Operating Systems – 234123

Homework Exercise 2 – Dry

מגישים:

נועה פריאנטה 206200305

pariente@campus.technion.ac.il

ששון שמואל למעי 325172351

sason-shmuel@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.
תשובה: הפקודה yes מקבלת מחרוזת כארגומנט ומדפיסה את המחרוזת אינסוף פעמים (כל פעם בשורה חדשה) עד שהתהליך נעצר (למשל על ידי שליחת סיגנל להרוג ctrl+C). אם אינה מקבלת ארגומנט הינה מדפיסה 'y' אינסוף פעמים עד שהתהליך נעצר.

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.
תשובה: כאשר הרצנו make oldconfig לבד, הפקודה לא רצה מכיוון שהיא אמרה שהיא מצפה לתשובות מהמשתמש לגביי האם להעביר קונפיגורציות שונות מהKERNEK הישן. כאשר אנו משתמשים ב-' ' yes כפי שציינו הוא מדפיס אינסוף פעמים את המחרוזת ' ' על ידי שימוש ב-pipe (מסומן ב-|) אני אומרים לפקודה yes שהערוץ פלט שלה הינה הפקודה לאחר מכן, כלומר make oldconfig ובכך אנו כאילו עונים לTERMINAL ועוברים לו 'כן' להכל.

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

```
GRUB_TIMEOUT=5
```

הסבירו מה היתרונות ומה החסרונות בהגדלת הפרמטר GRUB_TIMEOUT.
תשובה: GRUB_TIMEOUT זה משך הזמן (בשניות) המוצג על המסך את התפריט של GRUB (מאפשר לבחור באיזה מערכת הפעלה וכו' להשתמש) לפני שהמערכת מאתחלת אוטומטית למה שמוגדר כברירת מחדל שלה.
הערך -1 משמעותו הצגה ללא הגבלת זמן של התפריט והערך 0 טעינה של מערכת ההפעלה הדיפולטית.
יתרונות: נותן זה לבחור איזה מערכת הפעלה לעלות
חסרונות: מאט את עליית מערכת ההפעלה

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

```
944  static int run_init_process(const char *init_filename)
945  {
946        argv_init[0] = init_filename;
947        return do_execve(getname_kernel(init_filename),
948              (const char __user *const __user *)argv_init,
949              (const char __user *const __user *)envp_init);
950  }
```

נסו להחליף את הפונקציות זו בזו ובדקו האם הגרעין מתקמפל.
תשובה: ראשית כדי לענות את השאלה נציין מה ההבדל בין שתי הפונקציות:
execve() הינה פונקציית POSIX קריאת מערכת לשימוש של תוכניות C במצב userspace
do_execve() הינה פונקציית גרעין, כלומר ניתנת להרצה רק ברמת הרשאהkernel

run_init_process() הינה פונקציה בקוד הגרעין (אחראית ליצירת תהליך ה-init של מערכת ההפעלה) ולכן רצה בהרשאות הגרעין. היא אינה מכירה את הספרייה של posix אשר הינה ספריית שירות לקריאות מערכת לתוכניות חיצוניות לגרעין ולכן היא מריצה do_execve() ואם היא תנסה להריץ execve() נקבל שגיאת קומפילציה:

implicit declaration of function '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.

הנה פונקציה לקריאת מערכת אשר מקבלת בארגומנט הראשון מספר שמציין את הקריאה הרצויה ושאר הארגומנטים הינם בהתאם לסוג הקריאה המבוקש.

קריאות מערכת כפי שלמדנו בקורס הינה הדרך של תוכניות בסביבת משתמש לבקש ממערכת ההפעלה לבצע פקודות אשר רצות בהרשאות גרעין, לדוגמה כתיבה לקובץ.

ממומשת בספרייה GLIBC, ממה שמצאנו בREPO, בקובץ syscall.c ומוגדר כheader ב Unistd.h.

ערך ההחזרה של הפונקציה הוא בהתאם לקריאה, לרוב 0 מציין הצלחה.

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.

קריאת מערכת מספר 39 הינה הרצת קוד הגרעין sys_getpid אשר מבקשת את ה-id של התהליך שרץ כרגע.

קוד ברור יותר:

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

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

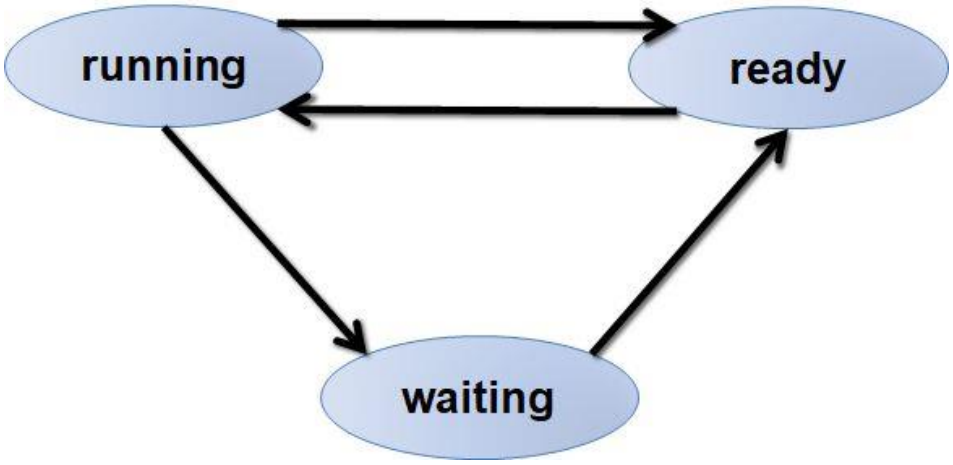
```
int main() {
    int x = get_weight();
    cout << "weight: " << x << endl;
    assert(x == 0);
    x = set_weight(5);
    cout << "set_weight returns: " << x << endl;
    assert(x == 0);
    x = get_weight();
    cout << "new weight: " << x << endl;
    assert(x == 5);
    cout << "===== SUCCESS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

הבדיקה בודקת שדיפולטית משקל תהליך הוא 0 ולאחר ששינינו את התהליך למשקל 5 השינוי אכן התרחש על ידי קריאת המערכת שממשנו.

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

running כאשר התהליך רץ על המעבד (המחסנית שלו טעונה) והוא עובר ל-ready כאשר מערכת ההפעלה מפריעה אותו ולכן מעבירה אותו לתור זה על ידי preemptive scheduler כמו למשל RR

b. ready→running

ready זה המצב ממנו התהליך מתחיל שהוא מסיים אתחול, כלומר נוצר התהליך והוא עובר ל-runing שהוא מקבל זמן מעבד.

c. running→waiting

running כאשר התהליך רץ על המעבד והוא ביקש ממערכת ההפעלה בקשת I\O ולכן מערכת ההפעלה העבירה אותו למצב waiting

d. waiting→ready

תהליך הוא המצב waiting כאשר הוא בהמתנה לאירוע. כאשר ה-event או בקשת ה-I\O שהתהליך חיכה לו הושלם הוא חוזר למצב ready כי הוא מוכן לחזור לרוץ.

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

a. מה היתרון בשימוש ב quantum גדול?

שימוש ב-quantum גדול מדמה את אלגוריתם FIFO כמו ב- None-preemptive Scheduling כך שאם תהליך רץ הוא ירוץ לזמן ארוך והתקורה של החלפת הקשר תתרחש פחות.

b. מה היתרון בשימוש ב quantum קטן?

שימוש ב-quantum קטן נקבל הדמיה של שני תהליכים הרצים במקביל בחצי המהירות של תהליך הרץ לבד וכן ייעול זמן ההמתנה.

c. במידה והמערכת עמוסה (מכילה הרבה תהליכים מוכנים לריצה), מדוע עדיף להוסיף תהליכים חדשים בסוף התור?

העדיפות להוספת תהליכים בסוף התור נובעת משני סיבות: סיבה ראשונה הינה שהוספת תהליכים בתחילת התור תיצור מצב בו התהליכים בסוף התור לא יקבלו זמן מעבד, כפי שקראנו לזה בהרצאה "הרעבה". סיבה שנייה הינה הוגנות, שהרי תהליכים שמחכים יותר לזמן מעבד יהיו במקום יותר גבוה בתור.

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

הפרמטר min_granularity מציין את הגודל המינימלי של quantum, כלומר הזמן המינימלי שתהליך ירוץ על המעבד. זה פותר את הבעיה בה כאשר תור התהליכים המוכנים לריצה עמוס, נמנע מהרבה החלפות הקשר אשר יש להן תקורה אשר פוגעות בביצוע המעבד.

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

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

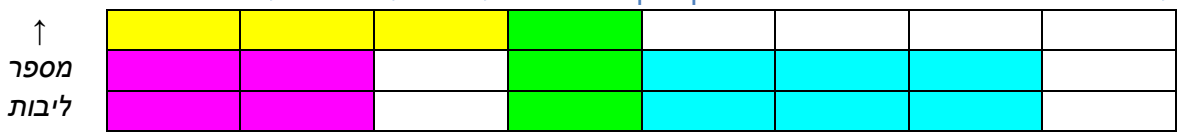
תהליך 1 דורש 2 ליבות וירוץ למשך 2 שניות עד לסיום.
תהליך 2 דורש 1 ליבות וירוץ למשך 3 שניות עד לסיום.
תהליך 3 דורש 3 ליבות וירוץ למשך 1 שניות עד לסיום.
תהליך 4 דורש 2 ליבות וירוץ למשך 3 שניות עד לסיום.

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

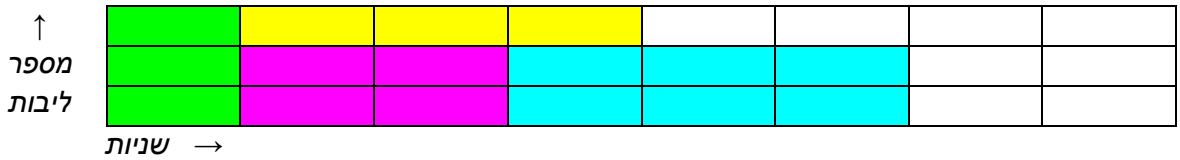
a. FCFS
b. SJF
c. EASY
d. תשובות b ו c נכונות.

נתאר את ההקצאה של המעבד כפי שלמדנו בהרצאה.

עבור FCFS סידור התהליכים לריצה הוא אך ורק לפי סדר הגעתם למערכת ההפעלה.



עבור SFJ סידור התהליכים לריצה הוא לפי הקצר יותר.



עבור EASY הרחבה של FCFS כל פעם שנוצר חור נמצא מבחינת זמן הגעה את התהליך הראשון שיכול למלא אותו.

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?

כפי שלמדנו בהרצאה במצב כזה אלגוריתם SRTF יתן לנו את זמן average wait האופטימלי מכיוון שבכל פעם שמגיעה עבודה חדשה או שעבודה כלשהי הסתיימה נתחיל את העבודה עם הזמן הריצה הנותר הקצר ביותר