

Λειτουργικά Συστήματα Δραστηριότητα 1  
2024-2025

AM: 1096060

Ονοματεπώνυμο: Μαρία-Νίκη Ζωγράφου

## ΔΡΑΣΤΗΡΙΟΗΤΑ 1

### Άσκηση 1:

Βρείτε το αρχείο στον πηγαίο κώδικα του xv6 όπου ορίζονται οι κλήσεις συστήματος οι οποίες θα είναι διαθέσιμες σε προγράμματα χρήστη. Έπειτα, διαλέξτε ένα από τα προγράμματα χρήστη στον φάκελο user. Εξηγήστε τη λειτουργία του κάνοντας συγκεκριμένες αναφορές στα system calls που καλεί:

Στο αρχείο **user.h** ορίζονται όλα τα system calls τα οποία είναι διαθέσιμα στον χρήστη. Στα προγράμματα χρήστη γίνεται include user.h ώστε να έχει εκεί πρόσβαση ο χρήστης.

Ορισμός αυτών των συναρτήσεων γίνεται στο **usys.S**. Εκεί ο preprocessor της C αντικαθιστά το SYSCALL(name) με:

```
.globl name; \
name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \int instruction causes the processor to generate a trap [1]
    ret
```

Τοποθετεί δηλαδή το «όνομα» της συνάρτησης που καλεί ο χρήστης στον καταχωρητή eax, έτσι ώστε, όταν το int \$T\_SYSCALL κάνει interrupt και ο έλεγχος μεταφερθεί στο Kernel, να κληθεί η κατάλληλη συνάρτηση.

Στην πραγματικότητα στον καταχωρητή eax τοποθετείται ένας αριθμός που αντιστοιχεί στο όνομα της συνάρτησης. Αυτή η αντιστοίχιση γίνεται στο αρχείο syscall.h με χρήση define.

(Όσον αφορά το kernel, οι κλήσεις συστήματος γίνονται define στον φάκελο kernel στο αρχείο syscall.c όπου υπάρχει ένας πίνακας από function pointers για όλες τις συναρτήσεις που αποτελούν κλήσεις του συστήματος. Οι συναρτήσεις αυτές ορίζονται (γίνονται define) στα αρχεία sysfile.c, sysproc.c.)

### Εξήγηση λειτουργίας του ls.c:

Το ls.c τυπώνει τα αρχεία στο current path αν δεν δοθούν args αλλιώς τυπώνει τα αρχεία για κάθε Path που δόθηκε. (argc ο αριθμός των ορισμάτων και argv τα ορίσματα, αν είναι πολλά διαφορετικά Path το loop θα καλέσει την ls χωριστά για το καθένα).

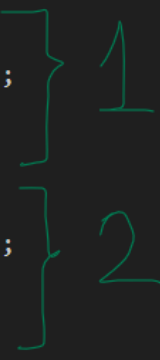
Στο τέλος και των δύο περιπτώσεων **καλείται η exit() system call**. Αφού έχει γίνει #include το user.h header (#include "user.h") ο linker μπορεί να συνδέσει το αρχείο με το usys.S. Η exit() τερματίζει το πρόγραμμα.

```
73 int
74 main(int argc, char *argv[])
75 {
76     int i;
77
78     if(argc < 2){
79         ls(".");
80         exit();
81     }
82     for(i=1; i<argc; i++)
83         ls(argv[i]);
84     exit();
85 }
```

Κλήση της ls:

1. Όταν καλείται η συνάρτηση ls() γίνεται, έλεγχος αν το path που δόθηκε μπορεί να «ανοιχτεί», αν όχι τότε γίνεται return στην main.
2. Ύστερα γίνεται κλήση της `fstat system call` (με την προϋπόθεση ότι ο πρώτος έλεγχος ήταν επιτυχής και υπάρχει fd). Από το βιβλίο ([1] Russ Cox, Frans Kaashoek και Robert Morris. The Design of the xv6 x86 Operating System. 2019), γνωρίζουμε ότι η fstat επιστρέφει πληροφορίες για το αντικείμενο στο οποίο αναφέρεται το file descriptor. Άμα επιστραφεί τιμή μικρότερη του μηδενός τότε η ls τερματίζεται (δεν επιστρέφει τίποτα και σταματάει). Πριν γίνει το return όμως κάνουμε `close(fd), system call` που απελευθερώνει το file descriptor fd ώστε να μην ξαναχρησιμοποιηθεί.

```
25 void
26 ls(char *path)
27 {
28     char buf[512], *p;
29     int fd;
30     struct dirent de;
31     struct stat st;
32
33     if((fd = open(path, 0)) < 0){
34         printf(2, "ls: cannot open %s\n", path);
35         return;
36     }
37
38     if(fstat(fd, &st) < 0){
39         printf(2, "ls: cannot stat %s\n", path);
40         close(fd);
41         return;
42     }
43 }
```



3. Με ένα switch statement ελέγχουμε διαφορετικές περιπτώσεις. Στην ειδική δομή δεδομένων stat (έχει οριστεί στο αρχείο stat.h) έχει δοθεί από την `fstat()` (system call) τιμή που δείχνει αν πρόκειται για αρχείο, directory ή άλλη συσκευή. Η fstat καλείται με όρισμα το &st, δηλαδή με έναν pointer που δείχνει στην stat st.

```
1  #define T_DIR 1 // Directory
2  #define T_FILE 2 // File
3  #define T_DEV 3 // Device
4
5  struct stat {
6      short type; // Type of file
7      int dev; // File system's disk device
8      uint ino; // Inode number
9      short nlink; // Number of links to file
10     uint size; // Size of file in bytes
11 };
```

4. Στην περίπτωση που πρόκειται για αρχείο, το πρόγραμμα τυπώνει το όνομα το αρχείου και κάποια χαρακτηριστικά του, που έχουν αποθηκευτεί από την fstat() στην stat st(st.type, st.ino, st.size). Για να τυπωθεί το όνομα του αρχείου χρησιμοποιείται η `fmtname()` η οποία απλά το επιστρέφει σαν string με συγκεκριμένο format.

```

44 switch(st.type){
45 case T_FILE:
46     printf(1, "%s %d %d %d\n", fmtname(path), st.type, st.ino, st.size);
47     break;
48 } 4
49 case T_DIR:
50     if(strlen(path) + 1 + DIRSIZ + 1 > sizeof buf){
51         printf(1, "ls: path too long\n");
52         break;
53     }
54     strcpy(buf, path);
55     p = buf+strlen(buf);
56     *p++ = '/';
57     while(read(fd, &de, sizeof(de)) == sizeof(de)){
58         if(de.inum == 0)
59             continue;
60         memmove(p, de.name, DIRSIZ);
61         p[DIRSIZ] = 0;
62         if(stat(buf, &st) < 0){
63             printf(1, "ls: cannot stat %s\n", buf);
64             continue;
65         }
66         printf(1, "%s %d %d %d\n", fmtname(buf), st.type, st.ino, st.size);
67     }
68     break;
69 } 5
70 close(fd);
71 } 6

```

5. Στην περίπτωση που πρόκειται για directory εκτελούνται περισσότερες ενέργειες. Αρχικά αν το Path είναι πολύ μεγάλο η συνάρτηση σταματάει (Break). Αν όχι, το directory το μεταφέρουμε στο buffer buf και βάζουμε '/' στο τέλος. Μετά με ένα loop διατρέχουμε τα περιεχόμενα. Καλούμε την **read() system call**. Όταν το directory δεν έχει άλλα περιεχόμενα για ανάγνωση, η read() θα επιστρέψει τιμή 0 και το loop θα σταματήσει. Αν το inum == 0 δεν έχει διαβαστεί σωστό αρχείο ή φάκελος οπότε πάμε παρακάτω. Αλλιώς προσθέτουμε το όνομα του αρχείου ή του φακέλου στο buffer που έχει το Path. Καλούμε την συνάρτηση stat() (από ulib.c) για την συγκεκριμένη εγγραφή(file or directory) στη συνέχεια. Αν επιστρέψει έγκυρη τιμή τυπώνουμε τα στατιστικά του αρχείου. Στην επομενη επανάληψη στο Buffer θα αντικατασταθεί το παλιο αρχείο με το όνομα του νέου κ.οκ.
6. Τέλος με την **system call close()** απελευθερώνουμε το file descriptor για αποφυγή data leaks. (δεν ελέγχουμε όμως αν έγινε επιτυχώς αυτό).

## Άσκηση 2:

Δημιουργήστε ένα πρόγραμμα χρήση για το xv6.

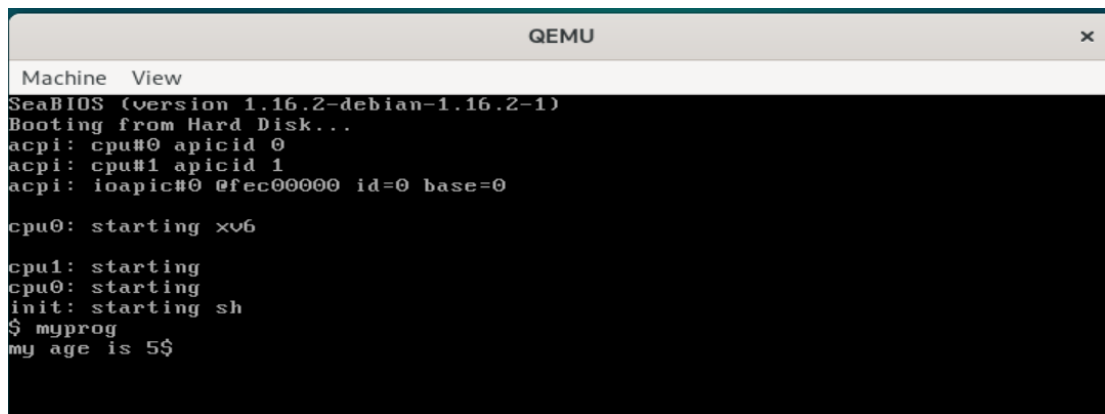
```

manya@debian: ~/xv6/user
GNU nano 7.2 myprog.c
#include "types.h"
#include "stat.h"
#include "user.h"

int
main(int argc, char *argv[])
{
    int i=5;
    printf(1, "my age is %d", i);
    exit();
}

```

Εκτέλεση προγράμματος από το terminal του xv6:



```
Machine View
SeaBIOS (version 1.16.2-debian-1.16.2-1)
Booting from Hard Disk...
acpi: cpu#0 apicid 0
acpi: cpu#1 apicid 1
acpi: ioapic#0 0fec00000 id=0 base=0

cpu0: starting xv6

cpu1: starting
cpu0: starting
init: starting sh
$ myprog
my age is 5$
```

### Άσκηση 3:

Αναζητώντας μέσα στον πηγαίο κώδικα του xv6, βρείτε την κλήση συστήματος uptime.

1. Βρείτε όλα τα σημεία του κώδικα του πυρήνα στα οποία γίνεται αναφορά σε αυτή.
2. Εξηγήστε την υλοποίησή της:

Η συνάρτηση uptime ορίζεται στο αρχείο sysproc.c. Αναζητούμε που αλλού αναφέρεται μέσα στο kernel:

```
root@debian: /home/manyaxv6/kernel# grep "uptime" -r
syscall.c:extern int sys_uptime(void);
syscall.c:[SYS_uptime] sys_uptime,
sysproc.c:sys_uptime(void)
```

Η uptime υλοποιείται ως εξής:

Ορίζουμε unsigned int xticks.

Καλούμε την acquire() η οποία μας δίνει πρόσβαση στην μεταβλητή ticks και κλειδώνει το ticksclock για να μην μπορούν να την χρησιμοποιήσουν άλλες διεργασίες.

Ύστερα η xticks παίρνει την τιμή ticks. Η ticks είναι μια global variable, της οποίας η τιμή αυξάνεται κάθε φορά που το ρολόι προκαλεί Interrupt. Έτσι δείχνει πόσα τικ έγιναν από όταν ξεκίνησε το σύστημα.

Με το release απελευθερώνουμε ξανά το ρολόι ticksclock και έχουν πρόσβαση σε αυτό οι άλλες διεργασίες. Η συνάρτηση acquire ορίζεται στο spinlock.c

```
112 // return how many clock tick interrupts have occurred
113 // since start.
114 int
115 sys_uptime(void)
116 {
117     uint xticks;
118
119     acquire(&tickslock);
120     xticks = ticks;
121     release(&tickslock);
122     return xticks;
123 }
```

#### Άσκηση 4:

Βρείτε τη συνάρτηση μέσα στον kernel κώδικα όπου τελικά εκτελούνται οι system calls. Καταγράψτε τη σειρά των ενεργειών που γίνονται όταν καλείται ένα system call:

##### Περιγραφή με κείμενο:

Το πρόγραμμα καλεί κάποια system call. Αυτό είναι εφικτό επειδή τα user προγράμματα έχουν `#include "user.h"`. Αφού το πρόγραμμα χρήστη γίνει compile, ο linker θα βρει τα implementations των συναρτήσεων στο `usys.S`. όπως εξηγήθηκε παραπάνω, στο αρχείο `usys.S` τοποθετείται στον καταχωρητή `eax` ο αριθμός της κλήσης συστήματος που ζητάμε(`SYS_name`, όπου `name` η αντίστοιχη συνάρτηση). Αυτός ο αριθμός ορίζεται στο `syscall.h`.

Η εντολή `int $T_SYSCALL`, προκαλεί διακοπή και ο έλεγχος περνάει από τον User στο kernel. Αν ο αριθμός του interrupt είναι έγκυρος καλείται η `syscall()` της οποίας ο κωδικός έχει τοποθετηθεί στον `eax`.

Η συνάρτηση `syscall()` ορίζεται στο `syscall.c`. Εκεί υπάρχουν τα declarations (ορισμοί) όλων των system calls στην μορφή `extern int sys_name(void)`. Το `extern` δηλώνει στον compiler ότι η συνάρτηση αυτή μπορεί να διαβαστεί και από άλλα αρχεία.

Στο ίδιο αρχείο υπάρχει ένας πίνακας από function pointers που αντιστοιχίζουν τους αριθμούς που τοποθετούνται στον `eax` με τα system calls. Με την εντολή `syscall()` ελέγχουμε αν είναι έγκυρη η τιμή του `eax = num`. Αν ναι, η εντολή `syscalls[num]()` καλεί την κατάλληλη συνάρτηση ορισμένη στα αρχεία `sysproc.c` και `sysfile.c`. Μετά την κλήση της system call, το αποτέλεσμα της αποθηκεύεται στον `eax`.

```
void
syscall(void)
{
    int num;

    num = proc->tf->eax;
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        proc->tf->eax = syscalls[num]();
    } else {
        cprintf("%d %s: unknown sys call %d\n",
                proc->pid, proc->name, num);
        proc->tf->eax = -1;
    }
}

int sys_settickets(void) {
    return -1;
}
```

Πίνακας αντιστοίχισης στο `syscall.c`: Κάθε `SYS_name` που έχει τοποθετηθεί στον `eax` αντιστοιχίζεται με τον κατάλληλο function pointer. (lines 149-173)

ΠΡΟΣΟΧΗ: Κάποιες συναρτήσεις στο `sysproc.c` καλούν συναρτήσεις από το `proc.c`.

Σχεδιάγραμμα σε μορφή εικόνας:

random\_user\_program.c

System call

user.h

### syscall.h

```
1 // System call numbers
2 #define SYS_fork 1
3 #define SYS_exit 2
4 #define SYS_wait 3
5 #define SYS_pipe 4
6 #define SYS_read 5
7 #define SYS_kill 6
8 #define SYS_exec 7
9 #define SYS_fstat 8
10 #define SYS_chdir 9
11 #define SYS_dup 10
12 #define SYS_getpid 11
13 #define SYS_sbrk 12
14 #define SYS_sleep 13
15 #define SYS_uptime 14
16 #define SYS_open 15
17 #define SYS_write 16
18 #define SYS_mknod 17
19 #define SYS_unlink 18
20 #define SYS_link 19
21 #define SYS_mkdir 20
22 #define SYS_close 21
23 #define SYS_getpinfo 24
24 #define SYS_settickets 27
```

Ο preprocessor  
αντικαθιστά το  
κάθε όνομα με  
έναν αριθμό

### usys.S

```
1 #include "syscall.h"
2 #include "traps.h"
3
4 #define SYSCALL(name) \
5     .globl name; \
6     name: \
7         movl $SYS_ ## name, %eax; \
8         int $T_SYSCALL; \
9         ret
11 SYSCALL(fork)
12 SYSCALL(exit)
13 SYSCALL(wait)
14 SYSCALL(pipe)
15 SYSCALL(read)
16 SYSCALL(write)
17 SYSCALL(close)
18 SYSCALL(kill)
19 SYSCALL(exec)
20 SYSCALL(open)
21 SYSCALL(mknod)
22 SYSCALL(unlink)
23 SYSCALL(fstat)
24 SYSCALL(link)
25 SYSCALL(mkdir)
26 SYSCALL(chdir)
27 SYSCALL(dup)
28 SYSCALL(getpid)
29 SYSCALL(sbrk)
30 SYSCALL(sleep)
31 SYSCALL(uptime)
32 SYSCALL(getpinfo)
33 SYSCALL(settickets)
```

### syscall.c

```
125 extern int sys_chdir(void);
126 extern int sys_close(void);
127 extern int sys_dup(void);
128 extern int sys_exec(void);
129 extern int sys_exit(void);
130 extern int sys_fork(void);
131 extern int sys_fstat(void);
132 extern int sys_getpid(void);
133 extern int sys_kill(void);
134 extern int sys_link(void);
135 extern int sys_mkdir(void);
136 extern int sys_mknod(void);
137 extern int sys_open(void);
138 extern int sys_pipe(void);
139 extern int sys_read(void);
140 extern int sys_sbrk(void);
141 extern int sys_sleep(void);
142 extern int sys_unlink(void);
143 extern int sys_wait(void);
144 extern int sys_write(void);
145 extern int sys_uptime(void);
146 extern int sys_getpinfo(void);
147 extern int sys_settickets(void);
```

```
/*πίνακας με function pointers*/
149 static int (*syscalls[])(void) = {
150     [SYS_fork] sys_fork,
151     [SYS_exit] sys_exit,
152     [SYS_wait] sys_wait,
153     [SYS_pipe] sys_pipe,
154     [SYS_read] sys_read,
155     [SYS_kill] sys_kill,
156     [SYS_exec] sys_exec,
157     [SYS_fstat] sys_fstat,
158     [SYS_chdir] sys_chdir,
159     [SYS_dup] sys_dup,
160     [SYS_getpid] sys_getpid,
161     [SYS_sbrk] sys_sbrk,
162     [SYS_sleep] sys_sleep,
163     [SYS_uptime] sys_uptime,
164     [SYS_open] sys_open,
165     [SYS_write] sys_write,
166     [SYS_mknod] sys_mknod,
167     [SYS_unlink] sys_unlink,
168     [SYS_link] sys_link,
169     [SYS_mkdir] sys_mkdir,
170     [SYS_close] sys_close,
171     [SYS_getpinfo] sys_getpinfo,
172     [SYS_settickets] sys_settickets
173 };
```

### /\*συνάρτηση syscall()\*/

```
175 void
176 syscall(void)
177 {
178     int num;
179
180     num = proc->tf->eax;
181     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
182         proc->tf->eax = syscalls[num]();
183     } else {
184         /*καλεί αντίστοιχη συνάρτηση*/
185         cprintf("%d %s: unknown sys call %d\n",
186             proc->pid, proc->name, num);
187         proc->tf->eax = -1;
188     }
189 }
```

με το **extern**, ο compiler  
ξέρει πώς πρέπει να  
αναζητήσει τα  
implementations των  
συναρτήσεων αυτών σε  
άλλο αρχείο

οι συναρτήσεις sys\_name() είναι στα αρχεία:

sysproc.c

sysfile.c

proc.c

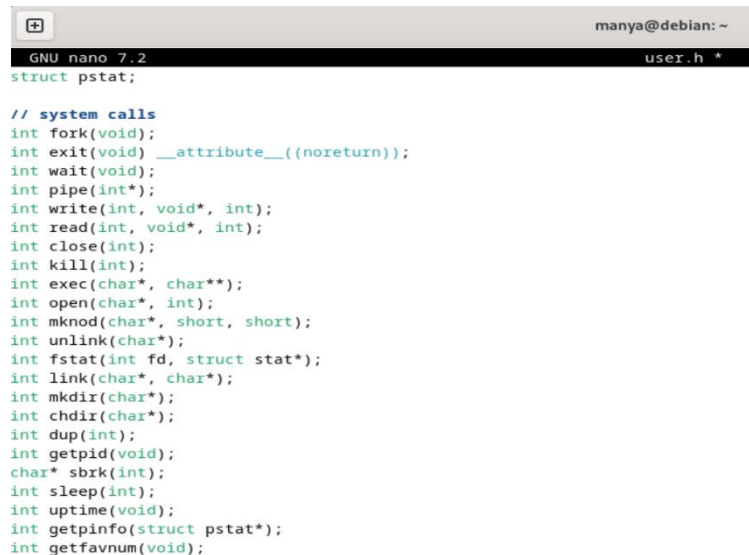
file.c

καλούν συναρτήσεις και απο άλλα αρχεία

## Άσκηση 5:

Προσθέστε μία κλήση συστήματος η οποία θα επιστρέφει τον αγαπημένο σας αριθμό. Η κλήση που θα φτιάξετε πρέπει να έχει το εξής πρωτότυπο: `int getfavnum(void)`. Μία τέτοια κλήση δεν έχει κανέναν πρακτικό σκοπό αφού δεν εκμεταλλεύεται κάποια πληροφορία ή πόρο που κατέχει το λειτουργικό σύστημα. Στόχος εδώ είναι να κατανοήσουμε τον τρόπο με τον οποίο προστίθενται κλήσεις συστήματος στο `xv6`.

Βήμα 1: Προσθήκη στο `user.h` του ορισμού `int getfavnum(void)`;



```
manya@debian: ~
GNU nano 7.2 user.h *
struct pstat;

// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(char*, int);
int mknod(char*, short, short);
int unlink(char*);
int fstat(int fd, struct stat*);
int link(char*, char*);
int mkdir(char*);
int chdir(char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int getpinfo(struct pstat*);
int getfavnum(void);
```

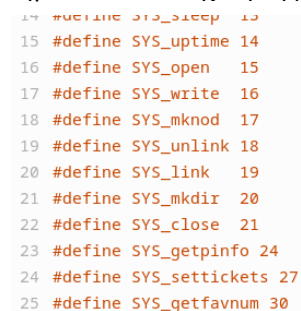
Βήμα 2: Προσθήκη SYSCALL(getfavnum) στο `usys.S`



```
manya@debian: ~
GNU nano 7.2 usys.S
movl $SYS_ ## name, %eax; \
int $T_SYSCALL; \
ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(getpinfo)
SYSCALL(settickets)
SYSCALL(getfavnum)
```

Βήμα 3: Αντιστοίχιση της νέας system call με αριθμό στο `syscall.h` header (π.χ. 30)



```
14 #define SYS_sleep 13
15 #define SYS_uptime 14
16 #define SYS_open 15
17 #define SYS_write 16
18 #define SYS_mknod 17
19 #define SYS_unlink 18
20 #define SYS_link 19
21 #define SYS_mkdir 20
22 #define SYS_close 21
23 #define SYS_getpinfo 24
24 #define SYS_settickets 27
25 #define SYS_getfavnum 30
```



Βήμα 4: Προσθέτουμε στο syscall.c του kernel, το declaration `extern int sys_getfavnum(void);` και ύστερα προσθέτουμε στον πίνακα από function pointers το ζεύγος `[SYS_getfavnum]` και `sys_getfavnum`.

Open ▾ +

syscall.c  
~/xv6/kernel

```

132 extern int sys_getpid(void);
133 extern int sys_kill(void);
134 extern int sys_link(void);
135 extern int sys_mkdir(void);
136 extern int sys_mknod(void);
137 extern int sys_open(void);
138 extern int sys_pipe(void);
139 extern int sys_read(void);
140 extern int sys_sbrk(void);
141 extern int sys_sleep(void);
142 extern int sys_unlink(void);
143 extern int sys_wait(void);
144 extern int sys_write(void);
145 extern int sys_uptime(void);
146 extern int sys_getpinfo(void);
147 extern int sys_settickets(void);
148 extern int sys_getfavnum(void);
149

```

Open ▾ +

syscall.c  
~/xv6/kernel

```

158 [SYS_fstat] sys_fstat,
159 [SYS_chdir] sys_chdir,
160 [SYS_dup] sys_dup,
161 [SYS_getpid] sys_getpid,
162 [SYS_sbrk] sys_sbrk,
163 [SYS_sleep] sys_sleep,
164 [SYS_uptime] sys_uptime,
165 [SYS_open] sys_open,
166 [SYS_write] sys_write,
167 [SYS_mknod] sys_mknod,
168 [SYS_unlink] sys_unlink,
169 [SYS_link] sys_link,
170 [SYS_mkdir] sys_mkdir,
171 [SYS_close] sys_close,
172 [SYS_getpinfo] sys_getpinfo,
173 [SYS_settickets] sys_settickets,
174 [SYS_getfavnum] sys_getfavnum
175 };

```

Βήμα 5: Υλοποίηση της `getmyfavnum()`. Είτε στο αρχείο `syscall.c` ή στο `sysproc.c` υλοποιούμε την συνάρτηση. Ένας απλός τρόπος ο εξής:

+ manya@debian: ~

GNU nano 7.2 exercise5.c \*

```

#include "user.h"

int main(int argc, char **argv){
int i;
i=getmyfavnum();
printf(1,"fav num= %d",i);
exit();
}

```

Βήμα 6: Φτιάχνουμε μια user εντολή που να ελέγχει την ορθή λειτουργία του system call. Το προσθέτουμε μετά στα προγράμματα που περιλαμβάνει το `makefile` και ύστερα καλούμε την εντολή αυτή από το terminal του xv6.

Open ▾ +

favourite.c  
~/xv6/user

```

1 #include "types.h"
2 #include "user.h"
3
4 int
5 main(int argc, char **argv)
6 {
7     int i=getfavnum();
8     printf(1,"favourite number is %d\n",i);
9     exit();
10 }

```

QEMU

Machine View

```

SeaBIOS (version 1.16.2-debian-1.16.2-1)
Booting from Hard Disk...
acpi: cpu#0 apicid 0
acpi: cpu#1 apicid 1
acpi: ioapic#0 @fec00000 id=0 base=0

cpu0: starting xv6
cpu1: starting
cpu0: starting
init: starting sh
$ favourite
favourite number is 27
$

```

## Άσκηση 6:

Προσθέστε μία κλήση συστήματος η οποία θα εκτελεί λειτουργία «shutdown». Η κλήση που θα φτιάξετε πρέπει να έχει το εξής πρωτότυπο: `void halt(void)`. Έπειτα, φτιάξτε ένα πρόγραμμα χρήστη με όνομα `shutdown` το οποίο θα καλεί την κλήση συστήματος που φτιάξατε, ώστε χρήστες του `xv6` να μπορούν πλέον να κάνουν `shutdown` το σύστημα χωρίς να χρειάζεται να το κάνουν από το `host` μηχάνημα. Σε αυτή τη κλήση συστήματος θα πρέπει να επικοινωνήσετε άμεσα με το `hardware` της εικονικής μηχανής (στην περίπτωσή μας το `QEMU`). Για την απενεργοποίηση του συστήματος το `hardware` περιμένει σε κάποια συγκεκριμένη διεύθυνση (`port`) να υπάρξει μία συγκεκριμένη τιμή. Οι αριθμοί για αυτά τα δύο ορίζονται από τον κατασκευαστή. Οδηγίες για την τιμή που πρέπει να στείλετε σε ποια πόρτα έχουν καταγραφεί για διάφορους `emulators` εδώ. Εξηγήστε τη λειτουργία της συνάρτησης `outw` την οποία θα χρησιμοποιήσετε.

*Σημείωση: Η συνάρτηση `outw` είναι ήδη υλοποιημένη στο `xv6` και βρίσκεται στο αρχείο `include/x86.h`.*

Πρέπει να κάνουμε κάποιες προσθήκες για την υλοποίηση της `halt`:

Βήμα 1: `void halt(void)` πρότυπο στο `user.h`

Βήμα 2: `#define SYS_halt 31` στο `syscall.h`

Βήμα 3: `SYSCALL(halt)` στο `usys.S`

Βήμα 4: Προσθέτουμε στο `syscall.c` του `kernel`, το `declaration extern int sys_halt(void)`; και ύστερα προσθέτουμε στον πίνακα από `function pointers` το ζεύγος `[SYS_halt] sys_halt`.

Βήμα 5: Υλοποίηση της `sys_halt`. Στο αρχείο `sysproc.c` υλοποιούμε την `int sys_halt(void)` η οποία καλεί την `void halt(void)` Που την υλοποιούμε στο `proc.c`

Για να γίνει `shutdown` ο `emulator` χρειάζεται η `outw(0x604, 0x2000)`;

<pre>int sys_halt(void){     halt();     return 0; }</pre>	<pre>void halt(void) {     outw(0x604, 0x2000);     return; }</pre>
--	---

Βήμα 6: Φτιάχνουμε πρόγραμμα `shutdown.c` στον φάκελο `User` και το προσθέτουμε στο `Makefile`. Τώρα με την εντολή `shutdown`, ο `qemu emulator` κλείνει.

**shutdown.c:**

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

int main(int argc, char** argv){
    halt();
    exit();
}
```

**terminal:**

```
SeaBIOS (version 1.16.2-debian-1.16.2-1)
Booting from Hard Disk...
acpi: cpu#0 apicid 0
acpi: cpu#1 apicid 1
acpi: ioapic#0 @fec00000 id=0 base=0

cpu0: starting xv6
cpu1: starting
cpu0: starting
init: starting sh
$ shutdown_
```

## Άσκηση 7:

Προσθέστε μία κλήση συστήματος η οποία θα επιστρέφει το πόσες φορές έχει εκτελεστεί μία συγκεκριμένη κλήση συστήματος η οποία θα περνιέται σαν όρισμα. Η κλήση που θα φτιάξετε πρέπει να έχει το εξής πρωτότυπο: `int getcount(int syscall)`.

Για να μετράμε τις φορές που θα κληθεί η κάθε συνάρτηση θα δημιουργήσουμε έναν κενό πίνακα στο `syscall.c`, τον `int numcall[sysc_max]`. Η τιμή `sysc_max` έχει γίνει `#define sysc_max 50` στο `syscall.h` έτσι ώστε ο πίνακας να χωράει όλες τις τιμές των `systemcalls` και να έχουμε περιθώριο να φτιάξουμε μερικές παραπάνω.

```
#include "syscall.h"

int numcall[sysc_max]={0};
```

Θέλουμε στην συνάρτηση `syscall()` του `syscall.c`, κάθε φορά που καλείται να αυξάνεται ο αριθμός στην αντίστοιχη θέση του πίνακα.

```
void
syscall(void)
{
    int num;
    num = proc->tf->eax;
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        numcall[num]++;

        proc->tf->eax = syscalls[num]();
    } else {
        cprintf("%d %s: unknown sys call %d\n",
            proc->pid, proc->name, num);
        proc->tf->eax = -1;
    }
}
```

Περνάμε στην υλοποίηση της `getcount` τώρα. Στο αρχείο `syscall.c` την καταγράφουμε με `extern int sys_getcount(void)`; ώστε να μπορεί να κληθεί από την `syscall` (ο compiler θα δει ότι είναι σε άλλο αρχείο). Επίσης την προσθέτουμε στον πίνακα των συναρτήσεων την `sys_getcount`, και τον αριθμό κλήσης της στο `syscall.h`.

### syscall.c:

```
static int (*syscalls[])(void) = { [SYS_fork] sys_fork,
    [SYS_exit] sys_exit,
    [SYS_wait] sys_wait,
    [SYS_pipe] sys_pipe,
    [SYS_read] sys_read,
    [SYS_kill] sys_kill,
    [SYS_exec] sys_exec,
    [SYS_fstat] sys_fstat,
    [SYS_chdir] sys_chdir,
    [SYS_dup] sys_dup, [SYS_getpid] sys_getpid, [SYS_sbrk] sys_sbrk, [SYS_sleep] sys_sleep, [SYS_uptime] sys_uptime,
    [SYS_open] sys_open, [SYS_write] sys_write, [SYS_mknod] sys_mknod, [SYS_unlink] sys_unlink, [SYS_link] sys_link,
    [SYS_mkdir] sys_mkdir, [SYS_close] sys_close, [SYS_getpinfo] sys_getpinfo, [SYS_settickets] sys_settickets,
    [SYS_halt] sys_halt, [SYS_getfavnum] sys_getfavnum, [SYS_getcount] sys_getcount
};

extern int sys_getcount(void);
```

## syscall.h

```
#define SYS_mkdir 20
#define SYS_close 21
#define SYS_getpinfo 24
#define SYS_settickets 27
#define SYS_getfavnum 30
#define SYS_halt 31
#define SYS_getcount 32
#define sysc_max 50
```

Υλοποιούμε τώρα στα αρχεία sysproc.c και proc.c την διαδικασία αυτή:

```
Open ▾ (+) • proc.c
~ /xv6/kernel

1 #include "types.h"
2 #include "defs.h"
3 #include "param.h"
4 #include "memlayout.h"
5 #include "mmu.h"
6 #include "x86.h"
7 #include "proc.h"
8 #include "spinlock.h"
9 #include "syscall.h"
10
11 extern int numcall[sysc_max];
12
13 struct ptable ptable;
14 static struct proc *initproc;
15 int nextpid = 1;
16 extern void forkret(void);
17 extern void trapret(void);
18 static void wakeup1(void *chan);
19
20 int getcount(int syscall)
21 {
22     return numcall[syscall];
23 }
24
```

```
Open ▾ (+) • sysproc.c
~ /xv6/kernel

1 #include "types.h"
2 #include "x86.h"
3 #include "defs.h"
4 #include "param.h"
5 #include "memlayout.h"
6 #include "mmu.h"
7 #include "proc.h"
8 |
9 int sys_getcount(void){
10     int id;
11     int count;
12     argint(0,&id);
13     count=getcount(id);
14     return count;
15 }
```

Το `argint(0,&id)` περνάει το πρώτο όρισμα που θα δώσει ο χρήστης ως τιμή στο `int`, δηλαδή το νούμερο της `system call` που θέλουμε να δούμε πόσες φορές κλήθηκε.

Τώρα φτιάχνουμε ένα User πρόγραμμα που να τυπώνει τα αποτελέσματα αυτής της εντολής και το προσθέτουμε στο `Makefile`.

```
GNU nano 7.2 count.c *
#include "types.h"
#include "user.h"

int main(int argc, char** argv){
    int number;
    number=atoi(argv[1]);
    printf(1,"system call number= %d    number of calls= %d\n",number ,getcount(number));
    exit();
}
```

```
QEMU x
Machine View
SeaBIOS (version 1.16.2-debian-1.16.2-1)
Booting from Hard Disk...
acpi: cpu#0 apicid 0
acpi: cpu#1 apicid 1
acpi: ioapic#0 @fec00000 id=0 base=0

cpu0: starting xv6
cpu1: starting
cpu0: starting
init: starting sh
$ count 1
system call number= 1    number of calls= 2
$
```

Καλούμε την `ls` και μετά ξανά την `count` για το 8(της `fstat` η οποία χρησιμοποιείται από την `ls`). Διαπιστώνουμε ότι όντως έχει κληθεί 26 φορές.

```
echo          2 4 14408
forktest      2 5 9376
schedtest     2 6 17304
grep          2 7 17024
init          2 8 14960
kill          2 9 14496
ln            2 10 14456
ls            2 11 16768
mkdir         2 12 14536
rm            2 13 14528
sh            2 14 25376
stressfs      2 15 15176
usertests     2 16 50080
wc            2 17 15608
zombie        2 18 14192
lotteryschedte 2 19 17888
myprog        2 20 14304
favourite     2 21 14368
shutdown      2 22 14224
count         2 23 14472
console       3 24 0
$ count 8
system call number= 8    number of calls= 26
```

### Άσκηση 8:

Προσθέστε μία κλήση συστήματος η οποία θα τερματίζει μια τυχαία διεργασία με πρότυπο `int killrandom(void)`. Θα χρειαστεί να προσθέσετε μία γεννήτρια τυχαίων αριθμών στο kernel. Πώς μπορείτε να βρείτε τα PID όλων των τρεχόντων διεργασιών στο σύστημα;

*Σημείωση: Για να σκοτώσετε μία διεργασία δοθέντος ενός συγκεκριμένου PID μπορείτε να χρησιμοποιήσετε την κλήση συστήματος `kill`.*

Από το βιβλίο [1] γνωρίζουμε ότι:

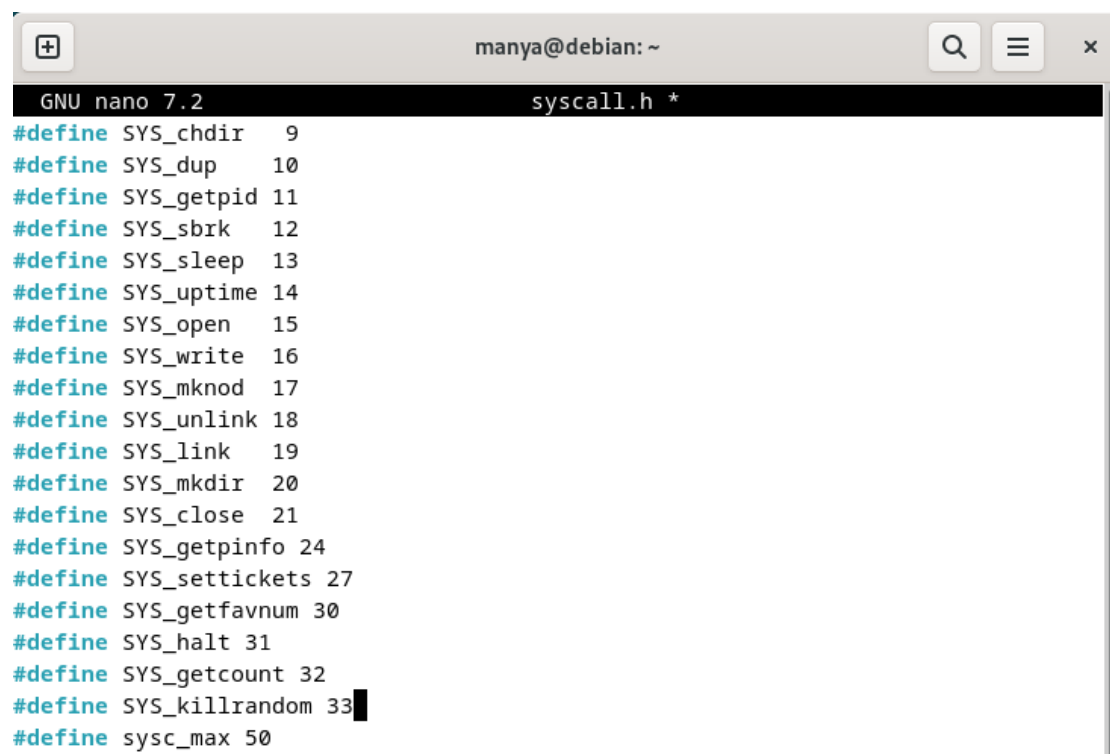
“... `kill` (2975) lets one process request that another be terminated. It would be too complex for `kill` to directly destroy the victim process, since the victim might be executing on another CPU or sleeping while midway through updating kernel data structures. To address these challenges, `kill` does very little: it just sets the victim's `p->killed` and, if it is sleeping, wakes it up. Eventually the victim will enter or leave the kernel, at which point code in `trap` will call `exit` if `p->killed` is set. If the victim is running in user space, it will soon enter the kernel by making a system call or because the timer (or some other device) interrupts.”

Η `kill` λοιπόν χρειάζεται απλά το `pid` μιας διεργασίας για να θέσει την κατάλληλη τιμή στο `ptable` της, και η `trap` αργότερα να την τερματίσει με την `exit()`.

Επομένως, πρέπει να φτιάξουμε:

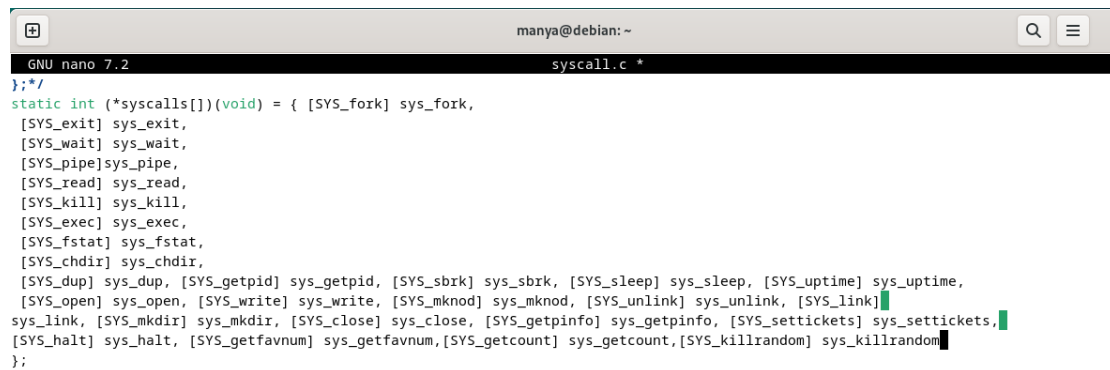
1. Μια γεννήτρια τυχαίων αριθμών
2. Μια system call που σκοτώνει ότι επιστρέφει η γεννήτρια `int killrandom(void)`.
3. Ένα User program που καλεί την system call `int killrandom(void)`.

Ορισμός αριθμού system call:



```
manya@debian: ~  
GNU nano 7.2 syscalls.h *  
#define SYS_chdir 9  
#define SYS_dup 10  
#define SYS_getpid 11  
#define SYS_sbrk 12  
#define SYS_sleep 13  
#define SYS_uptime 14  
#define SYS_open 15  
#define SYS_write 16  
#define SYS_mknod 17  
#define SYS_unlink 18  
#define SYS_link 19  
#define SYS_mkdir 20  
#define SYS_close 21  
#define SYS_getpinfo 24  
#define SYS_settickets 27  
#define SYS_getfavnum 30  
#define SYS_halt 31  
#define SYS_getcount 32  
#define SYS_killrandom 33  
#define sysc_max 50
```

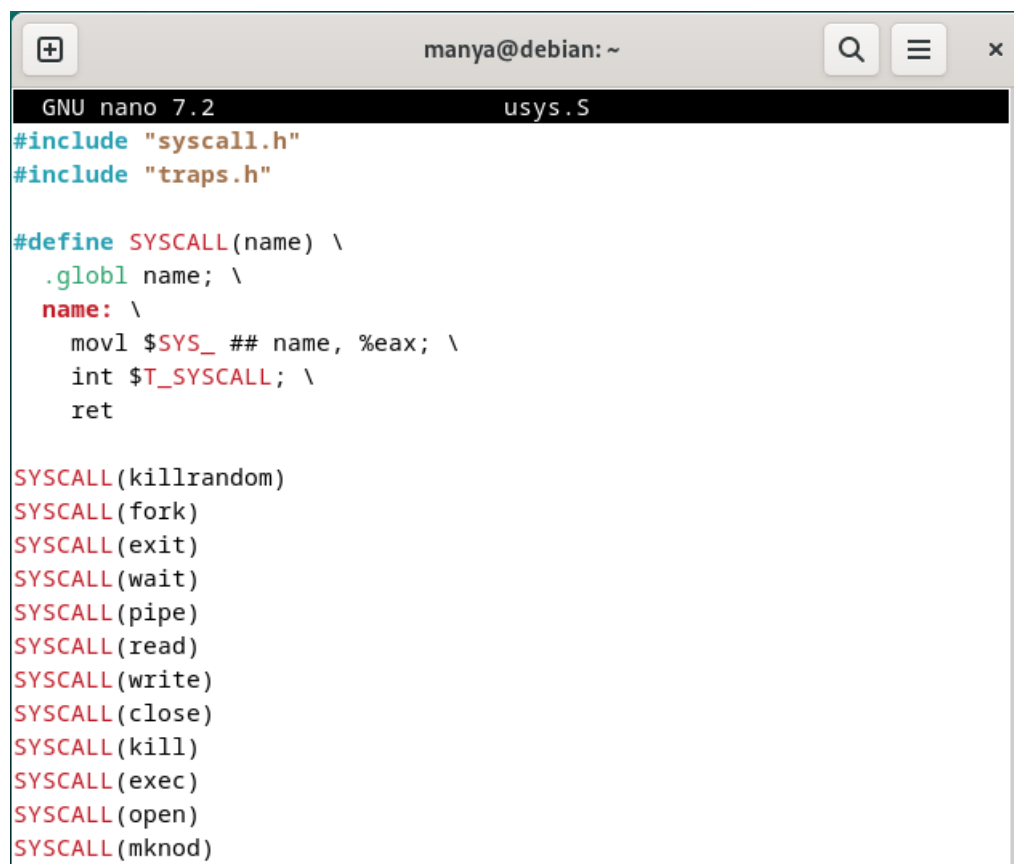
## Προσθήκη συνάρτησης στο syscall.c



```
GNU nano 7.2 syscall.c
};*/
static int (*syscalls[])(void) = { [SYS_fork] sys_fork,
[SYS_exit] sys_exit,
[SYS_wait] sys_wait,
[SYS_pipe] sys_pipe,
[SYS_read] sys_read,
[SYS_kill] sys_kill,
[SYS_exec] sys_exec,
[SYS_fstat] sys_fstat,
[SYS_chdir] sys_chdir,
[SYS_dup] sys_dup, [SYS_getpid] sys_getpid, [SYS_sbrk] sys_sbrk, [SYS_sleep] sys_sleep, [SYS_uptime] sys_uptime,
[SYS_open] sys_open, [SYS_write] sys_write, [SYS_mknod] sys_mknod, [SYS_unlink] sys_unlink, [SYS_link] sys_link, [SYS_mkdir] sys_mkdir, [SYS_close] sys_close, [SYS_getpinfo] sys_getpinfo, [SYS_settickets] sys_settickets,
[SYS_halt] sys_halt, [SYS_getfavnum] sys_getfavnum, [SYS_getcount] sys_getcount, [SYS_killrandom] sys_killrandom
};
```

`extern int sys_killrandom(void);`

## Προσθήκη στο usys.S:



```
GNU nano 7.2 usys.S
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
.globl name; \
name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(killrandom)
SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
```

## Υλοποίηση της συνάρτησης στο sysproc.c:

```
int sys_killrandom(void){
    int pid = rand();
    int res=kill(pid);
    if (res==0) return pid;
    return res;
}
```

Αν η kill είναι επιτυχής επιστρέφει 0 και η sys\_killrandom θα επιστρέψει το pid. Αλλιώς επιστρέφει -1 (θα χρησιμοποιηθεί για να τυπωθεί error message).

Γεννήτρια τυχαίων αριθμών:

Σημείωση: Θέλουμε η μέγιστη τιμή που να επιστρέφει να είναι όσες και ο μέγιστος αριθμός από διεργασίες που μπορούν να υπάρξουν. Αυτός ορίζεται στο header param.h που είναι included στο sysproc.c και ονομάζεται NPROC. (#define NPROC 64).

```
// The following functions define a portable implementation of
rand and srand.
static uint64 next = 1; // αρχικοποίηση του next

static unsigned long int next = 1; // NB: "unsigned long int" is
assumed to be 32 bits wide

int rand(void) // RAND_MAX assumed to be 32767
{
    next = next * 1103515245 + 12345;
    return (unsigned int) (next / 65536) % NPROC;
}
void srand(unsigned int seed)
{
    next = seed;
}
```

User program για έλεγχο σωστής λειτουργίας:



```
manya@debian: ~
GNU nano 7.2 test.c
#include "types.h"
#include "user.h"

int main(int argc, char** argv){
    int i= killrandom();
    if(i!=-1){
        printf(1,"process %d terminated kill returns 0 \n",i);
        exit();
    }
    else{
        printf(1,"kill returns  %d, no such process id \n",i);
    }
    exit();
}
```

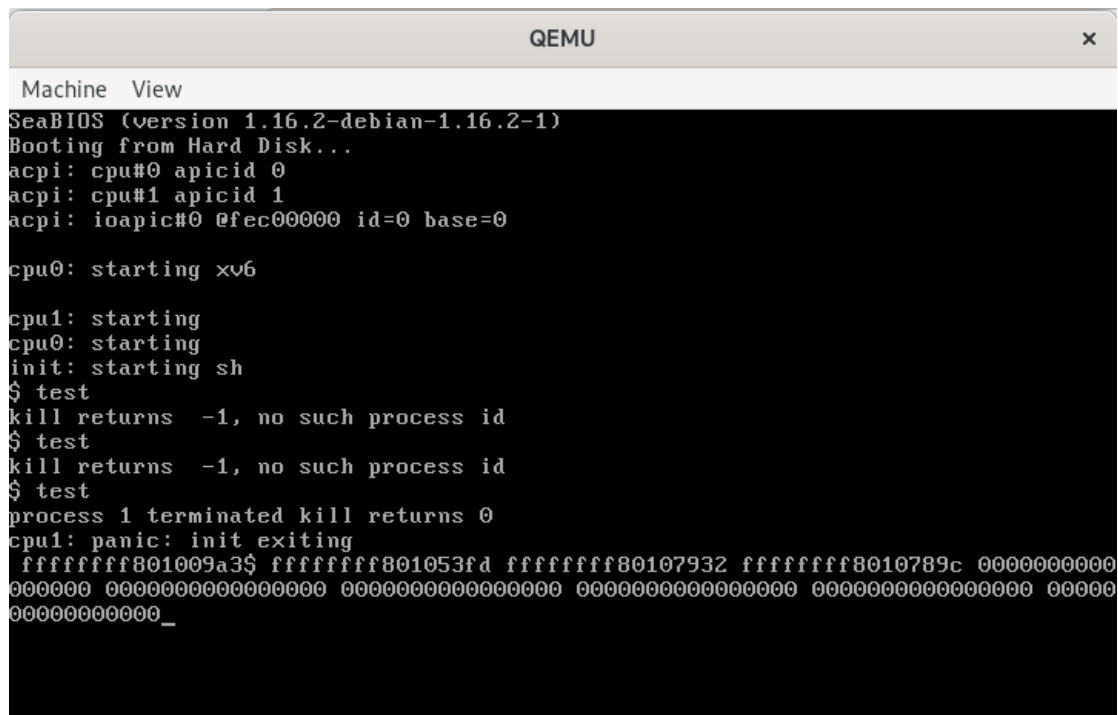
Αν δεν εκτελεστούν όλα με επιτυχία θα τυπωθεί πως δεν υπήρχε τέτοιο process id.

Αλλιώς θα τυπωθεί το process id της διεργασίας.

Προσθήκη στο Makefile με test\



Δοκιμή:



```
Machine View
SeaBIOS (version 1.16.2-debian-1.16.2-1)
Booting from Hard Disk...
acpi: cpu#0 apicid 0
acpi: cpu#1 apicid 1
acpi: ioapic#0 @fec00000 id=0 base=0

cpu0: starting xv6

cpu1: starting
cpu0: starting
init: starting sh
$ test
kill returns -1, no such process id
$ test
kill returns -1, no such process id
$ test
process 1 terminated kill returns 0
cpu1: panic: init exiting
 ffffffff801009a3$ ffffffff801053fd ffffffff80107932 ffffffff8010789c 0000000000
000000 000000000000000000 0000000000000000 0000000000000000 0000000000000000 00000
000000000000_
```

## Βιβλιογραφία

[1] ] Russ Cox, Frans Kaashoek και Robert Morris. The Design of the xv6 x86 Operating System. 2019. URL: <https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev10.pdf>.

[https://wiki.osdev.org/Random\\_Number\\_Generator#Pseudorandom\\_number\\_generators](https://wiki.osdev.org/Random_Number_Generator#Pseudorandom_number_generators)

<https://linux.die.net/man/2/fstat>