נושאים

- הקדמה לקורס.
- שיעור 1.
  - יוניקס.
  - לינוקס.
  - אפל.
  - רישיון לתוכנה חופשית.
  - הפצה של לינוקס.
- שיעור 2.
  - פונקציות ספריה.
  - Man (manual).
  - רמות הרשאה.
  - ניהול תהליכים.
    - fork(2).
    - wait(2).
    - execXX.
    - ניהול שגיאות.
- שיעור 5.
  - Blocking.
  - select(2).
  - poll(2).

הקדמה לקורס

פרטי קשר:

שם המרצה	מספר טלפון	כתובת אימייל	שעות קבלה
נצר זיידנברג	054-5531415	scipio@scipio.org	אחרי הקורס / ימי ראשון או חמישי ניתן לקבוע מראש

- ליצירת קשר, מומלץ לפנות בוואטצפ (מייל עלול להתפספס).

להלן מספר נקודות בנוגע להתנהלות הקורס:

- **השיעורים יוקלטו ללא התחייבות, ולא יהיו היברידיים.**
  לכן מומלץ להגיע לכל שיעור.
- **מפגשי יום חמישי בזום.**
- **מומלץ לעבור הסמסטר שכן סמסטר הבא המתכונת של הקורס תשתנה (תהיה קשה יותר).**
- **בקורס זה לא נלמד לכתוב מערכות הפעלה אלא נלמד לתכנת מערכות בסביבת Linux,**
  נכתוב דרייברים, נלמד Design Patterns ועוד...

להלן מספר נקודות בנוגע למטלות:

- **כל תוכניות הקוד ייכתבו בשפת C או $C++$.**
- **כל שבוע/שבועיים תעלה מטלה (פרט לשבועיים האחרונים של הסמסטר).**
- **יהיו סה"כ 7 מטלות בערך.**
- **המטלות הן 40 אחוז מהציון.**

להלן מספר נקודות בנוגע למבחן:

- **40 נקודות מהמבחן לשאלות אמריקאיות.**
- **3 שאלות פתוחות:**
  1. **מציאת באג.**
  2. **שאלת תכנון מערכת (לפרט כיצד נכתוב את הקוד).**
  3. **כתיבת קוד (נדרש דיוק סביר).**

אודותיי:

אני חבר סגל חדש, מלמד מערכות הפעלה מזה 10 שנים (לימדתי באוניברסטית תל-אביב, האוניברסיטה הפתוחה, אוניברסיטת בר-אילן ועוד...). יש לי חיבה לשפת C.

שיעור 1

- בשיעור זה יהיו מגוון סיפורי רקע והגדרות שחשובים לא פחות לבחינה.

Multics (מולטיקס) - מערכת מחשב שנחשבה מתקדמת באותה תקופה ("מחשב על") ואיפשרה מולטי-יוזר ומולטי-פרוסס מולטי פרוססור (יכולים לעבוד במחשב מספר משתמשים ותהליכים במקביל, והיו למחשב הרבה מעבדים).

Unix (יוניקס) - סטנדרט מסוים שמערכת הפעלה יכולה לקיים (אם עומדת במבחן POSIX).

POSIX (פוסיקס) - אוסף תקנים שמספקים בסיס לכלל מערכות ההפעלה.

יוניקס

1970: החברה AT&T הגיעו למסקנה שלא טוב שפעילות המחשבים תהיה בתוך החברה.
הייתה קבוצת פיתוח שישבה ב-Bell Labs (חברת בת בבעלות מלאה) והיא הייתה אחראית להמון פיתוחים טובים (ביניהם שפת C, יוניקס, ועוד דברים..).
ב-1969 הם רצו לקנות Multics אבל לא היה להם תקציב לזה.
בעקבות זאת הם מצאו מחשב פשוט יותר ודי נפוץ באותה תקופה (PDP-11) וכתבו מערכת הפעלה שמקיימת את התכונות של ה-Multics על המחשב הזה וקראו לה יוניקס.
הם התחילו להפיץ את הקוד של היוניקס שהם כתבו, והוא התחיל להתקדם ולהתפתח ויצאו עוד גרסאות שלו.
אחרי כמה זמן כל מיני חברות התחילו לפתח מחשבי יוניקס כאלה,
עם מערכות הפעלה שמקיימות את העקרונות של יוניקס.
הגיעו המנהלים של אותה קבוצה מפתחת ויצרו מבחן POSIX שבוחן האם מערכת הפעלה מסויימת מקיימת את הסטנדרט ועומדת בתקנים של מערכת יוניקס.

יש מערכות הפעלה שלא ניסו להיות תואמות יוניקס לדוגמה מערכת מקינטוש הקלאסית, מערכת דוס של מיקרוסופט, מערכת IBM MVS (מיין פריים) ועוד מערכות.
מבין המערכות שנועדו להיות תואמות יוניקס לא כל המערכות הפעלה שנכתבו נבדקו במבחן POSIX לכן נוצרו בעולם מערכות יוניקס שעברו את המבחן ומערכות דמויות יוניקס שמקיימות את העקרונות של יוניקס אבל לא עברו רישמית את המבחן (UNIX LIKE).
באזור שנות ה-80: AT&T התחילה לתבוע חברות שקראו למערכת הפעלה שלהם יוניקס למרות שלא באמת עברו את מבחן ה-POSIX.

- **Bell Labs** - חברה שהייתה מונפול התקשורת של ארה"ב שכן הייתה ספקית הטלפונים המרכזית באותה תקופה. ב-1984 בית המשפט חייב אותם להתפצל ל-8 חברות.
- **AT&T** - תאגיד תקשורת בינלאומי אמריקאי.
- **PDP-11** - סדרת מיני מחשבים שהיוותה אלטרנטיבה זולה למחשבים מרכזיים ואנלוגיים.

Linux (לינוקס) - מערכת הפעלה שפותחה בהתבסס על מערכת ההפעלה יוניקס.
בשונה מחלק גדול מהיוניקסים המסחריים, המערכת חינמית ועם קוד פתוח.
לינוקס נמצאת כמעט ברוב המכשירים: טלפונים, מצלמות אבטחה, מערכות בידור, נתבים, רכבים ואפילו 90% מטופ 500 מחשבי העל בעולם.
למעשה יוניקס נמצאת גם בתחנות עבודה אישיות (בעיקר בגרסת Apple) ולינוקס נמצאת בבערך כל מקום אחר.

לינוקס

- לידע כללי: 3 הפינים הכי מפורסמים בעולם הם שוודים (למעשה 10% מהפינים הם ממוצא שוודי).

1991: בפינלנד היה סטודנט עני ושמו "לינוס טורבאלדס".
הוא למד מדעי המחשב והגיע למסקנה שהוא רוצה להתקין יוניקס אצלו בבית.
באוניברסיטה שלו לימדו להשתמש במערכות דמויות לינוקס ולאוניברסיטה הזו היה צ'יפ 386 ואת נקודות הציפה היה מקיים צ'יפ 387 (היום כבר אין דברים כאלה).
בעבר האביזרים האלה נקנו ע"י מרצים עם תקציב יותר גדול ממה שהיה ללינוס שהיה להם כסף למעבדים האלה ולסטודנט העני שלנו לא היה את התקציב הזה אז הוא החליט שהפתרון הוא לכתוב מערכת הפעלה משלו מאפס וקרא לה "לינוקס".
באוגוסט 1991, הוא פירסם את הקוד שהוא כתב לאנשים והציע להשתמש בה כסוג של רשת חברתית (UNSET).
בגלל שהוא היה פיני AT&T לא הגיעו אליו לשלוח לו מכתב הזהרה לפני תביעה ולכן הוא המשיך להפיץ את זה, המערכת תפסה תאוצה וזה הגיע למצב שהרבה אנשים פיתחו ותיחזקו את המערכת עד שכבר לא היה את מי לתבוע ולא הייתה אפשרות להוריד את זה וכך נוצר ותפס תאוצה לינוקס.

אפל (לקריאה עצמית)

לאפל הייתה מערכת הפעלה משלהם שעבדה יותר טוב ממערכות הפעלה אחרות.
עם זאת, הלקוחות רצו מולטי-טסקינג ולכן אפל היו צריכים מערכת הפעלה חדשנית ומעודכנת שמבוססת יוניקס אבל הם נתקעו עם זה והם חיפשו מה לעשות.
הם תכננו להחליף את מערכת ההפעלה שלהם, ולכן קנו את NeXT והשתמשו במערכת ההפעלה NeXTSTEP שלהם שמבוססת יוניקס.
1999: אפל איחדו את NeXTSTEP לכדי יצירה של מערכת ההפעלה Mac OS X שלימים הפכה להיות MacOS.

היחס בין לינוקס לאפל: הארכיטקטורה של אפל שונה לגמרי אבל השימוש ב-System Programing זהה לגמרי בין אפל ללינוקס.

- **NeXT** - חברת מחשבים אמריקאית שנוסדה ע"י סטיב ג'ובס ב-1985. התפרקה לאחר שנקנתה ע"י אפל ב-1996.
- **NeXTSTEP** - מערכת הפעלה מבוסס יוניקס, מונחה עצמים התומכת במולטי טסקינג.

רישיון לתוכנה חופשית

שנות ה-80: באוניברסיטת MIT הייתה מדפסת מאוד גדולה ומרכזית.
ריצ'רד סטולמן הוסיף פיצ׳ר למדפסת שהדפיסה עוד הדפסת כותרת לדפים.
יום אחד Xerox החליטו לשדרג את התוכנה במדפסת מה שגרם לפיצ׳ר להיעלם.
ריצ׳רד ביקש שיכניסו לתוכנה שלהם את הפיצ׳ר שלו אך הם סירבו בטענה שזה יחשוף את הקוד שלהם. הם גם לא הבינו למה הוא מתלונן שכן קיבל שדרוג בחינם.
זה לא היה מקובל עליו, אז הוא שכר עורך דין וכתב את הרישיון לתוכנה חופשית, שידוע גם בתור "הרישיון הציבורי הכללי של GNU".
לפי הגדרה זו, תוכנית תוגדר כתוכנה חופשית אם היא עומדת ב-4 העקרונות הבאים:

א) התוכנה היא חופשית וניתן להריץ אותה בכל צורה ולכל מטרה.
ב) לא לוקחים אחריות על שום דבר.
ג) כל אחד יכול לקבל את הקוד של התוכנית תמורת עלות הפקת הקודת.
הערה: **כיום לא לוקחים תשלום, אך בעבר היה נהוג לגבות תשלום סמלי על הדיסקט או סידי.**
ד) התנאי היחיד להפיץ derived code הוא להשאיר אותו קוד פתוח.

- **Xerox** - חברה אמריקאית שמוכרת מדפסות.
- **MIT** - המכון הטכנולוגי של מסצ׳וסט (אוניברסטה פרטית).
- **Derived Code** - קטע קוד שנלקח כולו או בחלקו מקטע קוד קיים. (בעקרון מדובר בכל עבודה שנובעת מעבודה אחרת.
לפי GPL גם ספרייה שהתלנקג׳ו איתה יוצרת Derived Work. יש רישיון נוסף LGPL שמאפשר שימוש בספריות וסגירת הקוד).
בכל עבודה תעשיתית ובכל שימוש בספרית קוד פתוח מומלץ לבקש חוות דעת מהמחלקה המשפטית של הארגון).

הבהרה בנושאי רישיונות - הסיכום אינו מדויק ומיועד לתלמידי מדמ"ח ולא כתחליף לחוות דעת משפטית.

הפצה של לינוקס

- למעשה נרצה התקנה נוחה של לינוקס.

Linux Distribution (הפצה של לינוקס) - מערכת הפעלה המכילה את ליבת (קרנל) מערכת ההפעלה לינוקס, ספריות, קבצים שונים וסביבת עבודה לשם תפעול המחשב. לרוב מבוססת תוכנה חופשית וקוד פתוח.

יש 2 דרכים עיקריות להתקנת הפצה של לינוקס:
דרך 1 - באמצעות הפצות כגון Ubuntu, Red Hat ועוד... שמספקות כלי מערכת ניהול חבילות ואורזות את ההתקנה בצורה קלה ונוחה למשתמש עם תמיכה.
דרך 2 - לשבת ולהוריד חבילות קוד פתוח ולקמפל אותם אחד אחד יחד עם הקרנל, תוך כדי שמתאימים אותו למערכת (לחומרה) שלנו.
זהו תהליך ארוך ומייגע שחוזר על עצמו בכל פעם כשצריך לשדרג חבילה.
בעיה הנוספת היא שאין לנו גירסא שמישהו יכול לבדוק ולהבטיח שהתוכנה שלו עובדת על המחשב שלי.
דוגמה- נניח אני אורקל ואני רוצה להבטיח שהמוצר שלי עובד על הלינוקס של הלקוח, אז אני רוצה להתקין את אותה מערכת בדיוק אצלי אבל אם מה שמותקן אצלי זה כל מיני דברים שאספתי מכל מיני מקומות אז אורקל בחיים לא יצליחו להבטיח שהכל עובד.

- אנחנו לא נתעסק בניהול של לינוקס ולא בהבדלים בין מהדורות - רק בקידוד System Programming.

דרישה: לוודא שכל התרגילים מתקמפלים ורצים בגירסא Ubuntu 20/22 של הקורס לפני הגשה.
ניתן לקודד בכל סביבה (בדרך כלל השינויים מאד פשוטים).

שיעור 2

פונקציות ספריה

דוגמאות: printf(3), strcmp(3).

היתרון: שכותבים פעם אחת ואז הספרייה נגישה לכולם.

מתי זה מאוד חשוב? אם כתוב בפירוש שאסור לממש פונק׳ (למשל בעולמות הקריפטו קיימת הנחייה שכזו) או כאשר רוצים לחסוך זמן תכנות ובדיקות.

למה זה טוב שכולם משתמשים למשל באותה הצפנה וספרייה ולא כל אחד לעצמו? לפעמים המפתחים כותבים באגים, ובמוצר אבטחת מידע יהיו המון באגים ולכן המשתמשים יצטרכו לתקן את זה בעצמם.
בספרייה, לעומת זאת, אפשר לשנות את הקוד ולהשתמש בו שוב ושוב.
אז יש עותק אחד מאוד בדוק של קוד והסיכוי שיש בו באג הוא נמוך מאוד.

- מקרה חריג לדוגמה - באג כגון HEARTBLEED הפיל את כל העולם - התרחש ב-2014 ומאז לא קרה.
  לפני כן לא היה ידוע על באגים בספריה הזאת (OPENSSL) במשך 17 שנה.

האם פונק' send(2) המטפלת בסוקטים היא פונק' ספרייה? **לא**. פונק' ספריה הינה פונקציה שהיית יכול לכתוב בעצמך אבל מישהו כתב אותה בשבילך (ראה דוגמאות לעיל).
לעומת זאת, בפנייה לפונק' send(2) לא פונים לספריה אלא פונים למערכת ההפעלה כדי שתבצע שירות מסוים, כי למתכנת אין בכלל הרשאה לכתוב ישירות לכרטיס הרשת.
מכיוון שבדוגמאות לעיל זה דברים שבשליטת המתכנת לעומת send(2) שמבקש שירות שאינו בשליטת המתכנת אלא רק בשליטת מערכת ההפעלה, אז זוהי אינה פונקציית ספרייה.

למה זה ככה? במקרה שאין מנהל (Supervisor) אז יכולה להיות דריסה מיותרת בין תוכנות ועבודות.

אז איך נתאם? נגדיר את הפונק' ספריות הללו בתור מנהל. מנהל זה במחשב שלנו זוהי מערכת ההפעלה, וכל פעם שאנחנו רוצים לעשות דברים שנוגעים בחומרה או בפרוססים, אנו נפנה ל-Supervisor הנ"ל.

Man (manual)

- **חלק 1 – man(1) : פה נמצאות כל הפקודות שניתן להריץ בCMD, למשל ls(1), man(1) וכו'**

- **חלק 2 – man(2) : פה נמצאות כל הפונקציות שהן קריאות מערכת (SysCall).**
  למשל כל הפונק' שמכירים מקורס תקשורת.

- **חלק 3 – man(3) : פה נמצאות מרבית פונק' הספרייה. בדר"כ מדובר בספרייה סטנדרטית.**

מה קורה עם יוצאי דופן למשל PRINTF MAN? דף ה-man מתאר את הפונקציה הזאת (printf(1)) ולא את הפונקציה מהספרייה הסטנדרטית printf(3). לפי מה שלמדנו זהו אמור להיות בחלק 3,
אז מדוע הוא נמצא גם בחלק 1? ישנן דוגמאות נוספות לדברים הללו.
קיים גם SHELL BUILT IN בבאש (כלומר הוראה ב-bash(1) שהוא ה-SHELL שאנחנו משתמשים בו בברירת מחדל) המבצע עבודה הדומה ל-printf(3).

אנשים לא משתמשים בפונקי' הללו כמעט כבר, אך הן עדיין קיימות.
משמע יש כפילות כלשהי בשמות הפונקי', אלו פונקי' שונות לגמרי ולא חולקות את אותו ממשק, כמתואר לעיל.

- דוגמה נוספת להדגשת ההבדל - באג כגון הפונקי' kill(2).
  קיימת גירסא של קריאת מערכת (ראה שיעור 3) וגם גירסא המורצת משורת הפקודה (kill(1))

ברירת המחדל היא להציג את המדריך מהרמה הנמוכה ביותר ולעלות במקרה שיש יותר מדף אחד מתאים.
אם מוצג מדריך לא נכון ניתן להקיש בשורת הפקודה:

```
man 3 printf
man 2 kill
```

ולקבל את המדריך הנכון ל-printf(3) או kill(2).

Supervisor Vs. Hypervisor:

תהליכים לא אמורים לגעת בחומרה. בשביל זה יש את ה-SUPERVISOR.
נניח שיש לכם תוכנה כמו VM כדי לאפשר לשני מערכות הפעלה לגעת במכונה. אז אנחנו כן רוצים לנהל את התהליך בצורה יותר חכמה. בגדול זה HYPERVISOR שהוא מחוץ לתחום ולחומר הנדרש.
היפר-ויזור משמש כמו מערכת הפעלה למערכות הפעלה והוא מעל ל-Supervisor ומכאן שמו.

SYSCALL - זוהי הדרך שלנו בעצם לקרוא למערכת ההפעלה.
זוהי הוראת אסמבלר שונה מ-CALL המשמשת להיכנס לפונקציה.
לעומתה SYSCALL גם נכנסת לפונקציה (קריאת המערכת) וגם עולה הרשאות לרמת מערכת הפעלה ורשאית למשל לבצע פעולות על חומרה.

כיצד SYSCALL עובדת? הפרמטר שנכניס לפונקציה הוא בעצם מספר שאנו מכניסים ל-SYSCALL וככה מערכת ההפעלה יודעת איזה שירות אנו מבקשים מהOS לספק לנו. בין אם אני משתמש ב-send(2) בשפת C או בפסקל, הם יגשו לאותו SYSCALL.
אבל מה קורה בג'אווה או פייתון? יש לנו BYTECODE או סקריפט פייתון שאותו אני מריץ.
בעצם יש אינטרפרטר שקורא את ההוראות ושבסופו של דבר ממיר את הקוד לשפת מכונה ולקריאה ל-SYSCALL ולמערכת ההפעלה, למרות שזה לא נראה כך ברמת כתיבת הקוד.
בסופו של דבר - יש לנו רק ממשק אחד לחומרה וזה ממשק מערכת ההפעלה.
אם לדוגמא כתבנו קובץ אז בסוף פנינו לקריאות המערכת של מערכת ההפעלה גם אם הקוד נכתב בג'אווה או בפייתון ולא נראה שקראתי למערכת הפעלה.

רמות הרשאה

בארכיטקטורת אינטל יש לנו 2 רמות הרשאה בשימוש (רמה 1 מייצגת את ה-HYPERVISOR ולא נלמד).
הרשאות אלו נקראות גם RING.

- **רמה 0 : מייצגת את מערכת ההפעלה.**
- **רמה 3 : מייצגת את המשתמש (USER).**

בארכיטקטורת ARM יש (פרט להרשאות שמחוץ לסקופ של הקורס):

- **הרשאה EL0 : לתיאור המשתמש.**
- **הרשאה EL1 : לתיאור מנהל מערכת.**

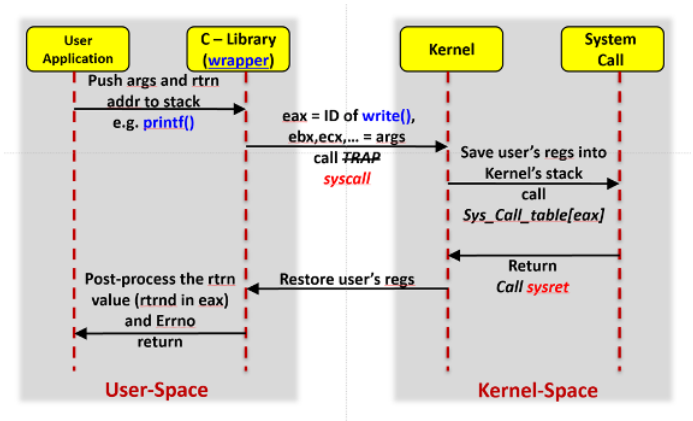
נשתמש במונח לעלות רמות הרשאה (כמו ב-ARM) כדי לתאר עליה למערכת ההפעלה למרות שבאינטל זה "לרדת". פשוט כי לעלות זה "יותר".

- קיים ממשק של הפרעות (פסיקות) למערכת ההפעלה באמצע ריצה. לא נדבר על כך בקורס.

חלוקה של קריאות המערכת:

- System calls can be roughly grouped into ***five*** major categories:
  - Process control (e.g. create/terminate process)
  - File management (e.g. read, write)
  - Device management (e.g. logically attach a device)
  - Information maintenance (e.g. set time or date)
  - Communications (e.g. send messages)

איך הדברים עובדים?



קונבציה: נהלי ה-MAN, כפי שמתואר לעיל, כאשר אנו שמים בחשיבות גבוהה את חלקים 1,2,3 אנחנו נקרא לפונקי' לפי החלק שלהם שהם נמצאים ב-MAN.

ניהול תהליכים

fork(2)

fork(2) - הפונק' היחידה ב-C שאנחנו קוראים לה פעם אחת וחוזרים ממנה פעמיים, פעם לאבא פעם לבן.
הפונק' חוזרת לאבא עם PID וחוזרת לבן עם הערך 0.
נניח יצרתי תהליך, פתחתי סוקטים, קובץ וכדומה. כאשר אני אשכפל אותם, הם יהיו משוכפלים גם כן לתהליך בן. (אם יש פרוסס עם כמה THREAD אז הוא לא משוכפל.)

- חשוב לציין: כל מה שקרה עד ה-fork(2) משוכפל אבל כל מה שקורה לאחר ה fork(2) ישפיע רק על התהליך שקרה (לאבא או לבן) אבל לא ישפיע על תהליכים אחרים.

קונבציה: המונחים בעברית בלשון זכר (אבא - בן) הם המונחים המקובלים בתעשייה.
המונחים באנגלית parent child גם הם מונחים מקובלים בתעשייה.
לא נהוג לדבר בעברית על תהליך הורה וצאצא או תהליך FATHER BOY באנגלית.

ההבדל בין Thread ל-Process? אם יש לנו THREAD, הכל (נדון בהמשך לא מדויק כרגע) ביניהם משותף לאורך כל הזמן, ואין שכפול.
לעומת זאת, אם אני תהליך אז הכל נפרד, טבלת פסיקות נפרדת, קבצים נפרדים, וכדומה.

בסטנדרט UNIX יש רק טרד ופרוסס. בלינוקס, יש לנו את הפונק' CLONE(2) שמתאימה רק ללינוקס, אנחנו לומדים UNIX לפי הסטנדרט אז לא נשתמש בה, אבל ב-CLONE(2) אתה יכול בעצם לברור ולהחליט מה משותף ומה לא. (לקורס הזה התאור של clone(2) מספיק. למידע נוסף - יש גוגל).

CopyOnWrite - כל התהליך הזה של העתקת זיכרון יכול להיות מאוד מבזבז זיכרון.
בשביל זה עושים אופטימיזציה באמצעות התכונה COPYONWRITE להבא (CoW) של fork(2) שהיא הרבה יותר יעילה בבה אנחנו נחזיק רק מצביעים לדפי זיכרון (דפים) ונשכפל אותם רק במידה ונשנה אותם.
(אם לא נשנה את הדף תהליך האבא והבן יכולים להצביע לאותו דף).

- דוגמת קוד על כישלון FORK ודוגמת הרצה של FORK ומה יודפס נמצאים במצגת.

flush() - פונק' אשר מרוקנת את הבאפר של מערכת ההפעלה.

האם cout ב-$C++$ כותב לבאפר? לא בהכרח. $C++$ משתנה כל תקופה.
בסטנדרטים המוקדמים של $C++$ אז cout עבד ללא באפרים.
בסטנדרטים האחרונים של $C++$ הם שינו את ההתנהגות (והוסיפו באפרים) להלכה.
זה אומר שנוכל לראות התנהגות שונה בין גירסאות שונות של הקומפיילר של אותו קוד
- לדוגמה - בגירסא ישנה נראה פלט ובגירסא חדשה לא נראה פלט - כי הוא יישאר בבאפר.
  - זה לא קורס CPP אז לא נתעמק בסטנדרטים וכו'.

מה שורת הקוד הראשונה שרצה כשמריצים תוכנה? הקוד שרץ לפני שמריצים את ה-MAIN זהו ה-START, CTOR מסוים שקורא ממערכת ההפעלה לשורת ה-MAIN בקוד.

wait(2)

אבא יכול לקבל את הערך שהבן חזר (הסתיים) איתו. האבא מקבל אינפורמציה איך הבן הסתיים,
כלומר מקבל עדכון כאשר הבן מסיים לרוץ והאם הוא סיים בהצלחה או כישלון.
(זהו הערך שה-MAIN של הבן מחזיר. אם הבן תמיד מחזיר אותו ערך אז כמובן אי אפשר לדעת).
על מנת לחכות לקבל את האינפורמציה הזו קיימת לנו פונקי' (2) WAIT.

wait(2) - מה שנקרא גם תהליך זומבי.
תהליך זומבי איננו תהליך (למעשה הוא לעולם לא יקבל מעבד או ישנה את מצבו או ישפיע על המערכת).
הוא פשוט מחכה שהאבא יקרא ל-wait(2).
כאשר האבא קורא ל-wait(2), הערך חזרה של הבן יחזור לאבא (לפרמטר status של wait(2)).
אם האבא לא יקרא ל-wait(2) ויסתיים, הבן יאומץ על ידי INIT.
במהלך הריצה של המערכת INIT לא עושה כלום – רק מחכה לכל הבנים וככה הזומבי יעלם בסוף.

יש לנו 3 פונק' WAIT מגרסאות שונות של סטנדרטים שונים.
בכל מיני הסתעפויות שינו את ה-SYSCALL הזה לפי דרישות ה-OS.

- **INIT** - תהליך מספר 1, האב הקדמון של כל התהליכים ב-UNIX.

execXX

כיצד להריץ תהליך: עד עכשיו אני יודע להריץ תהליך זהה לתהליך הקיים (שכפול).
על מנת להחליף את התהליך הקיים, קיימת פונקי' execve(2) ומשפחת פונקציות EXEC.
קיימות סה"כ 6 פונקציות EXEC השונות בפרמטרים וכולם קוראות ל-execve(2) בסוף.
כאשר אנחנו קוראים ל-execXX (כלומר לאחת הפונקציות במשפחה) לאחר fork(2) היא דואגת שה-OS
ישחרר את הזיכרון של התהליך ויתן את המשאבים לתהליך החדש שהורינו לרוץ (עם פונקציות execXX).

למה זה ככה? ככה. אך זה לא כזה גרוע כי השימוש ב-CoW אומר שאנחנו מפסידים הרבה פחות משאבים.

SYSTEM - ספריה שמריצה את 3 הפונקציות ברצף: WAIT-EXEC-FORK.

ניהול שגיאות

perror(3) - באמצעות המשתנה הגלובלי שנקרא ERRORNO (קבוע המסמן איזה ERROR קרה),
כאשר עושים perror(3) תודפס ההערה הרלוונטית לפי הטבלה הקיימת במערכת ההפעלה
על פי ה-ID ששמור ב-ERRORNO.
לא מדויק: המשתנה ERRORNO הוא THREAD SPECFIC STORAGE אותו לא נלמד, אז נשתמש במונח גלובלי.
הערה: גם ב-Windows יש פתרון דומה בעזרת פונקציה GETLASTERROR.

שיעור 5

Blocking

Blocking (חסימה) - ישנן פונקציות או קריאות מערכת שחוזרות,
לדוגמה - פונקציית power שחוזרת כאשר הסתיים החישוב.
וישנן פונקציות שחוזרות רק כשקורה משהו טוב (כלומר תנאי חיצוני),
לדוגמה - פונקציית scanf שתלויות משתמש כלומר אם המשתמש לא מקליד כלום,
הפונקציה תימשך לנצח - כלומר הפונקציה לא תחזור.
דוגמה נוספת - פונקציית wait בה אני מחכה לתהליך ילד (גם היא פונקציית בלוקינג
כי לא מובטח שהילד יסתיים).
להלכה תהליך יכול להימשך לנצח ולעולם לא לחזור.

מה הבעיה עם פונקציות Blocking? הבעיה איתן היא שאני לא יכול לעבוד במקביל.
למשל, אם יש שרת ואני מחליט לתת שירות ללקוח באמצעות פונקציית recv כדי להאזין לבקשה,
אז כרגע אני אתקע על אותה האזנה ולא אוכל לעשות דברים אחרים.

פתרונות אפשריים: ישנם מספר פתרונות אפשריים:

- **פיתרון 1 :** לפתוח כמה תהליכים עם fork(2) ככה שכל תהליך יטפל בלקוח אחר.

- **הבעיה :** הבעיה עם זה היא קודם שאנחנו יוצרים המון תהליכים וזה יקר.
בנוסף, במערכת שעושה אינטראקציה עם הלקוחות במקביל ואני רוצה לעשות השוואה
שדורשת זיכרון משותף זה לא יעבוד.

- **פיתרון 2 :** שימוש ב-Threads (הם עם זיכרון משותף ולא תוקעים אחד את השני).

- **הבעיה :** לא למדנו עדיין - נלמד בהמשך. (וגם זה עדיין יקר).

- **פיתרון 3 :** שינוי ה-File Descriptor לאחד שלא חוסם, כלומר אם נקרא ל-read או ל-receive
על הסוקט, היא לא תחסום אלא תחזור מיד אם לא הגיע שום דבר.

- **הבעיה :** זה פותר חלקית את הבעיה כי אם יש לי כמה סוקטים שהפכתי אותם ללא חוסמים,
ואני קורא אחד אחד לכל סוקט וייתכן שיש לו תוכן והוא חוזר או שאין לו והוא חוזר.
במצב כזה אנחנו עוברים בלי הפסקה בלולאת while ללקוחות וזה מביא את המעבד לניצולת
מקסימלית ללא עבודה מעשית (אנחנו "פול גז בניוטרל" שואלים "שלחת לי משהו?" "שלחת לי משהו?"
וכולי ללא הפסקה) וזה חונק את התהליכים האחרים.

למטרה זו נדבר על IO multiplexing שכוללות פונקציות שיאפשרו לנו להשיג תת קבוצה
של File Descriptors חמים כלומר כאלו שאם נעשה עליהם פעולה, מובטח שיחזרו מיד ואוכל לטפל בהם.
כלומר, נרצה פונקציה שתחכה על קבוצה של File Descriptors שהם בלוקינג ותחזיר לי את אלו שחמים.

לכן, היום נדבר על פונקציות **select** ו-**poll** שנועדו לעקוף את הבעיה הזאת.

select(2)

select - זו הפונקציה הישנה יותר בהשוואה ל-poll.

ה-select מקבל 3 קבוצות של File Descriptors ומחזיר תתי קבוצות של ה-File Descriptors החמים. הפונקציה מקבלת 5 פרמטרים.
הפרמטר הראשון מיועד לאופטימיזציה ואומר עד לאיזה File Descriptor נרוץ.
הפרמטר האחרון הינו Timeout לזמן שאני נותן לענות.
הפונקציה מחזירה int. היא תחזיר 1- במקרה של כישלון. אחרת הפונקציה הצליחה.

הערה: הפונקציה select בעייתית כי חלק מהפרמטרים נועדו גם לקלט וגם לפלט כלומר היא דורסת את הפרמטרים שהיא מקבלת ולכן צריך לשמור אותם.

השימוש ב-select: נכלול בקוד את ההדרים:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
```

ומבנה הפונקציה הינו:

```
int select(int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

- **numfds** - ה-fd המקסימלי שקיבלנו + 1.
- **readfs**, **writefs**, **exceptfds** - 3 קבוצות של fd שברצוננו לקרוא, לכתוב או לקבל מהם שגיאות.
- **timeval** - Struct שיגדיר לנו כמה זמן ה-select יחכה (הטיימאוט).

ה-strcut מוגדר באופן הבא:

```
struct timeval {
    int tv_sec;
    int tv_usec;
};
```

- **tv_sec** - כמה שניות אנו רוצים שיחכה. נמדד מ-1.1.1970.
- **tv_usec** - כמה מיקרו שניות אנו רוצים שיחכה.

ישנן מספר פונקציות שחשוב להכיר כדי לשחק עם הקבוצות:

פונקציה	שימוש
FD_SET(int fd, fd_set* set)	מכניס את fd לקבוצה
FD_CLR(int fd, fd_set* set)	מוציא את fd מהקבוצה
FD_ISSE(int fd, fd_set* set)	מחזיר אמת אם fd חלק מהקבוצה
FD_ZERO(fd_set* set)	מחזיר קבוצה ריקה

דוגמה בסיסית: מחכה 2.5 שניות עד שהמשתמש יגיב (יופיע משהו ב-standard input)

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#define STDIN 0  // file descriptor for standard input

int main(void)
{
    struct timeval tv;
    fd_set readfds;

    tv.tv_sec = 2;
    tv.tv_usec = 500000;

    FD_ZERO(&readfds); // איפוס הקבוצה
    FD_SET(STDIN, &readfds); // הכנסת הסטנדרט אינפוט לקבוצה

    // don't care about writefds and exceptfds:
    select(STDIN+1, &readfds, NULL, NULL, &tv);

    if (FD_ISSET(STDIN, &readfds)) // בדיקה אם הם חמים
        printf("A key was pressed!\n");
    else
        printf("Timed out.\n");

    return 0;
}
```

לחצו לדוגמה מורכבת יותר של שרת צ'אט

poll(2)

poll - זו הפונקציה החדשה יותר בהשוואה ל-select.
השוני מ-select הוא ש-poll לא מקבל קבוצות ולא דורס את הארגומנט שהוא מקבל.
הפונקציה מתחלקת ל-2 סוגים:

השימוש ב-poll: נכלול בקוד את ההדר הבא:

```
#include <poll.h>
```

ומבנה הפונקציה הינו:

```
int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

- ***Fds*** - מערך של fd's עליהם נרצה לנטר.
- **nfds** - כמות האיברים במערך.
- **timeout** - כמו ב-select (כמה זמן אנחנו מוכנים לחכות אבל במילישניות).
- pollfd - Struct שיגדיר לנו file descriptor ואיך אנו רוצים שה-poll יתייחס אליו.

ה-strcut מוגדר באופן הבא:

```
struct pollfd {
    int fd;
    short events;
    short revents;
};
```

- **fd** - ה-file descriptor.
- **events** - האירוע שאנו מעונינים בו. מדובר ב-bitwise OR של POLLIN עבור קריאה ו-POLLOUT עבור כתיבה.
- **revents** - כאשר פונקציית poll חוזרת, יכיל את האירועים שבאמת קרו.

דוגמה בסיסית: מחכה 2.5 שניות עד שהמשתמש יגיב (יופיע משהו ב-standard input)

```
 include <stdio.h>
#include <poll.h>

int main(void)
{
    struct pollfd pfds[1]; // More if you want to monitor more

    pfds[0].fd = 0;          // Standard input
    pfds[0].events = POLLIN; // Tell me when ready to read

    // If you needed to monitor other things, as well:
    //pfds[1].fd = some_socket; // Some socket descriptor
    //pfds[1].events = POLLIN;  // Tell me when ready to read

    printf("Hit RETURN or wait 2.5 seconds for timeout\n");

    int num_events = poll(pfds, 1, 2500); // 2.5 second timeout

    if (num_events == 0) {
        printf("Poll timed out!\n");
    } else {
        int pollin_happened = pfds[0].revents & POLLIN;

        if (pollin_happened) {
            printf("File descriptor %d is ready to read\n", pfds[0].fd);
        } else {
            printf("Unexpected event occurred: %d\n", pfds[0].revents);
        }
    }

     return 0;
   }
```

לחצו לדוגמה מורכבת יותר של שרת צ'אט