# מערכות הפעלה דו"ח מסכם - עבודה 1

מאור אסייג 318550746

רפאל שטרית 204654891

המחלקה להנדסת חשמל ומחשבים, התכנית להנדסת מחשבים

מערכות הפעלה להנדסת מחשבים 202.1.3071

## מטרות העבודה

במהלך הקורס אנו נלמד לעבוד עם מערכת ההפעלה xv6. מערכת זו מכילה את העקרונות החשובים של המבנה הארגוני של UNIX. אנו מסמלצים את מערכת ההפעלה הזו על גבי מערכת הפעלה LINUX בעזרת האימולטור QEMU.

## סקירת תשובות

### 1. חימום והכרת סביבת העבודה

This part of the assignment is aimed at getting you started. It includes a small change in xv6 shell. Note that in terms of writing code, the current xv6 implementation is limited: it does not support system calls you may use when writing on Linux and its standard library is quite limited.

- הרצנו את סביבת העבודה (מערכת הפעלה של לינוקס) . לאחר מכן ייבאנו את הקבצים של מערכת ההפעלה xv6 וקימפלנו. לאחר מכן סימלצנו את מערכת ההפעלה בעזרת האימולטור של qemu עם הפקודה make qemu.

כפי שראינו בתרגול ניתן גם להריץ קבצי c נוספים (רק לדאוג להוסיפם לקובץ ה makefile) – ראינו בתרגול שימוש פשוט של הדפסה לטרמינל "Hello xv6".

בשלב זה ניתן להריץ את הפקודה רק מתיקיית האב.

```
int main(int argc, char *argv[]){
    printf(2, "HelloXV6!\n");
    exit(0);
}
```

הקובץ לא מצורף לפרוייקט בכדי להימנע משגיאות קומפילציה מיותרות – הוא אינו מטרתנו בעבודה. הקוד זהה למה שראינו בתרגול (כמובן שהוספנו int לחתימת ה exit כפי שנראה בהמשך).

# 2. תמיכה במשתנה הסביבה PATH

**Support the PATH environment variable :**
When a program is executed, the shell seeks the appropriate binary file in the current working directory and executes it. If the desired file does not exist in the working directory, an error message is printed.

A ***"PATH"*** is an environment variable which specifies the list of directories where commonly used executables reside. If, upon typing a command, the required file is not found in the current working directory, the **shell** attempts to execute the file from one of the directories specified by the **PATH**. An error message is printed only if the required file was not found in the working directory or any of the directories listed in the ***PATH***.

Your first task is to add support for the ***PATH*** environment variable. This environment variable should be controlled by the **shell** (i.e., by a user space program called **sh.c**). First you should add a global PATH variable to the shell which should contain a list of directories where the shell should search for executables (for the sake of simplicity the list of directories should be limited to 10 entries). Next, "**set *PATH***" command should be added to the shell. It will change the value of PATH variable. Each directory name listed should be delimited by a colon **(':')**.

For example, if we wanted to add the root directory and the bin directory to the PATH variable one could use following command:
**set PATH */:bin/:***

Every time set ***PATH*** command is called, the new paths will replace the existing values of ***PATH***. Finally, the **shell** must be aware of ***PATH*** environment variable when executing a program.

The first place to seek the binary of the executed command should be the current working directory, and only if the binary does not exist, the shell must search it in directories defined by the ***PATH***. The list of directories can be traversed in random order and must either execute the binary (if it is found in one of the directories) or print an error message in case the program is not found.

Note that the user can execute a program by providing to the shell absolute path (i.e., the path which has '/' as its first character) or relative path. The search for the binary should be executed only on **relative paths**

Test your implementation by executing a binary which does not reside in current working directory but pointed by PATH. The tests must include commands which uses I/O redirection (i.e., file input/output and pipes).

- הוספנו תמיכה בPATH כלומר , במקום שנבדוק האם היעד שלנו (קובץ או תיקייה) נמצא בworking directory , אנחנו מחפשים בתיקיות נוספות שהם ב environment variable

```
case EXEC:
  ecmd = (struct execcmd*)cmd;
  if(ecmd->argv[0] == 0)
      exit(0);
  exec(ecmd->argv[0], ecmd->argv);
  char path[PATHSIZE];
  for (int i = 0; i < pathNum; ++i) {
    memmove(path,PATHS[i],strlen(PATHS[i]));
    memmove(path + strlen(path), ecmd->argv[0], strlen(ecmd-
>argv[0]));
    exec(path, ecmd->argv);
  }
```

אפשר לראות את המעבר שלנו על הenvironment variable בלולאת for , אנחנו בודקים בעזרת absolute path של כל environment variable האם יש לו את היעד שלנו .

- הוספנו תמיכה בהוספת ה environment variables , ההוספה מתבצעת כאשר נקלטת בטרמינל הפקודה set PATH. בנוסף מצורף מהם הpaths החדשים, נזכיר כי בכל כתיבה של הפקודה set path נמחקים ה environment variables ומוספים החדשים. המבנה של הוספת ה Path הוא :/ ואחרי זה השם של הabsolute path . לדוגמא :/:bin/ . במצב כזה רק ה path "bin" יהיה במערך הenvironment variables .

```
if(strcmp(buf2,"set PATH") == 0){
 char* StringOfPaths =buf + 9;
 int index = 0;
 pathNum = 0;
 while(*StringOfPaths != '\0' && pathNum < 10){
   if(*StringOfPaths ==':'){
     buf2[index] = '\0';
     strcpy(PATHS[pathNum],buf2);
     index = 0;
     pathNum++;
     printf(2,"Path number %d was added :
%s\n",pathNum,buf2);
   }else{
     buf2[index] = *StringOfPaths;
     index++;
   }
   StringOfPaths++;
 }

}
```

ניתן לראות שאנחנו רצים על string שהוכנס לנו , ומחפשים את הדפוס של כתיבת המסלולים (אפשר לראות ב if) ,ומוסיפים את המסלול למערך המסלולים שלנו , ומדפסים לשם בדיקה את מספר המסלול המוסף את המסלול.

הוספנו PATH לדוגמה :

```
QEMU
echo            2 4 12660
forktest        2 5 8144
grep            2 6 15560
init            2 7 13520
kill            2 8 12728
ln              2 9 12692
ls              2 10 14796
mkdir           2 11 12808
rm              2 12 12784
sh              2 13 24660
stressfs        2 14 13624
usertests       2 15 57784
wc              2 16 14204
zombie          2 17 12448
tee             2 18 13824
getpinfo        2 19 12484
console         3 20 0
txt2            2 22 8
$ set PATH /:bin/:a/:b/:
Path number 1 was added : /
Path number 2 was added : bin/
Path number 3 was added : a/
Path number 4 was added : b/
$ _
```

כעת אם ניצור תיקיות בשם a ו b ונכנס לתוכם ע"י cd עדיין נוכל להשתמש בפקודות הטרמינל אף על פי שהקבצים שמתארים אותן אינם נמצאות בתיקיות אלו אלא בתיקיות האב (מכיוון שהוספנו את תיקיית האב ל PATH)

```
$ mkdir a
$ cd a
$ mkdir b
$ ls
.               1 21 48
..              1 1 512
b               1 23 32
$ cd b
$ ls
.               1 23 32
..              1 21 48
$ _
```

# 3. הרחבת הפונקציונליות של xv6

## א. חימום לפני System calls

In order to enrich the functionality of the xv6 you are required to add a user space program called ***tee***. This user space program receives a single argument or two arguments.

If it received **single arguments** a filename. It should read from standard input and write to both standard output and the file provided as argument. Consider following command executed by shell:
***tee tmp.txt***

It must read the standard input and write it to standard output and file tmp.txt both of them If it received 2 arguments, it should read from filename and write to the second file that provided as second argument. Consider following command executed by shell:
**tee tmp1.txt tmp.txt**
It will read from ***tmp1.txt*** and write to ***tmp*.txt** only .

- בעצם התבקשנו להוסיף את הפונקציה מסוג tee שמקבלת או קובץ אחד או שני קבצים :
  מימשנו באמצעות קובץ חדש tee.c שיצרנו, והוספנו אותנו לכל מקום בו נדרשת תמיכה עבור הפקודה. בנוסף, הוספנו אותו לקובץ הקמפול makefile.

קובץ אחד

כתבנו לתוך buff ומשכנו את הקובץ בעזרת הפונקציה open (וזרקנו שגיאות במקרי קצה) ואז כתבנו לו בעזרת הפונקציה write.

```
if (argc == 2){
    gets(buff,512);
    printf(2,"\n");
    dest=open(argv[1], O_CREATE | O_RDWR);
    if (dest < 0 ){
        printf(2,"tee failed : destination file not found");
        exit(0);
    }

    write_status=write(dest,buff,sizeof(char)*strlen(buff));

    if (write_status < 0 ) {
        printf(2, "tee failed : destination file not
writable");
        exit(0);
    }
    printf(2,buff);
```

## שני קבצים

במקרה הזה התבקשנו להעביר את המידע שבקובץ 1 לקובץ השני, אז קראנו את הקובץ הראשון , יצרנו חיבור לקובץ השני וכתבנו לקובץ השני.

```
if (argc == 3){
    src = open(argv[1], O_CREATE | O_RDWR);
    if (src < 0 ) {
        printf(2, "tee failed : source file not found");
        exit(0);
    }
    read_status=read(src,buff,sizeof(char)*512);
    if (read_status < 0 ) {
        printf(2, "tee failed : source file not readable");
        exit(0);
    }
    dest=open(argv[2], O_CREATE | O_RDWR);
    if (dest < 0 ) {
        printf(2, "tee failed : destination file not
found");
        exit(0);
    }

    write_status=write(dest,buff,sizeof(char)*strlen(buff));
    if (write_status < 0 ) {
        printf(2, "tee failed : destination file not
writable");
        exit(0);
    }

}
```

- כעת נבצע הדגמה של הפקודות :

**קובץ אחד** : כתבנו לקובץ example.txt את השורה הבאה בטרמינל. מכיוון שלא קיים קובץ כזה במערכת, המערכת תיצור אותו ותכניס לתוכו את השורה שנכתבה בטרמינל.

```
$ tee example.txt
temp text for example.txt

temp text for example.txt
Writing successful :-)
$
```

בדיקת נוכחות לקובץ :

```
$ ls
.               1 23 48
..              1 21 48
example.txt     2 24 26
```

2 קבצים : המערכת תיצור את example2.txt ותכתוב לתוכו את תוכן example1.txt

```
$ tee example.txt example2.txt
Writing successful :-)
$ ls
.                1 23 64
..               1 21 48
example.txt      2 24 26
example2.txt     2 25 26
$ _
```

ניתן גם למחוק את הקבצים בצורה תקינה :

```
$ ls
.                1 23 64
..               1 21 48
example.txt      2 24 26
example2.txt     2 25 26
$ rm example.txt
$ ls
.                1 23 64
..               1 21 48
example2.txt     2 25 26
$
```

## ב. הוספת קריאת מערכת getpinfo

**getpinfo** is system call the print a list of currently process, it's displays information about the process

| <Number of Row> | <Process ID> |
|---|---|
| 0 | 28 |
| 1 | 23 |

Pay attention, when you add/change a system call, you must update both kernel sources and user space programs sources.

- נעזרנו בדוקומנטציה של MIT בכדי להבין היכן יש לבצע שינויים כדי לתמוך בפקודת מערכת חדשה. הדבר כולל שינוי כל קובץ שיש בו פירוט על הprocess הקיימים כמו zombie. בנוסף ביצענו שינוי בקובץ הקמפול לתמוך גם בקובץ getpinfo.c

```
$ getpinfo
<Number of Row>  <pif>
      0            1
      1            2
      2            19
$
```

# 4. קריאות מערכת wait ו exit

In most of the operation systems the termination of a process is performed by calling an exit system call. The exit system call receives a single argument called **"status",** which can be collected by a parent process using the wait system call , If a process ends without an explicit call to exit, an implicit call to exit is performed with the status obtained from the return value of the main function.

**This is not the case in xv6** – the exit system call does not receive a status and the wait system call does not return it, In addition, no implicit call to exit is performed. The following task will modify xv6 in order to support this common behaviour. In this part you are required to extend the current kernel functionality so as to maintain an **exit status** of a process and to endow the kernel with an ability to make an **implicit system call exit** when the process is done.

First, you must **add a field** to the process control block PCB ***(see proc.h – the proc structure)*** in order to save an exit status of the terminated process. Next, you have to change all system calls affected by this change ***(i.e., exit and wait).***

Finally, you must endow the current implementation of the exec system call with the ability to make an implicit system call exit at the time the process is exiting its main function without explicitly calling the exit system call.

## א. עדכון פונקציית exit()

Change the exit system call signature to void exit(int status). The exit system call must act as previously defined (i.e., terminate the current process) but it must also store the exit status of the terminated process in the proc structure.

• In order to make the changes in the exit system call you must update the following files: user.h, defs.h, sysproc.c, proc.c and all the user space programs that use the exit system call.

• Note, you must change all the previously existing user space programs so that each call to exit will be called with a status equal to 0 (otherwise they will fail to compile).

- הוספנו לכל אובייקט תהליך (process) מאפיין נוסף שהינו status. שיננו את חתימת הפונקצייה לקבל int. ביצענו שינויים בכל קבצי ה header כדי לתמוך בחתימה החדשה של exit \ במאפיין הנוסף status.

בתוך קובץ proc.c :

```
exit(int Astatus)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

  if(curproc == initproc)
    panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;
  curproc->status=Astatus;
```

הסטטוס מסמל האם יצאנו מתהליך בגלל שהוא הסתיים או בשל שגיאה.

בתוך proc.h מתארים את הstruct של כל תהליך :

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int status;
};
```

וכמובן עדכון חתימת כל המקומות בהם מוזכרת הפונקציה exit.

## ב. עדכון פונקציית wait()

Update the wait system call signature to int wait(int *status). The wait system call must block the current process execution until any of its child processes is terminated (if any exists) and return the terminated child exit status through the status argument.

• The system call must return the process id of the child that was terminated or -1 if no child exists (or unexpected error occurred).

• Note that the wait system call can receive NULL as an argument. In this case the child's exit status must be discarded.

• Note that like in task 4.1 (exit system call) you must change all the previously existing user space programs so that each call to wait will be called with a status equal to 0 (NULL) (otherwise they will fail to compile).

- פקודה זו ממתינה לכל הבנים שכרגע במצב ריצה, ורק לאחר שסיימו תוכנית האב תמשיך. כפי שנתבקש שיננו את הפונקציה כך שתחזיר את מספר הסטטוס שהתקבל מ exit (קיבלנו זאת מ exit על ידי מצביע לפני שהבן סיים או יצא עקב שגיאה).

```
int
wait(int *Astatus)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
      if(Astatus!=0)
          *Astatus = p->status;
      release(&ptable.lock);
      return pid;
    }
  }

  // No point waiting if we don't have any children.
```