Operating System
Assignment 1

Eran Aflalo 209343722

Omer Luxembourg 205500390

Task 1: Warm up ("HelloXV6"):

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.

We created the following C program and add it to the Makefile – added myprog.c to 'EXTRA' section and _myprog to 'UPROG' section.

Results below:

Task 2: Support the PATH environment variable:

In this task we edit *sh.c* to maintain an array of 10 paths which are variable length up to 100 chars, that will be the environment paths as we know from Linux.

First, we define a global array for the path:

```
char env_path[10][100];
```

Next we added to the 'main' function the following code:

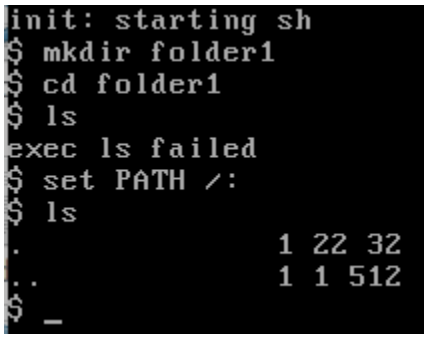
```
    // Read and run input commands.
    while (getcmd(buf, sizeof(buf)) >= 0) {
        if (buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' ') {
            // Chdir must be called by the parent, not the child.
            buf[strlen(buf) - 1] = 0; // chop \n
            if (chdir(buf + 3) < 0)
                printf(2, "cannot cd %s\n", buf + 3);
            continue;
        }
        // -- start edit 9/11 --
        // set PATH /:/bin
        if (buf[0] == 's' && buf[1] == 'e' && buf[2] == 't' && buf[3] == ' ' &&
            buf[4] == 'P' && buf[5] == 'A' && buf[6] == 'T' && buf[7] == 'H' && buf[8] == ' ') {
            // Set must be called by the parent, not the child.
            buf[strlen(buf) - 1] = 0; // chop \n
            char * paths_line = buf + 9; // now its only the paths
            int paths_len = strlen(paths_line);
            for (int i = 0; i < 10; i++) {
                if (paths_len != 0) {
                    int s_end = 0;
                    while ((paths_line[s_end] != ':') & (strlen(paths_line) > s_end)) {
                        env_path[i][s_end] = paths_line[s_end];
                        s_end++;
                    }
                    if (env_path[i][s_end-1] == '/') {
                        env_path[i][s_end] = '\0';
                    }
                    else{   // if no '/' add it to the path
                        env_path[i][s_end] = '/';
                        env_path[i][s_end+1] = '\0';
                    }
                    paths_line = paths_line + s_end + 1;        // it's a buffer - accumulate it by s_end+1
                    paths_len = paths_len - s_end;

                    // printf(1, "%d is: ", i);
                    // printf(1, "paths- '%s'  ", paths_line);
                    // printf(1, "path is- '%s'\n", env_path[i]);
                } else
                    memset(env_path[i], '\0', 100);
            }
            continue;
        }
        // -- end edit 9/11 --
        if (fork1() == 0)
            runcmd(parsecmd(buf));
        wait(0);
    }
    exit(0);
```

The code will allow us to run the command 'set PATH' and afterward the paths we want to add to the environment.

After this edition, we used the paths array to find commands to run from a current path, which are not the '*home*' path ("/") or order directories, which are not the current directory – the following code is in *sh.h* in the 'runcmd' function:

```
77          case EXEC:
78              ecmd = (struct execcmd * ) cmd;
79              if (ecmd -> argv[0] == 0)
80                  exit(0);
81              exec(ecmd -> argv[0], ecmd -> argv);
82              // -- start edit 9/11 --
83              if(ecmd -> argv[0][0] != '/'){      // if we are NOT looking for absolute path then search path array
84                  //printf(1, "We dont have absolute path... AND WE HOPE IT WILL RUN\n");
85                  for(int i=0; i<10; i++){
86                      int length = strlen(env_path[i]) + strlen(ecmd -> argv[0]);
87                      char str[length + 1];
88                      // store length of env_PATH in the length variable
89                      length = 0;
90                      while (env_path[i][length] != '\0') {
91                          str[length] = env_path[i][length];
92                          length ++;
93                      }
94                      str[length] = '/';
95                      // concatenate argv[0] to str
96                      for (int j = 0; ecmd -> argv[0][j] != '\0'; j++, length++) {
97                          str[length] = ecmd -> argv[0][j];
98                      }
99                      // terminating the str string
100                     str[length] = '\0';
101                     exec(str, ecmd -> argv);
102                 }
103             }
104             //else
105             //printf(1, "We did absolute path, BUT IT FAILED RUNNING...\n");
106             // -- end edit 9/11 --
107             printf(2, "exec %s failed\n", ecmd -> argv[0]);
108             break;
```

Results below:



Before setting the PATH, we can't use the *ls* command, because it is in the *home* directory. After we added the *home* directory, which is "/", we could run the command *ls*.

* note: if we run *"/ls"* for the directory *"/folder1"*, the command will work, as requested.

Task 3: Extend Functionality of XV6

3.1 We added user space program called *tee* as requested.

We created a C program and added it to the Makefile. The whole code is in *tee.c*.

The first section is for a single file:

```c
int main(int argc, char *argv[])
{
    if(argc == 2){
        char buf_read[512]; // creates a buffer
        if (open(argv[1], 0)>=0)    // a file already exists
            unlink(argv[1]);        // delete the file if exists...
        int fd_txt1 = open(argv[1], O_WRONLY | O_CREATE);    // create new fd if not exists, to write only
        while(1){
            int n = read(0, buf_read, sizeof(buf_read));    // return number of bytes
            if(n==0)
                break; // end of file
            else if(n<0) { // error in read
                printf(2, "read error\n");
                break;
            }
            else if(n>= 2 && buf_read[n-2]==3)
                break;// ctrl + c
            else {//write to file and console
                if (write(fd_txt1, buf_read, n) != n) { // wrriten bytes not equal to n = error!
                    printf(2, "write error\n");
                    break;
                }
                if (write(1, buf_read, n) != n) { // wrriten bytes not equal to n = error!
                    printf(2, "write to console error\n");
                    break;
                }
            }
        }
        close(fd_txt1); // closing file descriptor
```

The results of a single file:

Echo after each Enter

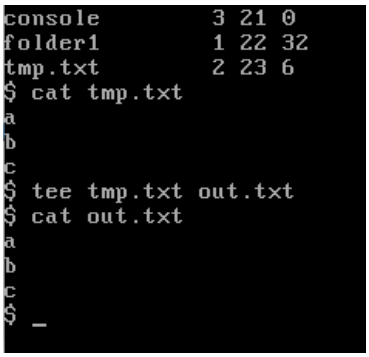
This is CTRL+C and then Enter

```
init: starting sh
$ tee tmp.txt
a
a
b
b
c
c
♥
$ cat tmp.txt
a
b
c
$ _
```

For two files we created the following code:

```
}else if(argc == 3){
    char buf_read[512]; // creates a buffer
    int fd_txt1 = open(argv[1], O_RDONLY);
    if (open(argv[2], 0)>=0)    // a file already exists
        unlink(argv[2]);        // delete the file if exists...
    int fd_txt2 = open(argv[2], O_WRONLY | O_CREATE);    // create new fd if not exists, to write only
    while(1){
        int n = read(fd_txt1, buf_read, sizeof(buf_read));    // return number of bytes
        if(n==0)
            break; // end of file
        else if(n<0) { // error in read
            printf(2, "read error\n");
            break;
        }
        else {//write to file and console
            if (write(fd_txt2, buf_read, n) != n) { // wrriten bytes not equal to n = error!
                printf(2, "write error\n");
                break;
            }
        }
    }
    close(fd_txt1); // closing file descriptor
    close(fd_txt2); // closing file descriptor
}else
    printf(2, "cannot tee, given %d arguments\n", argc);
exit(0);
}
```

The results are as follows:



As we can see the content of *tmp.txt* is in *out.txt* now.

3.2 *getpinfo* is a system call that print a list of currently running processes.

To implement it, we created a user space program and a kernel system call as follows:

Use space program only calls the system call –

```c
#include "types.h"
#include "stat.h"
#include "user.h"

#include "fcntl.h"

//passing command line arguments

int main(int argc, char *argv[])
{
    getpinfo();
    exit(0);
}
```

To create the kernel system call we added the following files with appropriate function calls and macros:

user.h ysys.S syscall.c syscall.h

In addition, we created the following function to *proc.c* file:

```c
void
getpinfo(void)
{
    struct proc *p;
    int i = 0;

    acquire(&ptable.lock);
    cprintf("<Number of Row>\t\t<Process ID>\n");
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if (p->state != UNUSED) {
            cprintf("\t%d\t\t\t%d\n", i, p->pid);
            i++;
        }
    }
    release(&ptable.lock);
}
```

Results:

```
init: starting sh
$ getpinfo
<Number of Row>          <Process ID>
        0                        1
        1                        2
        2                        3
$
```

* Note that the QEMU terminal does not identify “\t” as a regular tab.

Task 4: Wait and exit system calls

4.1 We updated the process structure with a status field in '*proc.h*' (and updated *user.h, defs.h, sysrpoc.c, proc.c*). In addition, we edited *exit* function (in *proc.c*) signature to receive an int status.

proch.h – added a field

```
// 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)
  // -- start edit 13/11 --
  int status;                  // added status as int
  // -- end edit 13/11 --
};
```

proc.c – exit function

```
void
exit(int status)    // 13/11 changed from void to int
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;
```

...

```
  // -- start edit 13/11 --
  curproc->status = status;      // assigning the status to the current process
  // -- end edit 13/11 --

  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

sysproc.c –

```
sys_exit(void)
{
    // -- start edit 13/11 --
    int status;
    if(argint(0, &status) < 0)
        return -1;
    exit(status);
    // -- end edit 13/11 --
    //exit();
    return 0;  // not reached
}
```

After changing *exit* function, we changed "*exit()"* to *"exit(0)"* where needed.

4.2 Updating the wait system call. We changed *wait* system call to resolve with the status given in exit.

Same as *exit* we updated the files:

proc.c -

```
// Return -1 if this process has no children.
int
wait(int *status)   // 13/11 changed from void
{
    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;
            // -- start edit 13/11 --
            if (status != 0)    // if status is NULL we cannot assign value to it
                *status = p->status;
            // -- end edit 13/11 --
            release(&ptable.lock);
            return pid;
        }
    }
```

The status in this function is a pointer to status field in ‘p’ process.

sysproc.c –

```
int
sys_wait(void)
{
    // -- start edit 13/11 --
    int *status;
    if(argptr(0, (void*)&status, sizeof(int*)) < 0)
        return -1;
    return wait(status);
    // -- end edit 13/11 --
    //return wait();
}
```

We created a user space program to test the functionality of the status with wait and exit. The following results show the status value that got from the function by calling “*exit(57)*”.

This is the user space program “wait_test”:

```c
int main(int argc, char *argv[])
{
    int pid = fork();
    if(pid == 0){
        printf(1, "I'm the child!\n");
        exit(57);
    } else if(pid < 0){
        printf (1, "This is fork failed\n");
        exit(0);
    }
    int status;
    wait(&status);
    printf(1, "I'm the parent, and got child status: %d\n", status);
    exit(0);
}
```

The terminal results:

```
init: starting sh
$ wait_test
I'm the child!
I'm the parent, and got child status: 57
$ _
```

4.3 Updating the wait system call – we adjust the “*exec.c*” file as requested.

exitf() – will create the function for the stack.

```c
void
exitf()
{
    asm volatile ("push %eax;");
    asm volatile ("pop 4(%esp);");
    asm volatile ("mov %0 , %%eax" : : "i" (SYS_exit));
    asm volatile ("int %0" : : "i" (T_SYSCALL));
}
```

In the *exec* function we updated the following:

```c
int
exec(char *path, char **argv)
{
    char *s, *last;
    int i, off;
    uint argc, sz, sp, ustack[3+MAXARG+1];
    struct elfhdr elf;
    struct inode *ip;
    struct proghdr ph;
    pde_t *pgdir, *oldpgdir;
    struct proc *curproc = myproc();

    //STEP 1 exit syscall size(byte)
    int exitsz = (int)exec - (int)exitf;
```

```
//Step 2 - stack pointer update and copy page from sp to exitf
sp = sz - exitsz;
copyout(pgdir, sp, exitf, exitsz);

// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
        goto bad;
    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
        goto bad;
    ustack[3+argc] = sp;
}
ustack[3+argc] = 0;

ustack[0] = sz - exitsz;  // NO fake return PC - return address of user space program
ustack[1] = argc;
ustack[2] = sp - (argc+1)*4;  // argv pointer
```

To test it we emitted the “exit(0)” in the user space program “wait_test.c” that we created before:

Makefile | wait_test.c

```
#include "types.h"
#include "stat.h"
#include "user.h"

#include "fcntl.h"

//passing command line arguments

int main(int argc, char *argv[])
{
  int pid = fork();
  if(pid == 0){
    printf(1, "I'm the child!\n");
    exit(57);
  } else if(pid < 0){
    printf (1, "This is fork failed\n");
    exit(0);
  }
  int status;
  wait(&status);
  printf(1, "I'm the parent, and got child status: %d\n", status);
  //exit(0);
}
```

QEMU - Press Ctrl-Alt to exit mouse grab

```
SeaBIOS (version 1.10.2-1ubuntu1)


iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF8DDD0+1FECDDD0 C980



Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ wait_test
I'm the child!
I'm the parent, and got child status: 57
$ 7_
```

As expected, the adjustment worked!