

ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

Δραστηριότητα 2

ΜΠΑΣΑΓΙΑΝΝΗ ΓΕΩΡΓΙΑ 1084016

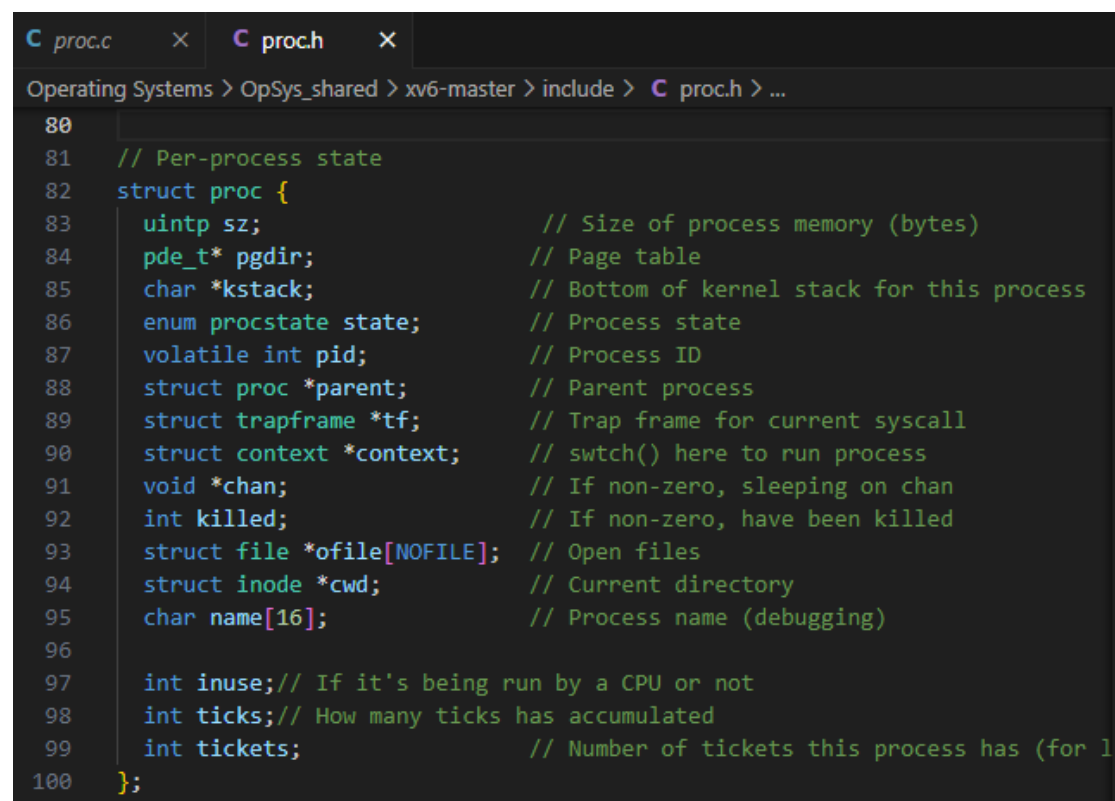
ΠΑΝΟΥΡΓΙΑΣ ΑΝΤΩΝΙΟΣ 1083996

ΣΤΕΡΓΙΟΠΟΥΛΟΣ ΓΕΩΡΓΙΟΣ 1083861

Άσκηση 1

Μια διεργασία αναπαριστάται στο xv6 kernel σαν μια δομή (struct) με όνομα `proc` όπου ορίζεται στο `xv6/include/proc.h`

Τα μέλη της δομής αυτής φαίνονται στην φωτογραφία και θα εξηγηθούν παρακάτω.



```
80
81 // Per-process state
82 struct proc {
83     uintp sz; // Size of process memory (bytes)
84     pde_t* pgdir; // Page table
85     char *kstack; // Bottom of kernel stack for this process
86     enum procstate state; // Process state
87     volatile int pid; // Process ID
88     struct proc *parent; // Parent process
89     struct trapframe *tf; // Trap frame for current syscall
90     struct context *context; // switch() here to run process
91     void *chan; // If non-zero, sleeping on chan
92     int killed; // If non-zero, have been killed
93     struct file *ofile[NOFILE]; // Open files
94     struct inode *cwd; // Current directory
95     char name[16]; // Process name (debugging)
96
97     int inuse; // If it's being run by a CPU or not
98     int ticks; // How many ticks has accumulated
99     int tickets; // Number of tickets this process has (for 1
100 };
```

Αξίζει να σημειωθεί ότι στο `xv6/kernel/proc.c` γίνεται η ανάθεση τιμών στα παρακάτω member για κάθε διεργασία καθώς και η δημιουργία ενός πίνακα που περιέχει όλες τις διεργασίες ο οποίος αξιοποιείται από το λειτουργικό σύστημα για την διαχείριση τους.

- ➔ `uintp sz`: Το μέγεθος του χώρου μνήμης που θα δεσμεύσει η διεργασία. (unassigned int)
- ➔ `pde_t* pgdir`: Είναι ο page table που έχει τις τιμές για το mapping μεταξύ των virtual και των physical addresses.
- ➔ `char * kstack`: Είναι η χαμηλότερη διεύθυνση του stack του kernel για την συγκεκριμένη διεργασία
- ➔ `enum procstate state`: Το state που βρίσκεται η διεργασία μπορεί να πάρει τιμές `UNUSED/EMBRYO/SLEEPING/RUNNABLE/RUNNING/ZOMBIE`
- ➔ `volatile int pid`: Ο μοναδικός αριθμός/κωδικός της διεργασίας.

- struct proc *parent: είναι δείκτης σε μια άλλη διεργασία και συγκεκριμένα στον γονέα της.
- struct trapframe *tf: Εδώ διατηρεί έναν δείκτη σε μια δομή trapframe η οποία έχει τις τιμές των καταχωρητών του userspace πριν αλλάξει σε kernel mode μέσω μιας syscall.
- struct context *context: Αποθηκεύει τις τιμές των καταχωρητών που χρειάζονται για την επιτύχη μεταγωγή περιβάλλοντος (context switch)
- void *chan: Όσο δεν είναι μηδενικό σημαίνει ότι η διεργασία είναι σε sleep mode αναμένοντας user input (μέσω κάποιου interrupt) για wake up.
- int killed: Όταν αυτή η μεταβλητή πάρει τιμή διάφορη του μηδενός σημαίνει ότι η διεργασία έχει γίνει killed. Διαφορετικά είναι ενεργή.
- struct file *ofile[NOFILE]: Εδώ έχουμε μια λίστα με τα αρχεία που είναι open κατά την εκτέλεση μιας διεργασίας. Αυτό χρειάζεται για την αποφυγή conflict απο διεργασίας που ζητάνε πρόσβαση σε ίδια αρχεία.
- struct inode *cwd: Εδώ είναι αποθηκευμένο το Current Working Directory.
- char name[16]: Είναι ένα string το οποίο αποδίδει ένα όνομα στην εργασία για διευκόλυνση στο debugging.
- int inuse: Η τιμή αυτή δείχνει αν η διεργασία αυτή την στιγμή τρέχει στην ΚΜΕ.
- int ticks: Το πλήθος των ticks απο την αρχικοποίηση της διεργασίας.
- int tickets: Πόσα tickets έχουν εκδοθεί για την συγκεκριμένη διεργασία. Αυτό θα αξιοποιηθεί για lottery scheduling.

Άσκηση 2

Η συνάρτηση void scheduler(void) ορίζεται στο `xv6/kernel/proc.c`:

```

258 void
259 scheduler(void)
260 {
261     struct proc *p = 0;
262
263     for(;;){
264         // Enable interrupts on this processor.
265         sti();
266
267         // no runnable processes? (did we hit the end of the
268         // if so, wait for irq before trying again.
269         if (p == &ptable.proc[NPROC])
270             hlt();
271
272         // Loop over process table looking for process to run
273         acquire(&ptable.lock);
274         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
275             if(p->state != RUNNABLE)
276                 continue;

```

```

281             proc = p;
282             switchvm(p);
283             p->state = RUNNING;
284             p->inuse = 1;
285             const int tickstart = ticks;
286
287             swtch(&cpu->scheduler, proc->context);
288
289             p->ticks += ticks - tickstart;
290
291             switchkvm();
292             // Process is done running for now.
293             // It should have changed its p->state
294             proc = 0;
295         }
296         release(&ptable.lock);
297     }
298 }
299

```

Η συνάρτηση αυτή αρχικά «μπαίνει» σε ένα μόνιμο loop (`for(;;)`) και ενεργοποιεί τα interrupts. Αν έχει φτάσει στο τέλος του process table στο προηγούμενο iteration τότε γίνεται `hlt()`. Στην συνέχεια αφού κάνει fencing τον process table μέσω των `acquire` και `release`, διατérνoν τον process table ελέγχοντας την τιμή του member `State` του `struct proc` μέχρι να βρεί μια διεργασία σε κατάσταση `RUNNABLE`. Όταν την βρεί κάνει switch στην συγκεκριμένη διεργασία με κλήση της `switchvm` και αλλάζει τόσο το state της διεργασίας σε `RUNNING` όσο και την τιμή `inuse` σε 1 (όπου εξηγήσαμε παραπάνω την σημασία της). Μέσω της `swtch` περνάει το context της συγκεκριμένης διεργασίας, δηλαδή φορτώνει στους καταχωρητές τις κατάλληλες τιμές για την εκτέλεση της. Τέλος μετράει τα ticks που πέρασαν κατά την εκτέλεσή της και επιστρέφει στο scheduler μέσω της `switchkvm()` αφού έχει αλλάξει και πάλι το state της.

Η συνάρτηση `void sched(void)` βρίσκεται στο `xv6/kernel/proc.c` :

```
303 void
304 sched(void)
305 {
306     int intena;
307
308     if(!holding(&ptable.lock))
309         panic("sched ptable.lock");
310     if(cpu->ncli != 1)
311         panic("sched locks");
312     if(proc->state == RUNNING)
313         panic("sched running");
314     if(readeflags() & FL_IF)
315         panic("sched interruptible");
316     intena = cpu->intena;
317     swtch(&proc->context, cpu->scheduler);
318     cpu->intena = intena;
319 }
```

Όταν η διεργασία που εκτελείται θέλει να εγκαταλείψει την CPU καλεί την συνάρτηση `sched`. Δηλαδή επιστρέφει την διεργασία στον scheduler και «παγώνει» προσωρινά, μέχρι να επιστρέψει αργότερα και πάλι στην CPU.

Παρατηρούμε την χρήση της συνάρτησης `swtch()` στην `scheduler()` και στην `sched()` για την υλοποίηση μεταγωγής περιβάλλοντος από scheduler σε process και από process στο scheduler αντίστοιχα.

Άσκηση 3 – Lottery Scheduling

Στα πλαίσια της υλοποίησης του Lottery Scheduling όπως αυτό περιγράφεται στην εκφώνηση της Δραστηριότητας 2 χρειάστηκε να κάνουμε τις παρακάτω τροποποιήσεις στον κώδικα:

- Αρχικά ορίσαμε με τον γνωστό τρόπο από την προηγούμενη δραστηριότητα την system call settickets(int n) που θα χρησιμοποιήσουμε για να αλλάζουμε την τιμή των tickets κάθε διεργασίας αργότερα στην schedtest.c

```
C user.h M X
include > C user.h > settickets(int)
25 int uptime(void); setticket
26 int getpinfo(struct pstat*
27 int getfavnum(void);
28 void halt(void);
29 int getcount(int);
30 int killrandom(void);
31 int settickets(int);
32
```

```
C syscall.h M X
include > C syscall.h > SYS_settickets
20 #define SYS_link 19
21 #define SYS_mkdir 20
22 #define SYS_close 21
23 #define SYS_getpinfo 24
24 #define SYS_getfavnum 23
25 #define SYS_halt 25
26 #define SYS_getcount 26
27 #define SYS_killrandom 27
28 #define SYS_settickets 28
```

```
C proc.c M ● C syscall.c 9+, M X
kernel > C syscall.c > syscalls
174 [SYS_mkdir] sys_mkdir,
175 [SYS_close] sys_close,
176 [SYS_getpinfo] sys_getpinfo,
177 [SYS_getfavnum] sys_getfavnum,
178 [SYS_halt] sys_halt,
179 [SYS_getcount] sys_getcount,
180 [SYS_killrandom] sys_killrandom,
181 [SYS_settickets] sys_settickets,
182 ];
```

```
ASM usys.S M X
ulib > ASM usys.S
30 SYSCALL(sleep)
31 SYSCALL(uptime)
32 SYSCALL(getpinfo)
33 SYSCALL(getfavnum)
34 SYSCALL(halt)
35 SYSCALL(getcount)
36 SYSCALL(killrandom)
37 SYSCALL(settickets)
```

```
162 int sys_settickets(void)
163 {
164     int n,f;
165     f = proc->tickets;
166     argint(0, &n);
167     proc->tickets=n;
168     cprintf("changed tickets of proccess : %d from %d to %d\n",proc->pid,f,proc->tickets);
169     return n;
170 }
```

- Ορίσαμε επίσης μια γεννήτρια τυχαίων αριθμών την οποία υλοποιήσαμε στο `xv6/kernel/proc.c` με την βοήθεια κώδικα από το διαδίκτυο. Η γεννήτριά μας βασίστηκε σε 2 συναρτήσεις `int rand(void)` και `void srand(unsigned int seed)`. Η κλήση της `srand` με όρισμα έναν unsigned int ορίζει μια τιμή στην static μεταβλητή `next` η οποία θα χρησιμοποιηθεί από την `rand` μετέπειτα σαν seed για να μας δώσει έναν τυχαίο ακέραιο. Επιλέξαμε, όπως θα δούμε παρακάτω να χρησιμοποιούμε ως seed σε κάθε κλήση της `rand` την τιμή των ticks της CPU, που μας προσδίδει την τυχαιότητα που απαιτείται για την συγκεκριμένη εφαρμογή.

```

503 int rand(void)
504 {
505     next=next * 1103515245+12345;
506     return (unsigned int) (next/65536)%100;
507 }
508
509 void srand(unsigned int seed)
510 {
511     next=seed;
512 }

```

- Στο αρχείο proc.c του kernel όπου γίνεται η αρχικοποίηση των διεργασιών τροποποιήσαμε τον κώδικα της συνάρτησης `allocproc` με τον τρόπο που φαίνεται παρακάτω έτσι ώστε κάθε νέα διεργασία που δημιουργείται (`p->state=EMBRYO`) να αποκτά default τιμή `tickets=50`.

```

33 static struct proc*
34 allocproc(void)
35 {
36     struct proc *p;
37     char *sp;
38
39     acquire(&ptable.lock);
40     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
41         //p->tickets=50;
42         if(p->state == UNUSED)
43             goto found;
44     }
45     release(&ptable.lock);
46     return 0;
47
48 found:
49     p->state = EMBRYO;
50     p->tickets=50;
51     p->pid = nextpid++;
52     release(&ptable.lock);
53
54     // Allocate kernel stack.
55     if((p->kstack = kalloc()) == 0){
56         p->state = UNUSED;
57         return 0;
58     }
59     sp = p->kstack + KSTACKSIZE;
60

```

- Στο αρχείο proc.c του kernel επιπλέον τροποποιήσαμε τον έλεγχο που κάνει ο scheduler όταν ψάχνει RUNNABLE διεργασίες ώστε να λαμβάνει υπόψιν του τον αριθμό των `tickets`. Πλέον για να πάρει μια διεργασία τον έλεγχο της CPU πρέπει να είναι RUNNABLE και να έχει αριθμό `tickets` μεγαλύτερο απο έναν τυχαίο κάθε φορά

ακέραιο αριθμό (κλήρωση). Με αυτόν τον τρόπο κατανοούμε ότι όσο μεγαλύτερο αριθμό tickets έχει μια διεργασία τόσο πιο πιθανό είναι αυτή να λάβει περισσότερο χρόνο στην CPU.

Πιο συγκεκριμένα, καθώς η συνάρτηση scheduler διαπερνά όλο τον πίνακα διεργασιών και όσο δεν βρίσκει κάποια RUNNABLE συνεχίζει το loop χωρίς να κάνει κάτι.

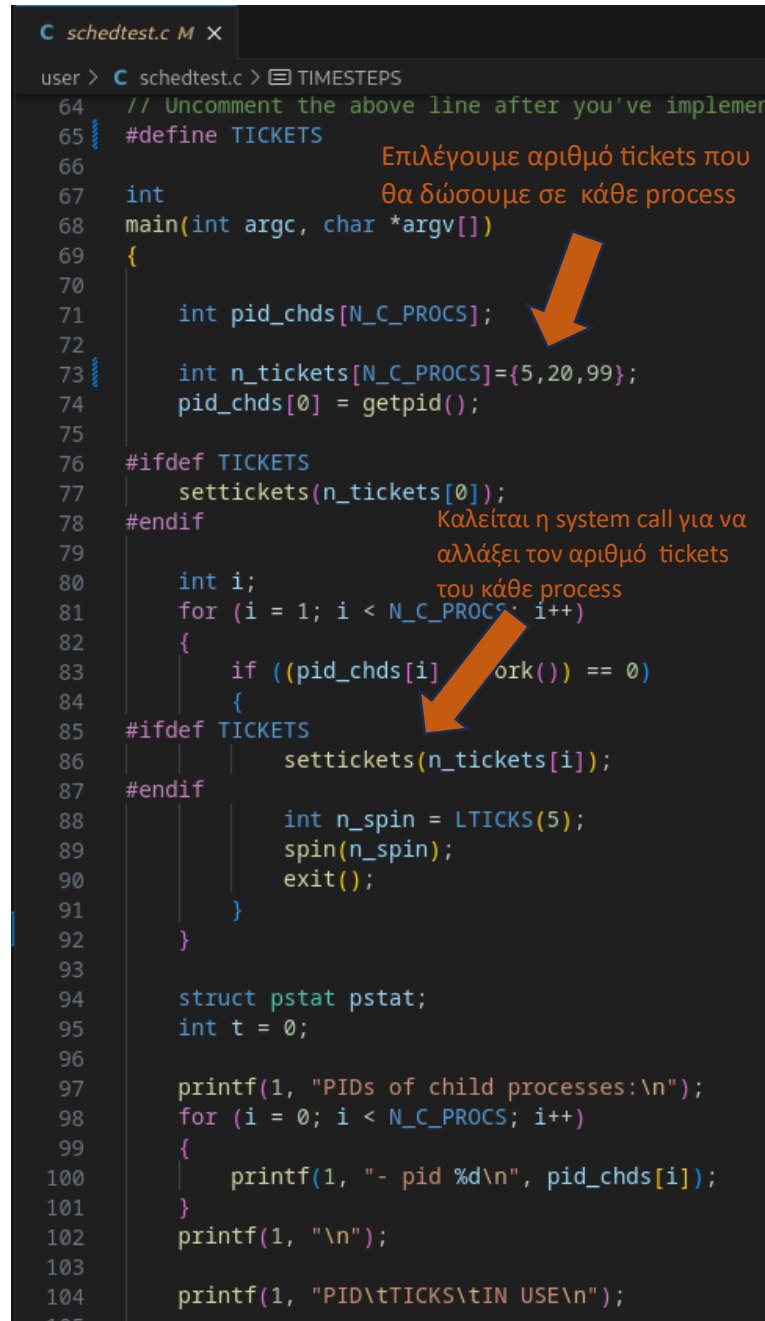
Έστω μια διεργασία είναι RUNNABLE, τότε θα προχωρήσει στον περεταίρω έλεγχο των tickets. Αν τώρα αυτά είναι λιγότερα από τον τυχαίο αριθμό της τελευταίας κλήρωσης, θα κάνει continue και θα προσπαθήσει να βρει άλλη διεργασία. Όταν, τελικά, βρει μια διεργασία RUNNABLE και με τον απαιτούμενο αριθμό tickets θα προχωρήσει κανονικά στον υπόλοιπο κώδικα του scheduler που κάνει το context switch και δίνει την CPU στην συγκεκριμένη διαδικασία.

```
267 void scheduler(void)
268 {
269     struct proc *p = 0;
270     int i = 0;
271     for(;;){
272         // Enable interrupts on this processor.
273         sti();
274
275         // no runnable processes? (did we hit the end of the t
276         // if so, wait for irq before trying again.
277
278         if (p == &ptable.proc[NPROC])
279             hlt();
280
281         //random number of tickets
282         srand(ticks);
283         int rand_num=rand();
284         // Loop over process table looking for process to run.
285         acquire(&ptable.lock);
286         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
287             //cprintf("%d\n",p->tickets);
288             if(p->state != RUNNABLE) |
289                 continue;
290             if(p->tickets < rand_num) // EDW ELEGXOS WINNER DRAS
291             {
292                 //cprintf("\n DEN ETREXE I %d\n",p->tickets);
293                 continue;
294             }
295         }
296     }
297 }
```

Δημιουργία τυχαίου
ακέραιου αριθμού με τις
συναρτήσεις που
περιγράψαμε παραπάνω
και με seed τα current ticks

Έλεγχος αν τα tickets της
διεργασίας είναι πάνω
απο τον τυχαίο αριθμό

- Τέλος εφόσον στο πρώτο βήμα υλοποιήσαμε την settickets system call, κάναμε τα απαραίτητα uncomment στο αρχείο xv6/user/schedtest.c έτσι ώστε να τροποποιήσουμε τον αριθμό tickets της κάθε διεργασίας και να μελετήσουμε την συμπεριφορά του λειτουργικού με το lottery scheduling. Πόσο δηλαδή χρόνο δίνει σε κάθε διεργασία με βάση τα tickets που αυτή έχει.



```

C schedtest.c M X
user > C schedtest.c > TIMESTEPS
64 // Uncomment the above line after you've implemented
65 #define TICKETS
66
67 int
68 main(int argc, char *argv[])
69 {
70
71     int pid_chds[N_C_PROCS];
72
73     int n_tickets[N_C_PROCS] = {5, 20, 99};
74     pid_chds[0] = getpid();
75
76 #ifdef TICKETS
77     settickets(n_tickets[0]);
78 #endif
79
80     int i;
81     for (i = 1; i < N_C_PROCS; i++)
82     {
83         if ((pid_chds[i] = fork()) == 0)
84         {
85 #ifdef TICKETS
86             settickets(n_tickets[i]);
87 #endif
88             int n_spin = LTICKS(5);
89             spin(n_spin);
90             exit();
91         }
92     }
93
94     struct pstat pstat;
95     int t = 0;
96
97     printf(1, "PIDs of child processes:\n");
98     for (i = 0; i < N_C_PROCS; i++)
99     {
100         printf(1, "- pid %d\n", pid_chds[i]);
101     }
102     printf(1, "\n");
103
104     printf(1, "PID\tTICKS\tIN USE\n");
105

```

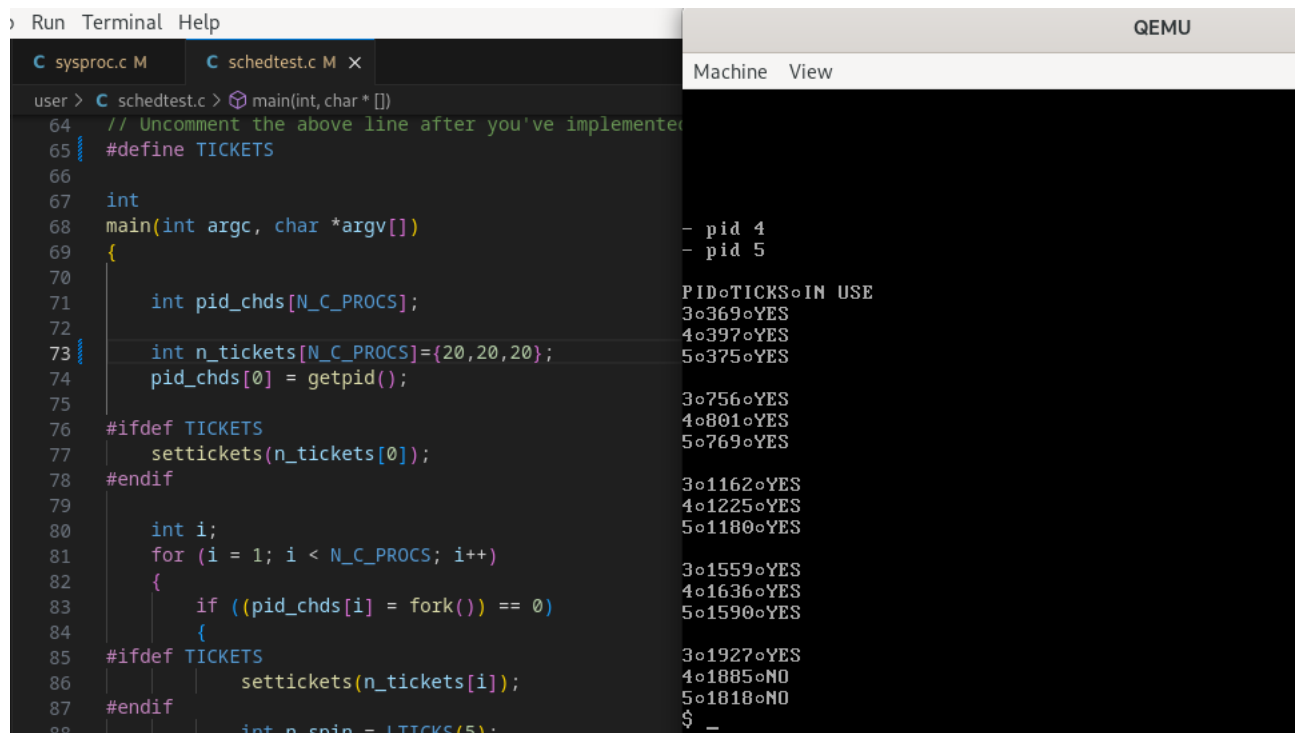
Επιλέγουμε αριθμό tickets που θα δώσουμε σε κάθε process

Καλείται η system call για να αλλάξει τον αριθμό tickets του κάθε process

Παραδείγματα Εκτέλεσης για διάφορες τιμές στα tickets

Σε αυτό το σημείο θα δούμε την συμπεριφορά του λειτουργικού που εφαρμόζει lottery scheduling για διάφορες αρχικές τιμές στα tickets των διεργασιών με την βοήθεια της schedtest.

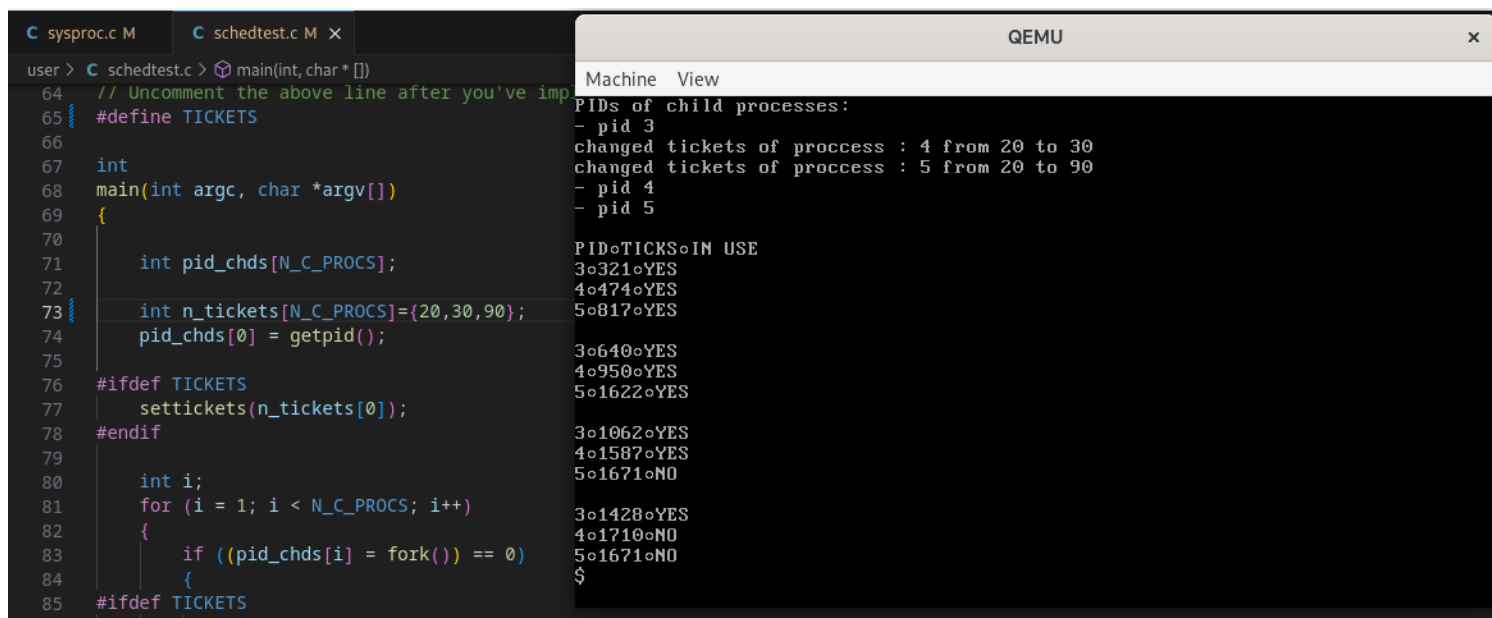
- Περίπτωση 1: Ίδιος αριθμός tickets



```
user > C schedtest.c > main(int, char * [])
64 // Uncomment the above line after you've implemented
65 #define TICKETS
66
67 int
68 main(int argc, char *argv[])
69 {
70     int pid_chds[N_C_PROCS];
71
72     int n_tickets[N_C_PROCS]={20,20,20};
73     pid_chds[0] = getpid();
74
75     #ifdef TICKETS
76         settickets(n_tickets[0]);
77     #endif
78
79     int i;
80     for (i = 1; i < N_C_PROCS; i++)
81     {
82         if ((pid_chds[i] = fork()) == 0)
83         {
84             #ifdef TICKETS
85                 settickets(n_tickets[i]);
86             #endif
87             int n_spin = TICKETS(5);
88         }
89     }
```

Παρατηρούμε ότι όλες οι διεργασίες μένουν στην CPU για περίπου ίσα χρονικά διαστήματα, λογικό εφόσον έχουν και τον ίδιο αριθμό tickets.

- Περίπτωση 2: Αυξανόμενος αριθμός tickets



```
user > C schedtest.c > main(int, char * [])
64 // Uncomment the above line after you've implemented
65 #define TICKETS
66
67 int
68 main(int argc, char *argv[])
69 {
70     int pid_chds[N_C_PROCS];
71
72     int n_tickets[N_C_PROCS]={20,30,90};
73     pid_chds[0] = getpid();
74
75     #ifdef TICKETS
76         settickets(n_tickets[0]);
77     #endif
78
79     int i;
80     for (i = 1; i < N_C_PROCS; i++)
81     {
82         if ((pid_chds[i] = fork()) == 0)
83         {
84             #ifdef TICKETS
85                 settickets(n_tickets[i]);
86             #endif
87             int n_spin = TICKETS(5);
88         }
89     }
```


Όπως ήταν αναμενόμενο με την νέα επιλογή tickets η Τρίτη διεργασία παίρνει παραπάνω χρόνο την CPU. Ο λόγος που στο τέλος είναι περισσότερα τα ticks της δεύτερης απο την 3^η είναι διότι η 3^η ολοκληρώθηκε ενα κύκλο νωρίτερα απο την 2^η.

- Περίπτωση 3: Η δεύτερη περισσότερα tickets απο τις άλλες δύο.

```

user > C schedtest.c M X
64 // Uncomment the above line after you've imple
65 #define TICKETS
66
67 int
68 main(int argc, char *argv[])
69 {
70
71     int pid_chds[N_C_PROCS];
72
73     int n_tickets[N_C_PROCS]={20,80,20};
74     pid_chds[0] = getpid();
75
76 #ifdef TICKETS
77     settickets(n_tickets[0]);
78 #endif
79
80     int i;
81     for (i = 1; i < N_C_PROCS; i++)
82     {
83         if ((pid_chds[i] = fork()) == 0)
84         {
85 #ifdef TICKETS
86             settickets(n_tickets[i]);

```

Machine View

```

PIDs of child processes:
- pid changed tickets of proccess : 4 from 20 t
3
- pid 4
- pid 5

PID0TICKS0IN USE
changed tickets of proccess : 5 from 20 to 20
303040YES
408170YES
503460YES

306130YES
4016030NO
507010YES

3010210YES
4016030NO
5012060YES

3014160YES
4016030NO
5017110NO
$

```

Η συμπεριφορά αυτή είναι αναμενόμενη και εξηγείται όπως παραπάνω. Έχει περισσότερα tickets, άρα παίρνει περισσότερο χρόνο στην CPU και άρα εκτελείται και γρηγορότερα (αυτό δεν ισχύει πάντα διότι εξαρτάται και από το μέγεθος της διεργασίας).

- Περίπτωση 4: Η πρώτη περισσότερα tickets απο τις άλλες δύο.

```

user > C schedtest.c M X
64 // Uncomment the above line after you've implemented the
65 #define TICKETS
66
67 int
68 main(int argc, char *argv[])
69 {
70
71     int pid_chds[N_C_PROCS];
72
73     int n_tickets[N_C_PROCS]={80,20,20};
74     pid_chds[0] = getpid();
75
76 #ifdef TICKETS
77     settickets(n_tickets[0]);
78 #endif
79
80     int i;
81     for (i = 1; i < N_C_PROCS; i++)
82     {
83         if ((pid_chds[i] = fork()) == 0)
84         {
85 #ifdef TICKETS
86             settickets(n_tickets[i]);
87 #endif
88
89         int n_spin = LTICKS(5);
90         spin(n_spin);
91         exit();
92     }

```

Machine View

```

3022830YES
409200YES
509120YES

3026040YES
4010520YES
5010410YES

3029270YES
4011840YES
5011700YES

3032470YES
4013150YES
5012990YES

3035940YES
4014570YES
5014460YES

3039460YES
4015770NO
5015050NO
$ -

```

Αυτή η περίπτωση έχει ιδιαίτερο ενδιαφέρον καθώς παρουσιάζει τον πιο αργό χρόνο συνολικής εκτέλεσης των διεργασιών.

Η λογική εξήγηση είναι ότι η 2^η και η 3^η διεργασία είναι παιδιά της 1^{ης} και ως εκ τούτου η 1^η δεν μπορεί να ολοκληρωθεί αν δεν τελειώσουν πρώτα τα παιδιά της. Επομένως ο επιπλέον χρόνος που χρειάζεται στην CPU απλά καθυστερεί τα παιδιά της από το να εκτελεστούν, με αποτέλεσμα να μην τερματίζεται γρήγορα.