חלק שני – ניהול תהליכים
Process Management

פרק 4: תהליכים Processes

מערכות מחשב מוקדמות הריצו תכנית בודדת בכל נקודת זמן. התכנית שלטה על המחשב בכללותו: המעבד, הזיכרון, הציוד, וקיבלה גישה לכל משאביו.

בימינו: מחשב מריץ מספר תכניות המצויות בו זמנית בזיכרון, ע"י שהמעבד מריץ כל תכנית לסירוגין למשך זמן קצר == שיתוף זמן.
כדי לאפשר אופנות פעולה כזאת, תוך שמירה שהתכניות לא תפגענה זו בזו יש להצר את צעדי התכנית: למנוע ממנה ביצוע של פעולות מסוימות באופן עצמאי. כדי לבצע פעולות אלה התכנית תיאלץ לבקשן כשירות ממ.ה. (לדוגמה: הקצאת שטח זיכרון, ביצוע ק\פ, ייצור תכנית נוספת שתורץ)

התוצאה: הולדת מושג התהליך =+- תכנית מורצת. או יחידת העבודה, 'ישות', במערכת שיתוף זמן מודרנית.

מעבר לכך: מטעמי יעילות ונוחות מ.ה. מבצעת משימות שונות עבור המשתמשים (ולא רק מאפשרת הרצת משימות ע"י המשתמשים): shell, דפדפן, משימות שונות כגון: שליפת התאריך, הצגת קבצים במדריך.
התכניות המבצעות משימות אלה נקראות תכניות מערכת (system programs), הן חלק ממ.ה., אך לא מגרעין מ.ה. הן מורצות במקביל לתכניות המשתמשים, במצב משתמש, ובמידת הצורך, כמו כל תכנית אחרת, פונות באמצעות ק.מ. לגרעין מ.ה. לשם קבלת שירות הדרוש להן.

גרעין מ.ה. אחראי על ניהול התהליכים, הקצאת הזיכרון הראשי, ניהול הציוד. מספק שירותים בסיסיים בלבד.

4.1 מושג התהליך Process Concept

במערכת אצווה (batch) הריצו ג'ובים כל ג'וב הורץ לבד ע"י המחשב. עת ג'וב א' הסתיים בחרה מ.ה., מבין הגו'בים הממתינים, ג'וב ב' והריצה אותו.
התוצאה: עת הג'וב המורץ מבצע ק\פ המעבד מושבת. לא חראם?

הרעיון\שיפור: מערכת ריבוי תכניות (multiprogramming): בזיכרון יוחזקו כמה תכניות\ג'ובים במקביל. עת ג'וב א' מבצע ק\פ, יריץ המעבד את ג'וב ב'. עת ג'וב ב', *מרצונו הטוב*, יוותר על המעבד (כדי לבצע ק\פ, או משום שהוא סיים) יחזור המעבד להריץ את ג'וב א' (בהנחה שהוא כבר סיים לבצע את הק\פ שלו).

הרחבה של הנ"ל: המעבד יעבור לבצע את ג'וב\תהליך\משימה ב', לא רק עת תהליך א' פנה לבצע ק\פ, אלא גם אם תהליך א' כבר רץ x יחידות זמן, ועל-כן עתה מפקיעים\גוזלים ממנו את המעבד לטובת משנהו (סיבה אחרת לגזלה: קדימות תהליך ב' גבוהה יותר). לשיטה זאת אנו קוראים: שיתוף זמן (time sharing).

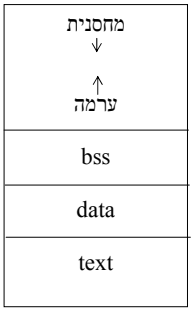
4.1.1 התהליך

תהליך =+- תכנית בהרצה

מרחב הכתובות (address space) של התהליך כולל:

א. פקודות התכנית (בשפת מכונה) = מקטע הטקסט (text segment)
ב. מחסנית התהליך (פרמטרים ומשתנים לוקליים)
ג. הערמה (להקצאה דינאמית)
ד. מקטע הנתונים (הסטטיים והגלובליים המאותחלים) data section
ה. bss = block started by symbol = גלובליים וסטטיים שאינם מאותחלים (וע"כ אינם נשמרים בַקובץ המכיל את התכנית)
ו. מרכיבים נוספים אותם נכיר בהמשך: מקטעי זיכרון משותף, קבצים ממופים לזיכרון, ועוד.

מעבר לכך, שומרת מ.ה. במבני נתונים שלה מידע נוסף אודות התהליך (איזה קבצים הוא פתח, קדימותו, כמה זמן הוא כבר רץ, מצב האוגרים שלו עת בוצעה החלפת הקשר, ועוד).



כל הנ"ל = מרחב הכתובות של התהליך (address space), לכתובות אלה, ורק אליהן, הוא רשאי לפנות.

לוגית אנו חושבים על מרחב הכתובות כעל רציף בזיכרון,
פיזית, עת נדבר על paging נראה שהוא אינו.

עת תהליך מוליד תהליכים ילדים,
יהיה כ"א מהם, לכל הפחות בהיוולדו,
עותק זהה אך נפרד של אביו.

מתי תהליך נוצר (נולד)

א. עת מ.ה. עולה היא מייצרת את התהליך (כמעט) הראשון במערכת: INIT, pid=1, אשר ירוץ לנצח.
ב. עת עושים login מייצר התהליך הנ"ל בן = ה: login shell. אשר עובר 'מוטציה' ונהפך מעותק של INIT ל: shell
ג. עת מקלידים פקודה מייצר ה: shell תהליך שיריץ את התכנית שמממשת את הפקודה.
ד. מ.ה. וכל תהליך אחר, עשויים לייצר תהליכים ממגוון של סיבות (כדי לתת שרות הדפסה, כדי למקבל עבודה הנעשית ע"י התכנית).
ה. עת נמסר מכלול של תהליכים (קובץ script) נוצרים סדרתית תהליכים.

מתי מסתיים תהליך

א. הוא סיים את פעולתו (בהצלה או מתוך כישלון) וביצע ק.מ. exit().
ב. הוא מועף ע"י מ.ה. בשל בעיה בהרצתו:
הוא חילק באפס,
חרג מהזיכרון,
חרג ממכסת הזמן שהוקצתה לו,
ניסה להגדיל קובץ מעבר למותר,
ניסה לבצע פקודה מיוחסת במצב משתמש.
ג. מ.ה. מורה לו לסיים 'שלא בעוונותיו': חסימה הדדית.
ד. אביו מחליט, מסיבות השמורות עימו, להרגו.
ה. המשתמש עושה logout.

כאמור, תהליך נולד ומת במצב גרעין (למה? בעצם)

4.2. מצב התהליך

תהליך (==תכנית בהרצה) משנה את מצבו עם הזמן.
מצב התהליך =+- מה התהליך עושה עתה.

מצבים אפשריים של תהליך:

א. חדש (new)= התהליך נוצר ('נולד') ע"י מ.ה.., מוקצים לו מבני הנתונים הדרושים שיאפשרו לו 'חיים'=לרוץ במערכת.

ב. התהליך רץ במצב משתמש (running, user mode)= הוא זכה במעבד, ומבצע פקודות בתכנית שאינן מחייבות שירות ממ.ה.

ג. התהליך רץ במצב גרעין (running, kernel mode) = התהליך זקוק לשירות כלשהו ממ.ה. (קריאת נתון, בדיקת התאריך). לשם כך הוא מריץ קטע קוד של מ.ה. המספק את השירות. ק.מ. היא הכלי שמאפשר לתהליך לעבור ממצב ב' ל-ג'.

ד. התהליך חסום\ישן (blocked/sleeping) = התהליך ממתין לאירוע כלשהו (יוזן לו נתון, יישלח לו סיגנל). למצב זה נעבור ממצב ג'.

ה. התהליך מוכן להרצה (ready) = האירוע לו התהליך המתין חל, והתהליך ממתין שהמעבד יוקצה לו (שוב) ע"מ שהוא יוכל להמשיך לרוץ. למצב זה נעבור ממצב ד', או, במערכת שיתוף זמן בסיסית, גם ממצב ב', במערכת שיתוף המאפשרת גזלה של המעבד מתהליך הרץ במצב גרעין, נוכל להגיע למצב זה גם ממצב ג' (גרסות מוקדמות של לינוקס, למשל, לא אפשרו גזלה של המעבד במצב גרעין. החל מגרסה 2.6 הדבר השתנה).

ו. התהליך סיים (ביצע ק.מ. exit), הוא zombie וממתין שאביו יתעניין בגורלו, ורק אז הוא יעלם סופית מהמערכת. למצב זה נעבור ממצב ג'.

ז. כפי שנראה בפרק 10#: לעתים, בשל עומס על המערכת, חלק מהתהליכים החסומים או המוכנים מוצאים מהזיכרון לדיסק (אשר מהווה מעין הרחבה של הזיכרון). תהליכים כאלה נקראים מושהים (suspend). על-כן את התהליכים החסומים והמוכנים נפצל לכאלה המצויים בזיכרון, ולכאלה שהושהו. הכנסה של תהליך מושהה לזיכרון נקראת עירור (activation).

הערות:

א. את המוכנים נחלק למוכנים במצב גרעין לעומת מוכנים במצב משתמש.

ב. למעשה את התהליכים הממתינים\חסומים אנו מחלקים לקבוצות, קבוצות (תורים, תורים) ע"פ האירוע לו התהליכים ממתינים (ק"פ מציוד א', קלט מהעכבר, נתונים מהרשת).

ג. כל תור מכיל את ה- PCB (כפי שיתואר מייד) של התהליכים הממתינים באותו תור.

ד. עת מתקבלת פסיקה, (ומטופלת בלי קשר לתהליך שמורץ עתה, או לזה שהפסיקה רלוונטית אליו) חלק מהטיפול בפסיקה יהיה, במידת הצורך, העברת התהליך המתאים מהתור בו הוא המתין לתור שני.

ה. תהליך נולד ומת במצב גרעין.

4.1.3 גוש בקרת תהליך

Process Control Block (PCB)

עבור כל תהליך, המהווה כזכור את האובייקט הבסיסי בו מתעניינת מ.ה., שומרת המערכת מבנה נתונים הנקרא גוש בקרת התהליך, והמכיל את כל המידע הדרוש אודות התהליך (בלינוקס נקרא מתאר התהליך וכולל כמאה שדות):

א. מספרו = המזהה שלו, PID = Process ID.

ב. מצבו (כפי שתואר קודם). במידת הצורך: האירוע לו הוא ממתין.

ג. ערך אוגרי התכנית עת התהליך 'נפרד' מהמעבד לאחרונה, בפרט ה- PC, IR, PSW האוגרים הכלליים, אוגר הבסיס והגבול ושפע האוגרים האחרים. כל החלפת הקשר טוענת\שומרת את ערכי האוגרים בשטח זיכרון זה.

ד. הכתובות בזיכרון (או הדפים) המוקצות לו.

ה. מידע תזמון אודות התהליך: קדימותו, כמה זמן עוד נותר לו לרוץ (בעידן הנוכחי).

ו. מידע חיוב: כמות זמן מעבד, וזמן אמיתי שהוא צרך. מגבלות על זמן הריצה, משתמש לו הוא שייך.

ז. מידע על מצב ק\פ: רשימת קבצים פתוחים.

ח. ועוד.

4.2.2 מתזמנים (Schedulers)

לאורך חייו תהליך נודד בין תורי תזמון שונים בהם הוא ממתין לקבלת שרות (מתזמן ארוך טווח, קצר מועד, מתזמן של ציוד כזה או אחר).

בכל שלב על מ.ה. לבחור איזה תהליך מבין אלה הממתינים בכל תור ישורת. הרוטינה שבוחרת את ה-'זוכה המאושר' נקראת מתזמן.

מתזמן ארוך טווח (long-term scheduler, מתזמן הג'ובים) קיים במערכות אצווה: בוחר מי מבין הממתינים בדיסק יוכנס לזיכרון.

מתזמן קצר-מועד (short-term scheduler, מתזמן המעבד): קיים בכל מערכת, מחליט מי מבין המוכנים לריצה יורץ (יזכה במעבד).

ביוניקס, כל מאית השנייה נשלחת פסיקה מהשעון, ומורץ המתזמן קצר המועד. כלומר עליו להיות מהיר.

המתזמן ארוך הטווח יורץ עת תהליך מסתיים ויש לטעון חדש לזיכרון. שולט על דרגת ריבוי התכניות (degree of multiprogramming).

משימתו: לאזן בין כמות התהליכים מוגבלי הק"פ (I/O bounded, אלה שצריכים הרבה ק"פ, ומעט זמן מעבד),

לבין כמות התהליכים מוגבלי המעבד (CPU bounded, אלה שצורכים הרבה זמן עיבוד, ומעט ק"פ),

כך שהזיכרון יכיל תמהיל טוב של שני הסוגים, וכך המעבד וכל מרכיבי הציוד יהיו עסוקים ככל שניתן.

ביוניקס לא קיים מתזמן ארוך טווח. חלקית מחליפו: מתזמן ביניים או משחלף (swapper) – עת כמות התהליכים המורצים גדולה, ולא ניתן לשרתם כהלכה, חלק מהתהליכים מושהים (suspend), ומוצאים לדיסק 'עד שירווח'.

4.2.3 החלפת הקשר (Context Switch)

החלפת התהליך המורץ ע"י המעבד מחייבת שמירה של מצב התהליך המוצא, וטעינת המעבד במצב התהליך המשוגר (dispatched). פעולה זאת נקראת החלפת הקשר.

ה<u>הקשר</u> של התהליך נשמר ב- PCB שלו.

החלפת ההקשר כרוכה בשמירה\טעינה של אוגרי המעבד; היא בגדר תקורה-נלווית (overhead) ועל-כן נשאף למזער את עלותה.

עלות הפעולה היא תלוית חומרה: בכמה אוגרים יש לטפל, והאם יש פקודות מכונה ייעודיות המאפשרות שמירה וטעינה של קבוצת אוגרים באופן אטומי.

4.3 פעולות על תהליכים Operations on Processes

שתי הפעולות הבסיסיות על תהליכים הן יצירה וסיום.

4.3.1 יצירת תהליך (Process Creation)

תהליך עשוי באמצעות ק.מ. ייעודית (ביוניקס שמה הוא fork) לייצר תהליך חדש. התהליך היוצר נקרא הורה, והתהליך הנוצר נקרא ילד. כך ניתן לצור עץ של תהליכים.

יש מערכות בהן הילד מקבל משאבי מערכת משלו, ויש כאלה בהן הוא חולק עם הורהו את משאבי ההורה (ואז תהליך יחיד לא יכול להעמיס את המערכת בצורה לא שוויונית על-ידי העמדת צאצאים רבים).

תהליך נוצר עשוי לקבל נתונים מאביו (בדומה לווקטור הארגומנטים המוכר לנו עת אנו מריצים תכנית מה- shell).

עת תהליך א' מוליד תהליך ב' ייתכן ש:

1. ההורה ימשיך לרוץ במקביל לילדו (זהו המחדל ביוניקס).
2. ההורה יושהה עד שבנו יסיים (ביוניקס ההורה יכול לבקש לנהוג כך).

מבחינת מרחב הכתובות, ייתכן ש:

1. הילד הוא העתק (שיבוט) של ההורה (זה המצב ביוניקס).
2. לילד נטענת תכנית משלו (ביוניקס דבר זה יבוצע כשלב אופציונאלי שני, נפרד; בחלונות ניתן לבקש שעת הילד נוצר הוא יריץ תכנית שונה מהורהו. במצב זה אנו אומרים שהילד מושרץ (spawn) ע"י ההורה.

ביוניקס, כאמור, עת תהליך הורה עם מספר תהליך ייחודי x מבצע ק.מ. fork() נוצר תהליך חדש, עם מספר תהליך שונה, y. שני התהליכים x, y מרצים את אותו קוד, החל בפקודת ה- fork() של ההורה.

כדי שבכל-אופן נוכל לדעת מי משני 'התאומים' הוא ההורה 'המקורי' ומי 'הילד' פקודת ה- fork() מחזירה את הערך אפס בילד, ואת מספר תהליך הילד (מה שקראתי מעל y) בהורה (כך ההורה יוכל גם לחכות לילדו, להרגו, וכולי).

בפרט, במידה ואנו מעוניינים בכך, (בד"כ) הילד יוכל לבצע ק.מ. בשם exec() (בעלת שישה וריאנטים שונים, הנבדלים מעט גם בשמם; נדון בחלקם בהמשך) וכך להחליף את מרחב הכתובות שלו, במלים אחרות, לעבור להריץ תכנית אחרת שתטען לזיכרון, ותרוץ מבראשית.

לדוגמה: עת אנו עושים login תהליך ה- INIT מבצע fork() ובכך יוצר עותק של עצמו. עתה הבן שנוצר מבצע exec() ועובר להריץ את תכנית ה- shell, וכך נוצר ה- login shell שלכם. עת אתם מקלידים ps ב- shell שוב נוצר תהליך...

ההורה עשוי להמשיך בפעולתו ככל העולה על רוחו לצד ילדו (&) או יכול להמתין לַילד (ע"י ביצוע ק.מ. wait() כפי שנכיר בהמשך). ההורה רשאי לסיים לפני ילדו, ואז הילד מאומץ ע"י INIT. מדוע לאימוץ זה חשיבות, ומה רע בתהליך שאין לו הורה (ולו מאמץ) נדון בהמשך.

מייד נפנה לדוגמה, תכנותית, אך לפני כן סטייה קטנה, שתיתן לנו מעט רקע דרוש על עבודה ביוניקס.

המשתנה errno והפונ' perror() ביוניקס

כזכור, הדרך בה תכנית המשתמש מבקשת שרות ממ.ה. (פתיחת קובץ, קריאת נתון מקובץ, יצירת תהליך) היא על-ידי ק.מ.. ק.מ. עלולה גם להיכשל. הנוהל ביוניקס הוא שעת ק.מ. מצליחה היא מחזירה את הערך אפס, ועת היא נכשלת היא מחזירה ערך שונה מאפס וכן מעדכנת משתנה גלובלי (פרטי לכל תהליך או ליתר דיוק פתיל) בשם extern int errno אשר יכיל את קוד השגיאה שחלה. עת ק.מ. מצליחה היא אינה משנה את ערכו של errno. (קובץ הכותר הדרוש: #include <errno.h> בו גם מופיעים הקבועים המתארים את קודי השגיאה השונים. בלינוקס: man 3 errno יציגם לכם)

הפונ': void perror(const char *);
(מצריכה: #include <stdio.h>) מקבלת מחרוזת ומדפיסה:
the given string : a description of errno

לדוגמה:

```
if (open(…) != 0) {
    perror("cannot open file") ;
    exit(EXIT_FAILURE) ;
}
```


במקום להשתמש ב- perror() ניתן להשתמש ב- strerror(errno) אשר מחזירה את תיאורו המילולי של הפרמטר שהועבר לה (בד"כ errno), ולא מדפיסה אותו.

דוגמה:

```
if (open(…) != 0) {
  fprintf(stderr, "%s cannot open %s:\
                  %s\n",
                  argv[0], argv[1],
                  strerror(errno)) ;
```

(שימושי בעיקר עת כמה תכניות מורצות ב- pipe ורוצים לדעת מי מביניהן נכשלה).

נהוג לחלק את השגיאות לשתי קבוצות:

- פטאליות: כאלה שלא ניתן להתאושש מהן.
- לא פטאליות: בד"כ מעידות על כך שלא ניתן (בינתיים) להקצות משאב (לייצר תהליך, להקצות מנעול).

עת חלה שגיאה מהסוג הראשון אין מנוס מסיום התכנית. עת חלה שגיאה מהסוג השני ניתן להמתין זמן מה, ואז לנסות שוב.

עד כאן הסטייה, ומכאן חוזרים למוטב, ל: fork() ולאכילה מנומסת בסכין ובמזלג:

דוגמה בסיסית ראשונה:

```
#include <stdio.h>
#include <stdlib.h>          // for exit()
#include <sys/types.h>

int main() {
   pid_t status  ;

   puts("Hi! I am a lonely parent\n") ;
   status = fork() ;
          ערך החזרה: בהצלחה: אפס, בכישלון: 1-
   if (status < 0) {
     perror("cannot fork") ;
     exit(EXIT_FAILURE) ;
   }
   puts("Hey! We r two already!!\n") ;
   if (status > 0)
                          לאב חוזר ה- pid של הילד
         puts("hello from father\n") ;
   else
                                  לילד חוזר תמיד אפס
         puts("hello from son\n") ;
    return(EXIT_SUCCESS) ;
}
```


קיבלנו: שני תהליכים המריצים אותה תכנית. לכ"א מהם מרחב כתובות משלו, בפרט משתנים משלו, לכן שינוי ערך משתנה אצל האחד לא יראה אצל משנהו. שניהם מתחילים לרוץ מהפקודה שאחרי הקריאה ל: fork() .

סדר הפלטים: אקראי.
נרחיב על כך:


```
#include <stdio.h>                // file: fork_race3.c
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>               // for fork()
//-------------------------------------------------
void do_parent() ;
void do_child() ;
//-------------------------------------------------
int main() {
   pid_t status = fork() ;

   if (status < 0) {
     perror("error in fork") ;
     exit(EXIT_FAILURE) ;
   }
   if (status > 0) {
       do_parent() ;
       return EXIT_SUCCESS ;
   }
   else {
       do_ child() ;
       return EXIT_SUCCESS ;
   }
   return EXIT_SUCCESS ;
}
//-------------------------------------------------
void do_parent() {
   puts("HELLO") ;
   puts("FROM") ;
   puts("A PARENT") ;
}
//-------------------------------------------------
void do_child() {
   puts("hello") ;
   puts("from") ;
   puts("a child") ;
}
```

נשים לב לסגנון התכנותי: ההורה והילד הולכים לפונ' משלהם, ואז מסיימים.


```
//-------------------------------------------------
```

והפלטים בהרצות שונות:

```
<216|0>yoramb@inferno-05:~/os$ a.out
HELLO
FROM
A PARENT
hello
from
a child
<217|0>yoramb@inferno-05:~/os$ a.out
hello
from
a child
HELLO
FROM
A PARENT
<218|0>yoramb@inferno-05:~/os$ a.out
HELLO
hello
from
a child
FROM
A PARENT
```

ראשית האם

מעורבב

<u>מסקנה</u>: מצב מרוץ == מה יקרה תלוי באופן המקרי בו מ.ה. תזמנה והריצה את שני התהליכים, ולא נקבע חד-משמעית על-ידינו.

דוגמה שנייה: ייצור כמה ילדים:

```
// file: fork2.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
  pid_t status ;
  int i ;

  i = getchar() ;

  for (i = 0; i< 3; i++) {
    status = fork() ;
    if (status < 0) {
      perror("error in fork") ;
      exit(EXIT_FAILURE) ;
    }

    if (status == 0)
    {
      printf("I am son #%d my pid= %d\n", i, getpid()) ;
      while(1) ;
      return(EXIT_SUCCESS) ;
    }
  }

  puts("father process end") ;
  return EXIT_SUCCESS ;
}
```


הרצה: (קלטים שלי מצוינים באות דגושה)

```
<210|1>yoramb@inferno-05:~/os$ gcc -Wall fork2.c
<211|0>yoramb@inferno-05:~/os$ a.out
```

עצירת התכנית ע"י **^z** עת היא ממתינה לקלט:

```
Suspended
<212|1>yoramb@inferno-05:~/os$ ps
  PID TTY        TIME CMD
24478 pts/12   00:00:00 tcsh
26956 pts/12   00:00:00 vim
27128 pts/12   00:00:00 a.out
27129 pts/12   00:00:00 ps
<213|0>yoramb@inferno-05:~/os$ fg
```

רץ עותק בודד:

החזרת התכנית לרוץ:

```
a.out
x
```

הזנתי לתכנית קלט, היא ממשיכה, מייצרת 3 ילדים, והאב מסתיים:

```
I am son #0 my pid= 27132
I am son #1 my pid= 27133
I am son #2 my pid= 27134
father process end

<214|0>yoramb@inferno-05:~/os$ ps
  PID TTY        TIME CMD
24478 pts/12   00:00:00 tcsh
26956 pts/12   00:00:00 vim
27132 pts/12   00:00:08 a.out
27133 pts/12   00:00:08 a.out
27134 pts/12   00:00:08 a.out
27135 pts/12   00:00:00 ps
```

בחזית רץ האב. עת הוא מסיים חוזר הפרומפט. (הילדים שברקע יכולים לק' מהמקלדת ולכ' למסך)

רצים שלושה עותקים:

נהרוג את הילדים (הרצים בלולאה אינסופית):

```
<215|0>yoramb@inferno-05:~/os$ kill 27132 27133 27134
```

ואין יותר עותקים של התכנית:

```
<216|0>yoramb@inferno-05:~/os$ ps
  PID TTY        TIME CMD
24478 pts/12   00:00:00 tcsh
26956 pts/12   00:00:00 vim
27156 pts/12   00:00:00 ps
```


דוגמה שלישית, בה ההורה ממתינה לבנותיה:

עת תהליך מסיים (מבצע מפורשות או באופן משתמע ק.מ. exit()) משאביו אינם נעלמים\ממוחזרים מיידית ע"י מ.ה., אלא הם נשמרים עד שאביו ('הביולוגי' או 'המאמץ' = INIT) מכיר בסיומו (מבצע עליו ק.מ. wait()).

על-כן אם הורה הוליד ילדים, הוא ממשיך לרוץ, ילדיו הסתיימו והוא לא ביצע עליהם wait הוא דומה למי שמותיר אחריו אשפה!

אם ההורה מסיים עת ילדיו רצים אין כך בעיה, שכן INIT (ולא הסב או הורה קדום אחר) יאמצם ויכיר בסיומם.

זיהוי של תהליך זומבי יכול להתבצע ע"י הרצת ps ואיתור תהליכים לצדם מופיע <defunct> (=נכחד) כשם הפקודה.

על-כן עתה נראה כיצד האב ממתין לילדיו:


```
#include <stdio.h>
#include <stdlib.h>          // for exit(), rand()
#include <sys/types.h>
#include <sys/wait.h>        // for wait()
#include <unistd.h>          // for sleep()
#include <time.h>

void do_child() ;

int main() {
  pid_t status ;
  int i ;

  srand((unsigned) time(NULL)) ;

  for (i = 0; i< 3; i++) {
    status = fork() ;
    if (status < 0) {
      perror("Cannot fork()") ;
      exit(EXIT_FAILURE) ;
    }
    if (status == 0)
    {
       do_child() ;
       exit(EXIT_SUCCESS) ;
    }
  }

  for (i = 0; i< 3; i++) {
    int son_id = wait(&status) ;
    printf("Son %d ended, exit status = %d\n",
                son_id,  WEXITSTATUS(status)) ;
  }
  puts("father process end") ;
  return EXIT_SUCCESS ;
}
```

```
//-----------------------------------------------------------------------------
void do_child() {
  sleep( rand() % 10) ;     // seconds
}

/*
<254|0>yoramb@inferno-05:~/os$ !!
a.out
Son 5042 ended, exit status = 0
Son 5043 ended, exit status = 0
Son 5041 ended, exit status = 0
father process end
<255|0>yoramb@inferno-05:~/os$
*/
```


הערות:

שימו לב שכדי להימנע מכל מיני מבוכות, כל ילד הולך לפונ' do_child() ואחר מסיים (כדי שלא יתחילו להיווצר כל מיני שושלות לא ברורות). מומלץ תמיד לקודד כך מבחינת סגנון תכנותי: לשלוח את הילדות להריץ פונ' משלהן, ואז לסיים (בתוך הפונ' או מייד מעבר לה).

הפונ' wait() מחכה לכך שילד כלשהו יסתיים.
היא מחזירה:

- 1- אם לא נוצרו ילדים, או שכבר המתינו על כל הילדים.
- מזהה של ילד שהינו זומבי. לפרמטר status מוכנסים דגלים המורים כיצד הילד סיים, בפרט ערך ההחזרה שלו. (אם אין לנו עניין בכך ניתן לזמנה: wait(NULL)).
- במידה ויש בנים אך הם עדיין רצים הפונ' תמתין עד לסיומו של אחד מהם, ואז תחזור כמו בסעיף מעל.
- קיים לפונ' וריאנט בשם waitpid() המאפשר להמתין על ילד מסוים, לא להמתין ולחזור עם כשלון אם אין ילד *שכבר* סיים. נִראה וריאנט זה בהמשך.
- הפונ' getpid(), getppid() מחזירות את מזהה התהליך והורהו בהתאמה. אם האב המקורי סיים אזי מזהה האב הוא 1 (=INIT).

היבט של fork(), שכפול חוצצים:

```
// file: fork_buffer.c
// A small program that fork()s.
// Before the fork(), the father prints "I am alone"
// as the output is buffered, it is not sent to the screen.
// When the child is born,
// and gets a copy of the address space of the father,
// it also gets this string. Therefore,
// when both send output (and break a line)
// this string occurs twice in the output (see bellow)

#include <stdio.h>
#include <stdlib.h>     // for exit()
#include <sys/types.h>
#include <unistd.h>     // for fork()

int main() {
  pid_t status ;

  printf("I am alone ") ;          הפלט של האב בחוצץ
  status = fork() ;                fork() משכפל גם החוצץ

  if (status < 0) {
    perror("cannot fork") ;
    exit(EXIT_FAILURE) ;
  }
  if (status > 0)
        puts("hello from father\n") ;   כאן החוצץ נפלט
  else
        puts("hello from son\n") ;      גם כאן החוצץ נפלט

  return EXIT_SUCCESS ;
}
```

הפלט:


```
/* a run:
  ======
<212|0>yoramb@inferno-05:~/os$ !a
a.out
I am alone hello from son

I am alone hello from father

<213|0>yoramb@inferno-05:~/os$
*/
```

עוד כמה הערות על wait()

- עת הורה הוליד ילד, כאשר הילד מסיים נשלח להורה סיגנל בשם SIGCHLD. ההורה יכול לא לבצע wait() כמו שראינו קודם, ואז להיחסם עד שמי מילדיו יסיים, אלא להגדיר טפלית סיגנל (signal handler), כלומר פונ' שתורץ אוטומאטית (בלי שקוראים לה מפורשות) עת הוא (ההורה) מקבל סיגנל זה. ההורה ימשיך בפעולתו 'הרגילה'. בטפלית יבצע ההורה wait() תוך שהוא כבר מובטח שיש ילד שסיים, ועל-כן לא יהיה עליו להמתין\להיתקע.
- לחילופין: יכול ההורה להכריז: signal(SIGCHLD, SIG_IGN); שמשמעו: אני מנער חוצני מילדי, עת הם מסתיימים איני רוצה להיות מדוּוַח על כך, והם, בהתאמה, לא יהפכו להיות זומבי עד שאני אכיר בסיומם, הם ימוחזרו מיידית.
- ייצור ילדים (fork()) עשוי להיכשל שכן יש גבול לכמות התהליכים שהמערכת יכולה לייצר, ועל כן אם היא הוצפה בתהליכים היא לא תאפשר יצירת חדשים.

ק.מ. exec()

בדוגמות שראינו עד כה הבת הריצה קוד שנכלל במרחב הכתובות (בתכנית) של אימה (אמרנו כי מבחינת סגנון תכנותי ראוי שהבת תריץ פונ' ייעודית מִשֵּלה, אולם אותה פונ' יכולה הייתה להיקרא גם על-ידי האם. כמובן שמבחינת הביצוע, ובפרט ערכי המשתנים, אין קשר בין הרצת הפונ' על-ידי האם וע"י הבת—הן שני תהליכים נפרדים).

עתה נראה כיצד הבת יכולה 'לעבור מוטציה' ולהריץ תכנית שונה לגמרי מזו שמריצה אימה.

נניח כי כתבנו את התכנית הבאה (המקבלת שתי מחרוזות, ממירה אותן למספרים, ומשווה ביניהם):

```
int main(int argc, char **argv) {
  if (argc != 3) {
    fputs("usage: my_cmp <arg1> <arg2>\n", stderr) ;
    exit(EXIT_FAILURE) ;
  }

  if (atoi(argv[1]) == atoi(argv[2]))
    puts("equals as ints\n") ;
  else
   puts("different as ints\n") ;

  return EXIT_SUCCESS ;
}
```

והיא שוכנת בקובץ ניתן להרצה בשם my_cmp.

נניח שהגדרנו:

```
char *args[] = {"my_cmp",
                "017", "17", NULL} ;
```

הפקודה:

```
execvp("my_cmp", args) ;
```

גורמת לתהליך שביצע אותה (בלי קשר לַאופן בו הוא נוצר) 'לעבור מוטציה', במילים אחרות לשנות את מרחב הכתובות שלו, ולהריץ את התכנית השוכנת בקובץ my_cmp, תוך שלתכנית מועברים כארגומנטים המחרוזות השוכנות במערך args.

על כן נוכל לכתוב את הקוד הבא:

```
#include <stdio.h>
#include <stdlib.h>        // for exit()
#include <string.h>
#include <sys/types.h>
#include <unistd.h>      // for sleep(), execvp()
#include <sys/wait.h>    // for wait()

void do_child(int) ;
```



```
int main() {
  pid_t status  ;
  int i ;

  for (i= 0; i < 2; i++) {
    status = fork() ;
    if (status < 0) {
      fputs("error in fork", stderr) ;
      exit(EXIT_FAILURE) ;
    }
    else if (status == 0) {
      do_child(i) ;
      fputs("we are not supposed to b here", stderr) ;
    }
  }
  return EXIT_SUCCESS ;
}
//-----------------------------------------------------------
void do_child(int n)  {
  #define MAX_STR_LEN   20
  char s1[MAX_STR_LEN], s2[MAX_STR_LEN] ;

  if (n == 0) {
    char *args[] = {"my_cmp", "017", "17", NULL} ;
    if (execvp("my_cmp", args) != 0) {
      perror("execvp() failed") ;
      exit(EXIT_FAILURE) ;
    }
  }
  strcpy(s1, "13") ;
  strcpy(s2, "12") ;
```

וריאנט שני (מתוך שישה) של exec():

```
  if (execlp("my_cmp", "my_cmp", s1, s2, NULL) != 0) {
      perror("execlp() failed") ;
      exit(EXIT_FAILURE) ;
  }
}
```

הפלט:

```
<237|1>yoramb@inferno-05:~/os$ a.out
different as ints

equals as ints

<238|0>yoramb@inferno-05:~/os$
```

אעיר כי ביוניקס כדי לייצר תהליך חדש שיריץ תוכנית שונה משל הורהו יש לבצע ()fork ואחריו ()exec; קיימות מ.ה., כדוגמת חלונות, בה ניתן לבצע את שתי הפעולות בעזרת ק.מ. בודדת. עת זה המצב אנו אומרים שהתהליך השריץ (spawn) תהליך שני.

4.4 שת"פ בין תהליכים Cooperating Processes

התהליכים שהרצנו עד כה היו בלתי-תלויים (idependant) – הם לא השפיעו על, ולא הושפעו ע"י תהליכים אחרים שרצו במקביל להם; לא הייתה להם שום אינטראקציה עם עמיתיהם.

לחילופין, תהליך עשוי להיות משתף פעולה (cooperating).

שת"פ בין תהליכים נועד לשם:

א. האצת חישוב – משימה תחולק בין כמה תהליכים, כל-אחד ייחשב חלק ממנה (ויורץ, אולי, ע"י מעבד נפרד).
ב. מודולאריות של משימה - (תהליך א' יקרא קלט מהמשתמש, תהליך ב' יתקשר עם הרשת, תהליך ג' יבדוק שגיאות כתיב).

אחת הפרדיגמות השכיחות לשיתוף היא פרדיגמת היצרן צרכן: תהליך יצרן (producer) מייצר מידע שנצרך בידי תהליך צרכן (consumer). דוגמות:

א. מהדר מייצר פלט שנצרך ע"י האסמבלר שמייצר פלט שנצרך ע"י הלינקר.
ב. שרת אינטרנט מייצר פלט אותו הוא שולח לדפדפן שפנה אליו לקבל דף נתונים.

כדי לאפשר את שיתוף הפעולה יש צורך ב-'מאגר' לתוכו יכניס היצרן את תוצרתו, וממנו ישלוף הצרכן את הנתונים.

יש לדאוג לסנכרון הפניה של היצרן והצרכן למאגר, כך שהיצרן לא 'יציף' את המאגר, והצרכן לא ישלוף פריטים שהיצרן לא השלים את יצירתם (קל וחומר שצרכן ימתין עת המאגר ריק).

לכאורה, מתכנת היצרן והצרכן יכול לתכנת את 'המאגר' בכוחות עצמו ובאחריותו (למשל: רשימה מקושרת בה היצרן מוסיף לזנב, והצרכן שולף מהראש). בפרט יהיה עליו לדאוג לסנכרון בין השניים.

למעשה, מערכות הפעלה בכלל, ויוניקס בפרט, מעמידות לרשותנו כלים שיקלו על התקשורת בין התהליכים: Inter Process Communication (IPC).

4.5 סיגנלים (Signals)

בפרק הנוכחי נכיר שפע של כלים לתקשורת בין תהליכים, והראשון ביניהם: הסיגנל (עמו ערכנו כבר הכרות ראשונית בפרק #2).

אציין כי הסיגנל הוא כלי מאוד, מאוד קדום ובסיסי להעברת מידע בין תהליכים. כיום, עת קיימים כלים מתקדמים ממנו, ממעטים לעשות בו שימוש לתקשורת בין תהליכים. עיקר תפקידו לאפשר למ.ה. לאותת לתהליך על שגיאה שהתהליך ביצע (חילק באפס, חרג מהזיכרון) או על אירוע כלשהו (ילדו סיים, טיימר שהוא הציב סיים).

הזכרנו שעת תהליך מזמן ק.מ., ק.מ. מחזירה ערך שונה מאפס אם היא נכשלת, והתהליך מצופה לבדוק זאת. במצב זה לא יישלח לתהליך סיגנל; סיגנל נשלח עת השגיאה חלה במקום 'מקרי' ובלתי צפוי, במלים אחרות עת השגיאה עלולה לחול בכל מקום שהוא.

בפרק #2 דברנו על הסיגנל כעל אנלוגי לפסיקה: דרך לאותת שחל אירוע המצריך טיפול. הפסיקה מאותתת זאת למ.ה., הסיגנל מאותת זאת לתהליך.

בפרק #2 ציינו שלכל סיגנל יש מספר ושם סימבולי (man 7 signal מציג את רשימת הסיגנלים, מספרם, שמם, והפעולה המחדלית המתבצעת עת הם נשלחים.)

עת סיגנל כלשהו s נשלח לתהליך, באופן מחדלי (אלא אם התהליך הגדיר אחרת), הסיגנל יגרום לפעולה קבועה מראש. בד"כ הפעולה היא העפת התהליך (וכן כן\לא שפיכת core). קיימים סיגנלים שהפעולה המחדלית עבורם היא התעלמות (SIG_CHLD).

תהליך עשוי לקבוע (מראש) שעת סיגנל כלשהו s נשלח אליו הוא מעוניין לנהוג באופן שונה מאשר המחדל. התהליך יכול לקבוע כי עת הוא מקבל את s הוא:

א. יתעלם ממנו.
ב. יתפוס אותו, כלומר יריץ פונ' מיוחדת, הקרויה <u>טפלית סיגנל</u> (signal handler).
ג. ינהג עם הסיגנל באופן המחדלי.

נרחיב על כך בהמשך.

סיגנל עשוי להישלח לתכנית באופן <u>סינכרוני</u>: בשל פעולה שהתכנית בצעה (בד"כ שגיאה), או באופן <u>א-סינכרוני</u>: בשל אירוע חיצוני לתכנית (ולכן הוא מגיע לתכנית במועד לא צפוי, לא מסונכרן, עם המשימה שהיא עושה (אולי היא בכלל ממתינה לקלט, או רצה במצב גרעין)).
סיגנל שיוצר הינו ראשית <u>תלוי</u> (pending), ושנית <u>נשלח</u> (delivered) לתהליך.

תהליך עשוי <u>לחסום</u> (block) קבלה של סיגנלים מסוימים, ואז הסיגנל נשאר תלוי עד הסרת החסימה.

את הסיגנלים: SIGKILL, SIGSTOP לא ניתן לחסום, לתפוס, או להתעלם מהם.

עת נשלח לתהליך סיגנל המעיד על שגיאה ראוי שהתהליך יסגור קבצים, ימחק קבצים זמניים, יחזיר את מצב המערכת למצבה המחדלי (למשל אם הוא כיבה את echo אזי ידליקו שוב). לבסוף יישלח התהליך לעצמו את אותו סיגנל, כדי לסיים כאילו לא הורצה טפלית.

<u>4.5.1 רשימת סיגנלים חלקית</u>

<u>סיגנלים המעידים על שגיאה</u>

- SIGFPE (בניגוד לשמו)נשלח בשל חלוקה באפס בשלמים.
- SIGSEGV חריגה מחוץ למרחב הכתובות של התהליך (פניה עם מצביע שלא אותחל, או הרחק מחוץ לגבולות מערך).
- SIGBUS פניה לזיכרון ביישור שגוי (למשל, פניה למשתנה מטיפוס long, שאמור לשכון בכתובת שהינה כפולה של ארבע, עם כתובת שאינה כנ"ל).
- SIGILL ניסיון לבצע פקודה לא חוקית: פקודה מיוחסת במצב משתמש, או פקודה לא מוכרת (קובץ ההרצה נדפק, הזיכרון נפגע בשל חריגה ממערך, מנסים לבצע נתונים, למשל משום שבמקום מצביע לפונ' הועבר מצביע לנתונים, התכנית כוללת פקודה שאינה מוכרת למעבד יחסית 'מיושן')

<u>סיגנלים המורים לתכנית להסתיים</u>

- SIGTERM מבקש בנימוס מהתכנית להסתיים. התכנית עשויה לדחות את הבקשה. kill מה-Shell שולח מחדלית סיגנל זה.
- SIGINT נשלח ע"י c^ (INT = interrupt).
- SIGQUIT נשלח ע"י \^ (מחדלית גם מביא לשפיכת core).
- SIGKILL על התכנית להסתיים מייד.

<u>סיגנלים המורים על סיום פרק זמן</u>

- SIGALRM טיימר סיים לקצוב זמן (אמיתי). נדון בו בהרחבה בהמשך.
- SIGVTALRM טיימר סיים לקצוב ריצה במעבד.

<u>סיגנלים הקשורים לבקרת תהליכים</u>

- SIGSTOP עוצר\משהה את התהליך שרץ בחזית (foreground). (דוגמה מייד בהמשך). הקשת z^ במקלדת שולחת את 'אחיו החלוש' של סיגנל זה: SIGTSTP (שניתן לתפיסה).
- SIGCONT גורם לתהליך שהושהה להמשיך לרוץ. פקודת ה- shell : fg שולחת אותו לתהליך שנעצר ע"י z^. (דוגמה מייד בהמשך).
- SIGCHLD נשלח להורה עת ילדו מסיים או מושהה (מחדלית מתעלמים ממנו).

סיגנלים הקשורים לשגיאות הקשורות למערכת

- SIGPIPE נשלח לתהליך שפתח צינור (pipe) או תור לכתיבה ואין קורא בצד השני, או שהקורא נעלם.

תהליך שניסה לכתוב על תושבת שלא חוברה.

- SIGXCPU (= exceeded CPU) נשלח לתהליך שעבר את ה- soft limit של זמן ריצה במעבד. אם התהליך לא יסיים בזריזות, ויעבור את ה- hard limit יישלח לו SIGKILL שישמידו סופית.
- SIGXFSZ נשלח לתהליך שניסה פעם ראשונה לכתוב על קובץ ולהגדילו מעבר למותר במערכת (בהמשך, הכתיבה 'סתם' תכשל, ל- errno יוכנס הערך EFBIG, ולא יישלח עוד הסיגנל).

סיגנלים שונים

- SIGWINCH = גודל החלון השתנה.
- SIGUSR1, SIGUSR2 = סיגנלים שנועדו לשימוש המתכנת. מחדלית גורמים להעפת התהליך. נראה אותם ביתר פרוט בהמשך.

4.5.1 שליחת סיגנל

אנו יכולים לשלוח סיגנל לתהליך בכמה אופנים:

א. מהמקלדת נוכל לשלוח את הסיגנלים הבאים:
   1. הסיגנל SIGINT נשלח ע"י הקלדת c^.
   2. הסיגנל SIGQUIT נשלח ע"י הקשת \^ .
   3. הסיגנל SIGSTOP נשלח ע"י הקשת z^.
   4. הסיגנל SIGCONT נשלח ע"י הקלדת fg.

ב. פקודת ה- shell: kill שולחת סיגנל רצוי לתהליך רצוי. לדוגמה: kill –INT 3879 שולחת את הסיגנל SIGINT לתהליך 3879 (kill 3879 שולח את הסיגנל SIGTERM לתהליך).

ג. ק.מ. ()kill מאפשרת לתהליך א' לשלוח סיגנל לתהליך ב'

נראה, ראשית, דוגמה לסעיף ג', בה תהליך שולח סיגנל לעצמו:


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

int main() {
  pid_t my_pid = getpid() ;
  int i, j ;

  for (i = 0; i< 10; i++) {
    for (j = 0; j < 10; j++)
        printf("(%d*%d = %d) ", i, j, i*j) ;
    putchar('\n') ;
    kill(my_pid, SIGSTOP) ;
  }

  return EXIT_SUCCESS ;
}
```

הערה קטנה: במקום לשלוח: kill(my_pid, SIGSTOP) ;
ניתן לשלוח: raise(SIGSTOP); השולח סינגל בהכרח לתהליך עצמו.

4.5.2 תפיסת סיגנל

הזכרנו כי בדרך כלל תהליך יכול לתפוס סיגנל, ע"י קביעת טפלית סיגנל שתקרא, באופן א-סינכרוני, עת הסיגנל נשלח לתהליך. (גם אמרנו כי קיימים מספר סיגנלים, כדוגמת KILL, STOP אותם לא ניתן לתפוס, ומהם לא ניתן להתעלם).

ישנם סיגנלים שיש טעם בתפיסתם וקביעת התנהגות ספציפית לתהליך עת הם נשלחים (כדוגמת CHLD); אחרים (כגון SEGV, BUS) ניתן לתפוס אולם לא ראוי לנסות להתאושש מהם, אלא לכל היותר לסיים את התכנית באופן נקי: לסגור משאבים (כגון קבצים פתוחים), לשלוח הודעת שגיאה, ולצאת.

אמרנו כי ק.מ.:
signal(<signal name>, <handler func>) ;
גורמת לכך שעת יישלח פעם יחידה(?) לתכנית הסיגנל <signal name> תופעל\תזומן הפונ' <handler func> אשר בהכרח מקבלת פרמטר יחיד מטיפוס int המציין את הסיגנל שגרם לזימון הטפלית (שימושי עת היא מופקדת על יותר מסיגנל אחד). הפונ' מחזירה void.

נראה דוגמה:

```
// file: catch_signals1.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>

//---------------------------------
void catch_int(int sig_num) ;

//---------------------------------
int main() {
  //signal(SIGINT, SIG_IGN) ;
  signal(SIGINT, catch_int) ;

   puts("in infinite loop") ;
   while (1) ;
   // while (1)
   //   pause() ;

  return EXIT_SUCCESS ;
}

//---------------------------------
void catch_int(int sig_num) {
  //signal(SIGINT, catch_int) ;
  puts("Na Na Banana") ;
}
```

אפשרות אחרת:

אפשרות אחת: התכנית בלולאה אינסופית. כל סיגנל גורם לה לשלוח את הפלט, ולחזור ללולאה.

אפשרות אחרת: pause() שולח את התהליך לשון עד קבלת סיגנל. על כך נחזור בלולאה.

למה פקודה זאת? אמפירית—לא חיונית(!?)

הערה קטנה: הפונ' pause(); גורמת לתהליך ללכת לשון עד קבלת סינגל.

שאלה קטנה: מדוע עדיף להשתמש בה ע"פ להריץ לולאה אינסופית?


```
/* a run:
   =====
<236|0>yoramb@inferno-05:~/os$ !a
a.out
in infinite loop
Na Na Banana
Na Na Banana
Na Na Banana
Quit
<237|131>yoramb@inferno-05:~/os$
*/
```

העפנו את התכנית ע"י \^ (שגם גורמת לשליחת ההודעה)

לחילופין, יכולנו להשהותה ע"י z^, ואז להעיפה ע"י: kill 3879 (שולח SIGTERM).

עד כאן ראינו כיצד נתפוס סיגנל.
לחילופין, אנו עשויים לרצות (ולו זמנית) להתעלם מסיגנל כלשהו (עד השלמת משימה כלשהי). הדרך לעשות זאת:

signal(SIGINT, SIG_IGN) ;

לבסוף, כדי להשיב את ההתנהלות בעת קבלת הסיגנל לזאת המחדלית נכתוב:

signal(SIGINT, SIG_DFL) ;

נראה עתה היבטים נוספים של שימוש בסיגנלים.

דוגמונת קצת צדדית

עת דנו ב- fork() ראינו דוגמה בה האם שולחת נתון לפלט הסטנדרטי, אך לא שוברת שורה, ועל כן הפלט נותר בחוצץ. עתה האם יוצרת בת המקבלת גם *עותק* של החוצץ של אימה, ועל כן עת כל אחת מהשתיים שולחת פלט, ושוברת שורה (ולכן החוצץ מרוקן למסך), הפלט *היחיד* ששלחה האם מוצג *פעמיים* על-גבי המסך.

עתה נציג את אותה תופעה עת הפלט נשלח לקובץ, וכן נפגוש (שוב) את ק.מ. pause(), כמו גם את טפלית הסיגנל הריקה.

לכאורה, עת האם פותחת קובץ ואז מולידה בת, ייתכן שלבת משוכפל גם מכוון הקובץ (שערכו בדוגמה שלנו אפס, שכן דבר טרם נכתב על הקובץ), ולכן עת ראשית כותבת האחת (החל ממקום אפס בקובץ), ושנית השנייה (החל ממקום אפס בקובץ) הכתיבה של השנייה 'תדרוס' את זו של הראשונה.
למעשה לא כך הוא: מכוון הקובץ הוא *יחיד*, ואינו מוחזק ב- PCB של תהליך כלשהו (אלא במקום אחר), ולכן גם אינו משוכפל ב- fork(). לכן עת הראשונה מהשתיים כותבת המכוון מקודם (כרגיל), והפלט של השנייה בא *מעבר* לפלט ששלחה הראשונה. (הקובץ בראש השקף הבא.)

היבט של fork() האם והבת כותבות על אותו קובץ *זו בהמשך לזו* (יש מצביע יחיד לקובץ):

```
// file:  fork_fprintf.c
// A small program that fork()s.
// Before the fork(), the father prints "I am alone" to a file
// as the output is buffered, it is not sent to the file.
// When the child is born,
// and gets a copy of the address space of the father,
// it also gets this string. Therefore, when both send output
// (and break a line)
// this string occurs twice in the file (see bellow)
// IN ADDITION: the father wakes up after his son finishes,
// and its output is APPENDED AFTER what his son
//  wrote; i.e.,
// BOTH SHARE A SINGLE FILE POINTER

/* The file out.txt is, therefore:
I am alone hello from son
I am alone hello from father
*/

#include <stdio.h>
#include <stdlib.h>     // for exit()
#include <sys/types.h>
#include <unistd.h>    // for fork()
#include <signal.h>


void catch_child(int signum) {}        // empty signal handler.
                                       // needed only
                                       // for the pause()
```

זה אינו פרוטוטיפ, אלא פונ' שלא עושה דבר. אך שניתן לקבוע כטפלית.
מייד נראה למה זה טוב

```
int main() {
  pid_t status ;
  FILE *fd = fopen("out.txt", "w");

  signal(SIGCHLD, catch_child) ;
                    // pause() will wake father up, only due
                    // to this declaration
  fprintf(fd, "I am alone ") ;

  status = fork() ;

  if (status < 0) {
    fputs("error in fork", stderr) ;
    exit(EXIT_FAILURE) ;
  }
  if (status > 0) {
    pause() ;
    fputs("hello from father\n", fd) ;
  }
  else
        fputs("hello from son\n", fd) ;

  return EXIT_SUCCESS ;
}
```

האב פותח קובץ:

האב קובע טפלית:

האב שולח פלט, אך לא שובר שורה, ולכן הפלט בחוצץ.

האב הולך לשון עד קבלת סיגנל אותו הוא תופס: SIG_CHLD.

לכן הפלט של הילד נשלח לפני זה של האב (רק עת הילד מסיים האב מוער).

(<u>הערה קטנה:</u> נשים לב שלמרות שהאב שלנו ממשיך לרוץ אחרי שילדו הסתיים הוא לא מכיר בסיומו (מבצע wait()), וזה לא יפה. מכיוון שהוא רץ רק מרחק קטן, זה גם לא נורא).

4.5.3 תפיסת הסיגנל SIGCHLD

כזכור, עת תהליך מסתיים הוא הופך להיות זומבי עד שאביו מכיר בכך. רק אז כל משאביו ממוחזרים סופית, והוא נעלם מהמערכת.

ראינו כיצד ההורה יכול לבצע את ק.מ. wait() אשר גורמת לו להיות מושהה (לשון) עד שבן כלשהו שלו מסתיים.

המגבלה של פתרון זה היא שלעתים איננו מעוניינים להשבית את האב בהמתנה לילדיו (אחרי הכל יש לו עוד כמה דברים לעשות בחיים, חוץ מאשר להמתין להם).

הזכרנו שעת הילד מסתיים נשלח לאמו הסיגנל SIGCHLD (שבאופן מחדלי מתעלמים ממנו). ניעזר בסיגנל זה כדי מצד אחד לא להשבית את האם, ומצד שני לא להותיר את בִיתה זומבית.

האֵם, במְקום להמתין לבת, תקבע טפלית סיגנל לסיגנל SIGCHLD. רק בטפלית, אליה נגיע אחרי שבת כלשהי תסיים, ולאם יישלח הסיגנל, תבצע האם wait() אולם אז הוא כבר לא יתקע אותה. (למעשה, כפי שנסביר, האם תבצע waitpid())

הדוגמה:


```
// catch_signals2.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>

//---------------------------------
void do_son(int num) ;
void do_dad() ;

void catch_chld(int sig_num) ;

//---------------------------------
int main() {
  pid_t pid ;
  int I ;

  srand(17) ;

  signal(SIGCHLD, catch_chld) ;

  for (I = 0; I < 3; i++) {
    pid = fork() ;
    if (pid == -1) {
        perror("fork failed") ;
        exit(EXIT_FAILURE) ;
    }
    if (pid == 0)
       do_son(i) ;
  }
  do_dad() ;

  return EXIT_SUCCESS ;
}
```

האב קובע כי עת ילדו מסתיים, תזומן הטפלית

כל ילד מקבל את מספרו = הכפולות שהוא ידפיס


```
//---------------------------------
void do_son(int num) {
  int I ;

  sleep(rand()%10) ;          // seconds
  for (I = 0; i<10; i++)
    printf(" %d", i*num) ;
  putchar('\n') ;

  exit(EXIT_SUCCESS) ;     // VERY IMPORTANT!!!
}


//---------------------------------
void do_dad() {
  int I ;

  for (I = 0; I < 17; i++) {
    printf("I am a busy dad\n") ;
    sleep(1) ;
  }
}
```

כדי שהילד יחיה קצת

לאבא יש הרבה עובדה חשובה לעשות

```
//---------------------------------
// waitpid() : -1 wait for any child,
//            NULL == we are not interested in the son's
//                  return status
//            WNOHANG == do not wait for a son,
//                    return immediately
//            returns the pid of a process that had finished
// As A SINGLE signal may wait for us even if a few
// children
// had finished (while we were sleeping or in kernel mode)
// we need to loop.

void catch_chld(int sig_num) {
   signal(SIGCHLD, catch_chld) ;
   while (waitpid(-1, NULL,  WNOHANG) > 0)
      puts("Another son has gone\n") ;
}
```



```
/* An example of an output:
   =======================
I am a busy dad
I am a busy dad
I am a busy dad
I am a busy dad
I am a busy dad
 0 0 0 0 0 0 0 0 0 0
Another son has gone
 0 1 2 3 4 5 6 7 8 9

Another son has gone

I am a busy dad
 0 2 4 6 8 10 12 14 16 18
Another son has gone

I am a busy dad
I am a busy dad
I am a busy dad
I am a busy dad
I am a busy dad
I am a busy dad
I am a busy dad
I am a busy dad
I am a busy dad
I am a busy dad
I am a busy dad

*/
```

האבא עושה את עבודתו

בן (הראשון) עושה את משימתו

עת הבן מסיים, האב נקרא לטפלית

ק.מ sigprocmask() מקבלת שלושה פרמטרים:

א. int how : קוד הפעולה שיש לבצע: האם יש להוסיף סיגנלים למסכה הקיימת (SIG_BLOCK), להסיר ממנה סיגנלים (SIG_UNBLOCK) או לקבוע מסכה חדשה שתחליף את הקיימת (SIG_SETMASK).

ב. const sigset_t *set : מצביע למסכת הסיגנלים שיש להוסיף\להסיר\לקבוע.

ג. const sigset_t *old_set : מצביע למסכת הסיגנלים שהייתה עד ביצוע הפעולה, וזאת למקרה שנרצה לשחזרה בהמשך (ניתן להעביר כארגומנט NULL).

כפי שנראה, באופן אופייני, לפונ' sigprocmask() נקרא מתוך טפלית סיגנל. במצב זה, עת הטפלית מסתיימת, מ.ה. משחזרת את מסכת הסיגנלים שהייתה לתהליך עם הכניסה לטפלית, ועל-כן איננו צריכים לשחזר את המסכמה המקורית בעצמנו (לפני היציאה מהטפלית).

4.5.4 מיסוך סיגנלים

לעתים ברצוננו למנוע מסיגנלים להטריד את מנוחתה של תכניתנו, למשל עת היא מבצעת משימה קריטית, או מטפלת בסיגנל, אולי אפילו באותו סיגנל שכבר הגיעה.

נציג שתי דרכים לחסימת סיגנלים (עד הסרת החסימה) כך שהם ימתינו בשקט. במלים אחרות, מ.ה. תאגור סיגנלים שנחסמו אצלה, ותֵשַלח אותם בתכנית רק אחרי הסרת החסימה.

הדרך הראשונה (והפחות מוצלחת, כפי שנסביר) היא בעזרת ק.מ.: sigprocmask().

לפני שנציג את הפונ' נומר כמה מלים על 'מסכה': מסכה היא מערך של סיביות. כל סיבית מייצגת דגל\אירוע. הסיבית יכולה להכיל את הערך אחד, משמע יש המסכה מתייחסת לאותו דגל\אירוע, או להכיל את הערך אפס, משמע המסכה אינה מתייחסת לדגל\אירוע. (השם 'מסכה' שכן במסכה יש מקומות בהן יש חורים, ובאחרים אין חורים)

מספר פונ' יעזרו לנו 'לייצר' את המסכה (לנקותה, להוסיף\לגרוע ממנה סיגנלים):

א. ריקון המסכה: sigemptyset(&set) ;
ב. מילוי המסכה כולה: sigfillset(&set);
ג. הוספת סיגנל למסכה: sigaddset(&set, SIGINT);
ד. הסרת סיגנל: sigdelset(&set, SIGINT);
ה. בדיקה האם סיגנל שייך למסכה: if (sigismember(&set, SIGINT)) puts("SIGINT is in mask");

נראה דוגמה:
התכנית שלנו מונה כמה פעמים נשלח לה c^.
אחרי שלושה היא שואלת האם לסיים וכן\לא עוצרת בהתאם לתשובה.
הנקודה המשמעותית לענייננו: עת אנו בטפלית ברצוננו לחסום סיגנלים אחרים שעלולים לזרוק אותנו לטפליות אחרות (תוך קטיעת הטפלית הנוכחית), או יקטעו הביצוע (יביאו להעפת התכנית). אנו רוצים לבצע הטפלית באופן בלתי מופרע.


```
// file:  mask_signals.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
//--------------------------------
void catch_int(int sig_num) ;

//--------------------------------
int main() {
  int i, j ;

  signal(SIGINT, catch_int) ;          קביעת הטפלית

  for (i = 0; i < 10; i++) {
    for(j = 0; j<i; j++)
      printf(" %d", j) ;

    putchar('\n') ;
    getchar() ;                 // here program waits
  }

  return EXIT_SUCCESS ;
}
//--------------------------------
```



```
void catch_int(int sig_num) {
  #define CTRL_C_THRESHOL 3
  static int ctrl_counter = 0 ;
  sigset_t curr_mask ;          המשתנה הדרוש:

  signal(SIGINT, catch_int) ;

  sigfillset(&curr_mask) ;          // mask all signals
                                    קבע את המסכה:
  sigprocmask(SIG_SETMASK, &curr_mask, NULL) ;

  מכאן ועד סוף הפונ' שום סיגנל (כמעט) לא יקטע את הביצוע
  ctrl_counter++ ;
  if (ctrl_counter >= CTRL_C_THRESHOL) {
      char c ;

      puts("Realy exit? [y/n]") ;
      c = getchar() ;
      if (c == 'y') {
         puts("OK, master");
         exit(EXIT_SUCCESS) ;
      }
      puts("Okay, I continue") ;
      ctrl_counter = 0 ;
  }
  // no need to restore mask. Done automatically:
  // sigprocmask(SIG_SETMASK, &prev_mask, NULL) ;
}
```

אם כל כך טוב, אז מה כל כך רע?

נניח שבתכנית שלנו קבענו גם טפלית ל-SIGQUIT (הנשלח ע"י הקלדת \^) המבצעת משימה אחרת בפונ' f().
עוד נניח שהמשתמש הקליד c^, ועל-כן נקראה הטפלית המתאימה (catch_int()).
לבסוף, נניח גם שהמשתמש הקליד \^
מה יקרה?
התשובה: תלוי.
אם המשתמש מאוד זריז, או לחילופין המחשב מאוד איטי, והסיגנל SIGQUIT התקבל טרם שבוצעה ה-sigprocmask() אזי ראשית תתבצע f() ורק אח"כ יחזרו ל- catch_int().

אם לעומת זאת ה- SIGQUIT יגיע לתכנית אחרי ביצוע ה- sigprocmask() אזי הוא יחסם כל עוד אנו ב- catch_int(), ורק עת נצא ממנה הוא ישולח בתכנית, ולכן f() תתבצע אחרי catch_int().

אם יש משמעות לסדר בו מבוצעות הטפליות אזי איננו אדישים לשאלה איזה תסריט יקרה, אולם התשובה תלויה בתזמון, *המקרי* מבחינתנו, בו מ.ה. הריצה את התכנית, לעומת שלחה את הסיגנל (המשתמש בשני התסריטים עשה אותן פעולות באותה מהירות).

למצב כזה, בו הפלט תלוי בתזמון המקרי בה מ.ה. הריצה את התכנית שלנו אנו קוראים מצב מרוץ (race condition), והוא נחשב למצב מאוד לא רצוי. (ראינו אותו כבר בהקשר בו תהליך הורה ייצר תהליך ילד ושניהם שלחו פלט, ונראה אותו שוב בעוד מצבים).

נראה דוגמה נוספת:

```
// file:  mask_signals2.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
//---------------------------------
void catch_int(int sig_num) ;
void catch_quit(int sig_num) ;

int x = 0 ;
//---------------------------------
int main() {
  int i, j ;

  signal(SIGINT, catch_int) ;
  signal(SIGQUIT, catch_quit) ;

  for (i = 0; i < 10; i++) {
    for(j = 0; j<i; j++)
      printf(" %d", j) ;

    putchar('\n') ;
    getchar() ;
  }
```

קבענו שתי טפליות, אחת לכל סיגנל


```
  printf("x = %d\n", x) ;

  return EXIT_SUCCESS ;
}
//---------------------------------
void catch_int(int sig_num) {
  sigset_t curr_mask ;

  signal(SIGINT, catch_int) ;

  sigfillset(&curr_mask) ;          // mask all signals
  sigprocmask(SIG_SETMASK, &curr_mask, NULL) ;

  x = 1 ;
}
//---------------------------------
void catch_quit(int sig_num) {
  sigset_t curr_mask ;

  signal(SIGQUIT, catch_quit) ;

  sigfillset(&curr_mask) ;          // mask all signals
  sigprocmask(SIG_SETMASK, &curr_mask, NULL) ;

  x = 2 ;
}
```

הטפליות זהות, פרט לערך שהן מכניסות ל- x

נניח שראשית הוזן c^ ואחריו \^.
השאלה: איזה ערך של x יודפס?
התשובה: תלוי. אם catch_int() הספיקה לחסום את הסיגנלים טרם שנשלח SIG_QUIT אזי יודפס 2, אחרת (היא תקטע, יפנו ל-catch_quit() , ואז יחזרו אליה ולכן) יודפס 1.

כיצד אם כן נמנע ממצב מרוץ?
נשתמש ב- sigaction() אשר מבצעת באופן אטומי את זימון הטפלית וקביעת מסכת הסיגנלים שתהיה תקפה בטפלית.
התוצאה: מכיוון שהמשתמש מקיש ראשית c^ אזי עת מזומנת catch_int() גם נחסמים הסיגנלים, ועל-כן לא משנה כמה מהר המשתמש יקיש את ה- \^ לבטח f() תבוצע *אחרי* catch_int(), כלומר סדר ביצוע הפונ' נשלט לגמרי על-ידינו.

על-כן נציג עתה את sigaction():
ק.מ. sigaction() מקבלת שלושה פרמטרים:
א. הסיגנל אותו יש לתפוס.
ב. מצביע למבנה: struct sigaction act ; המורה מה הטפלית שתופעל, ומה תהיה מסכת הסיגנלים שתקבע עם הכניסה אליה.
ג. מצביע למבנה מאותו טיפוס לתוכו נוכל להכניס את מצב הדברים שהיה טרם ביצוע ק.מ., לטובת שיחזור. אם איננו מעוניינים לשחזר ניתן להעביר NULL בארגומנט המתאים.

ראשית עלינו להזין את הערכים המתאימים למבנה act, ואז נוכל לזמן:
sigaction(SIGINT, &act, NULL) ;
אשר מבצעת את הפעולות של signal() ושל sigprocmask() באופן אטומי.

כמו כן הטפלית שאנו מייסדים (establish), תישאר בתוקף עד שנשנה אותה (ולא רק פעם יחידה כמו שקורה עם signal()). לכן בטפלית עצמה לא נצטרך לטעון את הפונ' שוב.

(ה- include הדרוש: signal.h)

עתה אציג את הערכים אותם נכניס למשתנה: struct sigaction act ; (טרם שאנו מזמנים את sigaction()):

ראשית, נקבע מהי הטפלית:

act.sa_handler = catch_int ;

שנית, נטפל במסכת הסיגנלים, וכמו קודם נחסום את כל הניתן לחסימה:

sigfillset(&act.sa_mask) ;

שלישית, נאפס דגלים שלא מעניינים אותנו (המעוניינים מוזמנים לקרוא אודותם ב- man):

act.sa_flags = 0 ;

ועתה, נוכל לזמן:

sigaction(SIGINT, &act, NULL) ;

והתכנית:


```
// file: mask_signals_sigaction.c
// mask signals like in mask_signals.c
// but in a better way: using sigaction
// which is preferred over signal()!

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
//---------------------------------
void catch_int(int sig_num) ;

//---------------------------------
int main() {
  int i, j ;
  struct sigaction act ;

  act.sa_handler = catch_int ;        // the handler function
  sigfillset(&act.sa_mask) ;          // mask all signals
  act.sa_flags = 0 ;                  //  flg we r not using
  sigaction(SIGINT, &act, NULL) ;
                                      // this shall remain until
                                      // we change it
                                      // (as oposed to signal()
                                      // NULL = we are not interested
                                      // in the old val of sigaction

  for (i = 0; i < 10; i++) {
    for(j = 0; j<i; j++)
      printf(" %d", j) ;

    putchar('\n') ;
    getchar() ;
  }
 return EXIT_SUCCESS ;
}
```

catch_int() נותרת בדיוק כמו קודם, ועל-כן לא כללתי אותה בדוגמה.

עד כה זימנו את הפונ' signal() ולא התעניינו בערך החזרה שלה. למעשה הפונ' מחזירה את הפעולה שננקטה עד כה ביחס לאותו סיגנל. (ערך מטיפוס: sighandler_t שהינו מצביע לפונ' עם הפרוטוטיפ המתאים [מקבלת שלם מחזירה כלום]).

<u>המטרה</u>: להיות מסוגלים לשחזר פעולה זאת.

<u>דוגמה פשוטה:</u>

```
if (signal(SIGINT, f) == SIG_IGN)
  sifnal(SIGINT, SIG_IGN) ;
```

במצבים אחרים נרצה לשחזר את הטפלית הקודמת רק אחרי השלמת משימה כלשהי.

<u>4.5.5 מימוש עיתאי\שעון-עצר (timer) בעזרת סיגנלים</u>

לעתים ברצוננו לקבל בתכנית איתות על כך שחלף פרק זמן כלשהו (ואם המשתמש לא הזין קלט אזי נעצור את ביצוע התכנית, או נפעיל שומר מסך; אם השרת אליו פנינו לא החזיר תשובה אזי...)

יוניקס לא מעמידה לרשותנו כלים מוגמרים לביצוע המשימה, אך כן מעמידה כלי די בסיסי, שמסָפיק במצבים פשוטים:

ק.מ. alarm(int sec); מבקשת מהמערכת לשלוח לתכנית את הסיגנל ALRM עוד sec שניות; במילים אחרות היא מדליקה טיימר.
(מכיוון שיוניקס אינה מערכת זמן אמת אזי ייתכן שהסיגנל יתאחר).
זימון הקריאה באופן: alarm(0); אומר שאיננו מעוניינים יותר בקבלת הסיגנל, במילים אחרות אנו מכבים את הטיימר.

שימוש בק.מ. יחד עם טפלית מתאימה לסיגנל יעזור לנו להשיג את הדרוש:

א. נקבע טפלית סיגנל לסיגנל הנ"ל, אשר תבצע את מה שברצוננו לעשות אם חלף פרק הזמן.
ב. נדליק טיימר לפרק זמן sec שניות.
ג. נפנה לביצוע המשימה שלה קצבנו sec שניות (לדוגמה: קריאת קלט מהמשתמש).
ד. נכבה את הטיימר.

אם לא נספיק להשלים את המשימה לה קצבנו זמן (למשל לא נקרא את הקלט) אזי בפרט לא נגיע לכיבוי הטיימר (פעולה שאנו מבצעים רק *אחרי* השלמת המשימה), ולכן אחרי sec שניות יישלח לנו סיגנל שיקפיץ אותנו לטפלית. בטפלית נוכל לעשות את מה שברצוננו לעשות עת אין עמידה בלוח הזמנים (למשל, לסיים את התכנית). אם הטפלית מכניסה אותנו למצב מיוחד (מדליקה שומר מסך), אזי שליחת סיגנל אחר תוציא אותנו מהמצב המיוחד, ותחזיר אותנו למצב 'רגיל' (בעזרת טפלית מתאימה).

מנגד, אם הספקנו להשלים את המשימה בַזמן (טרם שנשלח הסיגנל) אזי גם נפנה לפקודה signal(0); ובכך נכבה את הטיימר, ולכן לא יישלח לתכנית הסיגנל SIGALRM, ובפרט לא נגיע לטפלית, ולא נעצור את ביצוע התכנית.

נראה דוגמה פשוטה:

```
// file: timer_by_signal.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
//--------------------------------
void catch_alarm(int sig_num) ;
//--------------------------------
int main() {
  int num, j ;

  signal(SIGALRM, catch_alarm) ;

  for (; ; )
  {
    printf("Enter an int: ") ;
    alarm(10) ;
    scanf(" %d", &num) ;
    alarm(0) ;
    for(j = 0; j<num; j++)
      printf(" %d", j) ;

    putchar('\n') ;
  }

  return EXIT_SUCCESS ;
}
//--------------------------------
void catch_alarm(int sig_num) {

  //signal(SIGALARM, catch_alarm) ;      no need to

  puts("\nIt seems you gave up\nI am exiting...") ;
  exit(EXIT_SUCCESS) ;
}
```

קביעת הטפלית:

הדלקת הטיימר:

פניה לביצוע המשימה:

כיבוי הטיימר:

אם לא הייתה עמידה בזמנים, והטיימר לא כובה, מגיעים לטפלית:

דוגמה שנייה, מעט יותר מורכבת: אם המשתמש לא מזין קלט בזמן שקצבנו לו אנו מדליקים 'שומר מסך', אשר יִכבה עת המשתמש מזין ^c (הקלדת ^c עת שומר המסך כבוי אינה עושה דבר). סיום התכנית, למשל ע"י \^.
לכן בתכנית זו יש לנו שתי טפליות: האחת ל- SIGALRM והשנייה ל- SIGINT.

```
// file: timer_by_signal2.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
//--------------------------------
void catch_alarm(int sig_num) ;
void catch_int(int sig_num) ;
short screen_saver ;
```

למשתנה יכנסו אחד משלושה ערכים:
0 = שומר המסך כבוי, אנו בעבודה 'נורמאלית'
1 = שומר המסך דולק, שכן המשתמש 'נרדם'.
2 = שומר המסך דלק, המשתמש הקיש ^c ולכן עתה יש לחזור לאופנות עבודה רגילה; אולם אין להדפיס כפולות שכן לא נקרא מספר חדש.


```
//--------------------------------
int main() {
  int num, j ;
  struct sigaction act ;

  // must use sigaction() and not signal()
  // as the former terminates scanf()
  // while the latter does not, so the
  // 'screen saver' does not run with
  // signal() but does with sigaction()

  act.sa_handler = catch_alarm ;
  sigfillset(&act.sa_mask) ;
  act.sa_flags = 0 ;
  sigaction(SIGALRM, &act, NULL) ;

  signal(SIGINT, catch_int) ;
                          // ^C ends the 'screen saver'
  for (; ; )
  {
    screen_saver = 0 ;
    puts("Enter an int: ") ;
    alarm(10) ;
    scanf(" %d", &num) ;
    alarm(0) ;

    while (screen_saver == 1) {
      puts("screen saver is on") ;
      sleep(1) ;
    }
    if (screen_saver == 0)
    {
      for(j = 0; j<num; j++)
        printf(" %d", j) ;
      putchar('\n') ;
    }
  }
```

אנו באופנות עבודה רגילה

אם המשתמש נרדם, תקרא catch_alarm() שתכניס למשתנה את הערך 1.
^c יגרום לשינוי הערך ל-2

רק אם במשתנה יש אפס (ולא 2) יש להדפיס הכפולות (אם יש בו 2 ל- num אין ערך תקף)

```
  return EXIT_SUCCESS ;
}
//--------------------------------
void catch_alarm(int sig_num) {
  //signal(SIGALARM, catch_alarm) ;      no need to

  puts("\nIt seems you went away\n \
        I turn on screen saver...") ;
  screen_saver = 1 ;
}

//--------------------------------
void catch_int(int sig_num) {
  signal(SIGINT, catch_int) ;

  if (screen_saver == 0)
    return ;
  puts("\nGood Morning") ;
  screen_saver = 2 ;
}
```


<u>הנחיות והערות לשימוש בטיימר</u>

ביוניקס אין באפשרותנו להחזיק יותר מטיימר יחיד פעיל.

יש להקפיד להדליק את הטיימר *מיד* לפני הפנִיה לביצוע המשימה לה אנו מעוניינים לקצוב זמן, ולכבותו *מיד* אחרי השלמת המשימה (ולא, למשל, אחרי בדיקת הנתונים).

אם המשימה הושלמה במועד, אזי יש להקפיד לכבות את הטיימר בכל מהלך ביצוע (טעות אפשרית: הטיימר מכובה בסוף פונ', אך במצבים מסוימים יוצאים מהפונ' [עם return] לפני סיומה הרגיל).

<u>4.5.6 הסיגנלים SIGUSR1, SIGUSR2</u>

בתחילת דיוננו בנושא הסיגנלים ציינו שלסיגנלים יש שני תפקידים:

א. הם מאפשרים למ.ה. לשלוח איתות לתכנית שמשהו 'קרה', בד"כ שמשהו השתבש.

ב. הם עשויים לשמש ככלי IPC בסיסי\פרימיטיבי. מכיוון שכיום עומדים לרשותנו כלי IPC מתקדמים יותר, כפי שנראה בהמשך, אזי כיום ממעטים לעשות בסיגנלים שימוש לצורך זה. בכל-אופן נראה דוגמה.

יוניקס מעמידה לרשותנו שני סיגנלים: SIGUSR1, SIGUSR2 בהם נוכל לעשות שימוש חופשי, בפרט לשלחם מתהליך אחד למשנהו.

הדרך באמצעותה תהליך א' שולח סיגנל לתהליך ב' מתוך תכנית ב-C היא בעזרת ק.מ.:

kill(<process id>, <wanted signal>) ;

נציג עתה דוגמה לשליחת סיגנלים בין אב ובנו. נניח שהאב והבן מעוניינים לפעול לסירוגין פעם זה ופעם זה (אולי מפני שהאב מייצר משהו, והבן צורך אותו).


```
// file:  catch_usr_signal.c
// a dad and a son send signals to each other,
// and thus coordinate their progress.
// An example of an output at the end of the program

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>     // for fork()
#include <signal.h>
#include <sys/types.h>
//--------------------------------
void catch_sigusr1(int sig_num) ;
void do_son() ;
void do_dad(pid_t son_pid) ;

//--------------------------------
int main() {
  pid_t pid ;

  signal(SIGUSR1, catch_sigusr1) ;

  pid = fork() ;
  switch(pid) {
    case -1 : perror("fork() failed") ;
              exit(EXIT_FAILURE) ;
    case 0  : do_son() ;
              exit(EXIT_SUCCESS) ;
    default: do_dad(pid) ;
              exit(EXIT_SUCCESS) ;
  }
  return EXIT_SUCCESS ;
}
```

```
//---------------------------------
void do_dad(pid_t son_pid) {
  int i ;

  printf("pid of dad = %d, pid of son = %d\n",
         getpid(), son_pid) ;

  for (i=0; i < 5; i++) {
    pause() ;
    puts("Dad's turn") ;
    kill(son_pid, SIGUSR1) ;
  }
}
//---------------------------------
void do_son() {
  int i ;

  sleep(5) ;        // Very important! Gives dad enough
                    // time to send first printf,
                    // and issue pause().
                    // Otherwise son's puts() intrferes
                    // wth dd's prntf, and son's signal
                    // is lost, and we get a deadlock.
  for (i=0; i < 5; i++) {
    puts("Child's turn") ;
    kill(getppid(), SIGUSR1) ;
    pause() ;                // until u get a signal
  }
}
//---------------------------------
void catch_sigusr1(int sig_num) {
  signal(SIGUSR1, catch_sigusr1) ;
  printf(" process %d caught signal SIGUSR1\n",
         getpid()) ;
}

//---------------------------------
```

בשלב זה הילד אמור לשון

בלולאה, האב נותן לילד להתחיל, הוא הולך לשון עד שהילד יעירו, אז הוא עובד, מעיר הילד, והולך לישון

בלולאה, הילד מתחיל: ראשית: 'עובד', שנית: מעיר את האב, שלישית: הולך לישון


```
//---------------------------------

 /* An output:
   ==========
a.out
pid of dad = 1068, pid of son = 1069
Child's turn
 process 1068 caught signal SIGUSR1
Dad's turn
 process 1069 caught signal SIGUSR1
Child's turn
 process 1068 caught signal SIGUSR1
Dad's turn
 process 1069 caught signal SIGUSR1
Child's turn
 process 1068 caught signal SIGUSR1
Dad's turn
…

an output without son's sleep
============================
a.out
Child's turn                    <== son is scheduled first

                                <== dad ctches signal bfore
                                    it issues first printf()
process 1080 caught signal SIGUSR1
pid of dad = 1080, pid of son = 1081   <== dad's printf

dad enter loop, issues pause() and wait.
Son already sent his output, sent its signal,
and now he issues pause()
==> Both are waiting in a deadlock!!!

In any case we are in a race condition
Solution: semaphore as we shall learn
*/
```


4.5.7 כללי זהב לכתיבת טפלית סיגנל

א. ראוי שהטפלית תהיה קצרה ככל שאפשר. במידת הצורך היא תניף דגל (גלובלי) שייבדק ע"י התכנית הראשית, שגם תמשיך בַטיפול.

ב. עת תופסים SIGINT ראוי להניף דגל שיורה לתכנית להסתיים בהקדם, עת התכנית תגיע לנקודה בה נוח לה לסיים. (בפרק #5, בהקשר של פתילים, נקרא לכך: נקודת ביטוליות (cancelation point). כמובן שעל התכנית לבדוק את מצב הדגל מעת לעת.

ג. ראוי שהטפלית תמסך סיגנלים, ועדיף באמצעות sigaction() כדי לא לפתוח ולו חריץ בדלת למצבי מרוץ.

ד. עת נתפס סיגנל המעיד על באג אין לנסות להתחכם, יש לסגור הבאסְטה בזריזות ובבאסה (לסגור קבצים פתוחים, למחוק קבצים זמניים, להרוג ילדים), ולהתקפל. הקיפול ע"י החזרת מצב הטיפול באותו סיגנל למצב המחדלי, ואז שליחת אותו סיגנל לעצמה (raise()).

ה. סיגנל שנשלח לתהליך יתקבל ע"י התהליך רק עת התהליך במצב משתמש.

ו. אם סיגנל נשלח כמה פעמים ונותר תלוי (עדיין אינו מתקבל), הוא יתקבל פעם יחידה, וייחסם עד השלמת הטיפול בו.

ז. היזהרו מעצי באובאב ומטיימרים. זכרו שביכולתכם להחזיק רק טיימר יחיד, ותפעלוהו בשיקול דעת ובש˙ום שכל. זכרו: מה שלא יעשה השכל יעשה הזמן.

4.5.8 סיגנלים וקבוצת תהליכים

כל תהליך המורץ במערכת שייך לקבוצת תהליכים, מחדלית, התהליך נוסף לקבוצת התהליכים של אביו, אך עת ה- Shell מריץ תכנית כלשהי הוא מעביר את התהליך שמריץ את התכנית לקבוצת תהליכים נפרדת, הכוללת רק תהליך זה. מזהה קבוצת התהליכים שווה למספר התהליך (היחיד) הנכלל בה.

עת התכנית שלנו מבצעת fork() התהליך שנוצר משתייך, כרגיל, לקבוצת התהליכים של אביו. המוטיבציה: עת מעוניינים להשהות את התכנית, או להעיפה, רוצים לעשות זאת לכלל התהליכים הנכללים בתכנית.

כדי להעביר את תהליך הנוצר לקבוצת תהליכים נפרדת (הכוללת רק תהליך זה, ומזהה הוא מספר התהליך הנוכחי) נשתמש בק.מ.: setpgrp() .

בהתאמה, ק.מ. getpgrp() מחזירה את מזהה הקבוצה לה שייך התהליך.

נראה דוגמה:


```
//file: process_group.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>

void catch_int(int sig) ;

int main() {
  pid_t status ;

  signal(SIGINT, catch_int) ;

  printf("pid of father = %d = pgid = %d \n",
         getpid(),  getpgrp()) ;

  status = fork() ;
  if (status < 0) {
    perror("error in fork") ;
    exit(EXIT_FAILURE) ;
  }
  if (status > 0) {
     int i ;
     for (i=0; i < 10; i++) {
        puts("hello from father\n") ;
        sleep(1) ;
     }
     return(EXIT_SUCCESS) ;
  }
```

בינתיים, קיים תהליך יחיד, מספרו = מזהה קבוצתו

האב, בלולאה, מדפיס עשר פעמים את המשפט ...


```
  else
  {
     int i ;

     printf("son process %d belongs to set %d\n",
          getpid(),  (int)  getpgrp()) ;   // OR: getpgid(0)

     if (setpgrp() != 0) {                 // OR: setpgid(0, 0) ;
        perror("setpgrp() failed") ;
        exit(EXIT_FAILURE) ;
     }
     printf("son process %d belongs to set %d\n",
          getpid(),   getpgrp()) ;

     for (i=0; i < 10; i++) {
        puts("hello from son\n") ;
        sleep(1) ;
     }
     return(EXIT_SUCCESS) ;
}

//-----------------------------------------------------
void catch_int(int sig) {
  printf("process %d caught SIGINT, about to exit\n",
          getpid()) ;
  exit(EXIT_FAILURE) ;
}

//-----------------------------------------------------
```

הילד, מציג את מספרו, ואת הקבוצה לתוכה הוא נולד, אחר עובר לקבוצה נפרדת, ומציג שוב את אותם פרטים

מעבר לכך, גם הילד מציג עשר פעמים את המשפט שלו


```
/* a run:
 ========
<274|1>yoramb@inferno-05:~/os$ !g
gcc -Wall process_group.c
<275|0>yoramb@inferno-05:~/os$ !a
a.out
pid of father = 5841 = pgid = 5841
son process 5842 belongs to set 5841

son process 5842 belongs to set 5842
hello from son
hello from father
hello from son
hello from father
hello from son
hello from father

process 5841 caught SIGINT, about to exit
<276|1>yoramb@inferno-05:~/os$ hello from son

hello from son
hello from son
hello from son
hello from son
hello from son
hello from son

<276|1>yoramb@inferno-05:~/os$
```

מזהה האב וקבוצתו:
מזהה הילד וקבוצתו:
מזה הילד וקבוצתו אחרי ההעברה:

^c שולח סיגנל לאב, ורק לאב, הוא מסיים, בנו ממשיך לרוץ

```
a run in which setpgrp() is closed as a remark
==========================================
<279|0>yoramb@inferno-05:~/os$ !a
a.out
pid of father = 7767 = pgid = 7767
son process 7768 belongs to set 7767
son process 7768 belongs to set 7767
hello from son
hello from father
hello from father
hello from son
hello from father
hello from son
hello from son
hello from father
process 7768 caught SIGINT, about to exit
process 7767 caught SIGINT, about to exit

*/
```

בהרצה שניה, פקודת ה:
setpgrp() נסגרה בהערת תיעוד.
^c *יחיד* שהוקלד שולח SIGINT
לשני התהליכים (ושניהם
מסתיימים)

4.6 צינור (pipe)

כאמור, בפרק הנוכחי אנו מכירים מספר כלים לתקשורת בין תהליכים הקיימים במ.ה. יוניקס. אחד המוקדמים שבין הכלים הללו הוא הצינור. באופן רגיל, צינור מאפשר לתהליך הורה להתקשר עם צאצאיו, ולעתים (כתלות באופן ייצורו: אם הוא יוצר לפני שהם יוצרו) מאפשר לצאצאים להתקשר אלה עם אלה. במובן זה קיימת מגבלה על התהליכים שעשויים להעביר הודעות באמצעותו.

כבר נפגשנו במושג הצינור ביוניקס, כמשתמשים, עת כתבנו:

ls my_dir | grep "pipe" | more

נזכיר: הפלט של פקודת ה- ls my_dir מהווה קלט לפקודה "pipe" grep . והפלט של פקודה זאת הינו הקלט של פקודת ה- more.

נכיר ראשית את מושג הצינור כפי שמיוצר ע"י תהליך ביוניקס, ואחַר נראה כיצד אותו צינור משמש אותנו גם בְפקודות כגון אלה שמופיעות מעל.

דוגמה קטנה:

תכנית א' (קומפלה ל: prog1):

```
#include <stdio.h>
#include <stdlib.h>
//--------------------------------


int main() {

   puts("17") ;

   return EXIT_SUCCESS ;
}
```

תכנית ב' (קומפלה ל: prog2):

```
#include <stdio.h>
#include <stdlib.h>
//--------------------------------

int main() {

   int n ;

   scanf(" %d", &n) ;
   printf("%d\n", n*n) ;

   return EXIT_SUCCESS ;
}
```

הרצה: prog1 | prog2

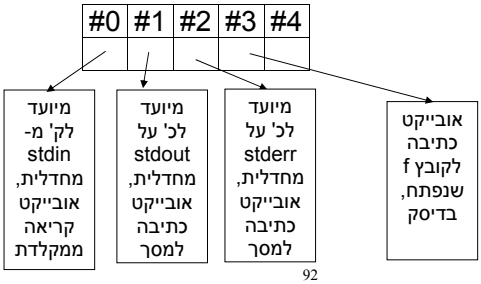
הפלט: 289

4.6.1 רקע

הזכרנו שבמ.ה. יוניקס כל הקלט והפלט של התהליך מול מגוון התקנים (מסך, קובץ על הדיסק, על התקן USB) נעשה באופן אחיד: למשל הכתיבה, תיעשה תמיד באמצעות ק.מ. ()write או לחילופין, באמצעות פונ' הספרייה ()fprintf (או מקבילותיה) של שפת C .

עתה נאמר שכל נתון שהתכנית פולטת אל מחוץ לה (לקובץ, לתהליך אחר, לרשת) או קולטת מבחוץ מגיע אליה באמצעות מתאר (descriptor) שהינו הכלי *האחיד* שמאפשר לתכנית ק\פ במובן הרחב של המילה.

מ.ה. מנהלת עבור כל תהליך מערך של מתארים (המוחזק ב- PCB של התהליך). על מנת שהתהליך יוכל לק\כ מ\להתקן צריך שבמערך תוקצה כניסה\תא מתאימ\ה. הכניסה המכילה מצביע שמורה על אובייקט שיודע לק\כ נתונים מ\על אותו התקן.



אנו מקבלים (לכל הפחות) שתי רמות בפניה לציוד:

א. <u>רמה לוגית</u> = תפקיד הכניסה מבחינת התהליך (לדוגמה: ק' מ- stdin).

ב. <u>רמה פיזית</u> = לאן הכניסה מצביעה, כלומר לאיזה ציוד בפועל נקשרת הכניסה (מהיכן וכיצד מגיע הקלט עת קוראים מ- stdin).

עת התהליך מתחיל לרוץ מוקצות לו (אוטומאטית) שלוש כניסות בטבלה:

א. כניסה 0# מתייחסת לקלט הסטנדרטי, ובאופן מחדלי מכילה מצביע לאובייקט שיודע לקרוא מידע מהמקלדת (אולם מחדל זה ניתן, כידוע, לשנות).

ב. כניסה 1# מתייחסת לפלט הסטנדרטי.

ג. כניסה 2# מתייחסת לקובץ השגיאה הסטנדרטי.

במידה והתכנית מעוניינת לקרוא\לכתוב קלט\פלט מ\לקובץ נוסף ('קובץ חיצוני' כפי שקראנו לו בשנה שעברה), או מ\לכל כל רכיב ציוד אחר (רמקול, רשת) עליה ראשית לפתחו (כפי שאנו זוכרים(?) מהתכניות שכתבנו בשפת C++/C: fopen() או ifopen()). פעולת הפתיחה מקָצה כניסה נוספת במערך המתארים, ומאתחלת את הכניסה עם מצביע כנדרש.

בתום הטיפול בציוד יש לסגור את הקובץ.

ניזכר בנושא בעזרת שתי דוגמות קטנות. הראשונה משתמשת בפונ' ספריה של C, והשנייה בק.מ:

```
// file: io-c-lib.c
// A small example of a program that performs i/o
// using C library routines

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
  FILE *fdr, *fdw ;                    // file descriptors in C
  char c ;

  if (argc != 3) {
    fputs("usage: a.out <inp file> <out file>", stderr) ;
    exit(EXIT_FAILURE) ;
  }

  fdr = fopen(argv[1], "r") ;          // openning the files
  fdw = fopen(argv[2], "w") ;
  if (!fdr || !fdw) {
    perror("failed to open input or output files") ;
    exit(EXIT_FAILURE) ;
  }

  while ((c = fgetc(fdr)) != EOF)   {     // reading/writing
    fputc(c, fdw) ;                       // a single char
    fputc(c, stdout) ;                    // from/to a file
  }
  fclose(fdr) ;                    // closing the files
  fclose(fdw) ;

  exit(EXIT_SUCCESS) ;
}
```



```
// file: io-system-calls.c
// A small example of a program that performs i/o
// using system calls
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>        // STDIN_FILENO, …

int main(int argc, char **argv) {
  int fdr, fdw ;                  // file descriptors
  char c ;                        למה int?
  if (argc != 3) {
    fputs("usage: a.out <inp file> <out file>", stderr) ;
    exit(EXIT_FAILURE) ;
  }
  fdr = open(argv[1],  O_RDONLY) ;   // open files
  fdw = open(argv[2], O_WRONLY |  O_CREAT) ;
  if (fdr < 0 || fdw < 0) {
    perror("failed to open input or output files") ;
    exit(EXIT_FAILURE) ;
  }

  while (read(fdr, &c, 1) )        // read/write a single char
  {                                // from/to the files
    if (write(fdw, &c, 1) != 1) {
      perror("write() failed") ;
      exit(EXIT_FAILURE) ;
    }
    write(1, &c, 1) ;              // echo char to stdout
  }
  close(fdr) ;                     // close the files
  close(fdw) ;
  exit(EXIT_SUCCESS) ;
}
```

למה 1 בארגומנט ראשון? STDOUT_FILENO

עד כאן על ק\כ מקבצים באופן כללי.

4.6.2 ייצור צינור

צינור מורכב מזוג מתארי קבצים: האחד מיועד לקריאה ומשנהו לכתיבה. במובן זה הצינור מדמה קובץ, אולם בפועל נתוניו מוחזקים בזיכרון, דבר לא מוקצה עבורו על-גבי מצע האחסון, ובהתאמה נפח הנתונים שהוא יכול להחזיק בכל נקודת זמן מוגבל מאוד (בד"כ, סד"ג של כעשרה קילו בית). הצינור נעלם עת כל התהליכים שהיו שותפים לו מסתיימים. כמו כל כלי ה- IPC הוא מאפשר רק לתהליכים שרצים *במקביל* להעביר ביניהם מידע.

צינור הוא אפיק תקשורת חד-כיווני המתנהל כתור, משמע כל תהליך עשוי רק לקרוא או רק לכתוב מ\על הצינור, ונתון שנכתב קודם גם ייקרא קודם.

פעולת ייצור הצינור מקצה את שני המתארים: (שתי כניסות במערך המתארים של התהליך) האחד לקריאה, ומשנהו לכתיבה.

נראה דוגמה:

```
#include <stdio.h>
#include <stdlib.h>     // for exit()
#include <string.h>     // for strlen()
#include <sys/types.h>
#include <unistd.h>    // for sleep(), execvp(), pipe()
#include <sys/wait.h>   // for wait()

#define        N        80

int main() {
  int pipe_descs[2] ;

  if (pipe(pipe_descs) == -1) {
    fprintf(stderr, "cannot open pipe\n") ;
    exit(1) ;
  }
```


4.6.3 ק\כ מ\לצינור

עתה, בהנחה שהצינור יוצר בהצלחה, אנו יכולים לקרוא ממנו נתונים, באמצעות ק.מ. read() דרך המתאר pipe_descs[0]:

```
char buff[N] ;
int nbytes ;

nbytes = read(pipe_descs[0], buff, N) ;
```

הכתיבה מתבצעת באופן סימטרי, באמצעות ק.מ. write(). (פירוט ערך ההחזרה של ק.מ. בהמשך.)

עד כאן, התהליך שייצר את הצינור יכול לתקשר באמצעותו אך ורק עם עצמו.

כיצד נשנה את המצב?
התהליך יבצע fork().
הילד שייוולד יהיה עותק של הורהו (כולל מערך המתארים המוחזק ב- PCB) בפרט יעמדו לרשותו שני מתארי הקבצים באמצעותם ניתן לכ\ק מ\על הצינור.

מה נעשה עתה?
על-פי צרכינו, אחד משני התהליכים יכתוב, ורק יכתוב, על הצינור; משנהו, יקרא, ורק יקרא, מהצינור.

בהתאמה:
ראשית, הקורא יסגור את pipe_descs[1], והכותב יסגור את pipe_descs[0] (שכן הם אינם זקוקים להם).

שנית, הם יוכלו לתקשר זה עם זה: הכותב יכתוב על הצינור (דרך התא pipe_descs[1] במערך המתארים), והקורא יקרא מהצינור (דרך pipe_descs[0]).

לבסוף, כל אחד מהם יסגור את מתאר הקובץ בו הוא השתמש: הקורא יבצע: close(pipe_descs[0]); , ובהתאמה לכותב.

נראה את התכנית השלמה:


```
// file:  pipes1.c

#include <stdio.h>
#include <stdlib.h>     // for exit()
#include <string.h>     // for strlen()
#include <sys/types.h>
#include <unistd.h>    // for sleep(), execvp(), pipe()
#include <sys/wait.h>   // for wait()

#define        N        80

int main() {
  int pipe_descs[2] ;

  if (pipe(pipe_descs) == -1) {
    fprintf(stderr, "cannot open pipe\n") ;
    exit(EXIT_FAILURE) ;
  }

  pid_t status = fork() ;

  if (status < 0) {
    fprintf(stderr, "error in fork\n") ;
    exit(EXIT_FAILURE) ;
  }
```

```
if (status == 0)                          // son process
{
     char s[] = "hi dad!" ;

     close(pipe_descs[0]) ;               // son writes to pipe
     write(pipe_descs[1], s, strlen(s)+1) ;
     close(pipe_descs[1]) ;
     exit(EXIT_SUCCESS) ;
}
else                                      // father process
{
     char buff[N] ;
     int nbytes ;

     close(pipe_descs[1]) ;               // father reads
     nbytes = read(pipe_descs[0], buff, N) ;
     printf("Got: %s (nbytes=%d) from son\n",
                                                buff, nbytes) ;
     close(pipe_descs[0]) ;
     exit(EXIT_SUCCESS) ;
  }

  return EXIT_SUCCESS ;
}
```


ק.מ. ()read עת מופעלת על צינור מחזירה:

א. אם בצינור קיימת כל כמות הנתונים הדרושה, אזי את הכמות המבוקשת; אחרת, אם קיימת כמות חלקית יוחזר מה שיש.

ב. אם הצינור ריק, אך יש כותבים פעילים (כאלה עם מתאר ק' פתוח) אזי התהליך ייחסם עד שמישהו יכתוב לצינור.

ג. אפס מוחזר אם אין אף מתאר כ' פתוח. כך יודע הקורא שעליו לסיים את לולאת הק'. (בקובץ רגיל, אפס מוחזר עת מגיעים ל- eof).

ד. 1- מוחזר על כישלון.

הקורא מצופה להמשיך בקריאה עד החזרת הערך אפס. אם הקורא סוגר את המתאר שלו, ואחַר הכותב מנסה לכתוב על הצינור, אזי לכותב נשלח הסיגנל SIGPIPE ומחדלית הוא מועף (עם הודעת השגיאה: broken pipe). אם הכותב אינו מועף אזי ()write מחזירה 1- ו- errno מתעדכן לערך EPIPE.

הקורא, על-כן יבצע משהו כמו:

```
while (read(my_pipe[0], &c, 1) > 0)
   do something with c
```


הכתיבה לצינור סימטרית לקריאה: אם יש מקום בצינור תיכתב כל הכמות, אחרת התהליך הכותב ייחסם. כאמור, כתיבה עת לא קיימים קוראים הינה בגדר שגיאה.

כתיבה לצינור של פחות מ- PIPE_BUF בתים הינה אטומית; כתיבה של יותר מכך עלולה להתערבב עם כתיבה של תהליך שני שכותב במקביל.

בדוגמה שבתכנית מעל לא בדקנו מהו ערך ההחזרה של read/write, זה לא יפה ולא ראוי: הבודק אמור לבדוק שכל מה שהוא רצה לכתוב אכן נכתב, הקורא בד"כ צריך לדעת כמה הוא קרא.

4.6.4 ק'\כ' מצינור בעזרת scanf/printf ושות'

נניח שפתחנו צינור (או כלי IPC אחר) בעזרת ק.מ. ()open (המחזירה int המציין מספר תא במערך המתארים), ולא באמצעות פונ' הספרייה ()fopen של שפת C (הפותחת זרם [stream] ומחזירה מצביע מטיפוס * FILE).

נניח שלמרות זאת ברצוננו לכתוב על 'הקובץ' שנפתח באמצעות פונ' הספרייה של שפת C (המקבלות משתנה מטיפוס * FILE).

כדי לאפשר זאת יהיה עלינו לפתוח את מתאר הקובץ שקיבלנו באמצעות פונ' הספרייה ()fdopen.
נראה דוגמה פשוטה:

```
// file:  pipe1_fdopen.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

#define      N      80

int main() {
  int pipe_descs[2] ;
  FILE *fdr, *fdw ;

  if (pipe(pipe_descs) == -1) {
    fputs("cannot open pipe", stderr) ;
    exit(EXIT_FAILURE) ;
  }
```

המשתנים: הן לפתיחת צינור, והן לק'\כ' מ\על קובץ זרם בעזרת פונק' הספרייה של שפת C

```
pid_t status = fork() ;

  if (status < 0) {
    fputs("cannot fork", stderr) ;
    exit(EXIT_FAILURE) ;
  }

  if (status == 0)                    // son process
  {
     char s[] = "hi-dad!" ;

     close(pipe_descs[0]) ;           // son writes to pipe
     fdw = fdopen(pipe_descs[1], "w") ;
     fprintf(fdw, "%s", s) ;
     fclose(fdw) ;
     exit(EXIT_SUCCESS) ;
  }
  else                                // father process
  {
     char buff[N] ;

     close(pipe_descs[1]) ;           // father reads
     fdr = fdopen(pipe_descs[0], "r") ;
     fscanf(fdr, "%s", buff) ;
     fclose(fdr) ;
     printf("Got: %s from son\n", buff) ;
     exit(EXIT_SUCCESS) ;
  }
  return EXIT_SUCCESS ;
}
```

כך 'נפתח' את קצה הכתיבה של הצינור לכתיבה ע"י פונ' של C

באופן סימטרי בקורא

4.6.5 הקשר בין צינור שנפתח ע"י תהליך לבין זה המוכר מה- Shell

עד כה ראינו כיצד צינור משמש להעברת מידע בין תהליכים שאנו יצרנו. מנגד, הזכרנו כי המונח מוכר לנו כמשתמשים בַ- shell. עתה נלמד את הקשר בין שני המושגים.

אמרנו שעת התכנית שלנו שולחת פלט, למשל:
puts("Hello World");
היא, מבחינתה, שולחת את הנתונים דרך הזרם (stream) הקרוי במונחים שלה stdout, ואשר קשור למתאר הקובץ #1. זרם זה התכנית אינה פותחת, ומ.ה. מנתבת דרך מתאר הקובץ #1 במערך המתארים, שם קיים מצביע אשר באופן מחדלי מורה על אובייקט המעביר נתונים למסך.

עתה נניח שייצרנו כבר צינור בשם pipe_descs, ביצענו fork(), והתהליך הכותב מבצע את זוג הפקודות:
close(STDOUT_FILENO) ;
dup(pipe_descs[1]) ;
מה עשינו? (נצייר):
הפקודה הראשונה מבין השתיים סגרה את מתאר הקובץ STDOUT_FILENO (כלומר את התא #1), והפכה אותו לכזה שאינו בשימוש.

הפקודה השנייה משכפלת את ערכו של התא pipe_descs[1] במערך המתארים (זה הכולל מצביע לאובייקט שיודע לשלוח נתונים לצינור) על התא הפנוי *הראשון* במערך המתארים: במקרה הנוכחי תא #1.

התוצאה: עת התכנית שלנו תכתוב ל- stdout, ותשלח את הפלט, על-כן, באמצעות האובייקט עליו מצביע התא #1 במערך המתארים, לאן יגיע הפלט?
לצינור.

אם הקורא יעשה דבר סימטרי נקבל שהפלט של היצרן נשלח לצינור, ועת הצרכן קורא את הקלט שלו מ- stdin הוא מקבל, דרך הצינור, את מה שכתב היצרן:


```
// file: pipes_dup.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

#define      N      80

int main() {
  int pipe_descs[2] ;

  if (pipe(pipe_descs) == -1) {
    fputs("cannot open pipe", stderr) ;
    exit(EXIT_FAILURE) ;
  }

  pid_t status = fork() ;

  if (status < 0) {
    fputs("error in fork", stderr) ;
    exit(EXIT_FAILURE) ;
  }
```

```
  if (status == 0)                    // son process
  {
     close(pipe_descs[0]) ;
     close(STDOUT_FILENO) ;
     dup(pipe_descs[1]) ;
     puts("hi dad! it's me") ;
     close(pipe_descs[1]) ;
     exit(EXIT_SUCCESS) ;
  }
  else                                // father process
  {
     char buff[N] ;

     close(pipe_descs[1]) ;
     close(STDIN_FILENO) ;
     dup(pipe_descs[0]) ;
     scanf(" %s", buff) ;
     printf("Got: %s from stdin\n", buff) ;
     scanf(" %s", buff) ;
     printf("Got: %s from stdin\n", buff) ;
     close(pipe_descs[0]) ;
     return(EXIT_SUCCESS) ;
  }

  return EXIT_SUCCESS ;
}
```

נצייר את מערך מתארי הקבצים

עתה נצעד צעד קטן קדימה, ונניח שהאב והבן מבצעים exec().
נזכור ש- exec() שומר על מערך הקבצים הפתוחים כמות שהוא (המערך מצוי ב- PCB של התהליך בגרעין, שאינו 'עובר מוטציה'.)

נראה דוגמה:


```
// file: pipes_dup_exec.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

#define     N     80
#define     IN    0
#define     OUT   1

int main() {
  int pipe_descs[2] ;

  if (pipe(pipe_descs) == -1) {
    fprintf(stderr, "cannot open pipe\n") ;
    exit(EXIT_FAILURE) ;
  }

  pid_t status = fork() ;

  if (status < 0) {
    fputs("error in fork", stderr) ;
    exit(EXIT_FAILURE) ;
  }
```



```
  if (status == 0)                    // son process
  {
     close(pipe_descs[IN]) ;
     close(STDOUT_FILENO) ;
     dup(pipe_descs[OUT])
     execlp("/bin/ls", "ls", "-l", NULL) ;
     // principally, we do not reach here
  }
  else                                // father process
  {
     close(pipe_descs[OUT]) ;
     close(STDIN_FILENO) ;
     dup(pipe_descs[IN]) ;
     execlp("/bin/sort", "sort", NULL) ;
     // principally, we do not reach here
  }

  return EXIT_SUCCESS ;
}
```

הבת משנה עורה להיות ls, השולח פלט ל- stdout

האם משנה חברבורותיה להיות sort, הקורא קלט מ- stdin

ובכל אופן, הפלט של האחת הוא הקלט של השנייה.
איזה יופי!

<u>הערה</u>

את זוג הפקודות close(STDIN_FILENO) + dup(pipe_descs[0]) ניתן להמיר בפקודה יחידה:

dup2(STDIN_FILENO, pipe_descs[0]) ;

המבצעת את שתי הפעולות יחד.

צינור, כפי שראינו, הינו מדיום חד כיווני. כדי ששני תהליכים יוכלו לשלוח מידע זה לזה יש להשתמש בשני צינורות: כל תהליך יכתוב לצינור אחד, ויקרא ממשנהו.

סידור כזה עלול להכניס אותנו לחסימה הדדית (deadlock):

א. נניח שמייד בתחילת ריצתם שני התהליכים מנסים לקרוא, כל אחד מצינור הקריאה שלו. התוצאה: שניהם נחסמים עד ש-'מישהו' יכתוב נתונים לצינור. רק ש-'מישהו' כזה אינו בנמצא.

ב. באופן סימטרי: אם שניהם מנסים לכתוב, כל אחד על צינור הכתיבה שלו, יותר נתונים מכפי שהצינור יכול להכיל, אזי שניהם ייחסמו עד ש-'מישהו' יקרא נתונים מהצינור.

4.7 תור (FIFO) או צינור משוים (Named Pipe)

צינור, כפי שראינו בסעיף הקודם, הוא 'אנונימי' – חסר שם, וכזה יכול לשמש רק להתקשרות של אב קדמון וצאצאיו. כדי שתהליכים שאין ביניהם 'קרבת דם' יוכלו להעביר מידע באופן דומה נשתמש ב-'צינור משוים' הקרוי גם 'תור' (שנאמר: קול התור נשמע בארצנו).

תור הוא קובץ במערכת הקבצים (אם יצרתם תור בשם ff ואחר תריצו ls תראו שקיים במדריך קובץ בשם ff שנפחו הוא אפס בתים). תהליכים *הרצים במקביל* ומעוניינים להעביר נתונים, יכתבו\יקראו על\מהקובץ (כמו על\מכל קובץ). עת רק ק' או רק כ' פותח את הקובץ, הוא נחסם, שכן אין לו עם מי להעביר נתונים.

את קובץ התור נוכל לייצר מה- shell באמצעות הפקודה:

mkfifo my_fifo

אם אחר נקליד: ls –l אזי יתקבל הפלט:

```
prw-r--r-- 1 yoramb teach     0 Aug 17  2006 my_fifo
```

מורה שזהו named pipe (בקובץ רגיל יופיע -, במדריך: d)

כדי לצור תור מתכנית בשפת C נכתוב:

mkfifo("my_fifo", S_IFIFO | 0644)

שם הקובץ | זהו תור | הרשאות: rw-r--r--

הפונ' מחזירה אפס בהצלחה, 1- בכישלון.
סיבה אפשרית לכישלון עשויה להיות שהתור כבר קיים. נחזור לכך בקרוב.

יצירת התור אינה פותחת אותו לק'\כ' (היא רק יוצרת את הקובץ), לכן מעבֵר ליצירה יש גם לפותחו.

אחרי שהתור נוצר, כך או אחרת, זוג תכניות הרצות במקביל יכולות להעביר נתונים באמצעותו. נראה דוגמה פשוטה (בהנחה שהתור כבר נוצר):

הכותבת:

```
// file: named_pipe_write.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
      FILE *fdw = fopen("my_fifo", "w") ;
      int i ;

      if (!fdw) {
         perror("cannot open my_fifo pipe") ;
         exit(EXIT_FAILURE) ;
      }
      for (i= 0; i<10; i++) {
         fprintf(fdw, " %d", i) ;
         fflush(fdw) ;

         printf("sent %d to my_fifo\n", i) ;
         sleep(3) ;
      }

     return EXIT_SUCCESS ;
}
```

דוגמה לתכניות שגם יוצרות את התור:

```
// file: named_pipe_write2.c
//adopted from
// http://www.ecst.csuchico.edu/~beej/guide/ipc/fifos.html
// Another example of fifo: Programs create the fifo
// themselves
// (if it exists then they do not fail)

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#define FIFO_NAME "a_fifo"
#define STR_LEN   100

int main()
{
      char s[STR_LEN];
      FILE *fdw;

      // errno == EEXIST if fifo already exists
      if (mkfifo(FIFO_NAME, S_IFIFO | 0644) == -1 &&
         errno != EEXIST) {
            perror("cannot create fifo file") ;
            exit(EXIT_FAILURE) ;
      }

      if (!(fdw = fopen(FIFO_NAME, "w"))) {
         perror("cannot open fifo file for w") ;
         exit(EXIT_FAILURE) ;
      }
```


והקוראת:

```
// file:  named_pipe_read.c
#include <stdio.h>
#include <stdlib.h>

int main() {
      FILE *fdr = fopen("my_fifo", "r") ;
      int status , num;

      if (!fdr) {
         perror("cannot open my_fifo pipe for read") ;
         exit(EXIT_FAILURE) ;
      }

      status = fscanf(fdr, " %d", &num) ;
      while (status != EOF)  {
         printf("got %d from my_fifo\n", num) ;
         status = fscanf(fdr, " %d", &num) ;
      }

      return EXIT_SUCCESS ;
}
```

ערכו של EOF בד"כ 1-, אך ראוי לא להסתמך על כך

ובת-זוגה, הקוראת:

```
// file: named_pipe_read2.c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#define FIFO_NAME "a_fifo"
#define STR_LEN   100

int main()
{
      char s[STR_LEN];
      FILE *fdr;

      if (mknod(FIFO_NAME, S_IFIFO | 0666, 0) == -1 &&
                errno != EEXIST) {
         perror("cannot create fifo file") ;
         exit(EXIT_FAILURE) ;
      }

      if (!(fdr = fopen(FIFO_NAME, "r")))   {
         perror("cannot open fifo file for r") ;
         exit(EXIT_FAILURE) ;
      }

      printf("got a writer--ready to get some stuff\n");
      while ( fscanf(fdr, " %s", s) != EOF )
         printf("Got: %s\n", s) ;

      return EXIT_SUCCESS ;
}
```

אופן מיושן יותר לצור תור. האפס בסוף חסר משמעות


```
      puts("got a reader--type some stuff\n");
      while ( fgets(s, STR_LEN, stdin) != NULL) {
         fprintf(fdw, " %s\n", s) ;
         fflush(fdw) ;          // <== important
      }

      return EXIT_SUCCESS ;
}
```

הערות אודות תור:

א. תהליך רשאי לפתוח את קובץ התור לק' *או* לכ', (אך לא לשתי הפעולות גם יחד).
ב. כאשר תהליך א' פתח את התור (לק' או לכ') ואין תהליך אחר שפתח את התור באופן המשלים, תהליך א' נחסם.
ג. המידע מגיע רק אחרי שהכותב מבצע fflush()
ד. כאשר הכותב סוגר את התור מוחזר לקורא EOF.
ה. הקורא, מצדו, מצופה להמשיך בק' עד קבלת EOF.
ו. כתיבה על תור אחרי סגירתו ע"י הקורא גורמת לשליחת הסיגנל SIGPIPE לכותב, ובאופן מחדלי מביאה להעפתו.
ז. בעת פתיחת התור ניתן להוסיף את הדגל: O_NONBLOCK.
כותב: שיוסיף דגל זה ייכשל בפתיחה אם לא ממתין לו קורא.
קורא: שיוסיף דגל זה, וינסה לקרוא מהתור עת אין בתור נתונים, לא יושהה, אלא יקבל חזרה אפס בתים (לא ניתן יהיה לדעת האם לא קיים כ', או שקיים כותב רק שהכותב לא כתב דבר).
ח. כתיבה של פחות מ- PIPE_BUF בתים לתור הינה אטומית. כתיבה של יותר מכך אינה.

4.8 תורי הודעות Message Queues

תורי הודעות (יחד עם זיכרון משותף שנכיר בסעיף הבא, וסמפורים שנכיר בסמסטר הבא) הם כלי IPC שמקורם ב- system V, ושחולקים ביניהם כמה מאפיינים משותפים כפי שנכיר. הם נקראים XSI IPC (XSI = X/Open System Interface) (X/Open הוא סטנדרט לינוקס מקביל ל- POSIX).

יתרונם של תורי ההודעות שהם יותר גמישים ופחות גולמיים מהתור (או הצינור):

א. בתור המידע מועבר כזרם של בתים. על השולח והמקבל לתאם ביניהם כיצד יפורש, 'יפורסס' (be parsed) הזרם. בתור הודעות, לעומת זאת, מועברות הודעות, הודעות. כל הודעה היא מערך של תווים ויש לה 'טיפוס', 'סוג' (= מספר טבעי > 0)
ב. בשל חלוקת ההודעות לסוגים, קורא מתור הודעות אינו חייב לשלוף את ההודעה שנשלחה ראשונה, ונמצאת בראש תור ההודעות, הוא יכול לבקש לקבל רק הודעות מטיפוס מסוים, ואז תועבר לו ההודעה הראשונה מטיפוס זה (בקשת הודעות מטיפוס אפס משמע רצון לקבל כל הודעה שהיא).

תור ההודעות מנוהל (ע"י הגרעין) כרשימה מקושרת, בה כל איבר חדש נוסף בסוף.

בניגוד לכמה כלי IPC אחרים, תורי הודעות או תכולתם אינם ממוחזרים עת כל התהליכים שהשתמשו בהם מסתיימים. הם נותרים במערכת (צורכים זיכרון, וניתן לעשות בהם שימוש גם בהמשך), עד מחיקתם המפורשת (או עד reboot).

בניגוד לכלי IPC אחרים הטיפול בהם אינו בדומה לטיפול בקבצים, ועל כן לא ניתן להשתמש בכלים המגוונים הקיימים לטיפול בקבצים (כפי שנכיר) ומנגד יש צורך בק.מ. מיוחדות עבורם.

ברוב הגרסות של יוניקס ניתן לייצר לכל היותר כמה עשרות תורי הודעות בסה"כ (בכל המערכת), וכולם יחד יכולים להכיל כמה עשרות הודעות, כל אחת בגודל כמה אלפי בתים.

כל תור הודעות (כמו גם מקטע זיכרון משותף, וסמפור) מזוהה *פנימית* עבור מ.ה. באמצעות מספר טבעי המכונה מזהה (identifier) המונפק ע"י מ.ה. בסדר מספרי עולה: כל תור חדש מקבל את המספר הבא בתור.

תהליכים המעוניינים לפנות לתור ההודעות (להוסיף או לשלוף ממנו הודעות), צריכים, כך או אחרת *להיפגש* (rendezvous), כלומר להכיר את המזהה. כיצד הם ידעו מהו המזהה?

לתור קיים גם מפתח חיצוני המועבר למ.ה. בפקודת ייצור התור, ויכול (ליתר דיוק צריך) להיות מוכר לתהליכים השונים המעוניינים להשתמש בתור, והוא שיעזור להם להפגש:

המפתח החיצוני ייקשר למפתח הפנימי. בעת ייצור תור הודעות חדש, או התקשרות לתור קיים, יסופק המפתח החיצוני, מ.ה. תחזיר את המפתח הפנימי, ובמפתח הפנימי יעשה שימוש בעת שליחת וקבלת הודעות.

עתה נשאל: כיצד ידעו התהליכים מהו המפתח החיצוני?

א. המייצר (בד"כ השרת אליו פונים באמצעות התור) ישמור אותו בקובץ כלשהו.
<u>החיסרון</u>: מחייב שימוש בקבצים.
ב. המייצר והאחרים יסכימו מראש על *מפתח* (חיצוני) אותו יספק היצרן לפקודת ייצור התור. אם כולם ישתמשו באותו מפתח הם יופנו לאותו תור, עם אותו מזהה תור (פנימי).
<u>החיסרון</u>: אם מישהו כבר ייצר תור תוך שימוש בַמפתח המוסכם פקודת הייצור תכשל.
ג. המייצר והאחרים יסכימו על קובץ במערכת הקבצים (לדוגמה: /tmp), ועל תו כלשהו 'המציין את הפרויקט' (לדוגמה: y). ויפעלו באופן הבא:

הקובץ והתו יסופקו לפקודה הנקראת ftok() ואשר תנפיק עבורם (הן עבור השרת, והן עבור הלקוחות) את המפתח, וכך נקטין את הסיכוי לשימוש חוזר במפתח קיים
לדוגמה:

```
key_t key ;
if ((key = ftok("~yoramb/os", 'y')) == -1) {
   perror("ftok() failed") ;
   exit(EXIT_FAILURE) ;
}
```


בהינתן המפתח, נוכל לצור תור הודעות חדש, או להצטרף לתור קיים, באמצעות הפקודה msgget():
נגדיר:

```
int msgid ;
```

כדי לייצר תור חדש תהליך השרת יבצע:

```
if ((msgid = msgget(key,
                    IPC_CREAT | IPC_EXCL | 0600))
    == -1) {
   perror(…) ;
   exit(…) ;
}
```

הארגומנט השני מורה כי: יש לייצר תור חדש, יש להיכשל אם התור כבר קיים, והרשאות הגישה לתור הן לק'+כ' למשתמש הנוכחי בלבד.

לעומת זאת, בלקוחות, שמעוניינים להקשר לתור שכבר נוצר ע"י השרת, נכתוב:

```
if ((msgid = msgget(key, 0)) == -1)  {
   perror(…) ;
   exit(…) ;
}
```

<u>נסביר</u>: הלקוחות מעוניינים רק להקשר לתור קיים (לא לייצרו, ובטח ובטח שלא להיכשל אם הוא כבר קיים).

עד כאן ייצרנו מפתח (חיצוני) לתור, וייצרנו תור חדש או נקשרנו לתור קיים, תוך שאנו מקבלים את מזהה התור (הפנימי, של מ.ה.). עתה נוכל להעביר הודעות בעזרת התור.

כל הודעה תכלול:
א. טיפוס ההודעה = מספר טבעי <u>חיובי ממש</u>.
ב. תוכן ההודעה = מערך של תווים בגודל המתאים.
את שני אלה נארוז במבנה שנגדיר (הן בשרת והן בלקוחות):

```
struct my_msgbuf {
   long mtype ;
   char mtext[MAX_TEXT] ;
} ;
```

ובמשתנה מטיפוס המבנה:

```
struct my_msgbuf my_msg ;
```


<u>שליחת הודעה</u>
תבוצע באופן הבא:

```
my_msg.mtype = 17 ;
scanf(" %s", my_msg.mtext) ;
status = msgsnd(msgid,
                (struct msgbuf *) &my_msg,
                srtlen(my_msg.mtext)+1,
                0) ;
```

<u>נסביר</u>:
א. המרת הטיפוס של הארגומנט השני היא לטיפוס המצביע שמקבלת הפונ'.
ב. הארגומנט השלישי כולל את מספר הבתים במרכיב ההודעה בלבד.
ג. אפס בארגומנט האחרון משמע: ההודעה תשלח עת בתור יהיה די מקום, ועד אז השולח נחסם. (לחילופין, בעזרת הדגל IPC_NOWAIT אפשר לבקש שבמידה ואין מקום בתור הפעולה תכשל).
ד. הערך המוחזר: 1- בכישלון, 0 בהצלחה.

קבלת הודעה

נעשית בדומה לשליחה (כמובן אחרי ביצוע msgget() + ftok()):

אמרנו כי מקבל הודעה יכול לקבוע האם ברצונו לקבל כל הודעה שהיא, או רק הודעות מטיפוס מסוים, על כן הוא יגדיר:

```
long int allowed_type ;
```

ויכניס למשתנה:

א. ערך אפס כדי לציין שברצונו לקבל כל הודעה שהיא.

ב. ערך חיובי ממש p כדי לקבל הודעות מהטיפוס p בלבד.

כמו כן יוגדר משתנה זהה לזה ששימש אותנו בשליחה:

```
struct my_msgbuf my_msg ;
```


פעולת הקבלה msgrcv() הינה:

```
status = msgrcv(msgid,
                (struct msgbuf *) &my_msg,
                MAX_TEXT,
                allowed_type,
                0) ;
```

נסביר:

הארגומנט הראשון הוא מזהה התור (כפי שחזר מ- msgget()).

הארגומנט השני הוא המבנה לתוכו תוכנס ההודעה וטיפוסה.

הארגומנט השלישי מציין את גודל ההודעה המרבית שביכולתנו לקלוט.

הארגומנט הרביעי מציין את טיפוס ההודעות שברצוננו לקלוט (כל הודעה, או רק מטיפוס רצוי).

בארגומנט החמישי נקבע, בעזרת logic or (|) מידע נוסף על אופן קבלת ההודעה:

א. הדגל IPC_NOWAIT מורה, כמו בשליחה, לא להמתין להודעה. אם לא ממתינה הודעה בתור יש לחזור עם כשלון.

ב. הדגל MSG_EXCEPT מורה שאם טיפוס ההודעות שציינו > 0 אזי יש לקבל כל הודעה שאינה מהטיפוס שצוין.

ג. הדגל MSG_NOERROR מורה שאם בתור ממתינה הודעה גדולה מדי אזי אין להכשל, אלא יש לקצצה.

ד. אפס בארגומנט האחרון מציין שאף אחד מהדגלים הנ"ל לא מונף.

ערך ההחזרה:

בכישלון: 1-

בהצלחה: נפח ההודעה שנקראה (הנתונים בלבד).

אופן השימוש האופייני בתור הודעות הוא שהשולח מייצר סדרת הודעות, הקורא(ים) קוראים את ההודעות (כל אחד הודעה מטיפוס מסוים, או כל אחד כל הודעה שהיא) עד כישלון (status == 1-), ש- hopefully מעיד על כך שהיצרן גמר לייצר את כל מה שהיה לו לייצר, ומחק את התור. לחילופין, הקורא יכול לצאת מלולאת הקריאה אם חלף x זמן timer ולא התקבלה כל הודעה (כפי שמורה שהודלק על-ידו).

שחרור תור הודעות יעשה בד"כ ע"י מי שיצרו, באמצעות הפקודה:

```
if (msgctl(msgid, IPC_RMID, NULL) == -1)
{
  perror("msgctl() failed") ;
  exit(EXIT_FAILURE) ;
}
```


להזכירכם, תור שלא נמחק ממשיך להתקיים!

הפקודה: ipcs מציגה תורי הודעות, זיכרונות משותפים, וסמפורים הקיימים במערכת, כולל מזהה המשאב.

גם מה- shell ניתן למחוק תור שיצרתם באמצעות הפקודה: ipcrm –q <msgid>

לדוגמה:

```
<208|1>yoramb@inferno-05:~/os$ ipcs

------ Shared Memory Segments --------
key    shmid   owner   perms   bytes    nattch   status

------ Semaphore Arrays --------
key      semid    owner    perms    nsems

------ Message Queues --------
key        msqid    owner   perms   used-bytes   messages
0x79050002 0        yoramb   600     0            0

<209|1>yoramb@inferno-05:~/os$ ipcrm -Q 0x79050002
```

```
// file: message_queue_write.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_MSG_LEN     200

struct my_msgbuf {
   long mtype;
   char mtext[MAX_MSG_LEN];
};

int main(void)
{
   struct my_msgbuf my_msg;
   int msqid;
   key_t key;

   // get an id for the queue
   if ((key = ftok("/tmp", 'y')) == -1) {
      perror("ftok failed");
      exit(EXIT_FAILURE);
   }


   // create the queue, if it exists then fail, prmssins = 0600
   if ((msqid = msgget(key,
                       0600 | IPC_CREAT| IPC_EXCL)) == -1) {
      perror("msgget failed");
      exit(EXIT_FAILURE);
   }
   printf("key = %ld, msqid = %d\n", (long) key, msqid) ;
```

יכיל את ההודעות למשלוח

מזהה התור (הפנימי)

מפתח התור (החיצוני)


```
   puts("Enter lines of text, ^D to quit:");

   // repeatedly, read data and send it to the queue
   my_msg.mtype = 1 ;
   while(scanf(" %s", my_msg.mtext) != EOF) {
      if (msgsnd(msqid,
                 (struct msgbuf *) &my_msg,
                 strlen(my_msg.mtext)+1,
                 0) == -1) {
         perror("msgsnd failed");
         exit(EXIT_FAILURE);
      }
      my_msg.mtype++ ;
      if (my_msg.mtype > 3)
         my_msg.mtype = 1 ;
   }

   // remove the queue
   // at this point receiver fails to read from queue
   // (msgrcv fail) so it exits
   if (msgctl(msqid, IPC_RMID, NULL) == -1) {
      perror("msgctl failed");
      exit(EXIT_FAILURE);
   }

   return EXIT_SUCCESS;
}
```

השולח יחסם אם אין מקום

טיפוס ההודעה חיובי **ממש!**


```
// file: message_queue_read.c
#include <stdio.h>
#include <stdlib.h>              // for exit(), EXIT_SUCCESS
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_MSG_LEN     200

// define this instead of the standard msgbuf,
// as the latter does not compile
struct my_msgbuf {
   long mtype;
   char mtext[MAX_MSG_LEN];
};

int main()
{
   int queue_id  ;
   int i ;
   long int allowed_type ;
   key_t key;

   if ((key = ftok("/tmp", 'y')) == -1) {
      perror("ftok failed");
      exit(EXIT_FAILURE);
   }

   queue_id = msgget(key, 0) ;    // open the existing queue
   if (queue_id == -1) {
      perror("msgget failed in reader") ;
      exit(EXIT_FAILURE) ;
   }

   printf("Enter type of messages to get from queue: ") ;
   scanf(" %ld", &allowed_type) ;
```



```
   for (i = 0; i < 10; i++) {
      struct my_msgbuf my_msg ;
      int status ;

      status = msgrcv(queue_id,
                      (struct msgbuf *) &my_msg,
                       MAX_MSG_LEN,
                       allowed_type,
                       0) ;
      if (status == -1) {
         perror("msgrcv failed") ;
         exit(EXIT_FAILURE) ;
      }
      printf("Got %s from queue\n", my_msg.mtext) ;
   }

   return(EXIT_SUCCESS) ;
}
```

דוגמה שנייה, מעט עשירה יותר, לשימוש בתור הודעות:

א. תהליך א' מנהל את ממשק המשתמש, וקורא (מהמשתמש האיטי) שני סוגי בקשות: לבדוק ראשוניות של מספרים, לבדוק פאלינדרומיות של מחרוזות.

ב. בקשות לבדיקת ראשוניות הוא שולח לשרת בדיקת ראשוניות (שיוכל לעבוד במקביל להמתנה לקלט נוסף מהמשתמש).

ג. בקשות לבדיקת פאלינדרומיות הוא שולח לשרת בדיקת פאלינדרומיות.

לשם הפשטות נשתמש בתור הודעות יחיד שיפַתח ע"י מנהל ממשק המשתמש (לחילופין, כל שרת יכול היה לפתוח תור הודעות משלו).


```
// file:  message_queue_write2.c
// A program that writes two kinds of messages
// to a queue:
// messages of type == 1 ==>
// a request for checking if num is prime
//    "    "   "   == 2 ==>
// a    "    "    "    " a string is a plindrom
// The program:  message_queue_read2_prime.c
// reads messages of type 1 from
// the queue (there is no program that checks
// for palinromicity).

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_MSG_LEN     200

struct my_msgbuf {
    long mtype;
    char mtext[MAX_MSG_LEN];
};

//---------------------------------------------
int main(void)
{
    struct my_msgbuf my_msg;
    int msqid;
    key_t key;
```



```
    // get an id for the queue
    if ((key = ftok("/tmp", 'y')) == -1) {
        perror("ftok failed");
        exit(EXIT_FAILURE);
    }

    // create the queue, if it exists then fail, prmissns = 0600
    if ((msqid = msgget(key,
                    0600 | IPC_CREAT| IPC_EXCL)) == -1) {
        perror("msgget failed");
        exit(EXIT_FAILURE);
    }

    puts("Enter:\n\
            1 to check if num is prime\n\
            2 to check if string is a palindrom\n\
            0 to quit") ;
    // repeatedly, read data and send it to the queue
    scanf(" %ld", &my_msg.mtype) ;
    while(my_msg.mtype != 0) {
        scanf(" %s", my_msg.mtext) ;
        if (msgsnd(msqid,
                    (struct msgbuf *)&my_msg,
                     strlen(my_msg.mtext)+1,
                     0) == -1) {
            perror("msgsnd failed");
            exit(EXIT_FAILURE);
        }
        scanf(" %ld", &my_msg.mtype) ;
    }
    // remove the queue
    if (msgctl(msqid, IPC_RMID, NULL) == -1) {
        perror("msgctl failed");
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}
```



```
// file:  message_queue_read2_prime.c
// run with: message_queue_write2.c
// Read messages of type 1 from the queue,
// and check whether the number
// in the message is prime. Send the output to the screen.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_MSG_LEN     200
#define ALLOWED_TYPE_PRIME      1

void check_prime( int num) ;

struct my_msgbuf {
    long mtype;
    char mtext[MAX_MSG_LEN];
};

//-------------------------------------------------------
int main()
{
   int queue_id  ;
   key_t key;

    if ((key = ftok("/tmp", 'y')) == -1) {
        perror("ftok failed");
        exit(EXIT_FAILURE);
    }
```

```
  queue_id = msgget(key, 0) ;    // open the existing queue
  if (queue_id == -1) {
    perror("msgget failed in reader") ;
    exit(EXIT_FAILURE) ;
  }

  while (1) {
    struct my_msgbuf my_msg ;
    int status ;
    int num ;

    status = msgrcv(queue_id,
                    (struct msgbuf *) &my_msg,
                     MAX_MSG_LEN,
                    (long int) ALLOWED_TYPE_PRIME,
                    0) ;
    if (status == -1) {
       perror("msgrcv failed") ;
       exit(EXIT_FAILURE) ;
    }
    // convert the string to an int
    num = atoi(my_msg.mtext) ;
    check_prime(num) ;
  }

  return(EXIT_SUCCESS) ;
}
```



```
//-----------------------------------------------------
void check_prime( int num) {
  int div ;

  for (div = 2; div <= num/2; div++)
    if (num % div == 0) {
         printf("%d is not prime\n", num) ;
         return ;
    }
  printf("%d is prime\n", num) ;
}
```


4.9 זיכרון משותף Shared Memory

כלי ה- IPC שראינו עד כה היו בעקרון סדרתיים: אם נתון א' נכתב לפני נתון ב' על אמצעי התקשורת אזי נתון א' גם יקרא לפני ב'.

זיכרון משותף (ז"מ), כשמו כן הוא, מאפשר לתהליכים שחולקים אותו לפנות לאותו קטע-זיכרון (= מערך) בגישה ישירה: הם יכולים לק'\לכ' על כל תא רצוי.

יתרון נוסף של ז"מ הוא מהירותו: המידע אינו עובר דרך תווך ביניים כלשהו בין הכ' לק'.

מנגד, האתגר בז"מ הוא כיצד לגרום לתהליכים לסנכרן\לתאם את הפניה לז"מ כך שתהליך ב' לא יקרא נתונים טרם שתהליך א' *סיים* להכינם לו. *בד"כ* הסינכרון יעשה באמצעות סמפורים, או מנעולים אותם נכיר בסמ' הבא. (להזכירכם, במחשבים מודרניים אפילו פעולת השמה על הזיכרון אינה אטומית!)

למותר לציין שבאופן בסיסי, אחד המנגנונים שמ.ה. מעמידה לרשותנו הוא הפרדה בין מרחב הכתובות של כל תהליך ותהליך, כך ששני תהליכים לא יוכלו לפנות לאותו קטע זיכרון (וזאת ע"מ שתהליך ב' לא יפגע חלילה בנתונים של תהליך א').

על כן, כדי להשיג את שיתוף הזיכרון יש להשתמש באמצעים מיוחדים.

כפי שציינו, זיכרון משותף, תור הודעות וסמפור, נקראים XSI IPC, מקורם משותף (System V) ויש ביניהם קווי דמיון, בפרט ברעיון של מפתח חיצוני לעומת מזהה פנימי, באמצעותם פונים לַאובייקט.

על-כן גם בזיכרון משותף נתחיל ביצור מפתח, באמצעות ftok():

```
key_t key ;
if ((key = ftok("~yoramb/os", 'b')) == -1) {
   perror(…) ;
   exit(…) ;
}
```

כמו עם תור הודעות (שהוקצה באמצעות msgget()), גם בזיכרון משותף נשתמש בַמפתח כדי להקצות את האובייקט הדרוש, ויוחזר לנו מזהה האובייקט:

```
int shm_id ;
shm_id = shmget(key,
                SHM_SIZE,
                IPC_CREAT | IPC_EXCL | 0600) ;
if (shm_id == -1) {
  …
}
```

<u>נסביר</u>: הארגומנט הראשון והשלישי דומים לאלה שראינו בפקודת ה- msgget(),
הארגומנט השני מציין את גודל שטח הזיכרון הרצוי בבתים (בד"כ, בפועל, יוקצו דפים שלמים, חלק מהדף האחרון המוקצה לא יעמוד לרשותנו). שטח הזיכרון המוקצה מאופס.

תהליכים נוספים שירצו להשתמש בז"מ שכבר הוקצה יבצעו:

```
key = ftok(כמו שראינו) ;
int shm_id ;
shm_id = shmget(key,
                0,          // OR: SHM_SIZE,
                0600) ;
if (shm_id == -1) {
  …
}
```

כלומר בארגומנט השלישי הם אינם מבקשים להקצות שטח ז"מ, קל וחומר לא להיכשל אם הוא כבר מוקצה, שכן ברצונם להקשר לקטע ז"מ קיים בגודל נתון. בדוגמה מעל התהליכים מעוניינים הן לק'-מ והן לכ'-על קטע הזיכרון, ולכן ההרשאות המבוקשות הן 0600.

בזאת מ.ה. הקצתה בזיכרון המחשב שטח ז"מ, ונתנה הרשאת גישה אליו לתהליכים שבצעו shmget(), אולם שטח הזיכרון טרם שולב במרחב הכתובות של תהליכים אלה; כלומר אין כתובות (עליהן מצביע פוינטר כלשהו) בתהליכים השונים באמצעותן הם יפנו לשטח הזיכרון המשותף.

בשלב הבא נרצה לשלב את קטע הז"מ במרחב הכתובות של כל תהליך:


```
char *shm_ptr ;
shm_ptr = (char *) shmat(shm_id,
                         NULL,
                         0) ;
if (shm_ptr == (char *) -1) {
  perror("shmat failed") ;
  exit(EXIT_FAILURE) ;
}
```

<u>הסבר</u>:

א. NULL בארגומנט השני, מורה שניתן לשלב את קטע הז"מ בכל מקום פנוי במרחב הכתובות שלנו. זה הערך המקובל *מאוד* לארגומנט. (בד"כ הוא ישולב בין המחסנית לערמה.)

ב. אפס בארגומנט השלישי מורה שברצוננו, בעזרת המצביע *הנוכחי* הן לק'-מ והן לכ'-על הז"מ. (בעזרת מצביעים אחרים, ופקודות shmat() אחרות, אולי נרצה רק לק', ואז נעביר את הדגל SHM_RDONLY).

עתה, נוכל באמצעות shm_ptr לטפל בשטח הזיכרון כבכל מערך. לדוגמה:

```
char input_str[MAX_STR_LEN] ;

fgets(input_str, MAX_STR_LEN, stdin) ;
strcpy(shm_ptr, input_str) ;
```

או:

```
for (i=0; i< 10; i++)
  shm_ptr[i] = '?' ;
```

עת איננו מעוניינים לפנות עוד לשטח הזיכרון באמצעות המצביע נבצע:

```
shmdt(shm_ptr) ;
```

בזאת איננו משחררים את שטח הזיכרון, אלא רק מנתקים את המצביע המסוים ממנו.

כדי לשחרר את שטח הזיכרון יכול כל תהליך שיש לו הרשאת ק'+כ' על השטח (ולא רק מקצהו) להגדיר משתנה עזר:

```
struct shmid_ds shm_desc ;
```

ולבצע:

```
if (shmctl(shm_id,
            IPC_RMID, &shm_desc) == -1)
{
  perror("shmctl failed") ;
  exit(EXIT_FAILURE) ;
}
```

שטח הזיכרון ימוחזר אחרי שכל התהליכים שעושים בו שימוש יסתיימו.

כמו עם תור הודעות, גם קטע ז"מ שלא שוחרר (למשל משום שהתהליך שאמור היה לשחררו עף בטרם עת) ממשיך להתקיים (ולגזול משאבי מערכת). בפרט, אם תנסו להקצותו שוב (עת תריצו שוב את תכניתכם), ההקצאה תכשל.

ראינו את פקודת ה- Shell: ipcs המציגה את משאבי ה- XSI IPC המוקצים.

פקודת ה- Shell: ipcrm –m <shm id>
תשחרר את שטח הזיכרון.

נביט בתקלה אפשרית הקשורה לשימוש בז"מ.
לפני שנביט בתקלה ניזכֵר כי הפעולה v++ (המגדילה את תא הזיכרון v באחד) למעשה, בד"כ, הופכת לשלוש פקודות מכונה:

```
reg ← v
reg++
v ← reg
```

(וכפי שאמרנו, אפילו זאת אינה אטומית!)

עתה נניח כי אנו מריצים שני תהליכים, החולקים ז"מ בן שני בתים, אליו הם פונים כאל מערך a. כל אחד מהתהליכים סופר, ומוסיף לז"מ, כמה אפסים וכמה אחדים הוא קרא (לתאים #0, #1 במערך).

קוד התהליכים:

```
while ((c = getchar()) != EOF) {
  if (c == '0' || c == '1')
    a[c – '0']++ ;
}
```

עתה נניח את התסריט\התזמון הבא:
עד כה נקראו 17 אפסים, עתה כל אחד משני התהליכים קרא אפס נוסף, וברצונו על-כן להגדיל את התא #0 במערך; ומ.ה. מריצה אותם, כולל מבצעת החלפת הקשר, באופן הבא:

תהליך ב'	תהליך א'
	reg1 = a[0]
reg2 = a[0]	
	reg1++
reg2++	
	a[0] = reg1
a[0] = reg2	

התוצאה: ערכו של a[0] הוא 18.
ערך שגוי! (הערך צריך להיות 19.)

<u>הסיבה</u>: שני התהליכים לא תאמו\סינכרנו את הטיפול שלהם בתא הז"מ.

<u>הפתרון</u>: שימוש בסמפור או במנעולים כלליים, כפי שנכיר בסמ' הבא.
<u>או</u>: בנעילת קטע הז"מ באמצעות מנעול יעודי כפי שלינוקס וסולאריס מאפשרות לנו לעשות (באופן שחורג מהסטנדרט. בלינוקס רק החל מגרסה 2.6.10 כל משתמש יכול לבצע פעולה זאת, בעבר רק משתמש מיוחס יכול היה לבצעה).

בהקשר של הדוגמה שלנו נבצע:

```
while ((c = getchar()) != EOF) {
  if (c == '0' || c == '1')
  {
    if (shmctl(shm_id,
               SHM_LOCK,
               &shm_desc) == -1) {
        perror("shmctl(lock) failed");
        exit(EXIT_FAILURE) ;
    }
    a[ c - '0']++ ;
    if (shmctl(shm_id,
               SHM_UNLOCK,
               &shm_desc) == -1) {
        perror("shmctl(unlock) failed") ;
        exit(EXIT_FAILURE) ;
    }
  }
}
```

אנו מבקשים לנעול את הז"מ. עד שהמנעול לא יינתן לנו אנו **תקועים\מושהים**

כאן, רק אנו מטפלים בז"מ באופן **בלבדי**

שחרור הז"מ שנעלנו

נציג זוג תכניות המשתמשות בז"מ:

א. תכנית א' 'מייצרת' מחרוזות (אותן היא קוראת מ- stdin) ומכניסה אותן לז"מ. (ייתכן שמספר תהליכים המריצים תכנית זאת ירוצו במקביל)

ב. תכנית ב' 'צורכת' את המחרוזות שייצרה תכנית א'. (כנ"ל)

כדי לסנכרן את הגישה שלהן לז"מ הן משתמשות בתא 0# בקטע הזיכרון:

א. עת ערכו – (מינוס) אין מחרוזת בז"מ.

ב. עת ערכו + (פלוס) יש מחרוזת בזיכרון.

(בתכנית יש ליקויים מבחינת סנכרון הגישה לזיכרון המשותף, נושא עליו נדון בפרק 7#, אך היא תקינה מהבחינה הטכנית ולכן תספק את צרכינו הנוכחיים.)


```
// file: shm_create_n_produce.c
/*  A program that allocates a block of shared memory,
 * then repeatedly 'produces' strings (it reads form stdin)
 * to the shm.
 * A second program: a 'consumer' consumes
 * these strings from the shm.
 * (shm_consumer.c)
 *
 * The programs do not utilize the random access
 * property of a shm
 */
#include <stdio.h>
#include <stdlib.h>     // for exit()
#include <string.h>     // for strcpy(), strcmp()
#include <unistd.h>     // for sleep()
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define MAX_STR_LEN     100
#define SHM_SIZE        MAX_STR_LEN + 1
// shm_ptr[0] holds whether the shm
// is empty (-) on full (+)
#define END_STRING      "END"

//-----------------------------------------------------------------------------
int main() {
  key_t key ;
  int shm_id ;

  char *shm_ptr ;

  char input_string[MAX_STR_LEN] ;

  struct shmid_ds shm_desc ;
```

המפתח החיצוני והפנימי לז"מ

המצביע בעזרתו נפנה לז"מ

מבנה לצורך שחרור הז"מ


```
// create a key for the shm
key = ftok("/tmp", 'y') ;
if (key == -1) {
    perror("ftok failed: ") ;
    exit(EXIT_FAILURE) ;
}
if ((shm_id = shmget(key,
                SHM_SIZE,
                IPC_CREAT  | IPC_EXCL  | 0600)) == -1) {
        perror("shmget failed: ") ;
        exit(EXIT_FAILURE) ;
}

shm_ptr = (char *) shmat(shm_id, NULL, 0) ;
if (!shm_ptr) {
        perror("shmat failed: ") ;
        exit(EXIT_FAILURE) ;
}
shm_ptr[0] = '-' ;          // sign shm is empty

puts("Now, (and only now!) reader can start") ;

printf("Enter a series of strings to be written\
       on the shm.\n\
       Enter %s to finish\n", END_STRING) ;
```

מרכיב זה נוריד אם כמה תהליכים עשויים לנסות להקצות את הז"מ

```
  for( ; ; ) {
     scanf(" %s", input_string) ;

     while (shm_ptr[0] == '+')    // while shm is not 'empty'
          sleep(1) ;

     if (shmctl(shm_id, SHM_LOCK, &shm_desc) == -1) {
               perror("shmctl LOCK failed: ") ;
               exit(EXIT_FAILURE) ;
     }

     strcpy(shm_ptr +1, input_string) ;
     shm_ptr[0] = '+' ;                    // the shm is not empty

     if (shmctl(shm_id, SHM_UNLOCK, &shm_desc) == -1
     {
               perror("shmctl UNLOCK failed: ") ;
               exit(EXIT_FAILURE) ;
     }

     if (strcmp(input_string, END_STRING) == 0) {
          if (shmctl(shm_id, IPC_RMID, &shm_desc) == -1)
          {
               perror("shmctl IPC_RMID failed: ") ;
               exit(EXIT_FAILURE) ;
          }
          return( EXIT_SUCCESS ) ;
     }

   }
   return( EXIT_SUCCESS ) ;
}
```

המתנה עסוקה


```
// file:  shm_consumer.c
// See documentation in: shm_create_n_produce.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define MAX_STR_LEN     100
#define SHM_SIZE        MAX_STR_LEN +1
#define END_STRING      "END"
//---------------------------------------------
int main() {
   key_t key ;
   int shm_id ;
   char *shm_ptr ;
   char output_string[MAX_STR_LEN] ;
   struct shmid_ds shm_desc ;

   // create THE SAME  key for the shm as the producer
   key = ftok("/tmp", 'y') ;
   if (key == -1) {
        perror("ftok failed: ") ;
        exit(EXIT_FAILURE) ;
   }

   // get the id of the block of memory
   // that was, hopefully, already created by the producer
   if ((shm_id = shmget(key,
                    SHM_SIZE,   // OR: 0
                    0600)) == -1) {
              perror("shmget failed: ") ;
              exit(EXIT_FAILURE) ;
   }
```



```
   shm_ptr = (char *) shmat(shm_id, NULL, 0) ;
   if (!shm_ptr) {
       perror("shmat failed: ") ;
       exit(EXIT_FAILURE) ;
   }

   for( ; ; ) {

     while (shm_ptr[0] == '-')
          sleep(1) ;

     // lock the shm before you operate on it
     if (shmctl(shm_id, SHM_LOCK, &shm_desc) == -1) {
         perror("shmctl LOCK failed: ") ;
         exit(EXIT_FAILURE) ;
     }

     strcpy(output_string, shm_ptr +1) ;
     shm_ptr[0] = '-' ;                    // the shm is empty

     if (shmctl(shm_id, SHM_UNLOCK, &shm_desc) == -1)
     {
         perror("shmctl UNLOCK failed: ") ;
         exit(EXIT_FAILURE) ;
     }
     printf("Got: %s from the shm\n", output_string) ;

     if (strcmp(output_string, END_STRING) == 0) {
       if (shmctl(shm_id, IPC_RMID, &shm_desc) == -1) {
               perror("shmctl IPC_RMID failed: ") ;
               exit(EXIT_FAILURE) ;
       }
       return( EXIT_SUCCESS ) ;
     }
   }
   return( EXIT_SUCCESS ) ;
}
```


<u>מספר הערות ביחס לז"מ</u>

א. בניגוד לזיכרון (פרטי) שהוקצה דינאמית ע"י malloc() וגודלו ניתן לשינוי ע"י realloc(), גודלו של שטח ז"מ שהוקצה אינו ניתן לשינוי.

ב. ניתן להגדיר:

```
struct S { … }
struct S *ptr ;
ptr = (struct S *) shmatt(כרגיל) ;
…
```

וכך לטפל במערך של מבנים:

```
ptr[0]. … =
```

ג. במידה ואנו מחזיקים מבנה נתונים כלשהו בז"מ יש להקפיד להחזיק את כל המידע אודותיו בז"מ (לדוגמה: כמה תאים כולל המערך, וכמה נמצאים כרגע בשימוש).

ד. מספר שלם בן ארבעה בתים יש לשמור בכתובת שהינה כפולה של ארבע. בד"כ הקומפיילר דואג לכך עבורנו. עת אנו מקצים ז"מ, ורוצים לאחסן בו נתונים (בפרט מבנים הכוללים חברים מטיפוסים שונים), חובת הדאגה לכך עוברת אלינו (אם לא נקפיד על כך ישלח לתהליך הסיגנל SIGBUS).

4.10 ממשק התושבת The Socket Interface

כלי ה- IPC שראינו עד כה אפשרו לנו להעביר הודעות בין תהליכים השוכנים באותה מערכת. התושבת הינה ממשק לפרוטוקולי תקשורת המאפשרים לנו להעביר מידע גם בין תהליכים המצויים במכונות שונות.

מהי כתובת IP?
ומהי נקודת-עגינה (port)?

4.10.1 רקע

בד"כ אנו דנים במודל בו קיים שרת (server), המורץ 'לנצח' והמספק מידע ללקוחות (clients) הפונים אליו עת הם זקוקים לנתונים.

נהוג להבחין בין שלושה סוגים עיקריים של תושבות:

א. תושבת זרם (stream socket): מאפשרת העברה רצופה של נתונים לאורך זמן. משתמשת בפרוטוקול TCP המבטיח שמירה על סדר ועל תקינות (ומסתמך על פרוטוקול IP הדואג לניתוב המידע). מחייבת יצירת קשר טרם העברת מידע. דומה לשיחת טלפון. המידע עובר כזרם של בתים. חלוקתו לחבילות שקופה למתקשרים.

ב. תושבת דטה-גרם (datagram socket): תשמש להעברת חבילה בודדת (לא מבטיחה שהחבילה תגיע, ולא סדר הגעה, אך אם החבילה תגיע היא תהיה תקינה). אינה מחייבת יצירת קשר טרם שליחת מידע. דומה לשליחת הודעת דואר.

ג. תושבת גולמית (raw socket) לשם פניה לפרוטוקולים נמוכים.

אנו נתמקד בתושבת זרם.

ממשק התושבת הוא חלק מסטנדרט POSIX ומקורו ב- BSD (בראשית שנות ה- 80').

4.10.2 תפעול תושבת על-ידי תהליך לקוח

נתחיל בדיון בשימוש בתושבת כלקוח.
כדי להכין תושבת לפעולה *כלקוח* עלינו לבצע את הפעולות הבאות:

א. להקצות מתאר קובץ, כלומר כניסה במערך המתארים של התהליך.

ב. לציין את כתובת ה- IP של המחשב עימו ברצוננו להחליף נתונים (השרת), ואת ה- port באותו מחשב דרכו מִתַקשר התהליך עימו ברצוננו לתקשר.

ג. לציין את כתובת ה- IP וה- port דרכם התהליך שלנו פונה לרשת (זאת, בלקוח, תוכל המערכת לעשות עבורנו, ולכן על צעד זה נוכל לדלג).

ד. לצור את הקשר עם השרת.

אחרי ביצוע ההכנות הללו נוכל, כאמור, לק' ולכ' מ\על התושבת כמן מכל קובץ רגיל.

נראה עתה את הפרטים הטכניים:

כפי שנראה, אחרי יצירת הקשר, התהליכים מחליפים מידע ע"י ק'-מ וכ'-על התושבת, כמו על כל קובץ במערכת הקבצים, תוך שימוש במתאר קובץ ובפעולות ()write ,()read, ועל-כן יכולים לעשות שימוש בכל ק.מ. האחרות המיועדות לטיפול בקבצים. הנתונים, כמובן, מגיעים אליהם ומהם לא מ\אל קובץ במערכת הקבצים, אלא מ\\אל הרשת. (במובן זה התושבת, כמו הצינור, ובניגוד ל- IPC XSI עושה שימוש בק.מ. הקיימות לטיפול בקבצים.)

כדי לקבל מתאר קובץ המאפשר שימוש בתושבת נשתמש בק.מ. (המקבילה ל- open):

```
int socket(int domain,  int type, int protocol);
```

לדוגמה:

```
int my_socket ;
my_socket = socket(AF_INET,
                   SOCK_STREAM,
                   0);
```

ק.מ. תחזיר מתאר קובץ, או 1- בכישלון.
הארגומנטים:
address family internet = AF_INET
(ערך שימושי חלופי: AF_UNIX לקשר בין תהליכים באותה מכונה)
SOCK_STREAM = סוג התושבת (כמתואר מעל)
0 = בחר את הפרוטוקול בעצמך, על-פי הארגומנט השני.

בכך קיבלנו תושבת (מתאר קובץ), אולם טרם קשרנו אותה לא לכתובת מקומית, לא לכתובת רחוקה (שתי כתובות ביניהן ברצוננו להעביר נתונים בעזרת הרשת), ובטח ובטח שטרם יצרנו קשר עם השרת.

את הקישור לכתובת המקומית בלקוח אפשר להניח למערכת לעשות בעצמה עת אנו יוצרים את הקשר עם השרת, ולכן על צעד זה נדלג.

אולם את כתובת השרת עלינו להזין.
טיפוס המבנה אותו יש לספק לפונ' יצירת הקשר באמצעות תושבת עם השרת הוא: struct sockaddr
(למעשה יש לספק מצביע למבנה מטיפוס זה).

לצרכינו יתאים יותר המבנה: struct sockaddr_in (in = internet) השווה לראשון בשטחו, וניתן להמרה ע"י cast מתאים, לטיפוס struct sockaddr

מרכיבי\חברי המבנה : struct sockaddr_in והטיפול בהם:

<u>א. משפחת הכתובות</u>

```
dest_addr.sin_family = AF_INET;
```

AF_INET מציין משפחת כתובות רשת.

<u>ב. כתובת השרת</u>

```
dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);
```

הפונ': inet_addr() מקבלת מחרוזת המתארת כתובת IP (לדוגמה: "10.2.10.24") ומחזירה אותה כמספר בן ארבעה בתים (בפורמט הרשת).
בדוגמה, הנחנו כי בראש התכנית קבענו:

```
#define    DEST_IP    "10.2.10.24"
```


<u>ג. פורט השרת</u>

```
dest_addr.sin_port = htons(DEST_PORT);
```

כתובת ה- IP, ומספר הפורט צריכים לעבור ברשת. על כן יש לדאוג שהם יהיו ב- Network Byte Order = בית יותר משמעותי קודם. כדי לדאוג שנתון אכן נמצא בפורמט זה עומדות לרשותנו הפונ': htons(), htonl() המתרגמות short, long (2, 4 בתים) מפורמט host ל- net.
(עמיתותיהן: ntohs(), ntohl() עושות את התרגום ההפוך)

בדוגמה, הנחנו כי בראש התכנית קבענו:

```
#define    DEST_PORT    3879
```

<u>ד. איפוס יתר המבנה</u>

```
memset(dest_addr.sin_zero, '\0',
       sizeof dest_addr.sin_zero);
```

הפונ': memset(), מקבלת מצביע (void *), ערך תו (בית) רצוי, וגודל שטח זיכרון, ו-'מורחת' את הערך על גוש הזיכרון (#include <string.h>).

בכך השלמנו את הכנת פרטי השרת במבנה dest_addr . מבנה זה, יחד עם התושבת שהקצאנו קודם, נעביר לפונ' יצירת הקשר עם השרת (והיא תקבע בעצמה את כתובת ה- IP ומספר הפורט שלנו):


```
if (connect(my_socket,
            (struct sockaddr *) &dest_addr,
            sizeof dest_addr) == -1)
...
```

פונ' יצירת הקשר הינה connect() והיא מקבלת שלושה פרמטרים:
א. תושבת.
ב. מצביע למבנה המכיל את פרטי השרת.
ג. גודלו בבתים של המבנה מסעיף ב'.
הפונ' מחזירה אפס בהצלחה, ו- 1- בכישלון.

בזאת השלמנו את תהליך יצירת הקשר עם השרת, ואנו יכולים לכתוב אליו ולקרוא ממנו נתונים באמצעות התושבת:

```
int read(int socket,
         char *buffer,
         int buflen) ;
```

תחזיר:
0 = הקשר נותק ע"י צד רחוק.
1- = שגיאה.
n = מספר הבתים שנקראו בפועל (עשוי להיות < buflen אם היו פחות בתים זמינים. אם אין כלל בתים התהליך ייחסם).

באופן סימטרי:

```
int write(int socket,
          char *buffer,
          int buflen) ;
```

מחזירה ערך דומה ל- read().

לבסוף יש לסגור התושבת:

```
int close(int socket) ;
```

נראה דוגמה של תכנית שלמה:


```
// file: socket_read_client_new.c
// run: a.out 10.2.10.25

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>          // for read/write/close
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>     // htonl(),…
#include <sys/socket.h>
#include <netdb.h>

#define  BUFLEN        1024
#define  DEST_PORT          3879

int main(int argc, char *argv[])
{
   int              rc;                    // ret code of s.c.
   int              my_socket;
   char             buf[BUFLEN+1];
   char*            pc;                    // pointer to buf
   struct sockaddr_in   dest_addr;

   if (argc < 2) {
      fprintf(stderr, "Missing host address\n");
      exit(EXIT_FAILURE) ;
   }

   my_socket = socket(AF_INET, SOCK_STREAM, 0);
   if (my_socket < 0) {
      perror("socket: allocation failed");
      exit(EXIT_FAILURE) ;
   }
```



```
   dest_addr.sin_family = AF_INET;
   dest_addr.sin_port = htons(DEST_PORT);
   dest_addr.sin_addr.s_addr = inet_addr(argv[1]);
   memset(dest_addr.sin_zero,
            '\0',
            sizeof dest_addr.sin_zero);

   rc = connect(my_socket,
                (struct sockaddr *)&dest_addr,
                (socklen_t) sizeof(dest_addr));
   if (rc) {
      perror("connect");
      exit(EXIT_FAILURE) ;
   }

   pc = buf;
   while ((rc = read(my_socket, pc, BUFLEN - (pc-buf)))) {
      pc += rc;
   }

   close(my_socket);

   *pc = '\0';
   printf("got from server: %s\n", buf);

   return EXIT_SUCCESS;
}
```


4.10.3 שרת ללקוח יחיד Single-Client Server

עתה נציג את השרת ממנו הלקוח קרא את ההודעה.

כזכור, השרת מאזין על פורט 3879,
הוא מקבל בקשות התחברות,
עבור כל בקשה הוא שולח את המחרוזת Hello World,
וסוגר את הקשר.

זאת עושה השרת בלולאה אינסופית.

גם השרת, כמו הלקוח, נזקק כמובן לתושבת:

```
int my_socket ;
my_socket=
   socket(PF_INET, SOCK_STREAM, 0);

if (my_socket < 0) …
```

עתה על השרת לעדכן את התושבת בכתובתו (מספר פורט + כתובת IP). בלקוח דילגנו על שלב זה, אולם כאן הוא הכרחי:

שוב נשתמש במבנה המיועד להחזקת כתובת:

```
struct sockaddr_in my_addr;
```

ונזין לו ערכים בדומה למה שעשינו בלקוח (שם, כמו כאן, אנו מזינים את כתובת השרת):

```
my_addr.sin_family = AF_INET;
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

הקבוע INADDR_ANY מוגדר בקובצי ה-include ומורה כי המערכת יכולה לקשר את השרת לרשת דרך כל אחת מהכתובות באמצעותן מחשב זה מחובר לרשת (ותתכנה כמה כתובות)

```
my_addr.sin_port = htons(MY_PORT);
```

MY_PORT צריך להיות מוגדר כקבוע על-ידנו, בדוגמה שלנו ערכו 3879.

```
memset(my_addr.sin_zero, '\0',
        sizeof my_addr.sin_zero);
```

בזאת בצענו בשרת אותן ארבע פקודות המכינות את המבנה כמו בלקוח.

עתה השרת אינו פונה דרך הרשת למחשב מרוחק, אלא הוא מוכן להיכנס להאזנה לפניות אליו, לשם כך, ראשית, עליו לקשור את כתובתו (כפי ששמורה ב- my_addr) לתושבת שהוקצתה (my_socket).
נעשה זאת ע"י:

```
if (bind(my_socket,
        (struct sockaddr *)&my_addr,
        sizeof my_addr) == -1) ...
```

בשלב זה השרת מוכן להיכנס להאזנה לפניות אליו.
הפקודה:

```
int rc ;
rc = listen(my_socket, 5) ;
```

מגדירה שהשרת מוכן לאפשר ללכל היותר חמש פניות להצטבר בתור אליו.
(כל פניה מעבר לחמש הממתינות תושב ריקם עם ההודעה: connection refused.)

כל פניה שמתקבלת תועבר לתושבת חדשה, לשם טיפולה. על כן נגדיר:

```
int my_socket2 ;
struct sockaddr_in her_addr ;
int size_her_addr= sizeof(her_addr);
```

וניכנס ללולאה:

```
while (1) {
  my_socket2 =  accept( my_socket,
                        (struct sockaddr *) &her_addr,
                        &size_her_addr) ;
```

בזאת אנו ממתינים ומקבלים פניות. (עד הגעת פניה נשלח לשון).
כל פניה תועבר לתושבת חדשה: my_socket2
לתוך her_addr תוכנס כתובת הפונה (לשם מעקב. אפשר להעביר כאן NULL.)
לתוך size_her_addr יוכנס מספר הבתים שבשימוש מתוך המבנה. כנ"ל מבחינת NULL

המשך גוף הלולאה:

```
  if (my_socket2 < 0)
    continue ;
```

אם יש בעיה בקשר נפנה לסיבוב נוסף בלולאה, ואחרת, כאמור, השרת שולח ללקוח את המחרוזת Hello World:

```
  if (write(my_socket2,
            SENT_STR,
            strlen(SENT_STR) +1) !=
                        strlen(SENT_STR) +1) …
```

ייתכן שלא כל מה שרצינו לכתוב אכן נכתב (בפרט אם רצינו לכתוב יותר מ- 1K).

בתום 'הטיפול בפניה' נסגור את התושבת שהוקצתה לה:

```
  close(my_socket2) ;
}  // end of while(1)
```

תכנית שלמה של שרת ללקוח יחיד:

```
// file: socket_write_server_new.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>        // for memset
#include <unistd.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>


#define  PORT   3879
#define  LINE   "hello world"

int main()
{
   int            rc;                    // return code
   int            main_socket;           //  עליה מתקבלות פניות
                                         // חדשות
   int            serving_socket;        // עליה משורתות פניות
                                         // קיימות\ישנות
   struct sockaddr_in  my_address;       // כתובת השרת
                                         // עבור bind()
   struct sockaddr_in  cliet_address;    // כתובת הלקוח עבור
                                         // accept()
   socklen_t  size_cliet_address;

   main_socket = socket(AF_INET, SOCK_STREAM, 0);
   if (main_socket < 0) {
      perror("socket: allocation failed");
      exit(EXIT_FAILURE) ;
   }
```



```
   memset(&my_address, 0, sizeof(my_address));
   my_address.sin_family = AF_INET;
   my_address.sin_addr.s_addr = INADDR_ANY;
   my_address.sin_port = htons(PORT);

   rc = bind(main_socket,
             (struct sockaddr *)&my_address,
             sizeof(my_address));
   if (rc) {
      perror("bind failed");
      exit(EXIT_FAILURE) ;
   }

   rc = listen(main_socket, 5);
   if (rc) {
      perror("listen failed");
      exit(EXIT_FAILURE) ;
   }
```



```
   size_cliet_address = sizeof(cliet_address);

   while (1) {
      struct sockaddr_in her_addr ;
      socklen_t addrlen = sizeof(her_addr) ;

                        קבל פניה חדשה, והפנה אותה לתושבת נפרדת
      serving_socket = accept(main_socket,
                        (struct sockaddr *)&cliet_address,
                        &size_cliet_address);
      if (serving_socket < 0)
         continue;
                        הכנס ל- her_addr את כתובת הלקוחה החדשה
      if (getpeername(serving_socket,
                 (struct sockaddr *) &her_ad
                 &addrlen) < 0) {
         perror("getpeername failed") ;
         exit(EXIT_FAILURE) ;
      }
      printf("Got request from %s\n",
             inet_ntoa(her_addr.sin_addr)) ;

      write(serving_socket, LINE, sizeof(LINE));
      close(serving_socket);
   }
   return(EXIT_SUCCESS) ;
}
```

הערה על פונ' אלה בשקף הבא. בינתיים נוכל להתעלם מהן.

ראוי מאוד, מאוד לבדוק כמה בתים נכתבו בפועל, ולהריץ לולאה שתדאג לכתוב הכל.

<u>מספר הערות</u>

א. במקום להשתמש ב- read/write ניתן להשתמש ב- send(), recv(). הן דומות מאוד (יש להן עוד ארגומנט, בו נהוג להעביר אפס. ניתן להעביר דגל שיבקש לא להיחסם).

ב. אם ברצונכם רק לקרוא או רק לכתוב על תושבת ניתן לבצע את הפקודה:
shutdown(my_socket, SHUT_RD) ;
ואז לא ניתן יהיה לקרוא עוד מהתושבת.
shutdown(my_socket, SHUT_WR) ;
ואז לא ניתן יהיה לכתוב עוד על התושבת.
ק.מ. מחזירות 1- בכישלון (אפס בהצלחה).

ג. לידיעתכם, מספרי פורט <= 1024 אינם מיועדים לבני תמותה. אם ברצונכם לבחור מספר פורט בעצמכם, בחרו ערך > 1024 וקטן מ: 65535.

ד. בלקוח ראינו את הפונ' inet_addr() המקבלת מחרוזת ("10.2.10.24") ומחזירה אותה ככתובת IP (מספר בן ארבעה בתים). בשרת ראינו את הפונ' ההפוכה לה המקבלת כתובת IP ומחזירה אותה כמחרוזת והינה: inet_ntoa() (מופיעה מייד בשקף הבא:)

ה. ייתכן שהשרת מעוניין לדעת מי הם הלקוחות שפונים אליו. הפונ' שתעמוד לרשותנו היא: getpeername() (שראינו קודם במסגרת השחורה).

נניח שהגדרנו (בשרת):

```
struct sockaddr_in   her_addr ;
socklen_t   addrlen = sizeof(her_addr) ;
int   serving_socket ;
```

ושהשרת ביצע:

```
serving_socket = accept(main_socket,
                        (struct sockaddr *)& her_addr,
                        & addrlen);
if (serving _socket < 0)
        continue;
```

עתה נוכל לכתוב:

```
if (getpeername(serving _socket,
                (struct sockaddr *) &her_addr,
                &addrlen) < 0) {
        perror("getpeername failed") ;
        exit(EXIT_FAILURE) ;
}
printf("Got request from %s\n",
                inet_ntoa(her_addr.sin_addr)) ;
```


4.10.4 המרת שם שרת בכתובתו

בדוגמה שראינו הלקוח פנה לשרת תוך ציון כתובת ה- IP שלו (כמחרוזת "10.2.10.24" שתורגמה לערך מספרי ע"י inet_addr()).
לעתים אנו יודעים רק את שמו של השרת (www.yes.no) אך לא את כתובתו. במקרה כזה נשתמש בפונ' gethostbyname() אשר מקבלת שם שרת ומחזירה את כתובתו (אותה נכניס למשתנה שמכיל את כתובת השרת).

כלומר במקום לכתוב:

```
dest_addr.sin_addr.s_addr =
        inet_addr(DEST_IP);
```

(עבור DEST_IP שמכיל את כתובת השרת כמחרוזת), כפי שכתבנו בעבר,
נכתוב עתה:

```
struct hostent *server_addr ;

if ((server_addr = gethostbyname(argv[1]))
                        == NULL) {
  perror("gethostbyname() failed") ;
  exit(EXIT_FAILURE) ;
}
dest_addr.sin_addr =
        *((struct in_addr *) server_addr->h_addr) ;
```

ומכאן נמשיך ל: connect() כמו קודם.
הרצת הלקוח תהיה:

a.out inferno-04.cs.hadassah-col.ac.il

4.10.5 שרת למספר לקוחות Multi-Clients Server

שרת למספר לקוחות עשוי להיות מתוכנן בשני אופנים חלופיים:

א. תהליך אב (master) ימתין לפניות. עבור כל פניה הוא ייצר תהליך יעודי לטיפול באותה פניה.
ב. תהליך יחיד ייתן מענה לכל הפניות

בגישה א' יש משהו אלגנטי, 'נקי' יותר, מנגד היא עלולה 'לאכול' הרבה משאבי מערכת (עת ייוצרו תהליכים רבים). גישה ב' הפוכה מבחינת יתרונות וחסרונות.

נדגים את גישה ב'.

הכלי המרכזי אותו עלינו להכיר לשם מימוש השרת הוא הפונ' select() המשמשת אותנו גם עם מתארי קבצים 'רגילים' (שאינם תושבות). (ולא נבלבל אותה עם ה- accept() שראינו בסעיף הקודם.)

נניח שברצוננו להיות מסוגלים להמתין לקלט מ- stdin, לפניה מלקוח חדש (על תושבת כדוגמת main_socket בדוגמה הקודמת), או לפניה מאחד מלקוחותינו הקיימים (על תושבת כדוגמת serving_socket מהדוגמה הקודמת).

אם נבצע read() או scanf() מ- stdin, או לחילופין מ- serving_socket, ולא ממתינים לנו נתונים נתקע\נושהה, כך גם אם נבצע accept() ולא ממתינה לנו פניה חדשה.

אם אנו נתקעים איננו יכולים לתת מענה לפניה שממתינה לנו על מתאר קובץ אחר.

<u>הערה:</u>
עת אנו פותחים קובץ, אנו יכולים לקבוע, (בעזרת fcntl() שנכיר בעתיד הרחוק), שעת אנו פונים למתאר, ולא ממתינים עליו נתונים, לא נושהה, אלא הפעולה תחזור מייד עם כשלון (1- והדלקת errno).

לכאורה זה פתרון: אולם הוא לא מוצלח שכן נצטרך בלולאה לבדוק את המתארים השונים, כך אנו עושים busy wait ואוכלים זמן מעבד לשווא.

על כן נרצה פתרון אחר:
נרצה להיות מסוגלים להגדיר קבוצה של מתארי קבצים (לדוגמה המתארים: 0#, 3#, 4#), ולקבוע שברצוננו להמתין לכך שאחד מתוך המתארים בקבוצה מוכן לקריאה (או כתיבה). עד *שאחד* מהמתארים בקבוצה יהיה מוכן (יחכו עליו נתונים) נלך לשון. עת נוּער (הפונ' תחזור) יהיה עלינו לבדוק איזה מתאר(ים) (מתוך אלה שקבענו) הם שגרמו לנו להתעורר, ומהם נקרא. אך אנו מובטחים שאיננו בודקים זאת לחינם.

כאמור, הפונ' (בעלת השם הלא מוצלח, לטעמי): select() היא שתעמוד לרשותנו.

פרוטוטיפ הפונ':

```
int select(int numfds,
           fd_set *rfd,
           fd_set *wfd,
           fd_set *efd,
           struct timeval *timeout) ;
```

ערך ההחזרה:
1- == חלה תקלה
אחרת == מספר מתארי הקבצים שמצריכים טיפול (ערך >= 0)

הארגומנטים לפונ':
א. numfds מכיל את ***מספר*** המתארים שיש לבדוק (במערך המתארים).
אם המשתנה int an_fd מכיל את מתאר הקובץ האחרון שפתחתם עד כה (וטרם סגרתם מתאר כלשהו), אזי an_fd+1 יהיה ערך מוצלח לארגומנט.
לחילופין: getdtablesize() או getrlimit(RLIMIT_NOFILE) מחזירות את גודלה של טבלת המתארים.

ב. fd_set *rfd מכיל את קבוצת המתארים מהם אנו ממתינים לנתונים (לקריאה).
ג. fd_set *wfd מכיל את קבוצת המתארים עליהם ברצוננו לכתוב (ואנו ממתינים שהדבר יתאפשר).
ד. fd_set *efd מכיל את קבוצת המתארים שעל חריגה בהם אנו רוצים להיות מדווחים.
ה. הארגומנט האחרון מורה כמה זמן ברצוננו להמתין עד שנתייאש, ואז הפונ' תחזור עם ערך אפס. ערך NULL בארגומנט מורה כי אין להתייאש כלל (לחילופין: ניתן להזין למבנה מספר של שניות ו\או של מיקרו שניות שברצוננו להמתין).

לתוך כל ארגומנט נכניס, טרם הקריאה לפונ', את המתארים שברצוננו להתעניין בהם (לדוגמה: 0#, 3#, 4#), בתום הקריאה לפונ' יחזרו בארגומנט המתארים עליהם ניתן לפעול (לדוגמה: 0#, 4#. על כן ייתכן שלפני הקריאה לפונ' נרצה לשמור את ערכו של הארגומנט, כדי להשתמש בו שוב בעתיד).

כיצד נכניס מתארים לקבוצה כלשהי (טרם הקריאה לפונ'), וכיצד נבדוק האם מתאר כלשהו נמצא בקבוצה (כפי שעודכנה ע"י הפונ'), ועל כן במתאר זה ברצוננו לטפל?

בעזרת מספר מאקרוים:
א. FD_ZERO(fd_set *fds) ירוקן את קבוצת המתארים.
ב. FD_SET(fd, fd_set *fds) יוסיף את המתאר fd לקבוצת המתארים fds.
ג. FD_CLEAR(fd, fd_set *fds) יסיר את המתאר fd מקבוצת המתארים fds.
ד. FD_ISSET(fd, fd_set *fds) יבדוק האם המתאר fd נכלל בקבוצת המתארים fds.

נפנה אם כן לתכנית השרת למספר לקוחות:

int main_socket = socket(…) ;

כפי שראינו בעבר. עליו נקבל פניות **חדשות**

struct sockaddr_in my_addr ;

יתופעל כפי שראינו בעבר:

```
my_addr.sin_family = … ;
my_addr.sin_addr.s_addr = … ;
my_addr.sin_port = … ;
memset( … ) ;
if (bind(main_socket,(…) &my_addr, …))
   error …

if (listen(main_socket, QUEUE_SIZE))
   error …
```

(אנו מוכנים לקבל QUEUE_SIZE פניות, כל אחת, כמו קודם, תופָנה לתושבת חדשה.
בניגוד לדוגמה הקודמת בה הפניות טופלו *בזו אחר זו*, עתה הן עשויות להיות מטופלות במקביל)

ועתה לחידושים שבדוגמה זאת:

```
fd_set rfd,
        copy_rfd ;
```

כאתחול, נכניס לקבוצה rfd את main_socket בלבד (בהמשך לקבוצה יתווספו ויגרעו תושבות) :

```
FD_ZERO(rfd) ;
FD_SET(main_socket, &rfd) ;
size_of_rfd_set = 1 ;
```

וניכנס ללולאה המרכזית של השרת:


```
while (1) {
   c_rfd = rfd;

   rc = select(main_socket + size_of_rfd_set,
               &c_rfd,
               NULL, NULL, (struct timeval *) NULL);

   if (FD_ISSET(main_socket, &c_rfd)) {
      serving_socket = accept(main_socket,
                      (struct sockaddr *)&her_addr,
                      (socklen_t*) &size_her_addr);
      if (serving_socket >= 0) {
         FD_SET(serving_socket, &rfd);
         size_of_rfd_set++ ;
      }
   }

   for (fd= main_socket +1;
        fd < main_socket + size_of_rfd_set;
        fd++)
      if (FD_ISSET(fd, &c_rfd)) {
         rc = read(fd, buf, BUFLEN);
         if (rc == 0) {
            close(fd);
            FD_CLR(fd, &rfd);
            size_of_rfd_set --- ;
         }
         else if (rc > 0)
            write(fd, buf, rc);
            // check that write went okay...
         else {
            perror("read() failed") ;
            exit(EXIT_FAILURE) ;
         }
      }
}
return(EXIT_SUCCESS) ;
```

בדוק מערך המתארים עד התא: main_socket + size_of_rfd_set, החזר את תת-הקבוצה של c_rfd עליה ממתינים נתונים לק'.

כאן מוחזרת תת-קבוצה של מה שהועבר

אם ה- main_socket בקבוצה==> לקוח חדש

העבירו לתושבת חדשה, הוספה לקבוצת המתארים מהם מצפים לנתונים, והגדל גודל הקבוצה

עבוֹר על יתר המתארים הפתוחים החל ב- main_socket+1 ואילך

עבוֹר כל מתאר: אם הוא ב: c_rfd ==> ממתינים עליו נתונים, לכן ק'

אם הק' החזירה אפס ==> הלקוח סגר הקשר ==> סגור המתאר, הסירו מהקבוצה, והקטן גודלה

אם קראת נתונים הדהד אותם ללקוח

התכנית השלמה (לעיונכם):

```
// file:  socket_server_multi_clients2.c
// A programs that opens a socket,
// and serves a few clients in parallel.
// for each new client a new socket is being opened
// Run the client:
// socket_client_multi_clients inferno-05.cs.hadassah-col.ac.il


#include <stdio.h>
#include <stdlib.h>
#include <string.h>              // for memset
#include <sys/types.h>
#include <netinet/in.h>          /* Internet address structures  */
#include <sys/socket.h>
#include <netdb.h>               /* host to IP resolution        */
#include <sys/time.h>            /* for timeout values           */
#include <unistd.h>              /* for table size calculations  */

#define PORT          5060    /* port of our echo server */
#define BUFLEN        1024  // buffer size
#define QUEUE_SIZE      10

int main()
{
   int            main_socket ;
   int            serving_socket ;
   int            size_her_addr;
   char           buf[BUFLEN+1];
   struct sockaddr_in  my_addr;
   struct sockaddr_in  her_addr ;
   fd_set         rfd;
   fd_set         c_rfd;
   int            size_of_rfd_set ;
   int            fd ;
   int            rc;

   main_socket = socket(PF_INET, SOCK_STREAM, 0);
   if (main_socket < 0) {
      perror("socket: allocation failed");
      exit(EXIT_FAILURE) ;
   }
```

```
    memset(&my_addr, 0, sizeof(my_addr));
    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(PORT);
    my_addr.sin_addr.s_addr = htonl(INADDR_ANY);

    rc = bind(main_socket,
              (struct sockaddr *)&my_addr, sizeof(my_addr));
    if (rc) {
        perror("bind failed");
        exit(EXIT_FAILURE) ;
    }

    rc = listen(main_socket, QUEUE_SIZE);
    if (rc) {
        perror("listen failed");
        exit(EXIT_FAILURE) ;
    }
```



```
    size_her_addr = sizeof(her_addr);

    FD_ZERO(&rfd);
    FD_SET(main_socket, &rfd);
    size_of_rfd_set = 1 ;

    while (1) {
        c_rfd = rfd;

        rc = select(main_socket + size_of_rfd_set,
                    &c_rfd,
                    NULL, NULL, (struct timeval *) NULL);

        if (FD_ISSET(main_socket, &c_rfd)) {
            serving_socket = accept(main_socket,
                                    (struct sockaddr *)&her_addr,
                                    (socklen_t*) &size_her_addr);
            if (serving_socket >= 0) {
                FD_SET(serving_socket, &rfd);
                size_of_rfd_set++ ;
            }
        }
        for (fd= main_socket +1;
             fd < main_socket+ size_of_rfd_set;
             fd++)
            if (FD_ISSET(fd, &c_rfd)) {
                rc = read(fd, buf, BUFLEN);
                if (rc == 0) {
                    close(fd);
                    FD_CLR(fd, &rfd);
                    size_of_rfd_set -- ;
                }
                else if (rc > 0)
                    write(fd, buf, rc);
                    // check that write went okay...
                else {
                    perror("read() failed") ;
                    exit(EXIT_FAILURE) ;
                }
            }
    }
    return(EXIT_SUCCESS) ;
}
```


הלקוחות: נריץ כמה כאלה במקביל. כל אחד שולח כמה מחרוזות לשרת ומקבלן חזרה. נריצם:

socket_client_multi_clients inferno-05.cs.hadassah-**col**.ac.il &

```
// A program that opens a socket with a server,
// sends/receives info to/from it.
// The server serves a few clients simulaneousel/in-parallely
// usage: <prog-name> <address of server>
// e.g.,:
// socket_client_multi_clients inferno-04.cs.hadassah-col.ac.il &
// and run a few like this in parallel.
// ('ping inferno-04' returns the full address of it)

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>             // for read/write/close
#include <sys/types.h>          /* standard system types       */
#include <netinet/in.h>         /* Internet address structures */
#include <sys/socket.h>         /* socket interface functions  */
#include <netdb.h>              /* host to IP resolution       */

#define  BUFLEN        1024   /* maximum response size  */
#define  PORT          5060   // set by server

int main(int argc, char *argv[])
{
    int             rc;
    int             s;
    int             i;
    char            rbuf[BUFLEN+1];
    char            wbuf[BUFLEN+1];
    struct sockaddr_in  sa;         /* Internet address struct        */
    struct hostent*     hen;        /* host-to-IP translation         */

    if (argc < 2) {
        fprintf(stderr, "Missing server name \n");
        exit(EXIT_FAILURE) ;
    }

    srand(17) ;
```



```
    hen = gethostbyname(argv[1]);
    if (!hen) {
        perror("couldn't resolve host name");
        exit(EXIT_FAILURE) ;
    }

    // create an address of host in a var, for the socket
    memset(&sa, 0, sizeof(sa));
    sa.sin_family = AF_INET;
    sa.sin_port = htons(PORT);
    memcpy(&sa.sin_addr.s_addr,
           hen->h_addr_list[0], hen->h_length);

    // allocate a socket
    s = socket(AF_INET, SOCK_STREAM, 0);
    if (s < 0) {
        perror("socket: allocation failed");
        exit(EXIT_FAILURE) ;
    }

    sleep(rand() % 10) ;
    // connect to server
    rc = connect(s, (struct sockaddr *)&sa, (socklen_t) sizeof(sa));
    if (rc) {
        perror("connect() failed");
        exit(EXIT_FAILURE) ;
    }
```

```
    // send and then receive messages from the server
    strcpy(wbuf, "a") ;
    for (i = 0; i < 7; i++) {
        rc = write(s, wbuf, strlen(wbuf)+1) ;
        // if (rc < strlen(wbuf)+1) …

        rc = read(s, rbuf,  strlen(wbuf)+1) ;
        // if (rc < strlen(wbuf)+1) …

        printf("%d got %s from server\n", (int) getpid(), rbuf) ;
        strcat(wbuf, "a") ;

        /* also possible:
        int num ;
        rc = write(s, &i, sizeof(i)) ;
        rc = read(s, &num,  sizeof(num)) ;
        printf("Got %d from server\n", num) ;
        */

        sleep(rand() % 5) ;
    }

    close(s);

    return EXIT_SUCCESS;
}
```


הרצה:

```
<217|1>yoramb@inferno-04:~/os>      socket_client_multi_clients
inferno-05.cs.hadassah-col.ac.il &
[1] 11269
<218|0>yoramb@inferno-04:~/os>      socket_client_multi_clients
inferno-05.cs.hadassah-col.ac.il &
[2] 11270
<219|0>yoramb@inferno-04:~/os>      socket_client_multi_clients
inferno-05.cs.hadassah-col.ac.il &
[3] 11271
<220|0>yoramb@inferno-04:~/os> 11269 got a from server
11270 got a from server
11271 got a from server
11269 got aa from server
11270 got aa from server
11271 got aa from server
11269 got aaa from server
11270 got aaa from server
11271 got aaa from server
11269 got aaaa from server
11270 got aaaa from server
11271 got aaaa from server
11269 got aaaaa from server
11270 got aaaaa from server
11271 got aaaaa from server
11269 got aaaaaa from server
11269 got aaaaaaa from server
11270 got aaaaaa from server
11270 got aaaaaaa from server

[1]    Done                socket_client_multi_clients inferno-
05.cs.hadassah-col.ac.il
<220|0>yoramb@inferno-04:~/os>
[2]    Done                socket_client_multi_clients inferno-
05.cs.hadassah-col.ac.il
```


4.11 קריאה לשגרה מרוחקת Remote Procedure Call (RPC)

הרעיון ב- RPC הינו שתכנית (לקוח) הרצה על מחשב א' מזמנת פונ' המורצת על מחשב ב' (על-ידי פניה לשרת המורץ על מכונה ב', וחושף לעולם שגרות בתכניתו).

לצערי, המהדר rpcgen, הכלי שיוניקס מעמידה לרשותנו למימוש הרעיון הינו מיושן, מקרטע, ומאוד לא ידידותי.
נכיר אותו, על-כן, רק במידה בסיסית.

הרעיון במהדר rpcgen הוא לחסוך למתכנת את כל ה-'גועל נפש' של פרטי התקשורת בין השרת ללקוח.

התכנית השלמה שאנו נכתוב תתחלק לשלושה חלקים:

א. תכניתונת בשפת rpcgen שתאפשר למהדר rpcgen לייצר עבורנו קבצים כפי שאתר מייד.
ב. מימוש הפונ' שהשרת מספק _פונ' אותן יכול הלקוח לזמן ממכונה מרוחקת).
ג. מימוש תכנית הלקוח, אשר תשתמש בכלי התקשורת שהמהדר- rpcgen ייצר עבורה (כדי לזמן את הפונ' מסעיף ב')

נראה את שלושת המרכיבים:

4.11.1 התכניתונת בשפת rpcgen

```
// file: rpc2/server_prog.x

/* rpcgen program
  Also in this project:
  1. server_routines.c :
          an implementation of the server routines.
  2. client_calling_rpc.c :
          a client program that issues the rpc.

  Compilation:
  1. rpcgen server_prog.x
    This creates: server_prog.h : an header file
                  server_prog_svc.c : a sekeleton of a server
                  server_prog_clnt.c : a skeleton for a client.
3.   Edit file: server_prog_svc.c
     change print_msg_1_svc to print_msg_1
     and so on with the other routines
     (my_sqrt_1_svc, my_sum1_svc).
  4. gcc server_routines.c
     server_prog_svc.c -lm -o server
     This creates the server. Do it on inferno-05
  5. gcc client_calling_rpc.c  server_prog_clnt.c -o client
    This creates the client. Do it in inferno-04
  6. On inferno-05, run: server &
  7. on inferno-04 run: client 10.2.10.25 "hello world"
 */
```

```
program REMOTE_PROG_SKELETON
{
   version MESSAGEVERS
   {

      // the three functions the server supplies:
      int MY_PRINTMESSAGE(string) = 1;
      double MY_SQRT(double *) = 2;
      int MY_SUM(int *) = 3;
   } = 1;
} = 99;
```

שם התכנית

מספר הגירסה

טיפוס המציין מערך של תווים שהינו NULL terminated (חריגה מ-C)

שמות הפונ'. נהוג לכתבם באות גדולה

מספרה של כל פונ': 1, 2, 3, ...

מספר גירסה

הפרמטר לכל פונ' הוא בהכרח מצביע יחיד, וערך ההחזרה הוא פרימיטיבי יחיד

מספר תכנית

עתה נהדר את התכניתונת:

rpcgen server_prog.x

נקבל שלושה קבצים:

א. server_prog.h
כולל הצהרות להן יזדקקו התכניות שנכתוב (הן תכנית השרת והן תכנית הלקוח).

ב. server_prog_svc.c
כולל את שלד תכנית השרת. תכנית זאת תזמן, על-פי בקשת הלקוח, את שלוש הפונ' שתיארנו בקובץ x. ונממש מייד. *בעיקרון*, בקובץ זה איננו נוגעים.

ג. server_prog_clnt.c כולל את הפונ' שתסיענה לנו בתקשורת בין הלקוח לשרת. יקומפל עם תכנית הלקוח.

<u>הערה</u>:
בשל באג(?) במהדר rpcgen נאלצתי אחרי ייצור קובץ השרת server_prog_svc.c להיכנס לקוד שלו, ועבור כל פונ', כדוגמת MY_SUM מקובץ ה-x. ושהפכה להיות: my_sum_1_svc בקובץ ה-c. למחוק את הסיפא svc_
יש לבצע שינוי יחיד עבור כל פונ' שהשרת מספק.
לא נעים,
לא נורא.

מספר גירסה

תוספת מיותרת ומזיקה

4.11.2 מימוש הפונ' המסופקות ע"י השרת

נפנה עתה לקובץ server_routines.c הנכתב כולו על-ידינו, וכולל את מימוש הפונ' המסופקות ע"י השרת.

כפי שראינו בקובץ ה- x., כל אחת מהפונ' המסופקות ע"י השרת מקבלת מצביע יחיד בהכרח למשתנה פרימיטיבי! ומחזירה ערך פרימיטיבי יחיד.

פונ' כדוגמת:

```
int MY_PRINTMESSAGE(string) = 1;
```

הופכת כאן להיות:

```
int * my_printmessage_1(char** msg, CLIENT *cl)
{
   static int result;            /* must be static! */

   if (puts(*msg) <= 0)
     result = 1 ;            // == failure
   else
     result = 0;             // == success
   return (&result);
}
```


<u>הערות</u>:

א. הפונ' חייבת להחזיר מצביע למשתנה סטטי (וזאת למרות שבקובץ ה- x. היא מוצהרת כמחזירה int).

ב. שם הפונ' מופיע כאן (בקובת ה- c.) באות קטנה.

ג. התוספת 1_ לשם הפונ' מציין את מספר הגרסה.

ד. הפרמטר ** char הוא המתאים לטיפוס string עליו הצהרנו בקובץ ה- x. .

ה. הפרמטר CLIENT *cl מסייע בתקשורת עם הלקוח. אין לו תפקיד מבחינתנו.

דין דומה חל עם שתי הפונ' הנוספות:

```
double MY_SQRT(double *) = 2;
```

הופכת כאן להיות:

```
double * my_sqrt_1(double *x, CLIENT *cl)
{
   static double result;

   result = (*x >= 0) ? sqrt(*x) : sqrt(-(*x));
   return (&result);
}
```

ולבסוף:

```
int MY_SUM(int *) = 3;
```

הופכת להיות:

```
// only A SINGLE param is allowed!
// a pointer cannot point to an array!
// Neither a pointer to a struct can not be simply passed.
// There are solutions to this problem
// (which I do not present)

int * my_sum_1(int *n1, /* int *n2, */ CLIENT *cl)
{
   static int result;

   result = (*n1 + *n1) ;
   return (&result);
}
```

להשלמת התמונה, ה- include הדרושים:

```
#include <math.h>
#include <stdio.h>
#include "/usr/include/rpc/rpc.h"
// not this one, but the above: #include <rpc.h>
#include "server_prog.h"    // generated by rpcgen
                            // from server_prog.x */
```


עד כאן השלמנו את צד השרת:

א. המהדר הכין לנו את המהדר הכין לנו את server_prog_svc.c : שלד תכנית השרת

ב. אנו כתבנו את server_routines.c : מימוש הפונ' שהשרת מספק.

ג. נקמפלם יחד:

```
gcc –Wall server_routines.c
          server_prog_svc.c
          -lm
          -o server
```

ולמעשה בזאת השרת שלנו מוכן, וניתן להריצו:

```
server &
```

והוא ירוץ ברקע ויחכה לפניות של הלקוחות.

נפנה עתה לתכנית הלקוח:

4.11.3 תכנית הלקוח

עתה נממש את תכנית הלקוח (אשר לשם התקשורת עם השרת תשתמש בקובץ server_prog_clnt.c שיוצר ע"י rpcgen.)

```
// fle: client_calling_rpc.c
// run: client 10.2.10.25 "Hello World"

#include <stdio.h>
#include "/usr/include/rpc/rpc.h"
#include "server_prog.h"       /* as  generated by rpcgen */

int main(int argc, char** argv)
{
   CLIENT *cl;     // דרוש לתקשורת לא מעניין אותנו

   int *result;       // משתני עזר שונים
   char *server;
   char *message;
   double val  ;
   double *dresult  ;

   int *arr = malloc(2*sizeof(int)) ;

   if (argc != 3)
   {
      fprintf(stderr, "usage: %s <host> <message>\n",
                        argv[0]);
      exit(EXIT_FAILURE);
   }
   /* Save values of command line arguments */
   server = argv[1];
   message = argv[2];
```


שם תכנית השרת, ומספר גרסה בקובץ x.

```
cl = clnt_create(server, REMOTE_PROG_SKELETON,
                         MESSAGEVERS, "tcp");
if (cl == NULL)
{
      /* Couldn't establish connection with server.
       * Print error message and die. */
      clnt_pcreateerror (server);
      exit(EXIT_FAILURE);
}
```

```
result = my_printmessage_1(&message, cl);
if (result == NULL)    // הייתה בעיה בקשר עם השרת,
{                      // הזימון נכשל
   clnt_perror (cl, server);
   exit(EXIT_FAILURE);
}
if (*result != 0)      // הייתה בעיה בהצגת המחרוזת בשרת
{
   fprintf(stderr, "%s: %s couldn't print your\
                   message\n",argv[0], server);
   exit(EXIT_FAILURE);
}
printf("Message was printed by %s.\n", server);

val = -17 ;
dresult = my_sqrt_1(&val, cl);
if (dresult == NULL)          //   בעיה בקשר\בזימון
{
   clnt_perror (cl, server);
   exit(EXIT_FAILURE);
}

printf("Result returned from server is: %f\n",
       (float) *dresult) ;
```



```
   arr[0] = 5 ; arr[1] = 7 ;
   result = my_sum_1(arr, cl);
   if (result == NULL)
   {
      clnt_perror (cl, server);
      exit(EXIT_FAILURE);
   }

   printf("sum returned from server is: %d\n", *result) ;

   clnt_destroy(cl) ;

   exit(EXIT_SUCCESS);
}

//-----------------------------------------------------
/* AN OUTPUT:
 * ==========
<214|0>yoramb@inferno-04:~/os/rpc2>
                                   client 10.2.10.25 "yosi"
Message was printed by 10.2.10.25.
Result returned from server is: 4.123106
sum returned from server is: 10
<215|0>yoramb@inferno-04:~/os/rpc2>
*/
```

השרת במכונה אחרת ולא יכול לגשת להמשך המערך

תכנית הלקוח תקומפל תוך שימוש ברוטינות התקשורת שיוצרו ע"י המהדר, ומצויות בקובץ server_prog_clnt.c:

```
gcc –Wall client_calling_rpc.c
          server_prog_clnt.c
          -o client
```

הרצת הלקוח:

client 10.2.10.25 "Hello World"

4.11.4 XDR = Extended/External Data Reperesentation

כפי שכבר הזכרנו בהקשר של תושבת, מכונות שונות מאחסנות נתונים בצורות שונות (האם הספרה הפחות משמעותי מאוחסנת בבית הפחות משמעותי או היותר משמעותי). על כן עת ברצוננו להעביר נתונים ממכונה למכונה יש לתת על-כך את הדעת.

rpcgen מסוגל לסייע לנו גם בכך: לייצר פונ' שתתרגמנה את הנתונים מהיצוג המקומי, ליצוג ב"ת: XDR ולהפך.

פונ' אלה תשכונה בקובץ נוסף (רביעי) שייוצר ע"י המהדר.

בתכנית שכתבנו קודם לא התייחסנו לסוגיה, וכך גם נותיר אותה: לא פתורה.

ואידַך זיל גְמוֹר.