

מערכות הפעלה דו"ח מסכם - עבודה 2

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 מסמלים את מערכת ההפעלה הזו על גבי מערכת הפעלה LINUX בעזרת האימולטור QEMU. בעבודה זו נתמקד בסיגנלים.

1. Signal Framework

This implementation will support only a single signal handler but unlike the signals you learned about in the practical session, this signal handler can receive two numbers - the pid of the sending processes and the other is a number that is sent by the sending process. The signal framework that you will create includes 3

```
//declaration of a signal handler function
typedef void (*sig_handler)(int pid, int value);

//set the signal handler to be called when signals are sent
sig_handler sigset(sig_handler );

//send a signal with the given value to a process with pid dest_pid
int sigsend(int dest_pid, int value);

//complete the signal handling context (should not be called explicitly)
void sigret(void);
```

system calls

The following subsection describes in detail the above system calls together with their implementation requirements. You are required to follow the instructions given and implement the requested data-structures and system calls.

סקירת תשובות

2. Storing and Changing the signal handler

In order to store the signal handler, you will add a new field to struct proc (see proc.h). This field will hold a pointer to the current handler (or -1 if no handler is set). As described in practical session 3, both fork and exec system calls modify the signal handlers.

We will copy the signal related behavior of fork and exec for our signals implementation.

The fork system call will copy the parent process' signal handler to the newly created child processes. The exec system call will reset the signal handler to be the default(1-).

The new sigset system call will replace the process signal handler with the provided one and return the previously stored signal handler.

- הוספנו שדה סיגנל הנדלר ב proc.h

```
// 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 trapframe *oldtf; // the old Trap frame
    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;
    sig_handler sighandler; // Signal handler
    struct cstack cstack;   // signals stack
    int worksig;            // check if the process work on signal
}
```

- דאגנו להעתיק שדה זה כשאנו עושים fork

```
np -> sighandler = curproc -> sighandler;
```

- וגם כשאנו עושים exec

```
curproc -> sighandler = (sig_handler)-1;
```

העברנו כתובת דיפולטית שלא מסמנת 1\0 וכו.

- מימוש sigset כפי שנתבקשנו

```
sig_handler
sigset(sig_handler new_sig)
{
    struct proc *curproc = myproc();
    sig_handler old_sig = curproc->sighandler;
    curproc->sighandler = new_sig;
    return old_sig;
}
```

Sending a signal to a process.3

The new sigsend system call sends a signal to a destination process. When a signal is sent to a process it is not handled instantly since the destination process may be already running or even blocked. This means that each process must store all the signals which were sent to it but still not handled in a data structure that we will refer to as the pending signals stack, Since multiple processes can send signals to the same recipient then he must save the signals in structure.

```
// defines an element of the concurrent struct
struct cstackframe {
    int sender_pid;
    int receipient_pid;
    int value;
    int used;
    struct cstackframe *next;
};

// defines a concurrent stack
struct cstack {
    struct cstackframe frames[10];
    struct cstackframe *head;
};

// adds a new frame to the cstack which is initialized with values
// sender_pid, receipient_pid and value, then returns 1 on success and 0
// if the stack is full
int push(struct cstack *cstack, int sender_pid, int receipient_pid, int value);

// removes and returns an element from the head of given cstack
// if the stack is empty, then return 0
struct cstackframe *pop(struct cstack *cstack);
```

The sigsend system call will add a record to the recipient pending signals stack. It will return 0 on success and -1 on failure (if pending signals stack is

- מימוש sigsend

```
int sigsend(int dest_pid, int value) {
    //send signal
    struct proc *p;
    struct proc *curr = myproc();

    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->pid == dest_pid) {
            if (push(&p->cstack, curr->pid, dest_pid, value) == 0) {
                release(&ptable.lock);
                return -1; // not in the stack
            }
            release(&ptable.lock);
            return 0;
        }
    }
    release(&ptable.lock);
    return -1;
}
```

נעזרנו בפונקציה שקיימת ב xv6 myproc() שתחזיר מצביע ל process הנוכחי

```
// Disable interrupts so that we are not rescheduled
// while reading proc from the cpu structure
struct proc*
myproc(void) {
    struct cpu *c;
    struct proc *p;
    pushcli();
    c = mycpu();
    p = c->proc;
    popcli();
    return p;
}
```

ראשית תפסנו את המנעול לטבלת התהליכים, לאחר מכן רצנו על הטבלה עד שנמצא את התהליך אליו אנו רוצים לשלוח את הסיגנל (השוואת שדה pid ל dest_pid).

לאחר מכן דחפנו למחסנית את הסיגנל (הערך value), במידה והפעולה הצליחה push תחזיר 1 הפונקציה תסתיים (נשחרר את המפתח) ותחזיר 0. במידה והפעולה נכשלה אנו ניכנס ל if, נשחרר את המפתח ונחזיר -1.

מימוש מחסנית

הצהרה על מחסנית

```
struct cstackframe {
    int sender_pid;
    int receipient_pid;
    int value;
    int used;
    struct cstackframe *next;
};

// concurrent stack
struct cstack {
    struct cstackframe frames[10];
    struct cstackframe *head;
};
```

הסבר : נדרשנו לכתוב מחסנית בגודל 10. מימוש סטנדרטי עם head ו next. תפקיד השדה used זה לסמן האם הlink הנוכחי משומש או לא, והאם ניתן לעדכן בו דברים.

מימוש Push

```
int
push(struct cstack *cstack, int sender_pid, int receipient_pid, int value) { // push signal to panding queue
    struct cstackframe *newSig;
    for (newSig = cstack->frames; newSig < cstack->frames + 10; newSig++){
        if (newSig->used == 1){
            if (newSig < cstack->frames + 10) {
                newSig->sender_pid = sender_pid;
                newSig->receipient_pid = receipient_pid;
                newSig->used = 0;
                newSig->value = value;
                newSig->next = cstack->head;
                cstack->head = newSig;
                return 1;
            }else{
                return 0;
            }
        }
    }
    return 0;
}
```

הסבר : נרוץ על ה linked-list מהאיבר ה 0 עד האחרון, במידה ומצאנו link לא משומש נעתיק אליו את כל הפרמטרים ונחזיר 1. אחרת, נחזיר 0.

מימוש Pop

```

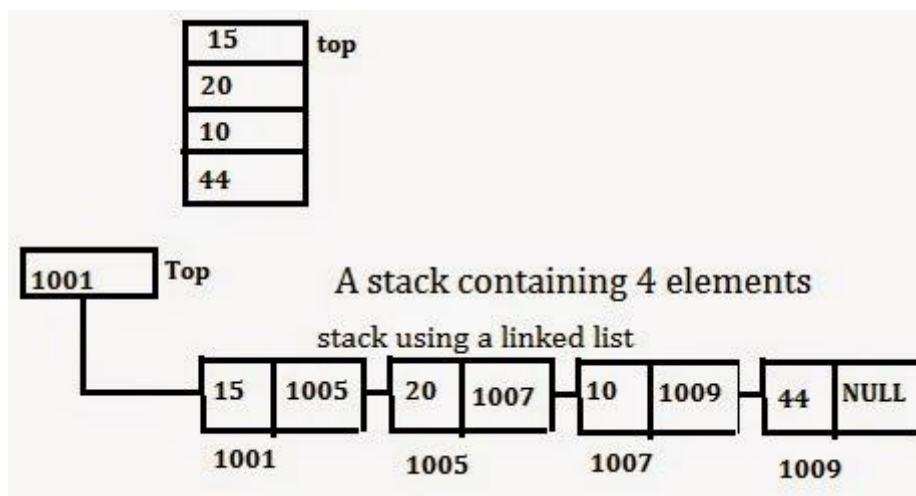
struct cstackframe
    *pop(struct cstack *cstack) {

    //pop the pending signal
    struct cstackframe *head;
    head = cstack->head;
    if (head != 0){
        cstack->head = head->next;

    }
    return head;
}

```

הסבר : נחזיר את ה link המשמש כ head ונגדיר את המצביע של ה head ללינק הבא ברשימה. מימוש סטנדרטי לחלוטין של מחסנית באמצעות רשימה מקושרת.



Signal Handling .4

When a process is about to return from kernel space to user space (using the function `trapret` which can be found at `trapasm.S`) it must check its pending signals stack. If a pending signal exists and the process is not already handling a signal (i.e., you should not support handling multiple signals at once) then the process must handle the signal.

The signal handling can be either discarding the signal (if the signal handler is default) or executing a signal handler when it returns to user space. In order to force the execution of the signal handler in user space you will have to modify the user space stack and the instruction pointer of the process. This requires knowledge of conventions of function call. You can refresh your your memory regarding function call conventions [here](#).

There are three major steps that must be performed in order to make a function call:

- Push the arguments for the called function on the stack
- Push the return address on the stack
- Finally jump to the body of the called function

Pushing arguments and the return address on the user space stack is a straightforward operation over the process trapframe once you understand the function call conventions.

In order to execute the body of the signal handler upon return to user space, one must update the instruction pointer's value to be the address of the desired function.

When the signal handler finishes its run, the user space program should continue from the point it was stopped before execution of the signal handler. Thus, one can naturally think that the return address that should be placed on the stack as the return address of the signal handler should be the previous instruction pointer (before changing it to point to the signal handler). However, this will not work. Since the signal handler can change the CPU registers values this can cause unpredictable behaviour of the user space program once jumping back to the original code.

In order to solve this problem, you must save the CPU registers values before the execution of the signal handler and restore them after the execution of the signal handler finishes. You should create a new field inside struct `proc` that will hold the original registers values. When the signal handler finishes, your code must return to

kernel space in order to restore them. This is the responsibility of the sigret system call, which will only restore the CPU registers values for the user space execution (you may backup the whole old trapframe in a new field inside struct proc).

The main problem here is that the signal handler can accidentally not call the sigret system call on exit and this may cause an unpredictable behavior in the user code. To solve this problem, you need to "inject" an implicit call to the sigret system call after the call to the signal handler. This can be done by putting this code onto the user space stack and setting the return address from

the signal handler to point to the beginning of the injected code. You can do so by using the memmove function which is located at string.c .

חימוש signalexe

```
void signalexe(void){
// exe the signal handler ,save the old state and reload the new state

    struct proc *curr = myproc();
    struct cstackframe *cstackframe1;
    if (curr == 0 || curr->worksig == 1 || (curr->tf->cs & 3) != DPL_USER )
        return;
    cstackframe1 = pop(&curr->cstack);
    if (cstackframe1 == (struct cstackframe *)0)
        return;
    if(curr->sighandler == (sig_handler)-1)
        return;
    curr->worksig = 1;
    memmove(curr->oldtf, curr->tf, sizeof(struct trapframe)); //backing up trap frame
    curr->tf->esp = curr->tf->esp - 8;
    memmove((void*)curr->tf->esp, callsigret, 8);

    *((int*)(curr->tf->esp-4)) = cstackframe1->value;
    *((int*)(curr->tf->esp-8)) = cstackframe1->sender_pid;
    *((int*)(curr->tf->esp - 12)) = curr->tf->esp;
    curr->tf->esp = curr->tf->esp - 12;
    curr->tf->eip = (uint)curr->sighandler; //setting the first instruction the be excuted after trapret
    cstackframe1->used = 1;
}
```

הסבר : בכדי שנוכל להתחיל בטיפול בסיגנל הנדלר, נרצה לוודא שכל הפרמטרים הקשורים אליו תקינים. נבדוק האם התהליך הנוכחי מטפל כעת בסיגנל, או האם מחסנית הסיגנלים הממתינים לביצוע ריקה, או האם הסיגנל הנדלר מצביע לכתובת זבל.

במידה וכל הבדיקות עברו, נוכל להתחיל לטפל בסיגנל. נעביר את מצב העבודה הנוכחי של התהליך ל 1 (worksig=1). נשמור את ה trapframe בזיכרון. נקדם את המצביע באוגר esp. נשמור בזיכרון גם את פונקציית החזרה שתפקידה לטעון מחדש את הזיכרון לאחר סיימנו את הטיפול בסיגנל, בדיוק כפי שלמדנו בתאוריה (callsigret).

```

1  #include "syscall.h"
2  #include "traps.h"
3
4  .globl callsigret; \
5  callsigret: \
6  movl $SYS_sigret, %eax; \
7  int $T_SYSCALL; \
8  ret
9

```

קובץ `sigretcall.S` הקורא לפונקציה `sigret` נעדכן את כל פרמטרי המערכת כשורה, כשנשים דגש על עדכון אוגר `eip` לכתובת הפונקציה שהוגדרה לסיגנל הנדלר - כך יתחיל ביצוע הסיגנל הנדלר שהמשתמש קבע.

חימוש `signalret`

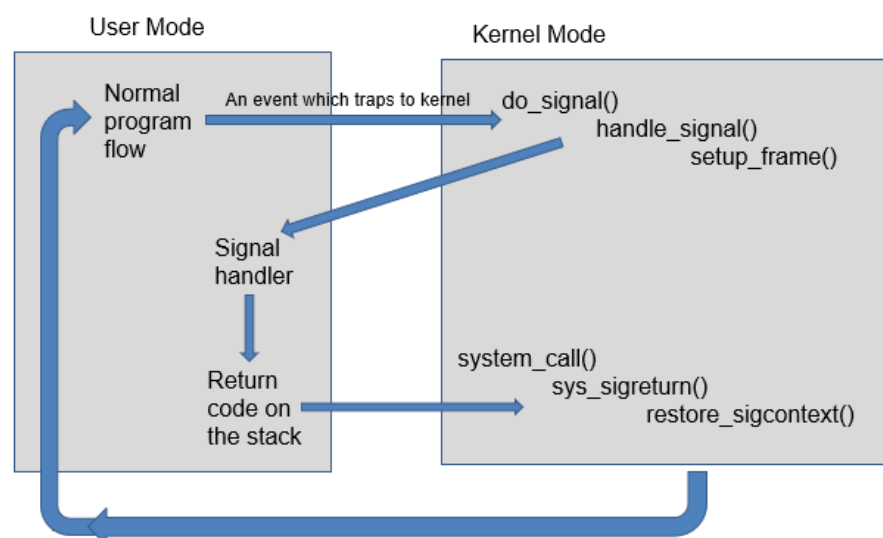
```

void sigret(void) {
    struct proc *curr = myproc();
    memmove(curr->tf, curr->oldtf, sizeof(struct trapframe));
    curr->worksig = 0; // enable handling next pending signal
}

```

הסבר : נטען חזרה את מצב התוכנית לפני הסיגנל הנדלר, ונעביר את מצב התהליך הנוכחי ל-0 (`worksig=0`) כלומר ניתן להמשיך ולטפל בסיגנלים.

Scheme of signal processing



בדיקות עצמיות - טסטים

```
int main(int argc, char *argv[]) {
    if(argc != 2){
        printf(1, "error argc is :%d instead of 2\n",argc);
        exit(num);
    }

    numOfProcess = atoi(argv[1]);
```

לקובץ הטסטים קראנו Tests.c, כשמריצים אותו מהטרמינל מתבקשים להכניס את מספר התהליכים שירוצו במקביל עבור כל טסט, לדוגמה 3,4 וכו'.

1. טסט inittest

```
printf(1, "\n");
printf(1, "init test: create %d process without sigset ,default handler\n",numOfProcess);
printf(1, "\n");

inittest();

void inittest(){
    int pid;
    int i;
    for (i = 0; i < numOfProcess; i++){
        pid = fork();
        if (pid == 0) { // child
            sleep(50);
            exit(num);
        } else { // father
            sigsend(pid,getpid());
            wait(&num);
        }
    }

    return;
}
```

הסבר : טסט זה הינו הטסט הראשון לבדיקה, ומשמש כאבן דרך בבדיקה עבור פעילות תקינה. אנו בודקים את פונקציונליות fork (יצירת תהליכים תקינה) וגם את sigsend עם parent pid. חשוב שטסט זה יעבור ללא יצירת זומבים (פעילות wait). המשתנה num הינו משתנה גלובלי עם הערך 0 בשל החתימה של הפונקציות wait, exit.

תוצאה : עבר בהצלחה לטסט הבא

```
init test: create 3 process without sigset ,default handler
test send signals : create 3 process with fork() and call sigsend
```

2. טסט sendsigtest

```
printf(1, "\n");
printf(1, "test send signals : create %d process with fork() and call sigsend \n", numOfProcess);
printf(1, "\n");

sendsigtest();

void sendsigtest(){
    sigset((sig_handler)&sendsigtesthandler);
    int pid;
    int i;
    for (i = 0; i < numOfProcess; i++){
        finish=1;
        pid= fork();
        if (pid == 0) { // child code
            while(finish){
                sleep(50);
            }

            exit(num);
        } else { // father code
            sleep(16);
            sigsend(pid, pid);
            wait(&num);
        }
    } // end of for
    return;
}
```

הסבר : ניצור numOfProcess בנים בעזרת fork, כל ילד ילך לישון עד שדגל finish לא יורם ל 1. דגל זה מתעדכן בסיגנל הנדלר sendsigtesthandler. מטרת טסט זה הינה לבדוק את הפעילות התקינה של system call sigsetn ושל sigsend.

```
void sendsigtesthandler(int pid, int value){
    finish = 0;
    if( value == getpid() ){
        printf(1, "test succeed: got value: %d and its equal to getpid() value: %d\n", value, getpid());
        return;
    }

    printf(1, "test fail: got value: %d and its not equal to getpid() value: %d\n", value, getpid());
    return;
}
```

תוצאה : הבדיקה עברה בהצלחה והמשכנו לטסטים הבאים.

```
test send signals : create 3 process with fork() and call sigsend
test succeed: got value: 7 and its equal to getpid() value: 7
test succeed: got value: 8 and its equal to getpid() value: 8
test succeed: got value: 9 and its equal to getpid() value: 9
```

3. טסט exectest

```
printf(1, "\n");
printf(1, "test exec: create process and then use exec()\n", numOfProcess);
printf(1, "\n");

exectest();
```

```
void exectest(){
    sigset((sig_handler)&exectesthandler);
    int pid = fork();
    char *argv[2];
    argv[0] = "echo";
    argv[1] = "test passed: exec() set default handler";
    if(pid==0){
        exec("echo",argv);
    } else {
        sleep(100);
        sigsend(pid, 0); // dont work because exec change signal handler
        wait(&num);
    }
    return;
}
```

הסבר : נגדיר הנדלר exectesthandler ונבצע fork עם הפקודה exec בקוד של הבן. מהאבא נשלח sigsend ונבדוק האם התבצע הסיגנל הנדלר, או שהוגדר הסיגנל הנדלר הברירת מחדל כפי שצריכה להיות הפעילות התקינה של exec.

```
void exectesthandler(int pid, int value){
    printf(1, "test fail: after exec() default handler%d\n", value, pid);
    finish = 0;
}
```

תוצאה : הבדיקה עברה בהצלחה, לא נכנסו לסיגנל הנדלר והמשכו לטסט הבא.

```
test exec: create process and then use exec()
test passed: exec() set default handler
test proc to proc: create 3 process with fork(), send from process to process
```

4. טסט sendsigProc

```
printf(1, "\n");
printf(1, "test proc to proc: create %d process with fork(), send from process to process\n", numofProcess);
printf(1, "\n");

sendsigProc();
```

```
void sendsigProc(){
    sigset((sig_handler)&sendsigProchandler);
    int processtable[10];
    int i;
    for (i = 0; i < numofProcess; i++){
        finish=1;
        processtable[i] = fork();
        if (processtable[i] == 0) { // child code
            if(i>0){
                sigsend(processtable[i-1], processtable[i-1]);
            }else {
                sigsend(processtable[i], processtable[i]);
            }
        }
        while(finish){
            sleep(20);
        }
        exit(num);
    }
} // end of for
for (i = 0; i < numofProcess-1; i++)
    wait(&num);
return;
```

הסבר : שליחת sigsend מתוך הבן לבן הקודם שנוצר. בדיקה זו בודקת תקינות התהליך שרץ ואת נכונות sigsend. קבענו את הסיגנל הנדלר להיות sendsigProchandler

```

void sendsigProchandler(int pid, int value){
    //pid should be the right one
    int mypid = getpid();
    finish = 0;
    if( value == mypid ){
        printf(1, "test succeed: got value: %d from %d and its equal to getpid() value: %d\n",value,pid,mypid);
        return;
    }

    printf(1, "test fail: got value: %d and its not equal to getpid() value: %d\n",value,mypid);
    return;
}

```

תוצאה : הטסט עבר בהצלחה והמשכנו לטסט הבא.

```

test proc to proc: create 3 process with fork(), send from process to process
test succeed: got value: 11 from 12 and its equal tto getpiest succeed: got val
ue: 12 from 13 and its equd() value: 11
al to getpid() value: 12
test change handler: create 3 process with fork(), change the signal handler

```

5. טסט changeHandler

```

printf(1, "\n");
printf(1, "test change handler: create %d process with fork(), change the signal handler\n",numOfProcess);
printf(1, "\n");

changeHandler();

```

```

void changeHandler(){
    sigset((sig_handler)&changeHandlerhandlerbed);
    int i;
    for (i = 0; i < numOfProcess; i++){
        finish=1;
        int pid = fork();
        if (pid == 0) { // child code
            sigset((sig_handler)&changeHandlerhandler);
            while(finish){
                sleep(20);
            }
            exit(num);
        }else {
            sleep(29);
            sigsend(pid, 0);
            wait(&num);
        }
    }
    return;
}

```


הסבר : בטסט זה אנו בודקים האם כל ילד יכול לשנות את הסיגנל הנדלר שלו מתוך הקוד שלו. בדיקת פעילות תקינה של fork, sigsend, exit, wait, sigset.

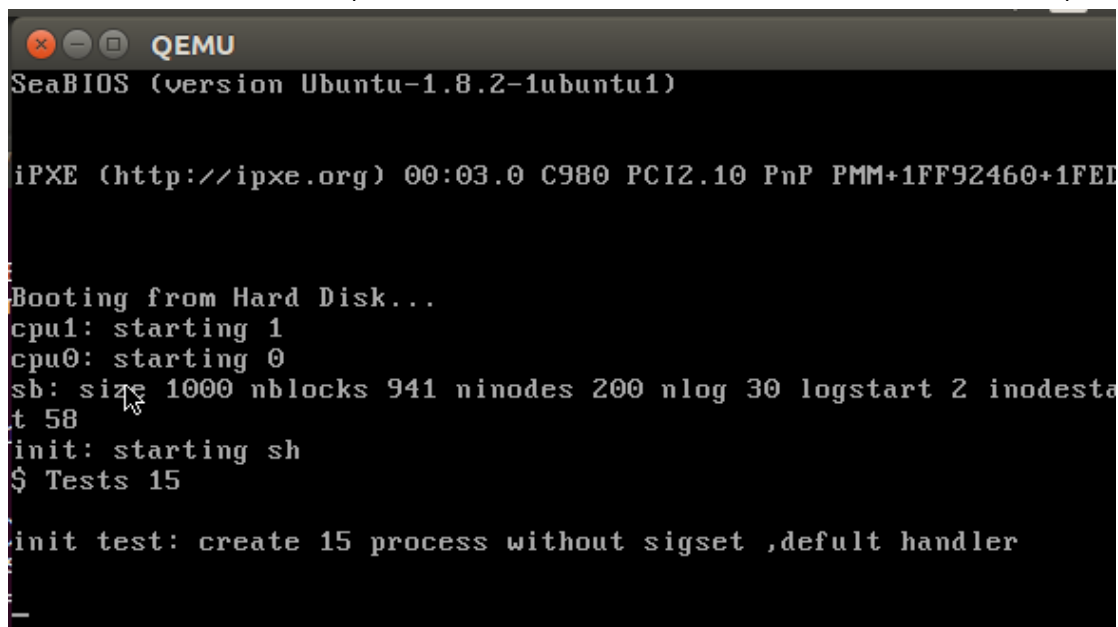
```
void sendsigProchandler(int pid, int value){
    //pid should be the right one
    int mypid = getpid();
    finish = 0;
    if( value == mypid ){
        printf(1, "test succeed: got value: %d from %d and its equal to getpid() value: %d\n",value,pid,mypid);
        return;
    }

    printf(1, "test fail: got value: %d and its not equal to getpid() value: %d\n",value,mypid);
    return;
}
```

תוצאה : בדקנו טסט זה עם 4 תהליכים לדוגמה, הטסט עבר בהצלחה.

```
test change handler: create 4 process with fork(), change the signal handler
test succeed : signal handler change in process 17
test succeed : signal handler change in process 18
test succeed : signal handler change in process 19
test succeed : signal handler change in process 20
test finished
```

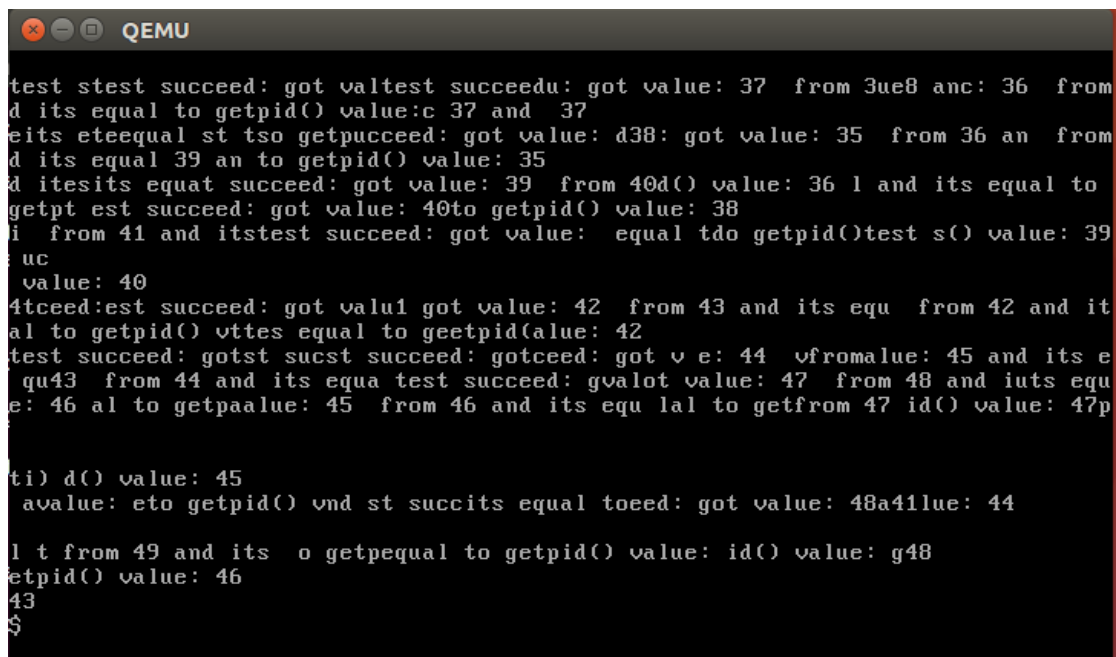
הערה : בדקנו את כלל הטסטים גם עם numOfProcess = 15 וקיבלנו :



```
QEMU
SeaBIOS (version Ubuntu-1.8.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF92460+1FED
Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodesta
t 58
init: starting sh
$ Tests 15

init test: create 15 process without sigset ,default handler
```



```

test test succeed: got valtest succeedu: got value: 37 from 3ue8 anc: 36 from
d its equal to getpid() value:c 37 and 37
eits eteequal st tso getpucceed: got value: d38: got value: 35 from 36 an from
d its equal 39 an to getpid() value: 35
d itesits equat succeed: got value: 39 from 40d() value: 36 l and its equal to
getpt est succeed: got value: 40to getpid() value: 38
i from 41 and itstest succeed: got value: equal tdo getpid()test s() value: 39
uc
value: 40
4tceed:est succeed: got valu1 got value: 42 from 43 and its equ from 42 and it
al to getpid() vttes equal to geetpid(alue: 42
test succeed: gotst sucst succeed: gotceed: got v e: 44 vfromalue: 45 and its e
qu43 from 44 and its equa test succeed: gvalot value: 47 from 48 and iuts equ
e: 46 al to getpaalue: 45 from 46 and its equ lal to getfrom 47 id() value: 47p

ti) d() value: 45
a value: eto getpid() vnd st succits equal toeed: got value: 48a41lue: 44

l t from 49 and its o getpequal to getpid() value: id() value: g48
etpid() value: 46
43
$

```

הסבר : הטסטים הסתיימו בהצלחה, אומנם בגלל שהעמסנו בהרבה פעולות
הדפסה ההדפסות "התבלגנו", אך קיבלנו תוצאה רצויה ולא קיבלנו panic או
שגיאות.