מערכות הפעלה 31261

פתרון מבחן סופי, מועד ב', סמסטר א' תשע"ד, 25/02/2014

<u>הוראות לנבחן:</u> משך הבחינה שלוש שעות. חומרי העזר המותרים הם השקפים של הקורס שנמסרו באתר הקורס בלבד. שימוש במחשבונים ומכשירים אלקטרוניים כלשהם אסור בהחלט. רשום את תשובותיך במחברת במצורפת לבחינה. ציין בבירור את מספר השאלה במחברת והשתדל להיות **<u>קצר ועניייני</u>** (תשובות ארוכות מדי או נסיונות לתת מספר תשובות אפשריות <u>ייפסלו</u> את תשובתך!). הקפד על כתב יד ברור ומסודר, ומחק את כל חומר הטיוטא. השימוש בשפה האנגלית מותר. השאלון מכיל 11 שאלות בשווי של 106 נקודות, ונפרש על פני 3 עמודים.
בהצלחה !

<u>הערה חשובה:</u> הפתרונות כאן ניתנים באריכות רבה יותר מהנדרש בכדי לספק את כל הפרטים וכל ההסברים להבהרת התשובות. במספר מקרים ניתנו תשובות אפשריות נוספות.
בפועל, התשובות הנדרשות מהתלמיד אמורות להיות יותר קצרות וענייניות!

<u>שאלה1</u> [12%]

א. תאר בקצרה את מנגנון מצבי המשתמש והגרעין (user mode, kernel mode) הקיים בכל יחידת עיבוד מודרנית, וכיצד מערכת ההפעלה משתמשת בו על מנת להגן על משאבי המערכת מפני נזקים אפשריים (מצד תהליכי משתמש רגילים)?

ב. תן לפחות שתי דוגמאות שונות של פעולות משתמש שכיחות, בהם מערכת ההפעלה מגנה על משאבי המערכת באמצעות מנגנון זה.

<u>תשובה:</u>

א. מנגנון שני המצבים של המעבד (user/kernel modes) מתבסס על הכנסת שינויים מתאימים בשני רבדים: רובד החומרה ורובד התוכנה. ברובד של החומרה אנו מוסיפים למעבד סיבית מצב חדשה ("סיבית מצב" mode bit) שבאמצעותה תתבצע הבחנה בין "מצב משתמש" (user mode) לבין "מצב מערכת" (system mode). בנוסף לסיבית המצב, פקודות המעבד (CPU instruction set) מתחלקות לשתי קבוצות: פקודות מוגנות (protected instructions) ופקודות רגילות. במצב מערכת (system mode) המעבד רשאי להפעיל כל פקודה אפשרית, ואילו במצב משתמש ניתן להפעיל פקודות רגילות בלבד ("פקודות משתמש"). נסיון להפעיל פקודה מוגנת במצב משתמש ייגרום לפסיקת חומרה שתחזיר את השליטה למערכת ההפעלה. כאן נכנס הרובד השני של המנגנון: על מערכת ההפעלה לספק ולנהל רשימה ארוכה של קריאות שרות (system calls) שיאפשרו לתהליכי משתמש לבצע פעולות מוגנות באופן מסודר, הוגן, ומאובטח. קריאות השרות הם שיגרות פשוטות בשפת C אשר תפקידן לאפשר ביצוע של פקודות מעבד מוגנות באופן בטוח, הוגן, ויעיל. רק מערכת ההפעלה רשאית לגשת לסיבית המצב. רק מערכת ההפעלה רשאית להעביר את המעבד למצב מערכת ולמצב משתמש.

ב. קריאת השרות ()open למשל יכולה להתבצע על ידי כל תהליך משתמש לשם פתיחת קובץ בדיסק. על ידי ביצוע ()open, התהליך מעביר את השליטה לידי מערכת ההפעלה ונכנס למצב sleep. מערכת ההפעלה תבדוק אם לתהליך יש את ההרשאות המתאימות לגשת אל הקובץ המבוקש, ובכך היא מגנה על חלקים של מערכת הקבצים מפני נזק אפשרי שעלול להיגרם להם על ידי משתמשים שאינם מורשים לגשת אליהם. במידה והתהליך עבר את הבדיקה, מערכת ההפעלה תשנה את סיבית המצב למצב מערכת, תפתח את הקובץ המבוקש, ולאחר מכן תחזיר שליטה לתהליך המשתמש (ותשנה את סיבית המצב חזרה למצב משתמש).
דוגמאות נוספת היא קריאות השרות execl, execv, execlp, execvp אשר בהם תהליך משתמש מבקש להפעיל תוכנית מערכת חיצונית (system program). גם כאן, מערכת ההפעלה תוודא שלתהליך המשתמש יש הרשאות הפעלה מתאימות ורק אז תרשה לו להפעיל את התוכניות המבוקשות. באופן זה מערכת ההפעלה מגינה על מערכת המחשב מפני נזקים אפשריים שעשויים להיגרם לה כתוצאה מהפעלת תוכניות על ידי משתמשים שאינם מורשים לעשות זאת.

שאלה 2 [12%]

תאר שלושה תפקידים מרכזיים של מערכת ההפעלה הקשורים לניהול **מערכת הזיכרון** (RAM) של מערכת המחשב, ותאר בדיוק שלושה סוגי בעיות שונות שמערכת ההפעלה צריכה להתמודד איתם במסגרת זו.

תשובה:

א. **הגנה על שטחי זיכרון של התהליכים השונים (Protection):** לכל תהליך שטח זיכרון פרטי שאינו חופף לשטחי התהליכים האחרים . על מערכת ההפעלה להגן על שטח הזיכרון הפרטי של כל תהליך שרץ על מערכת המחשב מפני נזקים כגון נסיון של תהליך אחד לפלוש לשטח הזיכרון של תהליך שני. המטלה הכי חשובה בהקשר הזה היא ההגנה על שטחי הזיכרון של מערכת ההפעלה עצמה מפני נסיונות קריאה או כתיבה מצד תהלכי משתמש רגילים שאינם מורשים לכך. מערכת ההפעלה בדרך כלל תפסיק באופן מיידי כל תהליך שינסה לכתוב או לקרוא מידע מתוך שטח הזיכרון של תהליך אחר (ללא הרשאה מתאימה).

ב. **הקצאה ושחרור של שטחי זיכרון (memory allocation/free):** על מערכת ההפעלה לבצע הקצאות זיכרון מתאימות לכל תהליך שעולה על מערכת המחשב בזמן לידתו ובהמשך חייו (בהם הוא עשוי לבקש שטחי זיכרון נוספים בצורה דינמית על מנת לבצע חישובים נוספים). על מערכת ההפעלה לנקות ולפנות שטחי זיכרון שהתפנו על ידי תהליכים שסיימו את השימוש בהם או סיימו את פעילותם לחלוטין.

ג. **פיתרון בעיות מצוקה בזיכרון:** במצבים בהם כמות הזיכרון הנתונה נוצלה עד תומה, על מערכת ההפעלה לנהל מנגנון שיחלוף זיכרון (memory swap) אשר בו שטחי זיכרון של תהליכים מסוימים יועברו לדיסק הקשיח באופן זמני, בכדי לאפשר לתהליכים חדשים לרוץ על מערכת המחשב. מערכת ההפעלה צריכה לפעול באופן הגון ויעיל, ולפנות לדיסק רק תהליכים אשר לא היו פעילים לאחרונה או זכו לזמן ריצה מספיק ארוך, ולא לפנות תהליכים קטנים מספיק או צעירים שעדיין לא הספיקו לבצע נתח עבודה הגון. בנוסף למנגנון השחלוף, מערכת ההפעלה מנהלת גם מנגנון מורכב מאוד של דיפדוף (paging) שמאפשר לטעון חלקים קטנים של תוכנית (דפים), ובכך לאפשר ליותר תהליכים לרוץ על שטח זיכרון מוגבל מאוד.

ד. **שמירת רצף של שטחי זיכרון (defragmentation):** לאחר שימוש ארוך במערכת הזיכרון שבו התבצעו מספר גדול מאוד של פעולות הקצאה ושחרור של זיכרון, נוצרים "חורים" רבים ברצף (הקווי) של מערכת הזיכרון שמקשים מאוד על הקצאות של שטחי זיכרון נוספים (בפרט אם נדרשים שטחים גדולים) עקב הפיזור השבור והמקוטע של השטחים הנוכחיים. במצבים כאלה, על מערכת ההפעלה לקחת פסק זמן (שאינו ממש מורגש, ומתבצע באופן מקבילי) בכדי לבצע איחוי (defragmentation או compaction) ובכך לאפשר ניהול מהיר ויעיל יותר של שטח הזיכרון הפנוי.

שאלה 3 [10%]

א. הסבר את מושג פונקציות **API** בהקשר של תיכנות מערכות הפעלה

ב. רשום שלוש דוגמאות של פונקציות **API** במערכת הפעלה **Linux**

ג. מנה שתי סיבות טובות להעדפת שימוש בפונקציות **API** בתיכנות מערכות הפעלה על פני שימוש ישיר בקריאות מערכת (system calls)

תשובה:

א. API הינו קיצור של Application Programming Interface. זוהי קטגוריה כללית מאוד של פונקציות בשפת תיכנות גבוהה כמו C, C++, Java, Python אשר מטרתם לאפשר גישה פשוטה ונוחה לקריאות שרות של מערכת ההפעלה (אשר לא תמיד קל להבינן ולהשתמש בהן ישירות). בחלק מהמקרים פונקציות API עשויות לספק למשתמש אפשרות נוחה לבצע פעולות מורכבות אשר לא ניתן לבצע באמצעות קריאת שרות אחת, ובכך ניתן לבצע באמצעות פונקציות API פעולות מערכת שכיחות ומשמעותיות יותר אשר קשה לבצע ישירות עם קריאות מערכת יסודיות.

ב. שלושה דוגמאות של פונקציות API במערכת ההפעלה Linux הן למשל: fopen, printf, fclose. קריאת המערכת האמיתית לפתיחת קובץ למשל היא open. הפונקציה fopen היא למעשה עטיפה נוחה של הפונקציה open.

ג. סיבה טובה ראשונה להעדפת השימוש בפונקציות API על פני שימוש ישיר בקריאות מערכת הוזכרה כבר בסעיף א': גישה נוחה, פשוטה, ובמקרים מסוימים בעלת ביצועים טובים יותר לקריאות מערכת. סיבה שניה היא אי-תלות בפלטפורמה: קיימות היום בשוק מספר גדול של גירסאות Linux (כולל Android) עם הבדלים עדינים בין קריאות המערכת. פונקציות API עוזרות מאוד לכתיבה של קוד אחיד שרץ על פני כל המערכות האלה ללא שינוי מאחר וכל מערכת מגדירה את פונקציות ה-API בהתאם למערכת הקריאות שלה. כך שגם אם מערכת הקריאות היסודית מתרחבת ומשתנה, פונקציות ה-API אינן משתנות בדרך כלל. למשל פונקציית ה-fopen API מתנהגת באופן דומה בכל מערכות ההפעלה (כולל Windows!).

שאלה 4 [10%]

תאר לפחות שלושה ייתרונות שונים לכך שמבנה הנתונים Pipe שמשמש לתיקשורת בין תהליכים שונים מנוהל על ידי הגרעין (kernel) של מערכת ההפעלה, ושהחוצץ שלו (buffer) שמור בשטח הזיכרון של מערכת ההפעלה (system memory) ולא בשטח הזיכרון של תהליך משתמש (user memory).

תשובה:

קיימים ייתרונות רבים ומגוונים לעובדה שהחוצץ של הצינור שמור בזיכרון של מערכת ההפעלה:

א. **שיתוף (sharing)**. העובדה שהחוצץ שמור בידי מערכת ההפעלה עצמה (ולא על ידי תהליך פרטי) מאפשרת למספר רב של תהליכי משתמש להשתתף בשטח הזיכרון הזה. למשל, מספר רב של תהליכים יוכלו לכתוב לצינור או לקרוא מאותו צינור במקביל.

ב. **איחסון מטמון (caching)**. מטרת ה-Pipe היא לאפשר העברה של נתונים בין תהליכי משתמשים. במקרים רבים, התהליך שאמור לקבל את הנתונים אינו פנוי לכך (עסוק בפעילות אחרת או במצב waiting). במצב כזה, החוצץ של הצינור משמש כמו זיכרון מטמון השומר את הנתונים עבור הלקוח.

ג. **תיאום בין מהירויות עבודה שונות של תהליכים (speed differences resolution)**. במקרים רבים, התהליך שמייצר את הנתונים עובד מהר יותר מהתהליך הצורך אותם. במקרים כאלה, מערכת ההפעלה שומרת חלק מהנתונים שהצרכן לא הספיק לקרוא על החוצץ עד שייתפנה לקרוא אותם.

ד. **עקיפה של בעיות user/kernel modes**. תהליכי משתמשים אינם יכולים לכתוב אחד לשטח הזיכרון של השני במצב user. הדרך היחידה לעשות זאת היא באמצעות מערכת ההפעלה בכך שהחוצץ נמצא ברשותה ומתווך בין שני התהליכים של המשתמשים במצב kernel בלבד.

שאלה 5 [8%]

א. תאר בקצרה את מנגנון ההרשאות (permissions) עבור קריאה, כתיבה, והפעלה של קבצים במערכת ההפעלה Unix (התייחס לקבצים רגילים בלבד, לא לתיקיות!).

ב. למשתמש dany יש ספריה בשם tasks בחשבון ה-Unix שלו: /home/dany/tasks. רשום תוכנית בשפת Python בכדי לפתוח את כל הקבצים בתיקיה tasks לכתיבה וקריאה עבור כל המשתמשים השייכים לקבוצה שלו. על התוכנית לבצע רק את הפעולות המבוקשות ולא לגרום לתוצאות לוואי שלא התבקשו!

תשובה:

א. לכל קובץ (רגיל או ספריה) יש 9 סיביות הגנה עבור אפשרות קריאה, כתיבה, או הפעלה: שלוש סיביות עבור המשתמש עצמו (שמסוגל לנעול את הקובץ לקריאה, כתיבה, או הפעלה אפילו עבור עצמו), שלוש סיביות עבור כל המשתמשים בקבוצה אליה הוא משתייך (כל משתמש משתייך לקבוצה מסוימת), ושלוש סיביות עבור כל שאר המשתמשים. רק בעל הקובץ או אדמיניסטרטור מסוגלים לשנות את כל אחת מתשע הסיביות הללו. הפקודה העיקרית שבאמצעותה ניתן לשלוט על מצב הסיביות היא chmod.

Permissions

user			group			other		
r	w	x	r	w	x	r	w	x

ב.

```
import os, subprocess

for path,dirs,files in os.walk('/home/dany/tasks'):
    for f in files:
        p = path + '/' + f
        subprocess.call(['chmod', 'g+rw', p])
    for d in dirs:
        p = path + '/' + d
        subprocess.call(['chmod', 'g+rx', p])
```

בכדי שחברי הקבוצה יוכלו להיכנס לתתי-תיקיות של /home/dany/tasks יש להדליק את סיביות ה-rx עבור כל תתי-התיקיות (רקורסיבית).

שאלה 6 [6%]

תאר במשפט אחד מה בדיוק מתבצע על ידי תוכנית Python הבאה:

```
import time, os
a = time.time()
b = ""
d = "D:/BRAUDE"
for path,dirs,files in os.walk(d):
    for f in files:
        if not f[-4:] == '.doc': continue
        p = os.path.join(path,f)
        x = os.path.getmtime(p)
        if x<a:
            a = x
            b = p
print b
```

הערה: הפונקציה **os.path.getmtime** מחזירה את זמן השינוי האחרון של קובץ (בשניות משנת 1970).

תשובה:

התוכנית מדפיסה את שם קובץ ה-Word (סיומת doc.) הכי ישן שכלול בתיקיה C:/BRAUDE (רקורסיבית).

שאלה 7 [5% + 5% bonus]

רשום תוכנית קצרה שבה תהליך אב P מוליד תהליך בן C כך שמתקיימים הדברים הבאים:

א. התהליך P מעביר את המשפט "Hello Son" לתהליך C, שמדפיס את המשפט הזה

ב. התהליך C מעביר את המשפט "Hello Father" לתהליך P אשר מדפיס גם הוא משפט זה

ג. סדר ההדפסה חייב להיות זהה למתואר בסעיפים הקודמים

ד. דרוש קיצור ושימוש מינימלי במשאבים!

הערה: תוכל להשתמש בשפת C או בשפת Python בכדי לרשום את התוכנית שלך

שאלת בונוס: תוספת בונוס של 5 נקודות יינתנו לכל מי שירשום תוכנית Python (או C) שתבצע את התסריט הנ"ל במלואו ובמדויק. (ניקוד יינתן לתשובה מדויקת בלבד! לא יינתן ניקוד חלקי עבור תשובה חלקית!)

תשובה:

```
import os, sys

rside, wside = os.pipe()

pid = os.fork()

if pid == 0:      # Child process
    print os.read(rside, 1024)
    os.write(wside, "Hello Father")
    sys.exit(0)
else:             # Parent process
    os.write(wside, "Hello Son")
    os.waitpid(pid,0)
    print os.read(rside, 1024)
```

א. תהליך האב P מוליד את תהליך הבן C על ידי ביצוע הפקודה pid=os.fork(). לאחר מכן יש לנו שני תהליכים שרצים במקביל: P, C

ב. תהליך הבן C (pid==0) קורא את הודעת האב ("Hello Son") ומדפיס אותה. לאחר מכן, הוא כותב לאותו צינור (אין צורך בשני צינורות!!) את המשפט "Hello Father" ומסיים.

ג. תהליך האב P רושם את ההודעה "Hello Son" לצד הכתיבה של הצינור (wside), ואז נכנס למצב המתנה לתהליך הבן (os.waitpid). מייד לאחר סיום תהליך הבן, תהליך האב יוצא ממצב ההמתנה, קורא את הודעת הבן ("Hello Father") ומדפיס אותה.

ד. העובדה שתהליך האב מחכה עד לסיום תהליך הבן מבטיחה שהדפסת ההודעות תתבצע על פי הסדר שהתבקש!

יש לזכור שמאחר שהצינור pipe נוצר בשלב מוקדם, הרי שהוא משותף לשני התהליכים! ולשני התהליכים יש רשות לכתוב ולקרוא מכל צד של הצינור. בפרקטיקה לא נעשה שימוש באפשרות לעבוד עם צינור בשני הכוונים מאחר וזה קשה לניהול ומסוכן, אבל מבחינה תאורטית הדבר אפשרי בהחלט (אפשר להוריד את התוכנית הנ"ל ולהריץ אותה על כל מערכת Linux).

שאלה 8 [10%]

הפלט של הפקודה ps של Linux (בתוספת הדגלים aux) נראה כך:

```
Linux> ps aux
        USER    PID %CPU %MEM    VSZ   RSS TTY    STAT START   TIME COMMAND
        root      1  0.0  0.4   5648  2540 ?      Ss   Apr20   0:01 /sbin/init showopts
        avahi   360  0.0  0.2   3248  1072 ?      Ss   Apr20   0:00 avahi-daemon
        root    363  0.0  0.1   5044   540 ?      Ss   Apr20   0:13 /usr/sbin/haveged -w 1024 -v 0 -F
        root   2617  0.0  0.1   3884   564 ?      Ss   Apr20   0:00 /usr/bin/kdm
        root   2700  0.0  0.2   5084  1180 ?      Ss   Apr20   0:03 /usr/lib/postfix/master
        dina   3513  0.0  0.6   9196  3428 ?      S    Apr20   0:00 xterm
        dany   2728  0.0  0.2   4528  1308 ?      Ss   Apr20   0:00 /bin/bash
        samy   2794  0.0  0.2   7060  1088 ?      Sl   Apr20   0:00 /usr/bin/VBoxClient --clipboard
        samy   3018  0.0  0.1   2496   740 ?      S    Apr20   0:00 python
        samy   3491  0.0  0.5   5656  2792 pts/2  Ss   Apr20   0:01 bash
        samy   9399 50.0  0.2   4476  1028 pts/2  R+   07:29   0:00 ps aux
```

על בסיס צורת הפלט הזו, רשום תוכנית Python המשתמשת בפקודה subprocess.Popen לשם הפעלת התוכנית ps aux, אשר מדפיסה את רשימת המשתמשים שצורכים יותר מ-10% מהזיכרון במערכת (%MEM)

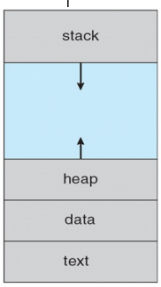
תשובה:

התוכנית הבאה מסבירה הכל בצורה ברורה

```
from subprocess import Popen, PIPE
import os, time

def high_mem_users():
    cmd = ['ps', 'aux']
    p = Popen(cmd, stdout=PIPE)
    title = p.stdout.readline() # skip title line
    users = []
    while not p.stdout.closed:
        line = p.stdout.readline().strip()
        if line == "":
            break
        fields = line.split()
        if len(fields)<4:
            continue
        user = fields[0]
        mem = float(fields[3])
        if mem>10:
            if not user in users:
                users.append(user)

    p.terminate()
    return users
```

שאלה 9 [8%]

```
#define DAY 18
int year = 2014 ;
char *message = "Hello World" ;
double price = 517.23 ;

int funky(int a1, int a2) {
  char *pstr ;
  if(a1<a2) {
    long int y1=a1, y2=a2 ;
    char buffer[500] ;
    pstr = new char[150] ;
    return y1+y2 ;
  } else if (a2>17) {
    return a1 ;
  }
  return DAY+a1+a2 ;
}
```



תוכנית C שלפניך היא חלק מקוד של תהליך P שרץ כרגע במערכת:

א. ערוך את רשימת כל המשתנים המופיעים בקוד זה

ב. עיין בקוד של התוכנית ונסה לשייך את כל המשתנים השונים שמופיעים שם לשטחי הזיכרון text, data, heap, או stack של התהליך P. רשום טבלה מתאימה שמציגה את תשובתך.

ג. ציין את הסכנה העיקרית שעשויה לנבוע כתוצאה מגידול בלתי מבוקר של שטחי ה-heap וה-stack ואיך היא נקראת בשפה מקצועית?

תשובה:

א. משתני התוכנית הם:

גלובליים: year, message, price

לוקליים: a1, a2, y1, y2, pstr, buffer

השם DAY הוא משתנה פרה-קומפילציה (pre-compile variable)

ב. המשתנים הגלובליים מאוחסנים בשטח ה-data. המשתנים הגלובליים הקבועים נשמרים בתוך שטח ה-text. כאשר מדובר במחרוזת טקסט כמו "Hello World" – המחרוזת עצמה נשמרת באזור data, אבל המצביע אליה נשמר באזור text.

המשתנים הלוקליים (a1, a2, y1, y2, pstr, buffer) נשמרים ב-stack. אך כאשר מתבצעת הקצאה דינמית של 500 בתים במקרה של buffer, הרי שהמערך של 500 בתים מאוחסן ב-heap אך המצביע buffer עצמו מאוחסן ב-stack. כמו-כן, המצביע pstr שנמצא ב-stack מצביע למערך של 150 בתים שנמצא ב-heap.

המשתנה DAY מוחלף בזמן קומפילציה למספר 18 בכל מקום באזור ה-text בו הוא מופיע (כך שהוא למעשה נעלם לחלוטין מהתוכנית ומוחלף בערך המספרי שלו).

STACK	A1, a2, y1, y2, pstr*, buffer*, buffer[500]
HEAP	char[150]
DATA	year, message[], price
TEXT	message*, DAY

ג. ככל ששטחי ה-stack וה-heap גדלים אחד לכוונו של השני, קיימת סכנת "התנגשות" ביניהן אשר כתוצאה מכך נתונים האמורים להיכנס לתחתית ה-stack ייכתבו לתוך ה-heap, מה שבסופו של דבר עשוי לגרום לשגיאות חמורות או אף קריסה של התוכנית. בשפה המקצועית, תופעה כזו נקראת stackoverflow.

שאלה 10 [8%]

במערכת מחשב בעלת 8 ליבות ידוע כי לכל תהליך מנת זמן CPU מקסימלית של 40ms ולכל פסיקה להחלפת תהליכים (context switch) נדרש 3ms. בהנחה (אידיאלית) שכל התהליכים במערכת מנצלים את מלוא המנה המוקצית להם בכל ריצה, מהו המספר המקסימלי של תהליכים שהמערכת מסוגלת להריץ בשניה אחת?
הערה: הנח שכל התהליכים התחילו לרוץ על המערכת לפני שעה כך שבשניה המדוברת שום תהליך חדש לא התחיל או סיים.

תשובה:

לכל מנת ריצה (40ms) נדרשת גם פעימת החלפת הקשר (3ms) חוץ מ- 8 המנות האחרונות (בכל 8 הליבות השונות). לכן אם n הוא המספר המקסימלי של המנות, אז הזמן הנדרש עבורם הוא 43n-24. מאחר ויש בידנו 8 ליבות, הרי שיש לנו תקציב של 8000 מילי-שניות. מוטל עלינו למצוא את הערך הגדול ביותר של n שמקיים את אי-השוויון: $43n-24 \leq 8000$. זה מוביל בקלות ל- $n \leq 8024/43$. לא קשה לקבוע (ללא-מחשבון!) כי n=186.
אבל עדיין צריך לוודא ש-186 התהליכים האלה יכולים להתפזר על פני 8 ליבות באופן כזה ששום ליבה לא תחרוג מתקציב של 1000ms. הפירוק הכי מאוזן ייתבצע באופן הבא: 24+24+23+23+23+23+23+23=186. קל לבדוק שכל ליבה מסוגלת להריץ 23 תהליכים בפחות מ-1000ms: 1000>986=40+43*22 (22 תהליכי עם החלפת

הקשר והאחרון ללא החלפת הקשר). לעות זאת בכדי להריץ 24 תהליכים, דרושים 1029ms (23*43+40).
לכן המספר הגדול ביותר של תהליכים שנוכל להריץ בשניה הוא 184.

שאלה 11 [12%]

עיין היטב בתוכנית producer.py אשר פגשנו מספר פעמים במהלך הקורס.

א. הסבר בשני משפטים מה התוכנית עושה וכיצד משתמשים בה מתוך שורת הפקודה של Unix (command line)?

ב. תוכנית המערכת /usr/bin/gzip היא תוכנית סטנדרטית במערכת ההפעלה Linux לשם דחיסת נתונים. בצרוף הדגל f- תוכנית זו מסוגלת לקלוט נתונים גם דרך הקלט הסטנדרטי (stdin), ולפלוט את הדחיסה שלהם לפלט הסטנדרטי (stdout). רשום תוכנית Python בשם zip_apples.py שמטרתה להפנות את הפלט הסטנדרטי של producer.py לקלט הסטנדרטי של התוכנית gzip לשם דחיסתו. לשם פשטות, התוכנית zip_apples.py תדפיס את האורך של הנתונים לאחר דחיסתם.

```
import sys, time

def make_apples(n):
    for i in range(n):
        time.sleep(0.01)
        print "Apple_%d" % i
        sys.stdout.flush()

if __name__ == "__main__":
    n = int(sys.argv[1])
    make_apples(n)
```

תשובה:

א. התוכנית מערכת producer.py מקבלת ארגומנט אחד n ומדפיסה n שורות מהצורה Apple_i כאשר i רץ מ-0 עד n-1. צורת ההרצה משורת הפקודה נראית כך:

```
samyz@brdlinux:~> producer.py 5
Apple_0
Apple_1
Apple_2
Apple_3
Apple_4
samyz@brdlinux:~>
```

ב. כך תראה הפעלה של producer.py ו-gzip על ידי צינור ביניהם:

```
samyz@brdlinux:~> producer.py 12 | gzip -f
gSs,( I 7 rлPJi(m
1 bsamyz@brdlinux:~>
samyz@brdlinux:~>
```

יש לשים לב לכך שהפלט הבינארי הדחוס נראה כמו "זבל" על המסך (זוהי הדחיסה של של שורות Apple_i, i=0,…,11).

בכדי להריץ את הצינור הזה דרך Python, נשתמש בחבילה subprocess (ולמעשה בשני צינורות):

```
from subprocess import Popen, PIPE

def zip_apples(n):
    p1 = Popen(['/home/samyz/vault/producer.py', str(n)], stdout=PIPE)
    p2 = Popen(['/usr/bin/gzip', '-f'], stdin=PIPE, stdout=PIPE, stderr=PIPE)

    while not p1.stdout.closed:
        apple = p1.stdout.readline()
        if not p1.poll() is None:
            break
        p2.stdin.write(apple)

    p2.stdin.close()
    print "zip length =", len(p2.stdout.read())

if __name__ == "__main__":
    from sys import argv
    n = int(argv[1])
    zip_apples(n)
```