עבודת בית מספר 1- מערכות הפעלה

שם סטודנטים ותעודת זהות :

בשאר בשותי - 207370248

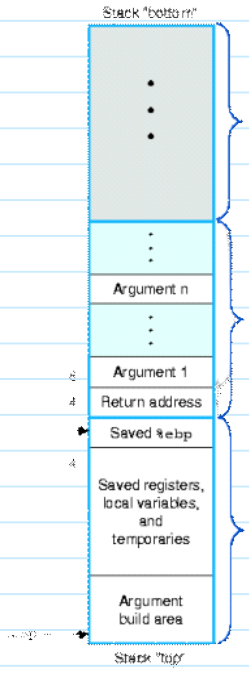
ויסאל עמאשה - 211579347

שנת לימודים : **2022/2023**

מרצה : **רחל קולודני**

שאלה 1 :

הסבירו במשפט אחד בלבד מדוע בקונבנציית GCC לקריאה לפונקציה, שומרים בסר הפוך (במחסנית) את הפרמטרים המועברים לפונקציייה הנקראית :



כלומר, הסבירו מדוע תחילה מבצעים $push$ לפרמטר ה- n (האחרון), ורק לבסוף מבצעים $push$ לפרמטר ה- 1 (ראשון).

פתרון לשאלה 1 :

קונבנציית של GCC לקריאה לפונקציה משתמשת במערך הפוך על המחסנית מכיוון שהיא מאפשרת רשימות ארגומנטים באורך משתנה ומקלה על יישום המאקרו המאפשר לפונקציה עם מספר משתנה של ארגומנטים לגשת לארגומנטים שלה בצורה ניידת.

במשמעות אחרת, הקונבנציה הזו מרשה הפונקציה לגשת פרמטר הראשון בצורה מהירה יותר. (מאחרי שה- Stack גדלה בכיוון מטה בזיכרון, הגישה לפרמטר הראשון שדחפנו אותו בסוף קל יותר מבחינת חישוב פוינטירים מאשר לגשת לאחרון כך שהוא יותר קרוב לראש המחסנית).

שאלה 2 :

ציינו את כל הפלטים האפשריים (למסך) של קטע הקוד הבא : **(נמקו!)**

```
pid_t pid = fork();
if (pid < 0)
{
        exit(1);
}
else if (pid > 0)
{
        printf("%d", getpid());
        exit(0);
}
else
{
        char *const argv[] = {"sleep", "1", NULL};
        execv("/bin/sleep", argv);
        printf("%d", getpid());
}
```

<u>ניתן להניח כי :</u>

$pid(father) = 8$
$pid(son) = 13$

פתרון לשאלה 2:

יש אופציות שונות לפלט וזה נקבע על ידי $fork(\square)$,$execv(/\text{bin}/\text{sleep},argv)$ והצלחתם.

- אופציה הראשונה - ה$fork()$ נכשל בריצה מואז הpid יהיה שלילי ולכן התוכנית יוצאת במצב 1. ואין הדפסה כמובן.
- אופציה השניה - בהנחה שלתהליך האב יש pid של 8 ולתהליך הילד יש pid של 13, הפלט של קוד זה יהיה:
  תהליך האב ידפיס את ה-Pid שלו, שהוא 8, ואז ייצא. לכן, הפלט יהיה:
  8
  תהליך הילד יבצע את קריאת ה-execv, שתחליף את תהליך הילד בפקודת השינה, מה שיגרום לתהליך הילד לישון למשך שנייה אחת. הצהרת printf לאחר קריאת execv לא תבוצע מכיוון ש-execv מחליף את תהליך הילד בפקודת sleep ולעולם לא חוזר.

לכן, הפלט הסופי יהיה:
8

- אופציה שלישית - כמובן יהיה לנו התהליך האב עם $pid = 8$ ותהליך הבן עם $pid = 13$. התהליך של הבן מתחיל לרוץ ואז $execv()$ נכשל ומודפס את pid שלו שהוא 13.
  אחר כך, תהליך האב מתחיל ומדפיס את ה pid שלו שהוא 8 .
  לכן הפלט הוא 138 .

- אופציה רביעית - יהיה לנו התהליך האב עם $pid = 8$ ותהליך הבן עם $pid = 13$. התהליך של הבן מתחיל אחרי הצלחת $fork()$ לרוץ ואבל $execv()$ נכשל ומודפס את pid שלו שהוא 1 ועוצר.
  אחר כך, תהליך האב מתחיל ומדפיס את ה pid שלו שהוא 8 .
  עכשיו תהליך הבן ממשיך עם הדפסה של 3 ,לכן הפלט הסופי הוא 183 .

- אופציה חמישית - תהליך הבן מתחיל ונעצר לפני שהוא עושה $execv$. תהליך האב מתחיל לרוץ ומסדפיס הpid- שלו שהוא 8 ויוצא עם ערך 0. אחר כך, תהליך הבן ממשיך ועושה את $execv()$ והוא מסתיים בהדפסה ה pid שלו שהוא 13. לכן, הפלט הסופי הוא 813 .

לסיכום, יש 5 אופציות שונות :

1. נכשל ה$fork$ - אין פלט
2. 8
3. 138
4. 183
5. 813

שאלה 3 :

ציינו את כל הפלטים האפשריים (למסך) של קטע הקוד הבא : **(נמקו!)**

```
int value = 0;
if (fork() != 0)
{
        wait(&value);
        value = WEXITSTATUS(value);
        value += 3;
}
printf("%d\n", value);
value += 4;
exit(value);
```

פתרון לשאלה 3 :

הפלטים האפשריים של קוד זה תלויים בהתנהגות של תהליך הילד, שנוצר על ידי הקריאה fork() .

- אם הקריאה fork() מצליחה, נוצרים שני תהליכים: תהליך האב ותהליך הילד. תהליך הילד מריץ את הקוד מיד לאחר הקריאה fork() בעוד שתהליך האב ממתין לסיום הילד ולאחר מכן מעדכן את ערך הערך.

תהליך האב מתחיל לרוץ ובגלל הפונקציה $wait$. יהיה במצב המתנה עד שתהליך הבן יסתיים. תהליך הבן ירוץ ומעדכן ערך ה$Value$ ל 4 ויוצא כאשר ערך החזרה שלו הוא 4.
תהליך האב מתעדכן אחרי סיום תהליך הבן כל שהערך $value$ משתנה בגלל פונקציה $WEXITSTATUS$ של הבן. ואחר כך הערך ה- $value$ מתעדכן אחרי הוספה 3. ואז $value = 7$.
אז מדפיס 7 שהוא ערך ה-$value$ על המסך ומעדכן אותו שוב בהוספה 4. כלומר $value = 11$ ויוצא מתהליך ומחזיר 11.

הפלט הסופי :	0 7

- אם הקריאה fork() נכשלת, הקוד בבלוק if אינו מבוצע, ותהליך האב ממשיך להתבצע כרגיל .

אז התהליך ה$fork$ לא מיצר תהליך בן. נכנס לבלוק ה if , מוסיפים 3 לערך $value$ ואז מדפיסים הערך שהוא כרגע 3 למסך. אחר כך מוסיפים עוד 4 שהפך ל 7 והתוכנית יוצאת .
הפלט הסופי : 3 .