

# Λειτουργικά Συστήματα Δραστηριότητα 2

2024-2025

AM: 1096060

Ονοματεπώνυμο: Μαρία-Νίκη Ζωγράφου

## Περιεχόμενα

Άσκηση 1: .....	3
Άσκηση 2: .....	5
Συνάρτηση scheduler: .....	5
Συνάρτηση sched:.....	6
Άσκηση 3: .....	9

## Άσκηση 1:

Πώς αναπαρίσταται μία διεργασία στο xv6 kernel; Εξηγήστε τη σημασία καθενός από τα members του struct το οποίο αντιπροσωπεύει μία διεργασία στο σύστημα:

Στο αρχείο proc.h στον φάκελο include ορίζεται η δομή δεδομένων proc που αναπαριστά ένα process/ μία διεργασία. Κάθε διεργασία έχει τα εξής στοιχεία:

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING,
ZOMBIE };
struct proc {
    uintp 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
    volatile 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 inuse; // If it's being run by a CPU or not
    int ticks; // How many ticks has accumulated
    int tickets; // Number of tickets this process has (for
lottery scheduling)
};
```

Επεξήγηση του κάθε στοιχείου:

1. `uintp sz`: μέγεθος της μνήμης της διεργασίας, που περιέχει κώδικα, δεδομένα stack, heap κτλ. (Σημείωση: `uintp`=pointer-sized unsigned integer)
2. `pde_t* pgdir`: pointer στο Page table directory της διεργασίας. Χρησιμοποιείται στην διαχείριση της εικονικής μνήμης, συσχετίζοντας τις virtual διευθύνσεις με τις πραγματικές φυσικές διευθύνσεις.
3. `char *kstack`: Αυτός ο δείκτης δείχνει στο κάτω μέρος της στοίβας του kernel για τη συγκεκριμένη διεργασία. Η στοίβα αυτή χρησιμοποιείται όταν η διεργασία είναι σε kernel mode.
4. `enum procstate state`: Το enum `procstate` είναι μια μεταβλητή που μπορεί να πάρει μόνο μία από τις προκαθορισμένες τιμές που έχουν οριστεί στην αρχή με το `enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE }`. Αυτές οι τιμές αντιστοιχίζονται σε έναν ακέραιο. Αντιπροσωπεύουν την κατάσταση της διεργασίας.

UNUSED	Η δομή δεν έχει χρησιμοποιηθεί.
EMBRYO	Η δομή μόλις δημιουργήθηκε.
SLEEPING	Η διεργασία είναι σε αναμονή
RUNNABLE	Η διεργασία είναι έτοιμη να εκτελεστεί.

RUNNING	Η διεργασία εκτελείται.
ZOMBIE	Μόλις μια διεργασία τελειώσει μπαίνει σε κατάσταση ZOMBIE. Αν ο γονέας διεργασία έχει τελειώσει πριν από αυτή, δεν θα λάβει το 'SIGCHLD' signal του τερματισμού της διεργασίας του παιδιού. Επομένως, αν και τερματισμένη, η διεργασία του παιδιού μένει στο process table σε ZOMBIE state.

5. `volatile int pid`: Το Process ID ένας μοναδικός αριθμός που δρα ως αναγνωριστικό της διεργασίας.
6. `struct proc *parent`: Δείκτης στην αντίστοιχη δομή `proc` της διεργασίας γονέα.
7. `struct trapframe *tf`: Δομή που αποθηκεύει την κατάσταση των registers τη στιγμή μιας διακοπής (trap).
8. `struct context *context`: Δομή που αποθηκεύει το context ώστε να γίνει το context switch, δηλαδή τις τιμές των καταχωρητών της cpu.
9. `void *chan`: Όταν η διεργασία είναι σε κατάσταση SLEEPING και περιμένει σε κάποιο κανάλι, πχ ένα pipe, αποθηκεύουμε εκεί τον pointer για το κανάλι.
10. `int killed`: Είναι 0 για μη σκοτωμένες διεργασίες.
11. `struct file *ofile[NOFILE]`: Πίνακας δεικτών για τα ανοιχτά αρχεία που χρησιμοποιεί η διεργασία.
12. `struct inode *cwd`: Η inode δομή αντιπροσωπεύει αρχεία ή φακέλους. Στον δείκτη `cwd` φυλάμε το current working directory της διεργασίας –το path στο οποίο λειτουργεί δηλαδή.
13. `char name[16]`: Περιέχει το όνομα της διεργασίας.
14. `int inuse`: Flag που δείχνει αν η διεργασία εκτελείται από την CPU.
15. `int ticks`: Το σύνολο των "ticks" (μονάδες χρόνου CPU) που έχει χρησιμοποιήσει η διεργασία
16. `int tickets`: Καθορίζει τον αριθμό "εισιτηρίων" (tickets) που έχει η διεργασία. Χρειάζεται για τον αλγόριθμο χρονοπρογραμματισμού με κλήρωση (lottery scheduling)

## Άσκηση 2:

Ποια είναι η συνάρτηση scheduler, ποια είναι η συνάρτηση sched. Εξηγήστε τη λειτουργία της καθεμίας.

Οι δύο αυτές συναρτήσεις βρίσκονται στο αρχείο proc.c.

### Συνάρτηση scheduler:

Η συνάρτηση scheduler αποτελείται από έναν ατέρμονα βρόχο. Οι διακοπές είναι όλες ενεργοποιημένες. Η συνάρτηση διατρέπει τον πίνακα διεργασιών. Αν φτάσει στο τέλος του, `(p == &ptable.proc[NPROC])`, καλείται η `hlt`, η οποία σταματάει την `cpu`. Δεν εκτελούνται instructions οπότε το σύστημα "κοιμάται" και περιμένει κάποιο hardware ή timer interrupt για να ξαναγίνει ενεργό.

Άμα υπάρχουν ακόμα διεργασίες και δεν έχει φτάσει στο τέλος του πίνακα, ζητάει αρχικά το `ptable.lock` δηλαδή την άδεια να επεξεργαστεί τον πίνακα διεργασιών με την εντολή `acquire(&ptable.lock);`

Αν της επιτραπεί, παίρνει το `lock` (άρα μόνο αυτή έχει πρόσβαση στον πίνακα διεργασιών) και διατρέπει τον πίνακα διεργασιών με ένα λούπ. Αν το `state` μιας διεργασίας είναι `RUNNABLE` πρέπει να μεταβεί σε αυτήν.

Αρχικά φυλάει την τιμή του `p` στην μεταβλητή `proc`, μία global μεταβλητή που αντιπροσωπεύει το current process και καλεί την `switchvm(p)`, η οποία κάνει εναλλαγή της εικονικής μνήμης, έτσι ώστε το πρόγραμμα να βλέπει την εικονική μνήμη της νέας διεργασίας.

Ύστερα η scheduler την κάνει `RUNNING`, αλλάζοντας την κατάστασή της και σημειώνει ότι χρησιμοποιείται από την `cpu` (`inuse=1`).

Τέλος, κάνει context switch με το `switch()`, το οποίο ευθύνεται και για την μετάβαση στην εκτέλεση της διεργασίας.

Όταν ο έλεγχος ξαναπεράσει στην scheduler, με την `switchkvm()` διασφαλίζει την επιστροφή στην εικονική μνήμη του πυρήνα, μηδενίζει την τιμή του `proc` και απελευθερώνει το `ptable` με την εντολή `release(&ptable.lock);`

```
Void scheduler(void){
    struct proc *p = 0;
    for(;;){
        // Enable interrupts on this processor.
        sti();
        // no runnable processes? (did we hit the end of the table last
        time?)
        // if so, wait for irq before trying again.
        if (p == &ptable.proc[NPROC])
            hlt(); // The hlt stops the CPU from executing further
        instructions until the next external interrupt is received
        // Loop over process table looking for process to run.
        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
```

```

    if(p->state != RUNNABLE)
        continue;
    // Switch to chosen process. It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
    proc = p;
    switchvm(p);
    p->state = RUNNING;
    p->inuse = 1;

    swtch(&cpu->scheduler, proc->context);

    switchkvm();
    // Process is done running for now.
    // It should have changed its p->state before coming back.
    proc = 0;
}
release(&ptable.lock);
}
}

```

#### Συνάρτηση sched:

Όταν η τρέχουσα διεργασία αλλάζει κατάσταση από **RUNNING** σε **SLEEPING** ή **ZOMBIE** ή **WAITING** ξαναδίνει τον έλεγχο στην scheduler. Για αυτήν την εναλλαγή είναι υπεύθυνη η συνάρτηση sched().

Η sched() εκτελεί όμως πρώτα κάποιους κρίσιμους ελέγχους πριν καλέσει τη συνάρτηση swtch(), η οποία θα κάνει το context switch:

```

Void sched(void)
{
    int intena;

    if(!holding(&ptable.lock))
        panic("sched ptable.lock");
    if(cpu->ncli != 1)
        panic("sched locks");
    if(proc->state == RUNNING)
        panic("sched running");
    if(readeflags() & FL_IF)
        panic("sched interruptible");
    intena = cpu->intena;
    swtch(&proc->context, cpu->scheduler);
    cpu->intena = intena;
}

```

### 1<sup>ος</sup> έλεγχος:

```
if(!holding(&ptable.lock))
```

Ο πρώτος έλεγχος διασφαλίζει ότι το ptable.lock έχει αποκτηθεί, δηλαδή ότι η sched() αποκλειστική πρόσβαση στον πίνακα διεργασιών. Αλλιώς καλείται το panic (επεξήγηση στο τέλος).

### 2<sup>ος</sup> έλεγχος:

Έλεγχος της τιμής του ncli της CPU:

```
if(cpu->ncli != 1)
```

Η μεταβλητή cpu->ncli στο xv6 είναι ένας μετρητής που καταγράφει το βάθος των κλήσεων της συνάρτησης cli() (clear interrupt), η οποία χρησιμοποιείται για την απενεργοποίηση των διακοπών (interrupts) στον επεξεργαστή. Συγκεκριμένα, το cpu->ncli αυξάνεται κάθε φορά που καλείται η cli() (απενεργοποίηση διακοπών) και μειώνεται κάθε φορά που καλείται η sti() (set interrupt), η οποία ενεργοποιεί τις διακοπές.

Αυτός ο έλεγχος επιβεβαιώνει ότι το ncli έχει την τιμή 1, κάτι που σημαίνει ότι η cli() έχει κληθεί μία φορά παραπάνω από την sti() και οι διακοπές είναι απενεργοποιημένες, αλλά δεν υπάρχει nesting κλήσεων cli(). Έτσι, το sched() διασφαλίζει ότι οι διακοπές είναι άνεργες άρα η μετάβαση θα γίνει με ασφάλεια ενώ ταυτόχρονα δεν υπάρχει κίνδυνος να μην μπορούν να ενεργοποιηθούν οι διακοπές από τον scheduler.

Σημείωση: Ο scheduler θέλει ύστερα να ενεργοποιήσει τα Interrupts οπότε αν το ncli>1 πχ ncli=2, το sti() δεν θα ενεργοποιήσει τις διακοπές! Αυτό αποτελεί πρόβλημα. Αν π.χ. τελειώσουν οι διεργασίες του ptable και κληθεί η hlt η CPU σταματάει οριστικά.

### 3<sup>ος</sup> έλεγχος:

```
if(proc->state == RUNNING)
```

Η sched() πρέπει να καλείται αν όλα λειτουργούν επιθυμητά όταν η τρέχουσα διεργασία έχει μεταβεί σε μια άλλη κατάσταση από την RUNNING, όπως SLEEPING, RUNNABLE ή ZOMBIE, ώστε να γίνει η επιστροφή στον scheduler, ο οποίος θα κάνει μια άλλη διεργασία RUNNING.

### 4<sup>ος</sup> έλεγχος:

```
if(readeflags() & FL_IF)
```

Η συνάρτηση readeflags() η οποία χρησιμοποιεί ενσωματωμένο assembly κώδικα για να διαβάσει την τιμή του καταχωρητή EFLAGS:

```
static inline uintp readeflags(void){
    uintp eflags;
    asm volatile("pushf; pop %0" : "=r" (eflags));
    return eflags;}

```

Επίσης στο mmu.h ορίζεται: `#define FL_IF 0x00000200 // Interrupt Enable`

Το FL\_IF είναι μια σταθερά (512 στο δεκαδικό) η οποία αντιστοιχεί με την τιμή του Interrupt Flag όταν τα interrupts είναι ενεργοποιημένα. Όλα τα υπόλοιπα bits της είναι 0. Επομένως με το bitwise AND &, θα επιστραφεί τιμή 1 όταν το IF bit είναι 1, αλλιώς θα επιστραφεί 0. Έτσι αν τα interrupts είναι ενεργά, ακόμα και το ncli είναι 1, δεν θα επιστραφεί ο έλεγχος στον scheduler.

5° βήμα:

```
intena = cpu->intena;  
swtch(&proc->context, cpu->scheduler);  
cpu->intena = intena;
```

Η τρέχουσα κατάσταση των διακοπών (cpu->intena) αποθηκεύεται στην τοπική μεταβλητή intena, ώστε να αποκατασταθεί η κατάσταση διακοπών αργότερα, όταν επιστρέψει ο έλεγχος στο sched.

6° βήμα:

```
swtch(&proc->context, cpu->scheduler);
```

Γίνεται εναλλαγή περιβάλλοντος με το swtch, η cpu εκτελεί τον scheduler.

Σημείωση: Το kernel panic είναι ένα μέτρο ασφαλείας που λαμβάνεται από το kernel ενός λειτουργικού συστήματος όταν εντοπίσει ένα σοβαρό εσωτερικό σφάλμα. Οι ρουτίνες του kernel που διαχειρίζονται το panic είναι σχεδιασμένες έτσι ώστε να εμφανίζουν ένα μήνυμα σφάλματος στο τερματικό.



## Άσκηση 3:

### Άσκηση 3

Τροποποιήστε το `xv6` kernel ώστε να υποστηρίζει την πολιτική προγραμματισμού Lottery Scheduling [1, 4]

Το Lottery Scheduling θα λειτουργεί ως εξής:

Μία διεργασία θα ξεκινά την εκτέλεσή της με ένα συγκεκριμένο αριθμό εισητηρίων (`tickets`), αυτό μπορεί να είναι ένας οποιοδήποτε αριθμός θέλετε. Ο scheduler κάνει μια κλήρωση για την κάθε διεργασία. Αν κάποια από αυτές “κερδίσει”, θα ξεκινήσει η εκτέλεσή της. Αυτή η διαδικασία θα επαναλαμβάνεται για κάθε κλήση στον scheduler.

Ο αριθμός εισητηρίων μιας διεργασίας μπορεί να αλλάξει με τη χρήση της system call `settickets(int n)` που θα πρέπει να υλοποιήσετε. Όταν ο χρήστης εκτελεί αυτή τη system call, η τρέχουσα διεργασία θα αλλάξει αριθμό εισητηρίων με βάση το όρισμα. Η τρέχουσα διεργασία που εκτελείται στον τρέχοντα πυρήνα (`core`) βρίσκεται στην `global` μεταβλητή `proc` μέσα στο kernel.

Τη διαδικασία της κλήρωσης μπορείτε να την υλοποιήσετε με διάφορους τρόπους. Μια απλή προσέγγιση είναι ο scheduler να υπολογίζει έναν τυχαίο αριθμό μεταξύ 0 και `total_tickets`. Έπειτα, για την κάθε διεργασία να γίνεται έλεγχος αν ο αριθμός των εισητηρίων της είναι μεγαλύτερος από αυτό το νούμερο. Αν ναι, κερδίζει την κλήρωση.

Θα χρειαστεί επίσης να προσθέσετε μία **γεννήτρια τυχαίων αριθμών** στο kernel.

Για να καταλάβετε αν λειτουργεί σωστά ο scheduler σας μπορείτε να χρησιμοποιήσετε το έτοιμο πρόγραμμα `sched-test`. Αυτό θα σας δώσει στατιστικά σχετικά με το χρόνο εκτέλεσης της κάθε διεργασίας. Εξηγήστε γιατί η υλοποίησή σας είναι σωστή με βάση τα αποτελέσματα. Επεξεργαστείτε τον κώδικα στο αρχείο `user/sched-test.c` αλλάζοντας τον αριθμό των εισητηρίων που δίνονται στην κάθε διεργασία. Εξηγήστε τα αποτελέσματα της εκτέλεσης.



**Βοήθεια:** Τα σημεία του πηγαίου κώδικα που θα πρέπει να επεξεργαστείτε είναι τα εξής:

- Συνάρτηση `scheduler` στο αρχείο `kernel/proc.c` ώστε να γίνεται η επιλογή της διεργασίας που θα εκτελεστεί σε κάθε κβάντο με βάση τον νικητή της κλήρωσης.
- Συνάρτηση `fork` στο αρχείο `kernel/proc.c`. μπορείτε να έχετε μία διεργασία να κληρονομεί τα εισητήρια από το γονέα της.
- Τα αρχεία `kernel/sysproc.c`, `kernel/syscall.c`, `include/sysproc.h`, `include/user.h`, `ulib/usys.S` για την προσθήκη της απαραίτητης κλήσης συστήματος ώστε να ελέγχεται ο αριθμός εισητηρίων μιας διεργασίας από κώδικα χρήστη.
- Συμπληρώστε τον κατάλληλο κώδικα στο αρχείο `include/random.h` για την υλοποίηση της γεννήτριας τυχαίων αριθμών.


Φροντίστε να δώσετε στην διεργασία `init` (PID 1) ένα τουλάχιστον εισητήριο.

Χρειάζεται πρώτα να σιγουρευτούμε ότι η δομή `proc` (PROCESS) έχει μεταβλητή για να φυλάει τα `tickets`. Υπάρχει όντως ένας ακέραιος `int tickets`.

Επομένως, πρέπει τώρα να διασφαλίσουμε ότι κάθε διεργασία θα λαμβάνει `tickets`.

Πώς δημιουργείται μια διεργασία στο `xv6`;

Πρώτη διεργασία που δημιουργείται είναι η `init`, στο `kernel/main.c`, στην `main()` καλείται η `userinit()`; `// first user process`. Η `userinit()` ορίζεται στο `proc.c` και καλεί πρώτα την `allocproc()`. Η `allocproc()`, `allocate process` επιστρέφει ένα `proc struct`. Επομένως πρέπει να δώσει στην κάθε διεργασία τουλάχιστον ένα εισητήριο.



```
manya@debian: ~
GNU nano 7.2 syscall.h
#define SYS_link 19
#define SYS_mkdir 20
#define SYS_close 21
#define SYS_getpinfo 24
#define SYS_settickets 27
```

- syscall.c: Διαγράφουμε τον παλιό ορισμό της settickets που επιστρέφει πλην 1. Παρατηρούμε επίσης ότι έχουν γίνει ήδη τα εξής:

```
extern int sys_settickets(void); //declaration συνάρτησης που θα είναι σε άλλο αρχείο.
[SYS_settickets] sys_settickets //Μέσα στην syscall() προσθήκη της sys_settickets στον πίνακα από function pointers.
```

- sysproc.c: Ορίζουμε την sys\_settickets

```
int sys_settickets(void){
    int tickets;
    if(argint(0,&tickets)<0 || tickets<0){
        return -1;
    }
    else{
        proc->tickets=tickets;
    }
    return 0;
}
```

- proc.c
  - ορίζουμε γεννήτρια τυχαίων αριθμών: Επιστρέφει έναν αριθμό μικρότερο από το max που θα είναι ίσο με το πλήθος των tickets, δηλαδή των μέγιστο αριθμό εισιτηρίων που θα μπορούσε να έχει μια διεργασία.

```
// Gennhtria tyxaiwn ari8mwn
static uint64 next1 = 1;
int prand(int max) // RAND_MAX assumed to be 100
{
    next1 = next1 * 1103515245 + 12345;
    return (unsigned int) (next1 / 65536) % max;
}
```

- Αλλαγή της scheduler, ώστε να υλοποιεί έναν αλγόριθμο κλήρωσης.

Αναλυτική εξήγηση με **σχόλια**:

```
void scheduler(void) {
    struct proc *p;

    for(;;) {
        // Enable interrupts on this processor.
        sti();

        int total_tickets = 0; //υπολογισμός συνολικού αριθμού tickets
        //των RUNNABLE διεργασιών στο rtable (υποψήφιος για επιλογή)
```

```

        acquire(&ptable.lock); //πρέπει να έχουμε το lock στο ptable
//για αποκλειστική πρόσβαση
        for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            if (p->state == RUNNABLE) {
                total_tickets += p->tickets;
            }
        }
//Αν συνολικός αριθμός tickets=0 : ο sched δεν πρέπει να κάνει τίποτα
//Ο scheduler πρέπει να απελευθερώσει το ptable και να περιμένει
        if (total_tickets == 0) {
            release(&ptable.lock); //άλλες διεργασίες έχουν access
            hlt(); // CPU δεν εκτελεί εντολές έως interrupt
            continue;
        }
        int test = prand(total_tickets); //τυχαίος αριθμός του lottery
        // Ξαναδιατρέχει το ptable έως ότου κάποια διεργασία να έχει
        παραπάνω εισιτήρια από το test
        for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            if (p->state != RUNNABLE) {
                continue; //αγνοώ όποια δεν μπορεί να τρέξει
            }
            if (test < p->tickets) { //έγινε η επιλογή
                proc = p; //current process = το p που επιλέξαμε
                switchvm(p); // Switch to user virtual memory
                p->state = RUNNING;
                p->inuse = 1;
                swtch(&cpu->scheduler, p->context); // Context switch
//διεργασία τελείωσε άρα αλλαγή virtual memory σε αυτή του kernel
                switchkvm();
                proc = 0; // Process stops running for now; reset proc
                break; // Exit the inner loop
//αλλιώς μόλις τελειώσει η επιλεγμένη διεργασία το loop θα συνεχιστεί
//από αυτήν και μετά
// Μόλις μια διεργασία εκτελείται και τελειώσει τη χρήση της CPU,
//μπορεί να αλλάξει την p->state της πριν επιστρέψει στον scheduler.
//Εάν συνεχιστεί το loop χωρίς break, μπορεί να επεξεργάζεται
//διεργασίες των οποίων η κατάσταση δεν είναι πλέον RUNNABLE. Θέλουμε
//επιστροφή στο εξωτερικό loop άρα κάνουμε break
            }
            else { //μειώνω το test κάθε φορά, καθώς άμα τύχει πολύ
                //μεγάλος αριθμός test ίσως να μην επιλεγεί καμία διεργασία! Π.χ. αν δυο
                //διεργασίες με 200 tickets και τύχει test 300 δεν θα τρέχει καμία
                test -= p->tickets;
            }
        } //όταν τελειώνουμε απελευθερώνουμε το lock πριν το ζητήσουμε
        //ανά στην αρχή της λούπας
        release(&ptable.lock);
    }
}

```

- Αλλαγή της allocproc() ώστε κάθε διεργασία που δημιουργείται να έχει τουλάχιστον ένα εισιτήριο. έτσι η Init θα έχει ένα εισιτήριο και ο scheduler θα την επιλέξει για να τρέξει (αλλιώς θα υπάρχει ατέρμονος βρόχος χωρίς καμία διεργασία να εκτελείται)

```
static struct proc*
allocproc(void)
{
    struct proc *p;
    char *sp;

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == UNUSED)
            goto found;
    release(&ptable.lock);
    return 0;

found:
    p->state = EMBRYO;
    p->pid = nextpid++;
    p->ticks = 0;
    p->tickets=1; //ΑΡΧΙΚΟΠΟΙΗΣΗ ΣΕ ΕΝΑ ΕΙΣΗΤΗΡΙΟ ΠΑΝΤΑ
//ΤΙΜΗ ΜΠΟΡΕΙ ΝΑ ΑΛΛΑΞΕΙ ΟΤΑΝ ΚΛΗΘΕΙ Η SETTICKETS
    release(&ptable.lock);
    ...
}
```

- user/lotteryschedtest: Χρειάζεται τώρα να τεστάρουμε τον scheduler μας. Στο ήδη έτοιμο lotteryschedtest.c διορθώνουμε ένα bug που έχει να κάνει με την εκτύπωση των πληροφοριών (έλειπε η αρχικοποίηση των στοιχείων των array σε -1):

```
void print_info() {
    int index[N] ;
    int ticks[N] ;
    int tticks = 0;
    for(int i=0;i<N;i++){
        index[i]=-1;
        ticks[i]=-1;} //όλες οι τιμές Initialized σε -1
    ...
}
```

### Έλεγχος:

```
$ lotteryschedtest
```

```
Forked 5 children to share ~3000 ticks (real 3000)
```

PID: 5	TICKETS: 200	TICKS: 676	CPU: 22.5%
PID: 6	TICKETS: 100	TICKS: 446	CPU: 14.8%
PID: 7	TICKETS: 500	TICKS: 1106	CPU: 36.8%
PID: 8	TICKETS: 50	TICKS: 176	CPU: 5.8%
PID: 9	TICKETS: 150	TICKS: 596	CPU: 19.8%

### Εξήγηση:

Ο αριθμός τεστ επιλέγεται τυχαία, όμως όσο μεγαλύτερος ο αριθμός των tickets που έχει μια διεργασία, τόσες περισσότερες οι πιθανότητες να είναι μεγαλύτερη του τεστ. Υπάρχει βέβαια και εξάρτηση από την σειρά των διεργασιών στο ptable, η πρώτη με αριθμό μεγαλύτερο του τεστ εκτελείται. Περιμένουμε λοιπόν όχι ακριβή αναλογία, αλλά μια συσχέτιση μεταξύ του αριθμού εισιτηρίων και χρόνο στη cru.

Όντως η διεργασία με τα 500 εισιτήρια έχει τον περισσότερο χρόνο και οι υπόλοιπες ακολουθούν με φθίνουσα σειρά (λιγότερα εισιτήρια λιγότερος χρόνος).

Το αποτέλεσμα είναι λογικό.