

Λειτουργικά Συστήματα

2η Δραστηριότητα

Κωνσταντίνος Τσάμπρας

up1083865

➔ 1)

Το σύστημα αντιμετωπίζει την κάθε διεργασία σαν ένα struct το οποίο έχει τα κατάλληλα πεδία για να αποθηκεύει τις πληροφορίες του κάθε process που χρειάζεται για να διαχειρίζεται όλες τις διεργασίες:

```
GNU nano 7.2                                proc.h
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 sche
};
```

Παραπάνω βλέπουμε την μορφή αυτού του struct (-τύπου δεδομένων).

- sz: μέγεθος της μνήμης της διεργασίας σε bytes
- pgdir: δείκτης για πίνακα (page dir) που βοηθάει στην αντιστοίχιση υλικής-φυσικής μνήμης
- kstack: στοίβα πυρήνα για syscalls και interrupts
- state: κατάσταση διεργασίας (Running, Runnable, Zombie)
- pid: μοναδικά αναγνωριστικό διεργασίας
- parent: δείκτης στην διεργασία-πατέρα, την διεργασία που δημιούργησε την διεργασία μας
- tf: χρησιμοποιείται για να αποθηκευτεί την κατάσταση του cpu για περιπτώσεις hardware interrupt, syscall και exceptions.
- context: χρησιμοποιείται για να αποθηκευτεί κάθε πληροφορία της διεργασίας (η κατάσταση του cpu, δείκτες αρχείων, αντιστοιχίσεις στη μνήμη κ.α.). Χρησιμοποιείται για το context-switch
- chan: λόγος που η διεργασία βρίσκεται σε «ύπνο»
- killed: σημαία που δείχνει αν έχει τερματιστεί η διεργασία
- onefile: πίνακας δεικτών σε αρχεία
- cwd: current working directory
- name: όνομα διεργασίας
- inuse: σημαία για το αν η διεργασία βρίσκεται σε χρήση (αν εκτελείται από τον επεξεργαστή)
- ticks: Αριθμός "ticks" του συστήματος για τα οποία η διεργασία έχει υπάρξει σε εκτέλεση
- tickets: Αριθμός εισιτηρίων για τον αλγόριθμο lottery.

➔ 2)

- Sched:

```
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;
}
```

Ο ρόλος της sched είναι να εναλλάσσει την εκάστοτε διεργασία με τον scheduler. Για να το πετύχει αυτό αρχικά κάνει διάφορους ελέγχους:

1. Έλεγχο για να είμαστε σίγουροι ότι έχουμε τον έλεγχο της διεργασίας.
2. Έλεγχο για το αν ο επεξεργαστής βρίσκεται σε κατάσταση διακοπής (interrupt)
3. Έλεγχο για το αν η διεργασία όντως είναι σε κατάσταση running (δηλαδή αν εκτελείται αυτήν την στιγμή)
4. Έλεγχο για το αν οι διακοπές είναι ενεργοποιημένες (δηλαδή αν επιτρέπονται)

Στην συνέχεια αποθηκεύει την κατάσταση του intena (interrupts enabled) και καλεί την swtch η οποία αποθηκεύει τα δεδομένα της διεργασίας υπό εκτέλεση στην μνήμη και φορτώνει τα δεδομένα του scheduler στον επεξεργαστή, δηλαδή διακόπτει την διεργασία για να συνεχίσει η εκτέλεση του δρομολογητή. Τέλος, όταν τελειώσει η εκτέλεση του δρομολογητή και επιστρέψει η συνάρτηση swtch, αποκαθίσταται η κατάσταση του intena του επεξεργαστή.

- Scheduler:

```

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();

        // 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

            // 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;
            const int tickstart = ticks;

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

            p->ticks += ticks - tickstart;

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

```

Ο ρόλος της scheduler() είναι ο κύριος ρόλος ενός λειτουργικού, η ανάθεση διεργασιών προς εκτέλεση στην CPU.

Αρχικά έχει ένα infinite loop το οποίο διακόπτεται μόνο από interrupts, τα οποία κάνουμε enable εντός του loop.

Αν δεν υπάρχουν runnable διεργασίες απλά περιμένουμε κάποια διακοπή.

Στην συνέχεια γίνεται η ανάζητηση μέσα στον πίνακα ptable.proc (αφού δεσμεύσουμε τον πίνακα κάνοντας acquire) όπου εξετάζουμε όλες τις διεργασίες μέσα σε αυτόν και

αν βρούμε μια διεργασία η οποία έχει κατάσταση runnable (έτοιμη για εκτέλεση) κάνουμε τα εξής κύρια βήματα:

1. Αλλάζουμε την εικονική μνήμη για την διεργασία
2. Θέτουμε την κατάσταση της σε running (εκτελείται)
3. Αποθηκεύουμε τον αριθμό των ticks του επεξεργαστή πριν ξεκινήσει η εκτέλεση της διεργασίας
4. Καλούμε την switch() για να αποθηκεύσει την κατάσταση του scheduler στην μνήμη και να φορτώσει στον επεξεργαστή την αποθηκευμένη κατάσταση της διεργασίας
5. Όταν η εκτέλεση της διεργασίας τελειώσει ή διακοπεί και επιστρέψει σε εκτέλεση ο δρομολογητής, αποθηκεύουμε το πλήθος των ticks για τα οποία εκτελείτο η διεργασία στο πεδίο ticks του struct της διεργασίας
6. Αλλάζουμε πάλι την εικονική μνήμη

Τέλος απελευθερώνουμε το lock για τον πίνακα διεργασιών με την release.

➔ 3)

Υλοποίηση system call:

Με την γνωστή διαδικασία από την προηγούμενη δραστηριότητα κάνουμε τις κατάλληλες αλλαγές στα αρχεία του συστήματος για να δημιουργήσουμε μια νέα κλήση συστήματος. Η υλοποίηση της βρίσκεται στο proc.c

```
GNU nano 7.2      proc.c *
}

void
settickets(int n){
    proc->tickets=n;
//    cprintf("\ncurrent (%d) tickets: %d\nn: %d\n\n",proc->pid, proc->tickets, n);
    return;
}
```

Υλοποίηση lottery scheduling:

Για την υλοποίηση του lottery scheduling θα κάνουμε αλλαγές την συνάρτηση scheduler() στο αρχείο proc.c.

1. Αρχικά ορίζουμε τις μεταβλητές που θα χρειαστούμε:

```
void
scheduler(void)
{
    struct proc *p = 0;
    for(;;){
        int total_tickets = 0;
        int counter=0;
        int lucky_ticket=0;
```

2. Στη συνέχεια διαπερνάμε τον πίνακα των διεργασιών αθροίζοντας τον αριθμό των εισιτηρίων της κάθε μεταβλητής στην μεταβλητή total_tickets και υπολογίζουμε το τυχερό εισιτήριο :

```
        for(p=ptable.proc; p< &ptable.proc[NPROC];p++){

            if(p->state != RUNNABLE)// && p->state!=RUNNING)
                continue;
            total_tickets+=p->tickets;
        }

        if(total_tickets==0){total_tickets=1;}
        lucky_ticket = rand()%total_tickets;
```

3. Τέλος διαπερνάμε πάλι τον πίνακα διεργασιών για να εντοπίσουμε την τυχερή διεργασία η οποία στην συνέχεια θα εκτελεστεί

```
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->state != RUNNABLE)// && p->state != RUNNING)
                continue;

            if((counter<= lucky_ticket) && (lucky_ticket < (counter+p->tickets)))
            { // Lucky winner found
            }

            else { // No luck here
                counter+=p->tickets;
                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;
```

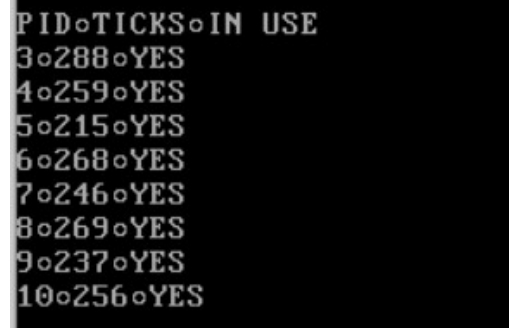
4. Δοκιμές:

A) Έστω 8 διεργασίες με την παρακάτω κατανομή εισιτηρίων:

```
int
main(int argc, char *argv[])
{
    int pid_chds[N_C_PROCS];

    int n_tickets[N_C_PROCS]={10,10,10,10,10,10,10,10};
    pid_chds[0] = getpid();
```

Με το εξής αποτέλεσμα:



PID	TICKS	IN USE
3	288	YES
4	259	YES
5	215	YES
6	268	YES
7	246	YES
8	269	YES
9	237	YES
10	256	YES


Βλέπουμε δηλαδή ότι όλες οι διεργασίες είχαν παρόμοιους χρόνους χρήσης της κεντρικής μονάδας επεξεργασίας, με μικρές διαφοροποιήσεις όπως αναμενόταν λόγω της τυχαιότητας της διαδικασίας.

B) Έστω 8 διεργασίες με την παρακάτω κατανομή εισιτηρίων:

```
int
main(int argc, char *argv[])
{
    int pid_chds[N_C_PROCS];

    int n_tickets[N_C_PROCS]={10,10,10,10,10,10,20,1000};
    pid_chds[0] = getpid();
```

Με το εξής αποτέλεσμα:



PID	TICKS	IN USE
3	264	YES
4	270	YES
5	207	YES
6	275	YES
7	232	YES
8	222	YES
9	427	YES
10	1565	NO

Βλέπουμε δηλαδή ότι η διεργασία με τα 1000 εισιτήρια (pid=10) είχε άπλετο χρόνο στην κεντρική μονάδα επεξεργασίας και τελείωσε πριν καν γίνει η πρώτη «καταγραφή» των χρόνων της κάθε διεργασίας.

Ακόμη βλέπουμε την διεργασία με τα 20 εισιτήρια (pid=9), διπλάσια από τις υπόλοιπες εκτός της 10, έχει σχεδόν διπλάσιο χρόνο στην κεντρική μονάδα επεξεργασίας από όλες τις υπόλοιπες, όπως αναμέναμε.

Γ) Έστω 8 διεργασίες με την παρακάτω κατανομή εισιτηρίων:

```
int
main(int argc, char *argv[])
{

    int pid_chds[N_C_PROCS];

    int n_tickets[N_C_PROCS]={4,4,8,8,16,16,32,32};
    pid_chds[0] = getpid();
```

Με το εξής αποτέλεσμα:

PID	TICKS	IN USE
3	144	YES
4	95	YES
5	225	YES
6	226	YES
7	383	YES
8	383	YES
9	693	YES
10	702	NO

Βλέπουμε έναν (κατά κανόνα) σχεδόν διπλασιασμό των χρόνων εκτέλεσης, αναλογικά με τον αριθμό εισιτηρίων. Η πρώτη διεργασία συχνά διαφέρει από τον αναμενόμενο αφού ξεκινάει πρώτη και δημιουργεί τις υπόλοιπες διεργασίες, οπότε έχει μερικά ticks ασυναγώνιστου χρόνου στην CPU.