Λειτουργικά Συστήματα Δραστηριότητα 2

2024-2025

ΑΜ: 1096060

Ονοματεπώνυμο: Μαρία-Νίκη Ζωγράφου

Περιεχόμενα

[Άσκηση 1: 3](#_Toc181474868)

[Άσκηση 2: 5](#_Toc181474869)

[Συνάρτηση scheduler: 5](#_Toc181474870)

[Συνάρτηση sched: 6](#_Toc181474871)

[Άσκηση 3: 9](#_Toc181474872)

## 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 };

structproc {

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 αρχη 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. |

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

      switchuvm(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;

}

1ος 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).

2ος 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 ταυτοχρόνως 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.

3ος 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.

4ος 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.

5ο 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.

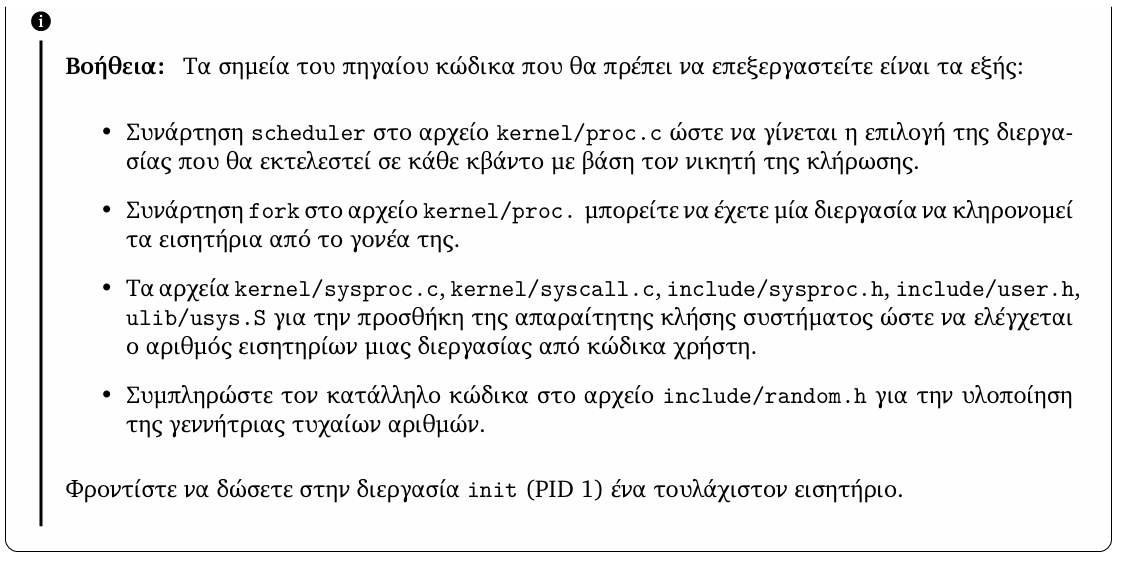
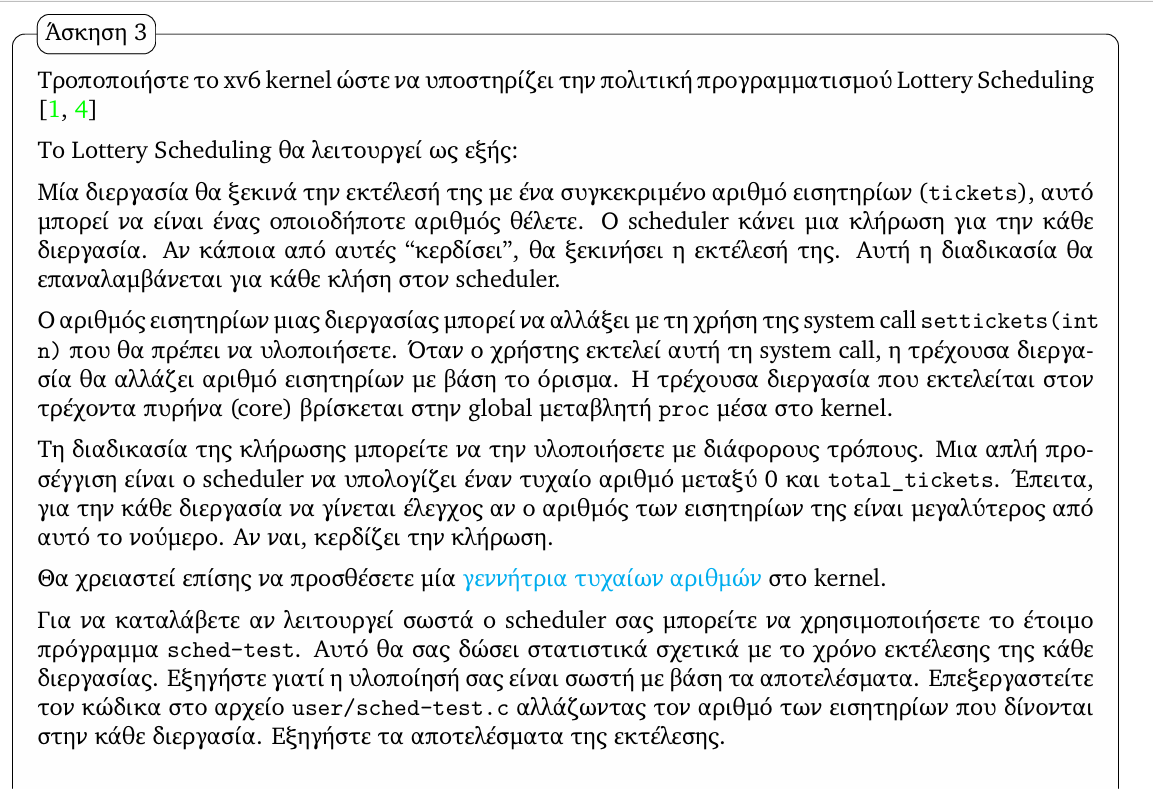
6ο βήμα:

  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:



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.

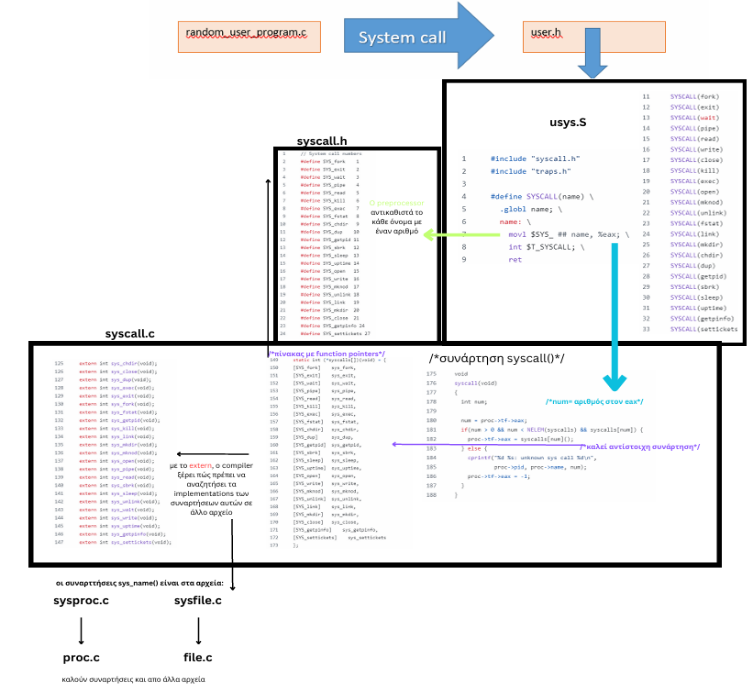
Πώς δημιουργείται μια διεργασία στο xv6;

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 εισιτήριο.

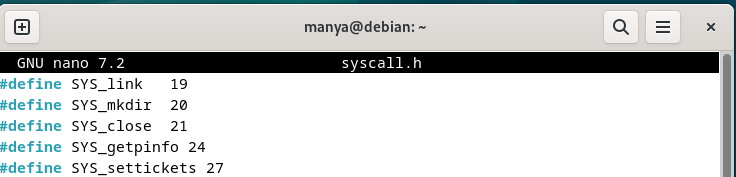
Στο αρχείο user/sh.c από τις συναρτήσεις main() και runcmd() παρατηρούμε ότι καλείται η fork1() κάθε φορά που ο χρήστης γράφει κάποια εντολή. Είναι γνωστό πως για να δημιουργηθεί ένα νέο Process καλείται η fork(). Επομένως πρέπει να φροντίσουμε ότι η διεργασία παιδί κληρονομεί εισιτήρια από την γονική της διεργασία.

Πώς θα δίνονται τα εισιτήρια;

Τα εισιτήρια θα πρέπει να δοθούν από τον χρήστη όταν καλέσει την system call settickets. Αυτό γίνεται τεστ από το αρχείο user/lotteryschedtest.c

Υλοποίηση της settickets:

Από το διάγραμμα της προηγούμενης εργασίας γνωρίζουμε ότι πρέπει να επεξεργαστούμε όλα τα αρχεία που συμμετέχουν στην διαδικασία κλήσης ενός system call.

* user.h: Προσθέτουμε την int settickets(int n); στο αρχείο include/user.h.
* usys.S: Υπάρχει ήδη ορισμός της SYSCALL(settickets)
* syscall.h: Στην συνάρτηση sys\_SETTICKETS έχει ήδη δοθεί αριθμός αντιστοίχισης.
* 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\_setickets

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 ptable (υποψήφιες 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 πρεπει 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

                switchuvm(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 τεστ 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 ξανά 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 εισητήριο 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; //ΑΡΧΙΚΟΠΟΊΗΣΗ ΣΕ ΕΝΑ ΕΙΣΗΤΗΡΊΟ ΠΆΝΤΑ

//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] ;

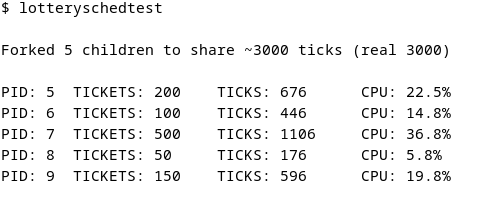
    int tticks = 0;

    for(int i=0;i<N;i++){

        index[i]=-1;

        ticks[i]=-1;} //όλες οι τιμές Initialized σε -1

  ...

**Έλεγχος:**

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 cpu.

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.