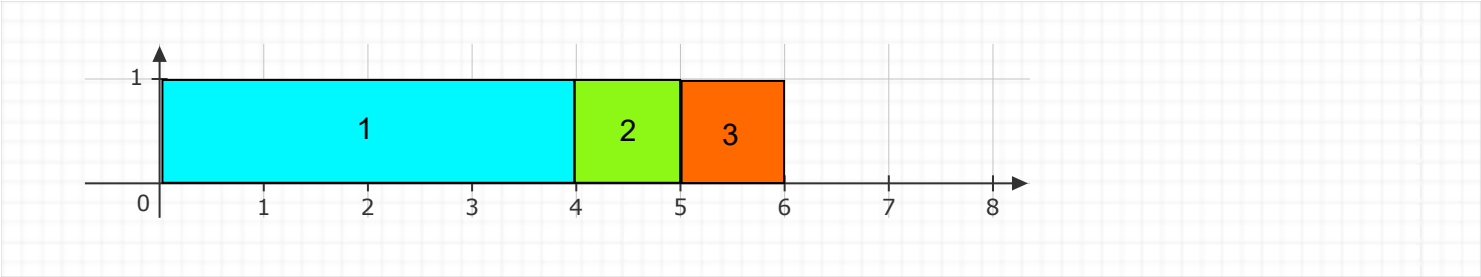
נמצא את ה-*average response time* עבור כל אלגוריתם:

:RR (round robin) algorithm



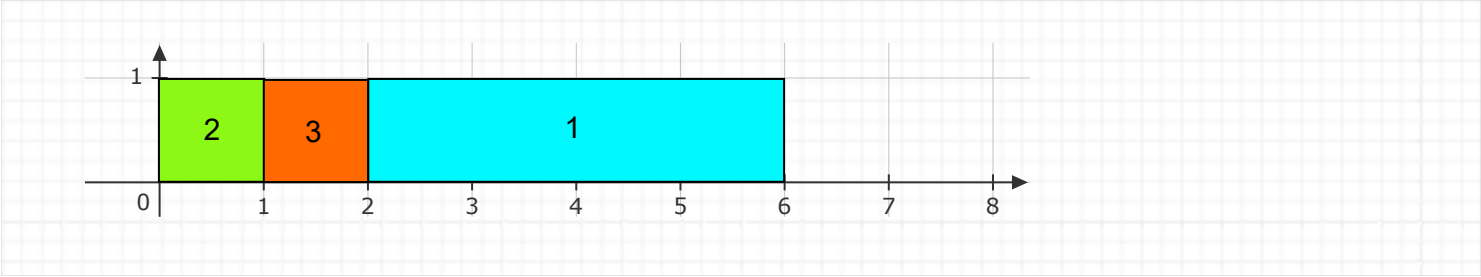
$$averageResponseTime = \frac{6+2+2.5}{3} = 3.5\ sec$$

:FCFS (first come first served) algorithm



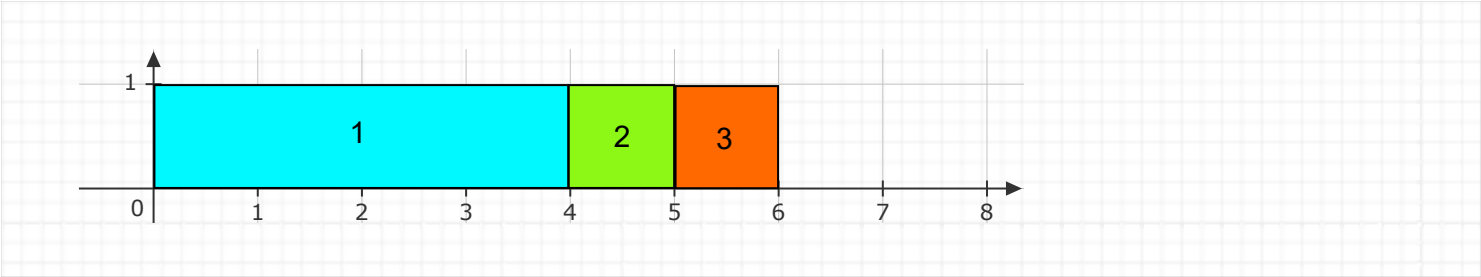
$$averageResponseTime = \frac{4+5+6}{3} = 3.5\ sec$$

:SJF (shortest job first) algorithm



$$averageResponseTime = \frac{1+2+6}{3} = 3\ sec$$

EASY (FCFS + back-filling) algorithm

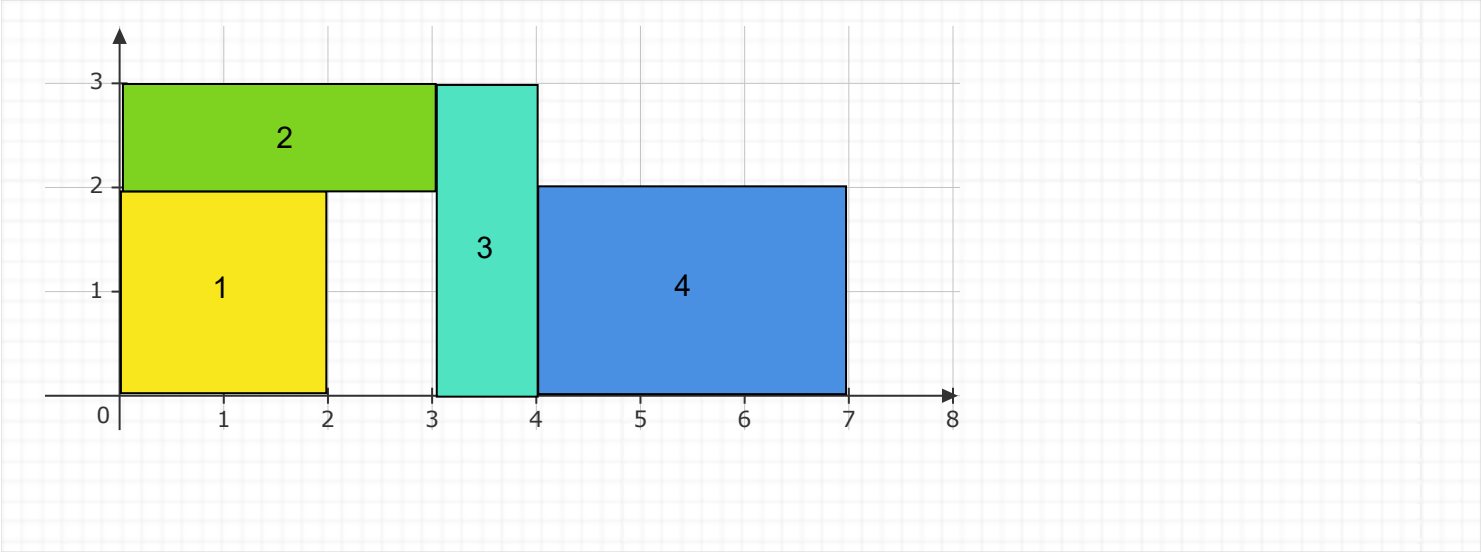


$$averageResponseTime = \frac{4+5+6}{3} = 3.5\ sec$$

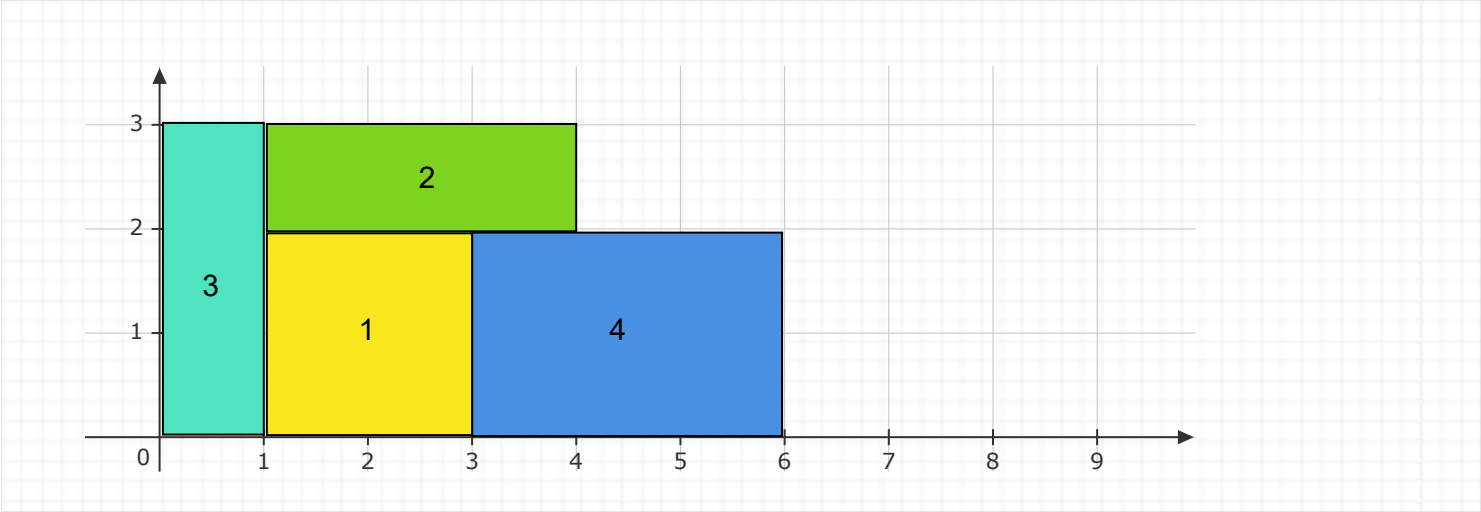
קיבלנו כי עבור דוגמה זו ה-$AverageResponseTime$ של אלגורימים $FCFS,\ RR,\ EASY$ ארוך משל אלגוריתם SFJ ולכן הם בהכרח לא ממזערים את זמן התגובה.
לכן, אלגוריתם ה-SFJ ימזער את זמן התגובה.

5. תשובה b.

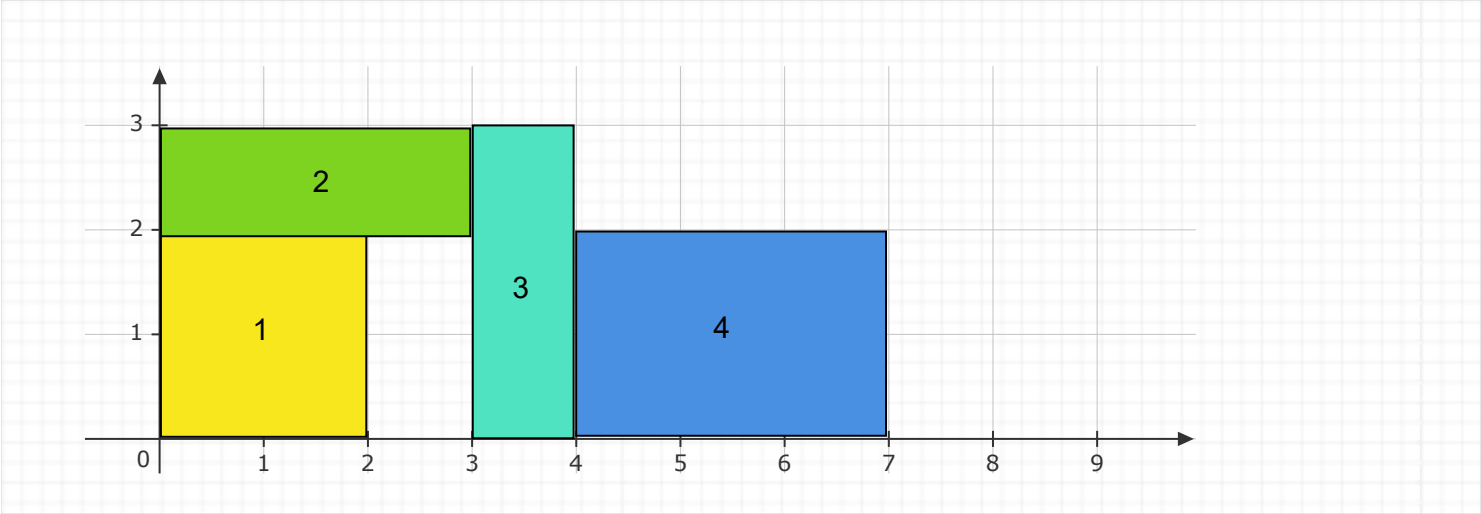
באלגוריתם $FCFS$ האלגוריתם יסיים לרוץ בתוך **7 שניות**



באלגוריתם SJF האלגוריתם יגרום לסיום כל התהליכים בתוך **6 שניות**.



באלגוריתם $EASY$, האלגוריתם יגרום לסיום כל התהליכים בתוך **7 שניות**.

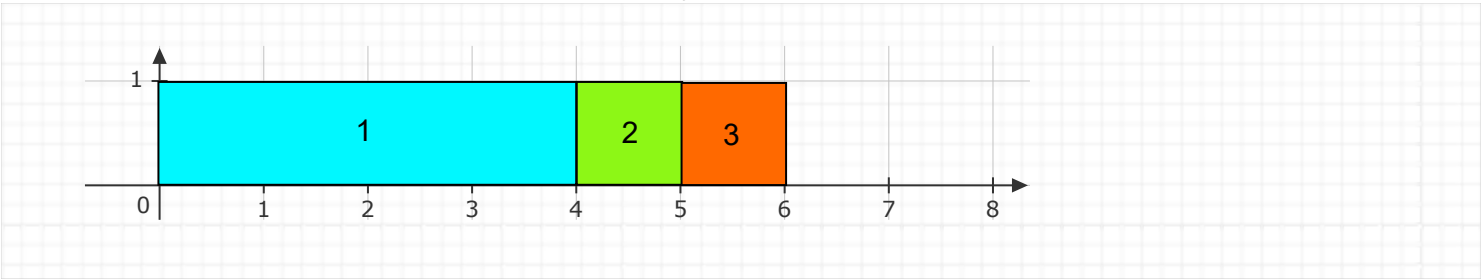


לכן, אלגוריתם ה-SJF יגרום לסיום כל התהליכים ראשון.

6. בהנחה שזמן החלפת הקשר הוא זניח (נמוך ביחס לזמן הריצה שאותו מבקשים התהליכים) במערכת בה תהליכים מגיעים בזמנים שרירותיים נעדיף להשתמש באלגוריתם $SRTF$ מפני שיכול להיווצר מצב שבו הגיעה ראשונה עבודה שלוקחת זמן רב מאוד, ומעט זמן אחריה הגיעו הרבה עבודות שלוקחות זמן קצר. במקרה זה, אלגוריתם SJF יריץ קודם את העבודה הארוכה, ויגרום לכל העבודות הקצרות לחכות עד לסיום המטלה הארוכה, דבר שיאריך את זמן ההמתנה הממוצע של עבודה מרגע הגעתה עד לרגע התחלת ריצתה.
לעומת זאת, באלגוריתם $SRTF$, כאשר מגיעה עבודה שזמן ריצתה קטן יותר מזמן ריצת העבודה הנוכחית ולכן נותר לה פחות זמן לרוץ והעבודה הקצרה תתחיל לרוץ ותסיים לפניה. בצורה זו, כל העבודות הקצרות ירוצו ויסיימו ורק לאחר מכן תחזור העבודה הארוכה לרוץ ותסיים, וכך זמן התגובה הממוצע שהוא הזמן בין ההגעה לסיום המשימה, יקטן משמעותית.

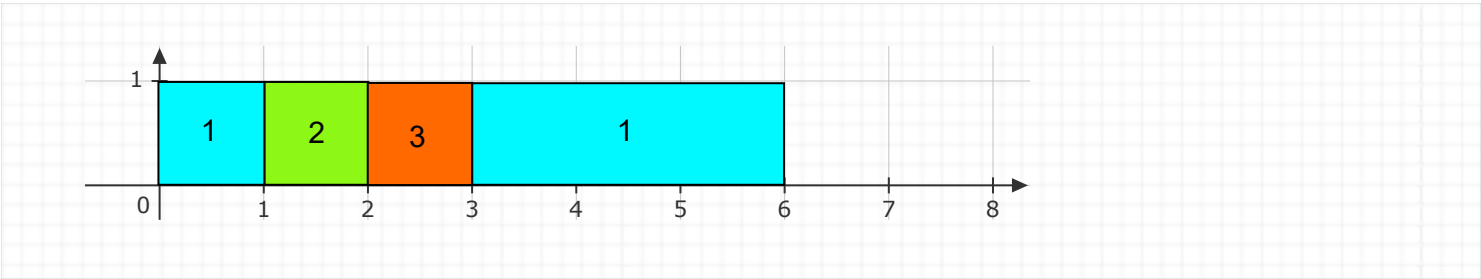
נתבונן בדוגמה הבאה כאשר זמן החלפת ההקשר הוא זניח:
בזמן $t = 0$ מגיע תהליך 1 המבקש לרוץ 4 שניות.
בזמן $t = 1$ מגיעים תהליכים 2, 3 כך שכל אחד מהם מבקש לרוץ שנייה אחת.

אלגוריתם SJF:



$$averageResponseTime = \frac{(4-0)+(5-1)+(6-1)}{3} = 4.333\ sec$$

אלגוריתם SRTF



$$averageResponseTime = \frac{(6-0)+(2-1)+(3-1)}{3} = 3\ sec$$

אך אם נניח כי זמן החלפת ההקשר גבוה כך שבהינתן מספיק תהליכים, השימוש בהפקעות והצורך בהחלפות הקשר מרובות יגרום ל-$SRTF$ להיות איטי בהרבה מ-SJF. לכן, תחת ההנחה הזו נרצה להשתמש באלגוריתם SJF כדי למזער את זמן התגובה הממוצע הקטן ביותר.