מערכות הפעלה תרגול 4

Process

מתרגל-יורם סגל
rmisegal@cs.colman.ac.il

Contents

תהליכים

תהליך (process) הוא אבסטרקציה של תכנית, פעולות שהתכנית מבצעת לפי הסדר.

כמו תאומים

בן המלך והעני

מילים ולחן: ירון כפכפי

איזה פלא!
להיות דומים כאלה.
רק החלפנו את הבגדים
אתה עכשיו אני,
והנסיך עני,
כמו תאומים זהים, אחים בדם
זה מדהים איך הבגדים עושים את האדם.

אם נקביל קובץ תוכנית למדים של חיילים

אזי כל חייל הלובש מדים הוא תהליך בפעולה

מושגי יסוד בתהליכים

- **Multiprogramming**: ישנם מספר תהליכים שצריכים להתבצע.
- **Scheduling**: איזה תהליך ירוץ על המעבד ומתי.
- **Context**: המערכת שומרת לכל תהליך **הקשר** משלו כדי שתוכל להריץ ולהחליף בין מספר תהליכים במקביל
  - הסביבה של התהליך (הזכרון שלו, זהות והרשאות, מצב הרגיסטרים עבור התהליך) שבה הוא חי.

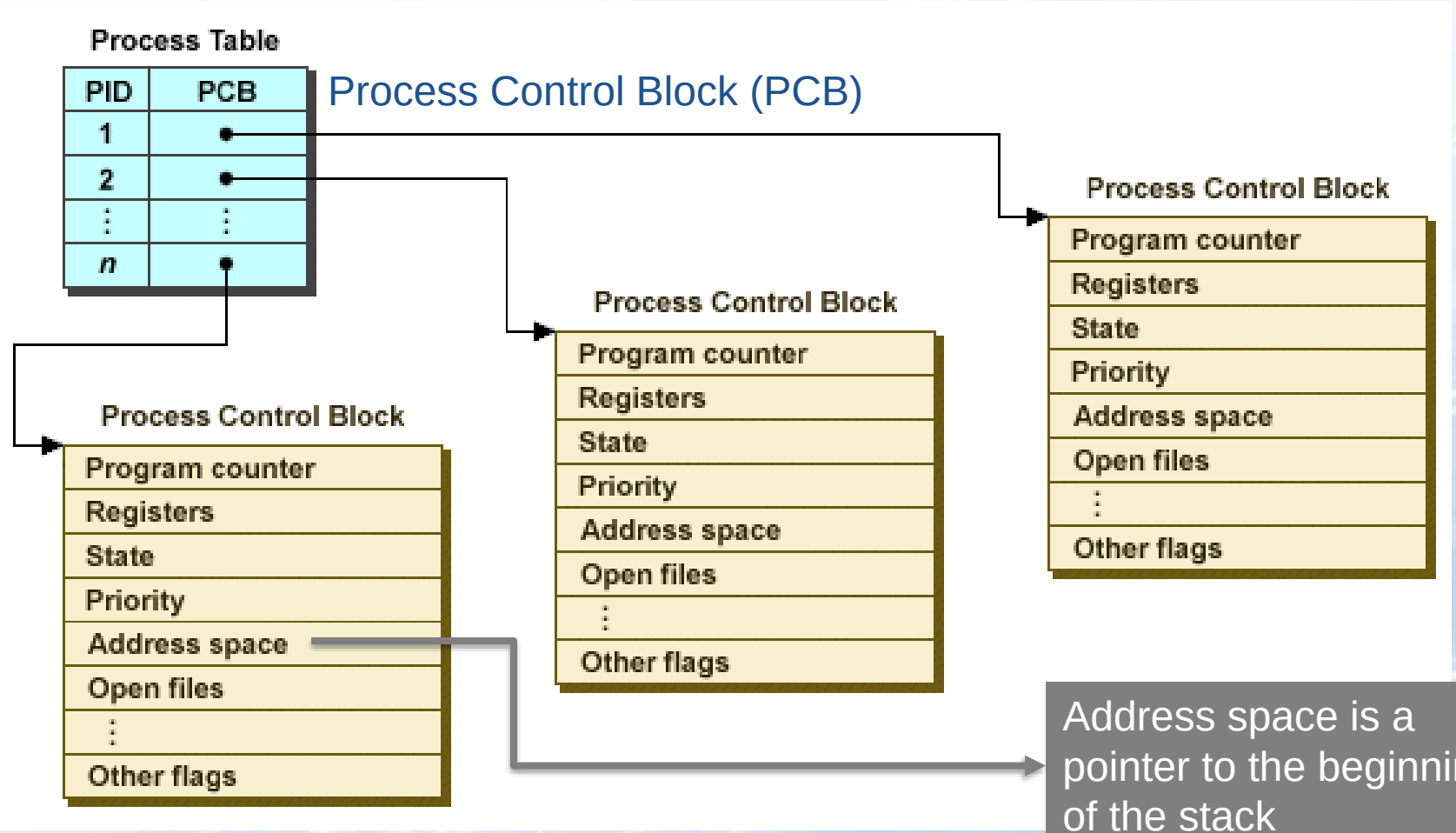
הקשר של תהליך

- **PID** – process identifier, מספר יחודי שמתקבל בזמן יצירת התהליך.
- **UID** – user identifier, מספר יחודי של המשתמש שמריץ את התהליך.
- **PPID** – parent PID, המזהה של התהליך שיצר את התהליך הנוכחי.

הקשר של תהליך

- **Mode** – מצב התהליך (מוכן, ממתין) בהמשך...
- **Priority** – מספר המסמל את עדיפות הרצת התהליך, הרחבה בהמשך.
- מערכת ההפעלה שומרת לכל תהליך אזור הנקרא **U-area** המכיל מידע הספציפי לתהליך. למשל: קבצים פתוחים,הקוד של התהליך, הזכרון של התהליך.

Process Table

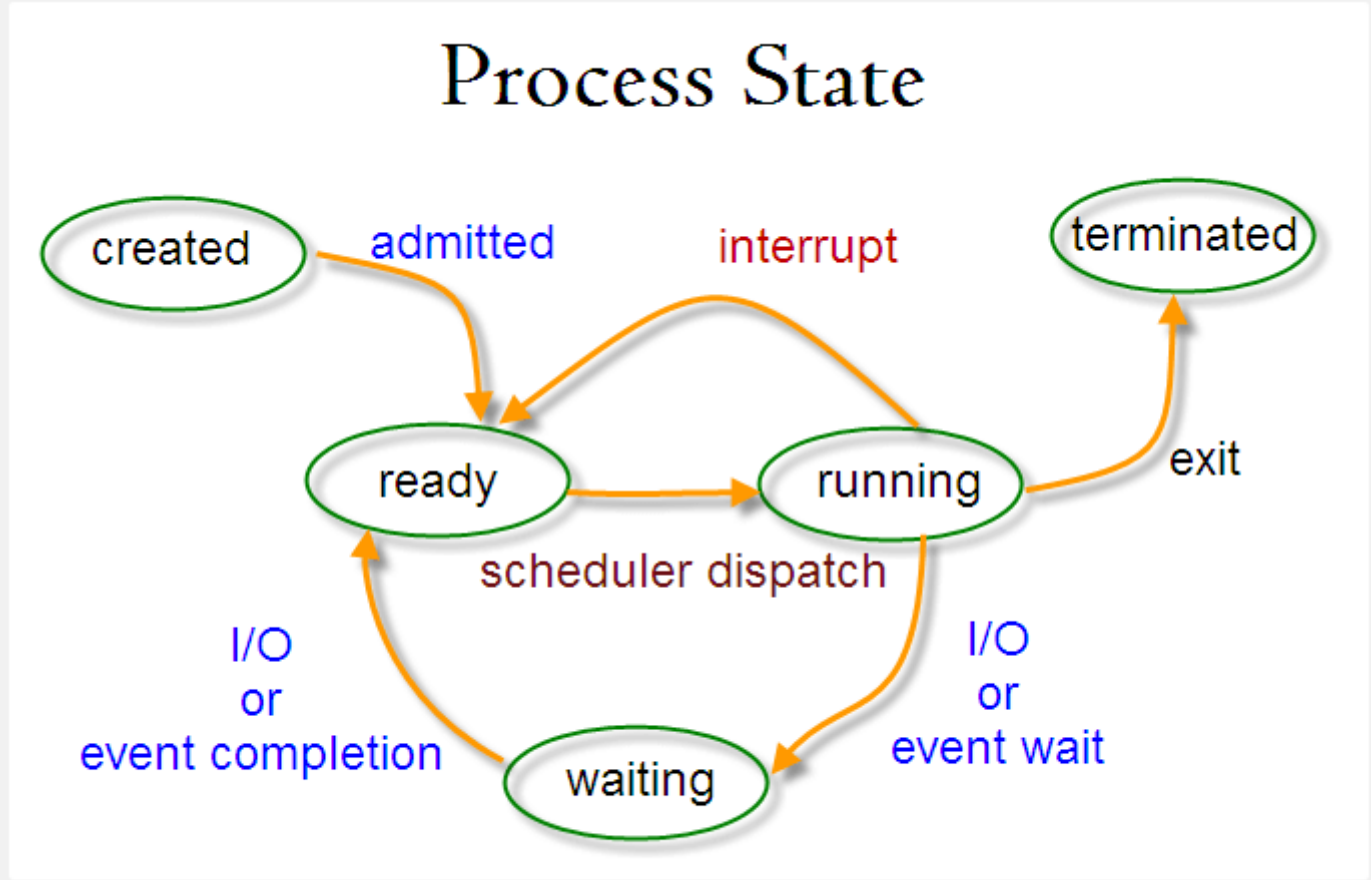


Address space is a pointer to the beginning of the stack

איך תהליך נוצר

- כאשר מערכת ההפעלה נטענת הגרעין יוצר 2 תהליכים:
  - Swapper: pid=0, אחראי על ניהול זכרון
  - Init: pid=1, האב של כל התהליכים, כל התהליכים צאצאים שלו.
- **בלינוקס כדי לייצר תהליך משתמשים בפקודה fork()**
- התהליך שנוצר הוא העתק של התהליך שיצר אותו (האב).

Process States



Linux Process State

פונק' fork (1)

- קריאת המערכת fork()
  - תחביר:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork();
```

  - פעולה: מעתיקה את תהליך האב לתהליך הבן וחוזרת בשני התהליכים
    - קוד זהה (**ומיקום בקוד זהה בשני התהליכים**)
    - זיכרון זהה (משתנים וערכיהם, חוצצים)
    - סביבה זהה (קבצים פתוחים, file descriptors, ספרית עבודה נוכחית)
  - פרמטרים: אין
  - ערך מוחזר:
    - במקרה של כישלון: (-1) לאב (מינוס אחד)
    - במקרה של הצלחה:
      - לבן מוחזר 0
      - ולאב מוחזר ה-pid של הבן

פונק' fork (2)

- לאחר פעולת fork() מוצלחת, אמנם יש לאב ולבן את אותם משתנים בזיכרון, אך בעותקים נפרדים (פרט לקבצים)
  - כלומר, שינוי ערכי המשתנים אצל האב לא ייראה אצל הבן, וההיפך
- כמו כן, תהליך הבן הוא תהליך נפרד מתהליך האב לכל דבר. בפרט, יש לו pid משלו
- מה יכול להדפיס הקוד הבא?

```
main() {
    fork();
    printf("hello");
}
```

פונק' fork (3)

- תשובות אפשריות (בהנחה ש-()fork הצליחה):

אפשרות ראשונה hellohello

אפשרות שניה hheellollo

אפשרויות נוספת .....

- הסיבה: שני תהליכים כותבים פלט בצורה לא מתואמת.
- מסיבה זו, נוסף תקן הקובע כי פונקציות מסוימות, וביניהן printf, "ינעלו" את הגישה לפלט בזמן שהן מדפיסות כדי למנוע ערבוב. לאחר שprintf מסיימת להדפיס, היא משחררת את הנעילה. פעולה כזו נחשבת "אטומית".
- לא ניתן לדעת בוודאות האם מערכת מסויימת עומדת בתקן או לא, לכן יש להניח כי אם לא נדאג לכך, פלטים יכולים "להתערבב"

Question

- A process execute the code :

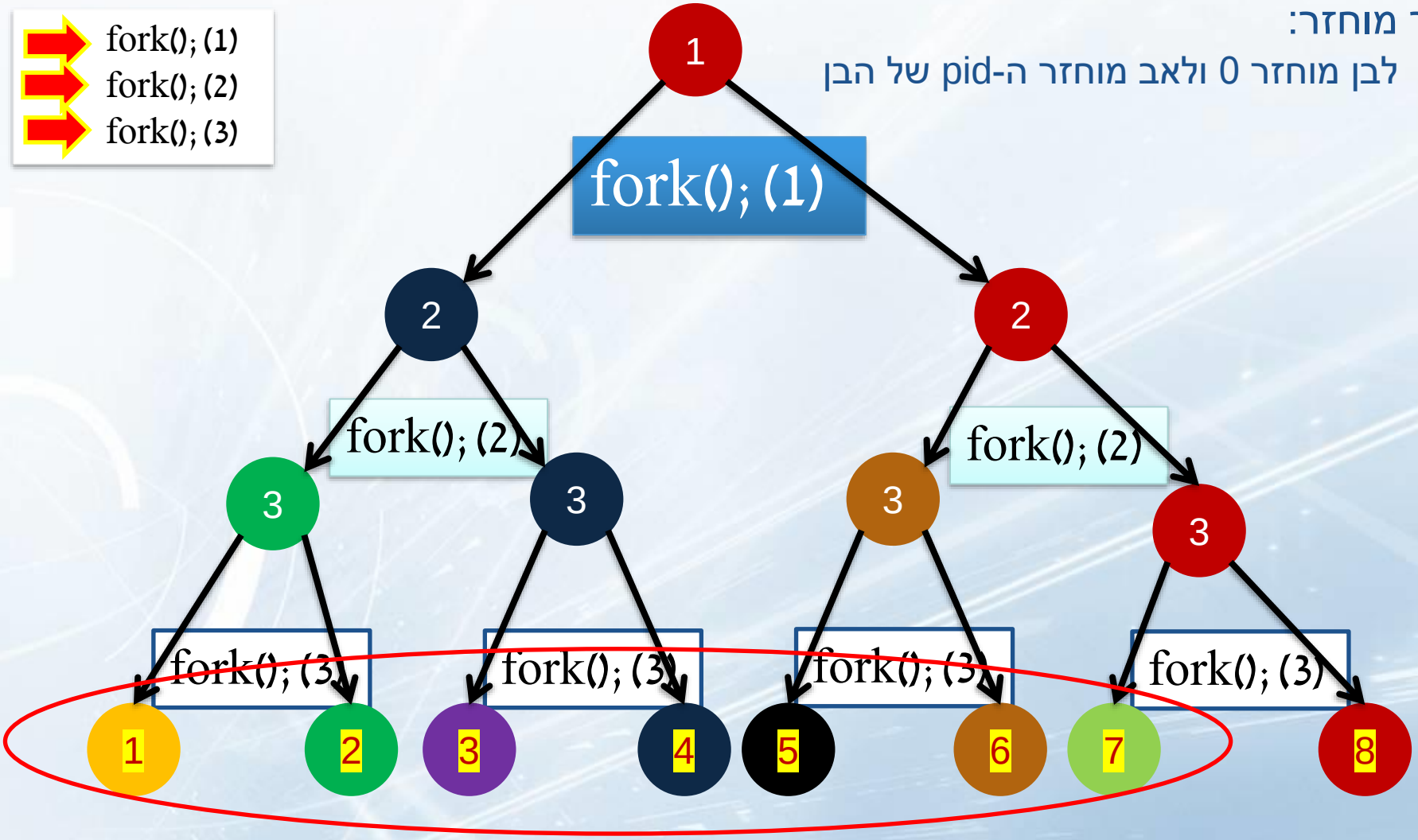
```
fork(); (1)
fork(); (2)
fork(); (3)
```

What will be the total number of **child** processes created ?

Answer

ערך מוחזר:

לבן מוחזר 0 ולאב מוחזר ה-pid של הבן

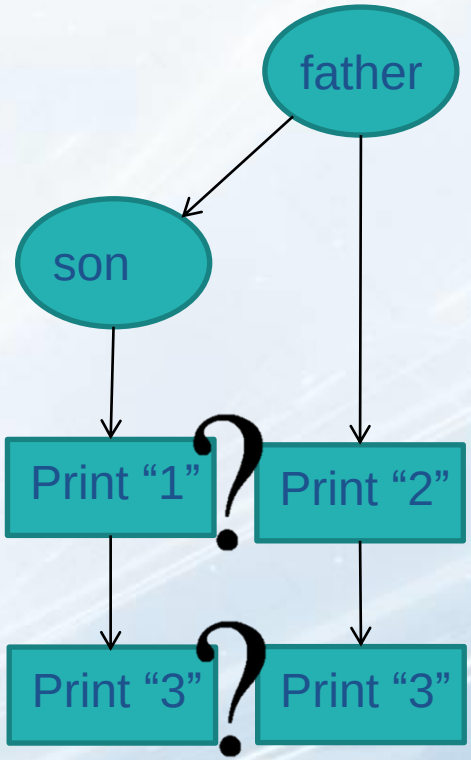


קוד ראשי

דוגמאות (1)

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

void main()
{
    pid_t pid;
    if (  (pid = fork()) == 0)
        printf("1");
    else
        printf("2");
    printf("3");
}
```



- Output: 2133 or 1233 or 2313 or 1323 or 23 (3312 is not possible)
  - גם אם printf אטומית, לא ניתן לשלוט בסדר בו תהליכים יתבצעו.

יתומים ואימוץ

- כאשר אב מסיים את פעולתו והבן עדיין חי אז הבן נחשב orphan.
- התהליך init מאמץ את כל היתומים.

דוגמאות לזיהוי תהליכים בזמן ריצה(2)

```
int main(){
    int i;
    for (i = 0; i < 3; i++)
        if (fork() == 0)
            while(1);
}
gcc -o example2 2_6.c
./example2
ps
UID   PID  PPID   NI STAT TT    TIME COMMAND
…
...
8385  1668   1     20  R   pts/0   0:12  example2
8385  1669   1     20  Z   pts/0   0:14  example2
8385  1670   1     20  S   pts/0   0:13  example2
kill -KILL 1668 1669 1670
```

מציג את רשימת התהליכים הפעילים ע"י המשתמש המחובר

State of a process:
O - Running.
S - Sleeping (process is waiting for an event to complete).
R - Runnable (process is on run queue).
Z - Zombie (process terminated and parent not waiting)
T - Stopped.

fork

- **pid_t getpid()**
  - **Returns:** process ID of calling process.
- **pid_t getppid()**
  - **Returns:** parent process ID of calling process.
- **uid_t getuid()**
  - **Returns:** user ID of calling process.
- **uid_t getgid()**
  - **Returns:** group ID of calling process.

דוגמא למספור תהליכים עקב fork

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

int main()
{
        pid_t val;
        printf("PID before fork: %d\n",(int)getpid());
        val = fork();
        if(val>0)
                printf("parent PID: %d\n",(int)getpid());
        else if(val ==0)
                printf("child PID: %d\n",(int)getpid());
        else {//if val==-1 print error to screen}
}
```

שיטה להבחנה בין תהליך האבא לתהליך הילד שרק נוצר

דוגמא

- כמה תהליכים ייווצרו לאחר הרצת קטע הקוד הבא? נמקו תשובתכם.

```
pid_t pid;
pid = getpid();
while (fork() == 0) {
    if (pid == getpid())
        break;
}
```

Negative Value: creation of a child process was unsuccessful.
Zero: Returned to the newly created child process.
Positive value: Returned to parent or caller. The value contains process ID of newly created child process.

דוגמא

- <u>תשובה</u>
- התהליך המקורי קורא ל- ()fork, יוצר בן ולאחר מכן מסיים (כי הערך שחוזר אצלו מה- ()fork שונה מאפס).
- תהליך הבן שנוצר נכנס לתוך הלולאה ואצלו לא מתקיים התנאי ()pid = getpid, כיוון שמזהה התהליך שלו שונה מהמזהה של תהליך האב. לכן הוא חוזר לראש הלולאה וקורא שוב ל- ()fork. כתוצאה מכך הוא יווצר תהליך נכד חדש ותהליך הבן יסתיים (כי שוב הערך שחוזר אצלו מה- ()fork שונה מאפס).
- תהליך הנכד נכנס ללולאה ושוב אצלו התנאי ()pid = getpid לא מתקיים, לכן הוא יחזור לראש הלולאה, וכך הלאה. בסופו של דבר ייווצרו אינסוף תהליכים.

דוגמא (t3_1.c)

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

int     glob = 6;                       /* global variable */
char    buf[] = "a write to stdout";

int main(void)
{
        int     var;                    /* automatic variable on the stack */
        pid_t   pid;

        var = 88;
        if(puts(buf)==EOF)
                printf("error in writing to stdout");
        printf("before fork\n");

        if ( (pid = fork()) < 0)
                printf("fork error");
        else{
                if (pid == 0)
                {       /* child */
                        glob++;         /* modify variables */
                        var++;
                }else
                { /* parent */
                        sleep(2);
```

```
shani@shani-VirtualBox:~/Dropbox/Operating Systems/Tirgul/T4$ ./a.out
a write to stdout
before fork
pid = 2802, glob = 7, var = 89
pid = 2801, glob = 6, var = 88
```

משתנה סביבה של לינוקס

- env
- echo $PATH
- top | grep 1234

Exec family

- מוטיבציה: ה shell צריך להריץ תוכנית מסויימת, שהקוד שלה שונה מהקוד שקורא לתוכנית.
- הוא יכול לייצר עוד תהליך עם ()fork אבל הקוד של התהליך הזה יהיה זהה לתהליך של ה shell
- ה shell צריך ליצר עוד תהליך שתוכן הקוד שלו יהיה שונה מהקוד של התכנית החדשה שצריכה לרוץ וה data יהיה גם כן של התכנית החדשה.
- הפונקציה חוזרת רק במקרה של שגיאה.
- פונקצית הרצת קוד: execl:

```
int execl(char *pathname, char *arg0,…., NULL)
```

ls(תזכורת)

- פקודה זו מדפיסה את התוכן של ספריה מסויימת.
- במידה ונריץ את הפקודה ללא פרמטרים, היא תדפיס את תוכנה של הספריה הנוכחית
  - ls
- שימוש בפורמט הדפסה ארוך.
  - ls –l
- מיון לפי תאריך השינוי האחרון.
  - ls -t
- הפיכת סדר התוצאות לאחר המיון.
  - ls -r

t3_exec.c

דוגמא execl

```
#include<stdio.h>
#include<unistd.h>
int main()
{
        execl("/bin/ls","/bin/ls","-l",NULL);
        printf("can only get here on error\n");
}
```

- תוכנית האם שקוראת לתוכנית משנה לא תמשיך לרוץ.
- רק תוכנית המשנה תמשיך לרוץ.
- תוכנית האם תישאר בכוננות רק למקרה של שגיאה בתוכנית המשנה ברגע הקריאה אליה.
- תוכנית האם תסגר.
- אם תקרה שגיאה בתוכנית המשנה (ברגע הקריאה אליה) אזי תוכנית האם תמשיך לרוץ מהנקודה שבא קראנו לתוכנית המשנה.

משפחת exec

e - an array of pointers to **e**nvironment variables is explicitly passed to the new process image.

l - command-line arguments are passed individually (a **l**ist) to the function.

p - Uses the **P**ATH environment variable to find the file named in the *file* argument to be executed.

v - Command-line arguments are passed to the function as an array (**v**ector) of pointers.

משפחת exec

int execl(const char ****path*, const char ****arg***, ...);**

int execlp(const char ****file*, const char ****arg***, ...);**

-look for the command in the path

int execv(const char ****path*, char *const** ***argv*****[]);**

int execvp(const char ****file*, char *const** ***argv*****[]);**

-look for the command in the path

int execle(const char ****path*, const char ****arg** **, ..., char * const** ***envp*****[]);**

int execve(const char * ***path***, **char *const** ***argv*** **[], char *const** ***envp*****[]);**

-pass the environ

int execl(const char ****path*, arg0, arg1, arg2,…..) – the command line arguments including arg0 which is the executed file**

משפחת exec

```
execl("/bin/ls", “/bin/ls", "-r", "-t", "-l", NULL);

execlp("ls", "ls", "-r", "-t", "-l", NULL);


char *args[] = {"/bin/ls", "-r", "-t", "-l", NULL };
execv("/bin/ls", args);


char *args[] = {"ls", "-r", "-t", "-l", NULL };
execvp("ls", args);
```

שימוש ב exec לקריאה לפקודת env מתוך קוד C

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

int main(void)
{
    int status;
    char *argv[] = { "/bin/env", 0 }; // Use the "which env" command to find your env location
    char *envp[] =
    {
        "HOME=/",
        "PATH=/bin:/usr/bin",
        "TZ=UTC0",
        "USER=beelzebub",
        "LOGNAME=tarzan",
        0
    };

    status=execve(argv[0], &argv[0], envp);
    if (status==-1)
        printf("exec failed\n")
    return -1;
}
```

env

קריאה לפקודה

העברת פרמטרים לפקודה

פקודה | פרמטרים | סביבה חדשה

e - an array of pointers to **e**nvironment variables is explicitly passed to the new process image.

v - Command-line arguments are passed to the function as an array (**v**ector) of pointers.

wait

- מה ההורה עושה בזמן שהילד רץ?

1. ממשיך לבצע פעולותיו.
2. ממתין לילד שיסיים.

State change :
- child terminated,
- child was stopped by a signal,
- child was resumed by a signal

wait

```
#include<sys/wait.h>
pid_t wait(int *status);
```

האב ימתין עד שהמצב (state) של אחד מבניו ישתנה.

State change – child terminated, child was stopped by a signal, child was resumed by a signal

מה status יכיל?

- האבא ממתין לשינוי STATE - שינוי STATE יתרחש רק כאשר הSTATUS מתקיים
- לאותו הSTATE יכולים להיות מוגדרים מספר STATUS-ים

WAIT status

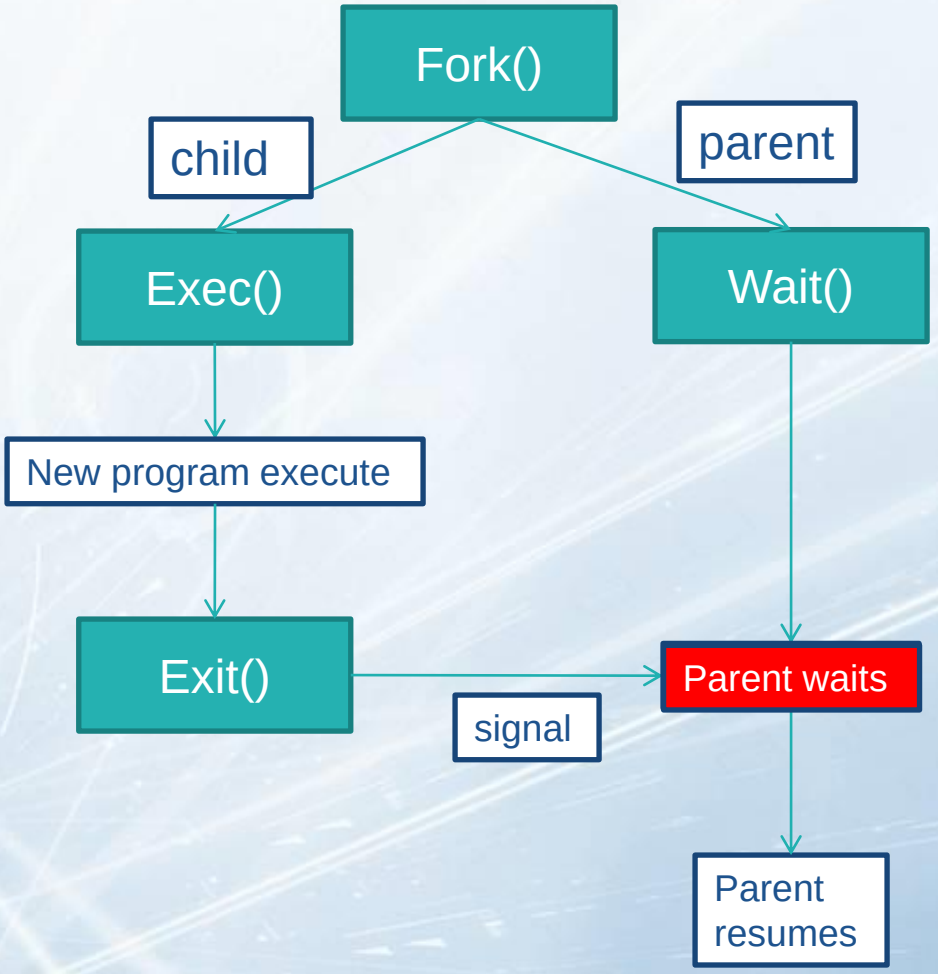
- **WIFEXITED(*status*)** returns true if the child terminated normally, that is, by calling exit(3) or _exit(2), or by returning from main().
- **WEXITSTATUS(*status*)** returns the exit status of the child. This consists of the least significant 8 bits of the *status* argument that the child specified in a call to exit(3) or _exit(2) or as the argument for a return statement in main(). This macro should only be employed if **WIFEXITED** returned true.
- **WIFSIGNALED(*status*)** returns true if the child process was terminated by a signal.
- **WTERMSIG(*status*)** returns the number of the signal that caused the child process to terminate. This macro should only be employed if **WIFSIGNALED** returned true.

status

- **WCOREDUMP(*status*)** returns true if the child produced a core dump. This macro should only be employed if **WIFSIGNALED** returned true.
- **WIFSTOPPED(*status*)** returns true if the child process was stopped by **delivery** of a signal;
- **WSTOPSIG(*status*)** returns the number of the signal which caused the child to stop. This macro should only be employed if **WIFSTOPPED** returned true.
- **WIFCONTINUED(*status*)** (since Linux 2.6.10) returns true if the child process was resumed by delivery of **SIGCONT**.

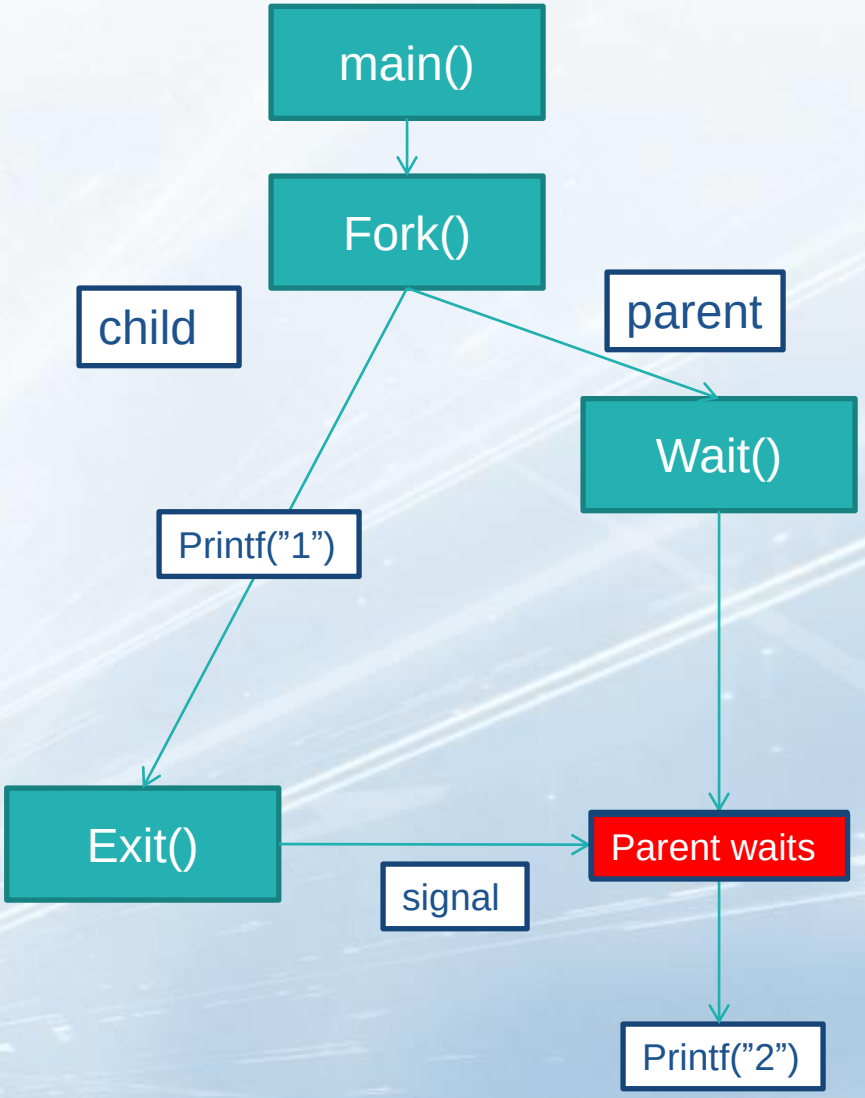
Wait flow

נדגיש לתוכנית
החדשה יש PID
זהה לזה של
childולכן
התהליך של הילד
לא מסתיים עד אשר
התוכנית החדשה
מסיימת



Wait example

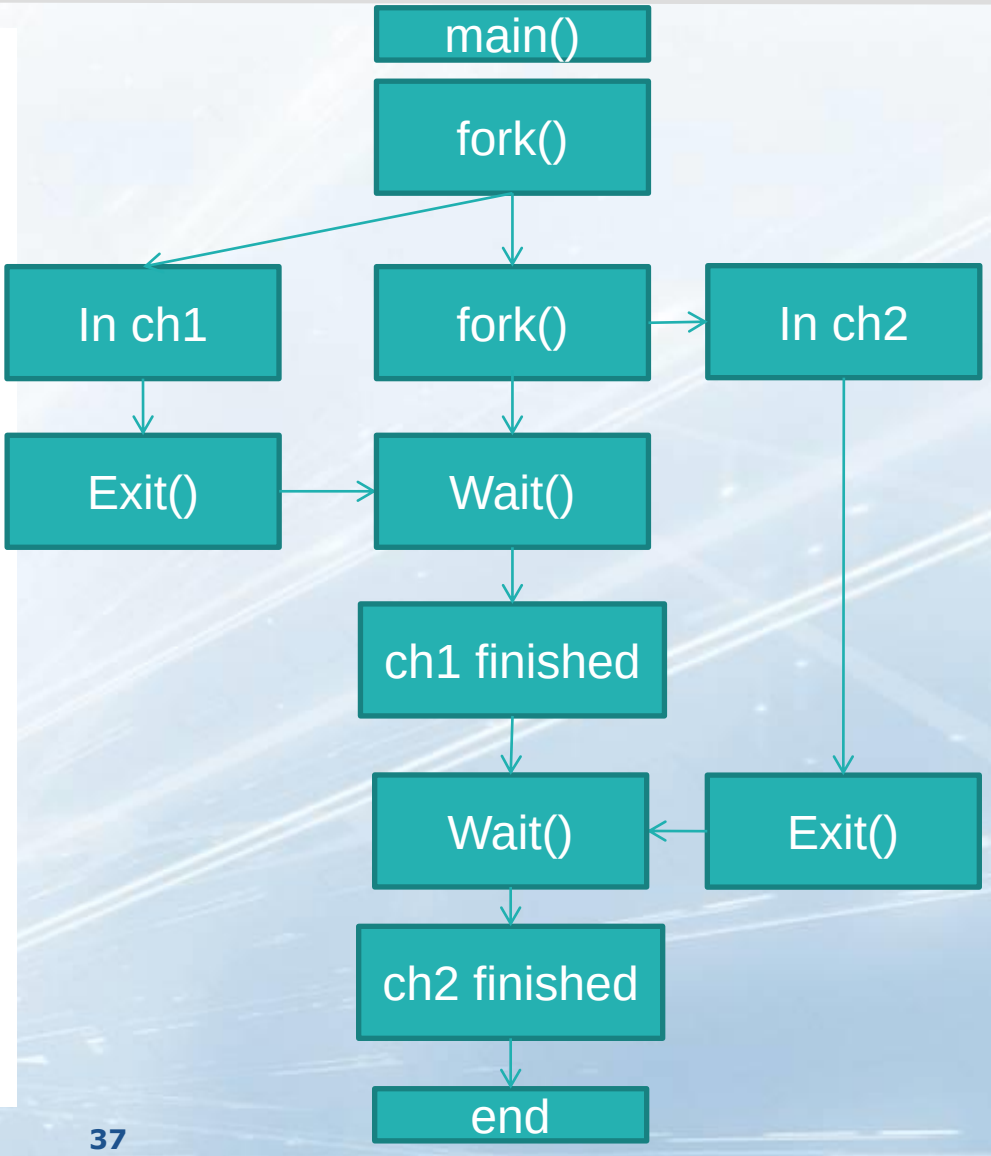
```
#include <unistd.h>
#include <sys/types.h>
#include<sys/wait.h>
#include<stdio.h>
int main()
{
    pid_t pid;
    int stat;
    if ( (pid = fork() ) == 0)
            printf("1\n");
    else
    {
            wait(&stat);
            printf("2\n");
    }
}
```



Wait example(2)

```
int main()
{
    int status;
    pid_t pid, pid1, pid2;
    if ((pid1 = fork()) == 0)
        printf("in child 1\n");
    else
        if ((pid2 = fork()) == 0)
            printf("in child 2\n");
        else
        {
            pid = wait(&status);
            if (pid == pid1)
                printf("child 1 finished\n");
            if (pid == pid2)
                printf("child 2 finished\n");
            pid = wait(&status);
            if (pid == pid1)
                printf("child 1 finished\n");
            if (pid == pid2)
                printf("child 2 finished\n");

        }
}
```



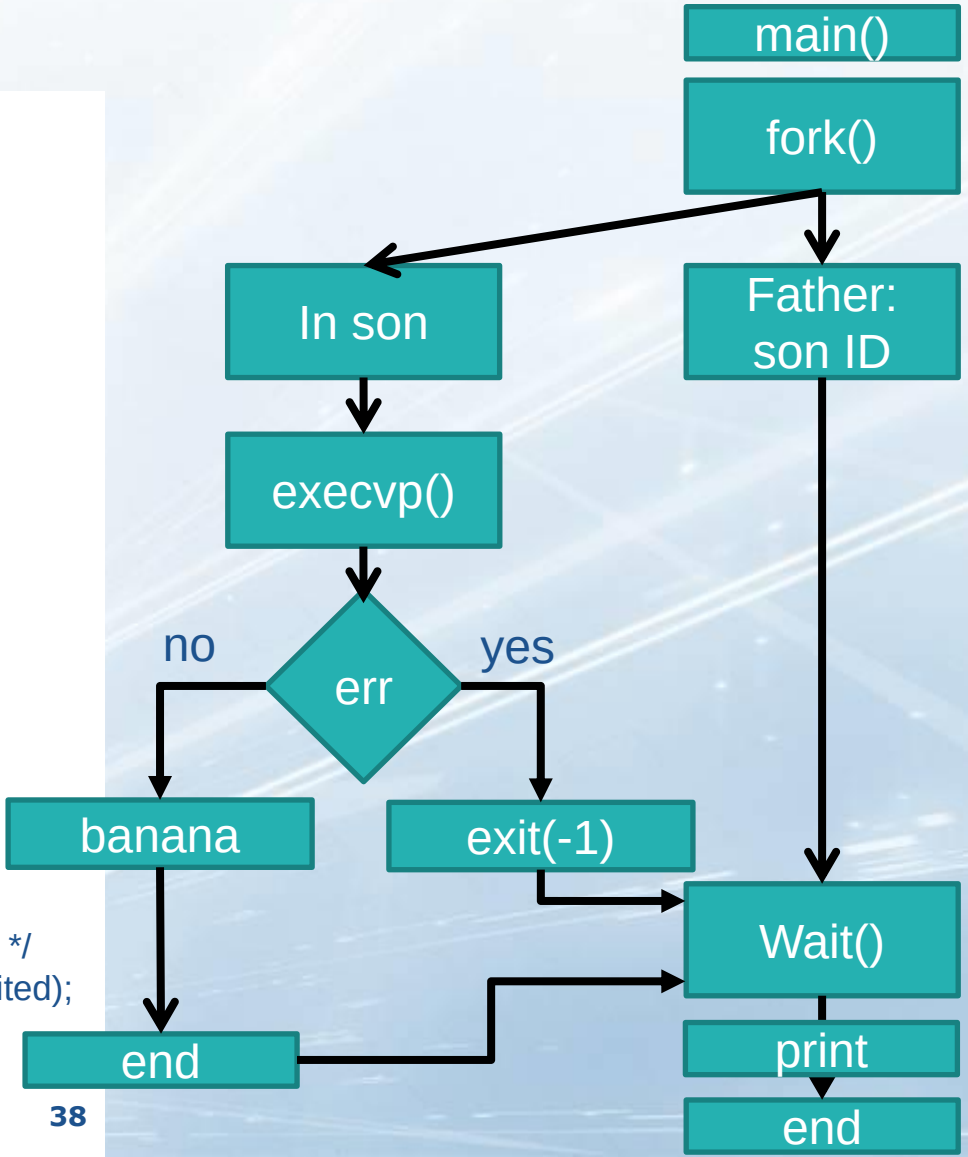
Wait() returns son pid

execvp + wait

```
int main(int argc, char* argv[])
{
        int stat, waited, ret_code;
        pid_t pid;
        pid = fork();
        if (pid == 0)
        {  /* Son */
                ret_code = execvp(argv[1],&argv[1]);
                if (ret_code == -1)
                {
                                perror("exec failed ");
                                exit(-1);
                }
                else
                                printf("Banana");
        }
        else

        {  /* Father */
           printf("Father: after fork, son proc id is %d \n",pid);
           waited = wait(&stat);   /* stat can tell what happened */
           printf("Father: Son proc completed,  id is %d \n", waited);
        }

}
```



ריבוי תהליכים ו-Context Switching

- תהליך יחיד ראשון init נוצר ומקבל PID1.
- במהלך ריצתו הוא יוצר תהליך נוסף שעתיד להתחרות בו על משאבי המערכת, נאמר PID2.
- מערכת ההפעלה תחליף בין התהליך PID1 וכל הנתונים הנלווים לו לבין PID2 תוך שמירה על חציצה בין התהליכים.
- Context Switching - לפני שתהליך "מורדם" נתוניו נשמרים ולפני שהוא הוא "מתעורר" נטענים אותם נתונים לאותם מקומות בדיוק.
- בשעת ריצת התהליך, הוא רוצה את כל משאבי המערכת שנחוצים לריצתו. אם משאב נחוץ תפוס ע"י תהליך אחר הוא יורדם שוב עד לתור הבא לניסיון ריצתו.
- תהליך יורדם לרוב לאחר בקשת I/O בשל הצפי למהירות גישה איטית מרכיבי החומרה.

מבט על fork

- נאמר כי התהליך שיצר הינו ה "אב" של תהליך ה"בן" (ראינו שאב יכול לייצר מספר רב של בנים)
- תהליך ה"בן" הינו עותק מלא של המקור
- בן איננו יצור מסורתי ויכול לבצע כל פקודה ללא קשר לאביו, אך הינו בעל ייחוס זהה
  - למשל, על-ידי קריאת המערכת ()execv
- תהליך אב רשאי לפקח על הבנים שלו
  - אך לא על "הדור הבא" או על "אחים"
  - לדוגמא ע"י שימוש ב ()wait מסוגיו השונים

waitpid

pid_t waitpid(pid_t pid, int *status, int options)

Pid:

- > 0 meaning wait for the child whose process ID is equal to the value of *pid*.
- < -1 meaning wait for any child process whose process **group ID** is equal to the absolute value of *pid*.
- -1 meaning wait for **any child process**.
- 0 meaning wait for any child process whose process **group ID** is equal to that of the **calling process**.

❖ Options:

- **WNOHANG** return immediately if no child has exited.
- **WCONTINUED** also return if a stopped child has been resumed by delivery of **SIGCONT**.

Waitpid example – t3_2.c

```
int main(void){
        pid_t       pid;
        if ((pid = fork()) < 0) {
                printf("fork error");                   }
        else        {
                if (pid == 0) {     /* first child */
                                    printf("first child\n");
                                    if ((pid = fork()) < 0)
                                                printf ("fork error");
                                    else {
                                                if (pid > 0){
                                                /* parent from second fork == first child */
                                                            if (waitpid(pid, NULL, 0) != pid)
                                                                        printf("waitpid error");
                                                            exit(0);
                                                }
                                                /*  We're the second child; */
                                                sleep(2);
                                                printf("second child, parent pid = %d\n", getppid());
                                                exit(0);
                                    }
                }
                if (waitpid(pid, NULL, 0) != pid)                   /* wait for first child */
                                    printf("waitpid error");
                /** We're the parent (the original process); we continue executing, knowing that we're not the parent of the second
        child.*/
                printf("original parent done\n");
                exit(0);
        }
}
```

EXIT

exit functions

There are three ways for process to terminate normally and two abnormal terminations.

- Normal termination:
  1. Executing **return**.
  2. Calling **exit** function.
  3. Calling **_exit** function. This function called by **exit.**

- Abnormal termination:
  1. Calling **abort**. (Will explain it on the next lecture/tirgul, generates SIGABRT signal)
  2. When the process receives certain signal. The signal can be generated by the process itself or by other process. (Next lecture/tirgul)

exit

```
#include<stdlib.h>
```

void _exit(int status); //system call

_exit: סיום התהליך הקורא "מייד", אין חיוב שיעשה flush to buffer.

void exit(int status); //c library function

exit: יבוצע flush to buffer.

ניתן להגדיר עוד פונקציות שיתבצעו בזמן קריאה ל exit באמצעות atexit.

atexit

int atexit(void (*function)(void));

- פונקציית ;()atexit "רושמת" פונקצייה נתונה להיות הפונקציה הנקראת בסיום נורמלי של תהליך (או דרך ***exit****(3)* *או דרך* return מה main של התוכנית).
- הפונקציות שמוגדרות נקראות בסדר הפוך לסדר הרישום שלהן.
- אין ארגומנטים שמועברים.
- אותה פונקציה יכולה להיות "רשומה" מספר רב של פעמים ונקראת פעם אחת לכל "רישום".
- הפונקציה ;()**atexit** מחזירה את הערך 0 אם הצליחה, אחרת היא מחזירה ערך שונה מאפס.
- _SC_ATEXIT_MAX – קבוע עבור הפונקציה sysconf שמכיל את מספר הפונקציות המקסימלי שאפשר לרשום ל ;()atexit

Atexit - example

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
 void fnExit1 (void) {
     puts ("Exit function 1.");
}

void fnExit2 (void) {
     puts ("Exit function 2.");
}

int main () {
     atexit (fnExit1);
     atexit (fnExit2);
     atexit (fnExit2);
     puts ("Main function.");
     return 0;
}
```

Output:

Main function.
Exit function 2.
Exit function 2.
Exit function 1.

t3_3.c

Atexit: כמה פונקציות יכולות לעשות רישום ל exit

בדיקה כמה פונקציות יכולות לעשות רגיסטר ל exit

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

void bye(void) {
    printf("Let's go home, we learned enough today\n");
}

int main(void)
{
    long a; int i;
    a = sysconf(_SC_ATEXIT_MAX);  /*get configuration information at run time*/
    printf("ATEXIT_MAX = %ld\n", a);
    i = atexit(bye);
    if (i != 0)
    {
        fprintf(stderr, "cannot set exit function\n");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

דוגמא מסכמת exit v.s _exit

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

void done(){
    printf("see ya!\n");
}
int main(){
    int status;
    atexit(done);
    if(fork())
    {       /*Parent*/
            wait(&status);/*status will get the 55 value from the child exit(55)*/
            printf("parent PID = %d\n",getpid());
            printf("exit status= %d\n",WEXITSTATUS(status));
            _exit(73); /*This _exit will not call the done()*/
    }else
    {       /*Child*/
            sleep(2);
            printf("child PID = %d\n",getpid());
            exit(55); /*The exit will perform the done() function as well as part of the exit)
    }
}
```

```
child PID = 31015
see ya!
parent PID = 31014
exit status= 55
```

זהירות זומבים!!!

- כאשר בן מסתיים והאב לא המתין לו, הבן הזה נחשב זומבי...