ΝΑΤΑΛΙΑ ΡΟΥΣΚΑ
1092581
10/11/2024

Δραστηριότητα 2: Χρόνο-προγραμματισμός Διεργασιών στο xv6 Kernel

ΑΣΚΗΣΗ 1

Στον φάκελο */xv6/include* βρίσκεται το header file *proc.h*, το οποίο περιέχει τη δομή δεδομένων proc που αναπαριστά μία διεργασία στο σύστημα.

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
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)
};
```

Members του struct proc:

- *unintp sz*: μέγεθος της διεργασίας στη μνήμη (unintp: unsigned integer type with pointer size)
- *pde_t* pgdir:* δείκτης στο page table της διεργασίας. Το page table αποθηκεύει τις αντιστοιχίσεις των εικονικών διεθύνσεων που χρησιμοποιεί η διεργασία με τις φυσικές διευθύνσεις του συστήματος
- *char *kstack:* δείκτης στη βάση (αρχική διεύθυνση) της στοίβας kernel. Κάθε διεργασία έχει user stack και kernel stack. Το kernel stack χρησιμοποιείται όταν η διεργασία εκτελείται σε kernel mode (μετά από ένα system call ή ένα interrupt).
- *enum procstate state:* Η μεταβλητή state είναι τύπου procstate, δηλαδή μπορεί να πάρει μία από τις σταθερές που έχουν οριστεί στο enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE }

UNUSED	Ένα process slot στον πίνακα διεργασιών δεν χρησιμοποιείται, είναι διαθέσιμο για μία νέα διεργασία
EMBRYO	Μία νέα διεργασία δημιουργήθηκε, αλλά δεν έχει αρχικοποιηθεί πλήρως

ΝΑΤΑΛΙΑ ΡΟΥΣΚΑ
1092581
10/11/2024

SLEEPING	Η διεργασία περιμένει να ελευθερωθεί κάποιος πόρος ή κάποιο συμβάν π.χ I/O operation
RUNNABLE	Η διεργασία είναι έτοιμη να τρέξει στη CPU και περιμένει τον scheduler να την επιλέξει
RUNNING	Η διεργασία εκτελείται στη CPU
ZOMBIE	Η διεργασία έχει τερματιστεί, αλλά δεν έχει αποδεσμευτεί πλήρως η μνήμη, ώστε να μπορεί ο γονέας να ανακτήσει το exit status της.

- *volatile int pid*: Ένας μοναδικός αριθμός για μία διεργασία. Με την λέξη κλειδί volatile, ο compiler δεν μπορεί να κάνει optimizations-faster execution (π.χ. να διαβάζει την μεταβλητη από έναν CPU register-cached value, πρέπει κάθε φορά να διαβάζει από την μνήμη)
- *struct proc *parent:* Δείκτης στη δομή proc της γονικής διεργασίας
- *struct trapframe *tf:* Δείκτης στη δόμη trapframe της διεργασίας, όπου αποθηκεύονται οι τιμές των CPU registers ώστε να συνεχιστεί η λειτουργία της από το σημείο που σταμάτησε μετά από μία διακοπή trap (προκύπτουν κατά την εκτέλεση ενός προγράμματος π.χ. λόγω system call)
- *struct context *context:* Δείκτης στη δομή context της διεργασίας, όπου αποθηκεύεται η κατάσταση της διεργασίας και μπορεί να ανακτηθεί για την εναλλάγη περιβάλλοντος- context switch
- *void *chan:* Δείκτης στο κανάλι στο οποίο κοιμάται η διεργασία (ο δείκτης δείχνει σε οποιοδήποτε τύπο δεδομένων)
- *int killed:* Αν το flag είναι 1, η διεργασία πρέπει να τερματιστεί.
- *struct file *ofile[NOFILE]:* Πίνακας 16(constant-NOFILE) δεικτών ofile που δείχνουν στη δομή file. Περιέχει τους δείκτες στα ανοιχτά file descriptors της δειργασίας.
- *struct inode *cwd:* Δείκτης cwd (current working directory) σε μία δομή inode που αναπαριστά το directory στο βρίσκεται η διεργασία
- *char name[16]:* Πίνακας χαρακτήρων με το όνομα της διεργασιάς
- *int inuse:* Το flag είναι 1 αν εκτελείται η διεργασία από κάποιο core
- *int ticks:* Ο αριθμός ticks (χρονικές μονάδες της CPU) που έχει συγκεντρώσει η διεργασία
- *int tickets:* Ο αριθμός των εισητηρίων που έχει η διεργασία, χρησιμοποείται από τον αλγόριθμο lottery scheduling. Η διεργασία με τον μεγαλύτερο αριθμό εισητηρίων έχει μεγαλύτερη πιθανότητα να επιλέγει από τον scheduler.

ΝΑΤΑΛΙΑ ΡΟΥΣΚΑ
1092581
10/11/2024

ΑΣΚΗΣΗ 2

Οι συναρτήσεις *scheduler* και *sched* βρίσκονται στο */kernel/proc.c*

```
//PAGEBREAK: 42
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the 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
      // 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);

  }
}
```

Μόλις το λειτουργικό σύστημα κάνει boot, αρχικοποιεί τον κάθε πυρήνα της CPU. Όταν γίνει το setup του κάθε πυρήνα, καλεί ατέρμονα την scheduler και αναζητεί διεργασίες που είναι σε κατάσταση runnable ώστε να εκτελεστούν. Αν βρεί κάποια έτοιμη διεργασία αλλάζει την κατάσταση της σε running και κάνει εναλλάγη swtch() στο context της διεργασίας ώστε να εκτελεστεί. Όταν η διεργασία τερματιστεί ή καλέσει τη system call yield() (στέλνει σήμα ότι δεν χρειάζεται άλλο τη CPU) γίνεται πάλι εναλλαγή περιβάλλοντος στην scheduler.

Αναλυτικά μέσα στο infinite loop της scheduler:

- Οι διακοπές είναι ενεργοποιημένες sti().
- Αν έχουμε φτάσει στο τέλος του πίνακα διεργασιών p == &ptable.proc[NPROC] και δεν υπάρχουν runnable processes, η CPU σταματάει hlt() προσωρινά μέχρι να συμβεί κάποια διακοπή.

ΝΑΤΑΛΙΑ ΡΟΥΣΚΑ
1092581
10/11/2024

- Πριν διατρέξει τον πίνακα διεργασιών, αποκτάει το κλείδωμα για το ptable με την εντολή acquire(&ptable.lock), ώστε να έχει πρόσβαση σε αυτόν μόνο η scheduler και να μην δημιουργηθούν race conditions με άλλες συναρτήσεις διεργασίες που θέλουν να προσπελάσουν και να αλλάξουν τον πίνακα.
- Διατρέχει τον πίνακα διεργασίων και όταν βρεί κάποια με state RUNNABLE, κάνει εναλλαγή της εικονικής μνήμης switchuvm(p) με τον χώρο διευθύνσεων που βλέπει η συγκεκριμένη διεργασία. Αλλάζει την κατάσταση της σε running και την μεταβλητή της δομής της inuse σε 1.
- Με την κλήση της swtch(&cpu->scheduler, proc->context) γίνεται το context switch. Αποθηκέυται η τρέχουσα κατάσταση της CPU στη διεύθυνση cpu->scheduler και ανακτάται η αποθηκευμένη κατάσταση της διεργασίας proc->context ώστε να τρέξει
- Όταν ο kernel επανέρχεται στη scheduler μετά από κάποια διεργασία καλείται η switchkvm() και γίνεται εναλλαγή στην εικονική μνήμη του πυρήνα.
- Με την εντολή release(&ptable.lock) απελευθερώνεται το lock της ptable.

```
// Enter scheduler.  Must hold only ptable.lock
// and have changed proc->state.
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;
}
```

Η συνάρτηση sched() καλείται από την yield() αν κάποια διεργασία δεν χρειάζεται άλλο τη CPU και θέλει να κάνει εναλλαγή περιβάλλοντος στη scheduler.

Οι έλεγχοι που γίνονται πριν εκτελεστεί η swtch(&p->context, cpu->scheduler):

- if(!holding(&ptable.lock)) Διασφαλίζει ότι έχει αποκτηθεί το κλειδωμα στο ptable, ώστε να έχει μόνο η sched() πρόσβαση στον πίνακα, αλλιώς καλείται το panic από τον kernel για να σταματήσει το σύστημα.
- if (cpu->ncli != 1) Η μεταβλητή ncli είναι ένας μετρητής που αυξάνεται όταν καλείται η cli() (clear interrupts), η οποία απενεργοποιεί τις διακοπές και μειώνεται όταν καλείται η sti(). Έτσι όταν έχει ακριβώς την τιμή 1, οι διακοπές είναι σίγουρα απενεργοποιημένες και η εναλλαγή περιβάλλοντος θα γίνει με ασφάλεια.
- If(proc->state==RUNNING) Η sched() καλείται όταν η τρέχουσα διεργασία έχει εγκαταλείψει τη CPU, δηλαδή είναι σε καταάσταση RUNNABLE, SLEEPING

ΝΑΤΑΛΙΑ ΡΟΥΣΚΑ
1092581
10/11/2024

- if (readeflags() & FL_IF) Η συνάρτηση readeflags() διαβάζει τα CPU flags από τον καταχωρτητή EFLAGS και ελέγχει αν το bit Interrupt Flag είναι 1. Αυτός ο έλεγχος εξασφαλίζει ότι είναι απενεργοποιημένες οι διακοπές.
- intena=cpu->intena Αποθηκεύει την κατάσταση του interrupt flag ώστε να την επαναφέρει μετά το switch
- Με την κλήση της swtch(&proc->context, cpu->scheduler) γίνεται το context switch. Αποθηκεύται η τρέχουσα κατάσταση της διεργασίας στη διεύθυνση proc->context και ανακτάται η αποθηκευμένη κατάσταση της scheduler cpu->scheduler ώστε να τρέξει

ΑΣΚΗΣΗ 3

Δημιουργία system call settickets(int n) ώστε ο χρήστης να μπορεί να αλλάξει τον αριθμό των tickets της τρέχουσας διεργασίας

- Προσθέτω τον ορισμό int settickets(int n) της system call για να είναι διαθέσιμη στον χρήστη στο /include/user.h

```
 GNU nano 7.2                                    user.h *
struct stat;
struct pstat;

// system calls
int settickets(int);
int fork(void);
int exit(void) __attribute__((noreturn));
```

- Στο usys.s υπάρχει ήδη ο ορισμός SYSCALL(settickets)
- Στο /include/syscall.h έχει ήδη δοθεί αριθμός αντιστοίχησης 27 στη SYS_settickets

```
natalia@debian: ~/xv6/include
 GNU nano 7.2                          syscall.h *
#define SYS_close  21
#define SYS_getpinfo 24
#define SYS_settickets 27
```

- Στο /kernel/syscall.c υπάρχει ήδη ο ορισμός της και στον πίνακα από function pointers το ζεύγος [SYS_settickets] sys_settickets. Σβήνουμε την υλοποίηση που επιστρέφει -1.

```
 GNU nano 7.2                                  syscall.c *
extern int sys_uptime(void);
extern int sys_getpinfo(void);
extern int sys_settickets(void);

static int (*syscalls[])(void) = {
[SYS_settickets]   sys_settickets,
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
```

- Στο sysproc.c υλοποιούμε την συνάρτηση

```
int sys_settickets(void){
  int arg;
  if(argint(0,&arg)<0){ //fetces first integer argument of sys_settickets and stores it in arg
    return -1;
  }
  else{
    proc->tickets=arg;
    return 0;
  }
}
```

ΝΑΤΑΛΙΑ ΡΟΥΣΚΑ
1092581
10/11/2024

Τροποποιούμε την υλοποίηση της fork() στο kernel/proc.c , ώστε μία διεργασία να κληρονομεί τα εισητήρια από το γονέα της προσθέτοντας την εντολή `np->tickets = proc->tickets;`

Στο kernel/proc.c προσθέτουμε μία γεννήτρια ψευδοτυχαίων αριθμών

```
static unsigned long int next=1;
int myrand(int total_tickets){
  next=next*1103515245 + 12345;
  return  (unsigned int)(next/65536)%total_tickets;//rand_number in range of [0,total_tickets]
}
```

Τροποποιούμε την shcheduler(), ώστε να υλοποιεί lottery scheduling. Δηλαδή υπολογίζει έναν τυχαίο αριθμό μεταξύ 0 και total_tickets μέσω της γεννήτριας myrand(total_tickets). Έπειτα, για την κάθε διεργασία που είναι σε κατάσταση RUNNABLE γίνεται έλεγχος αν ο αριθμός των εισητηρίων της είναι μεγαλύτερος από αυτό το νούμερο. Αν ναι, κερδίζει την κλήρωση και ανατίθεται στη CPU.

ΝΑΤΑΛΙΑ ΡΟΥΣΚΑ
1092581
10/11/2024

```
void
scheduler(void)
{
  struct proc *p = 0;
  for(;;){
    // Enable interrupts on this processor.
    sti();
    int total_tickets=0;
    // 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();
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state == RUNNABLE){
        total_tickets+=p->tickets;//calculate total_tickets for all runnable processes
        }
    }
    if(total_tickets>0){
      int winning_ticket=myrand(total_tickets);//random lootery ticket
      // Loop over process table looking for process to run.
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state == RUNNABLE){
        // Switch to chosen process.  It is the process's job
        // to release ptable.lock and then reacquire it
        // before jumping back to us.
        if(p->tickets>winning_ticket){//a process found that has more tickets than the lottery ticket
          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;
          break;//exit loop after context switch
        }
     }
    }
  }
  release(&ptable.lock);
  }
}
```

Η allocproc() καλείται κάθε φορά που δημιουργείται μία διεργασία ώστε να δεσμεύσει μία δομή proc στο process table για την νέα διεργασία. Προσθέτουμε την εντολή `p->tickets=1;` ώστε κάθε διεργασία που δημιουργείται να έχει τουλάχιστον ένα εισιτήριο. Έτσι η init (first user_space process in the system) θα έχει ένα εισητήριο και ο scheduler θα την επιλέξει για να τρέξει .

Για να ελέγξουμε αν λειτουργεί σωστά ο scheduler χρησιμοποιούμε το έτοιμο πρόγραμμα sched-test. Αυτό δίνει στατιστικά σχετικά με το χρόνο εκτέλεσης της κάθε διεργασίας. Στο ήδη έτοιμο lotteryschedtest.c διορθώνουμε ένα bug που έχει να κάνει με την printinfo(), αρχικοποίηση των στοιχείων των array σε -1.

ΝΑΤΑΛΙΑ ΡΟΥΣΚΑ
1092581
10/11/2024

```
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;
    }
```

Παρακάτω βλεπουμε τα αποτελέσματα της εκτέλεσης. Το ποσοστό των εισητηρίων που κατέχει μία διεργασία θέλουμε να αναπαριστά το χρόνο που τρέχει στη CPU. Για παράδειγμα η διεργασία 7 με 500 tickets, θα θέλαμε να καταλαμβάνει το 50% του χρόνου της CPU. Η τυχαιότητα στο lottery scheduling εισάγει στοχαστικότητα στα αποτελέσματα και τελικά παρατηρούμε ότι η διεργασία χρησιμοποιεί το 57.9% του χρόνου. Όσο περισσότερο ανταγωνίζονται οι διεργασίες για χρόνο στη CPU, είναι πολύ πιθανό να λάβουμε τα επιθυμητά ποσοστά. Ωστόσο, φαίνεται ξεκάθαρα ότι όσα περισσότερα εισητήρια έχει η διεργασία τόσος περισσότερος χρόνος της ανατίθεται στη CPU και οι υπολοιπες ακολουθούν σε φθίνουσα σειρά.

```
$ lotteryschedte

Forked 5 children to share ~3000 ticks (real 1473)

PID: 5  TICKETS: 200    TICKS: 593      CPU: 42.3%
PID: 6  TICKETS: 100    TICKS: 80       CPU: 5.0%
PID: 7  TICKETS: 500    TICKS: 799      CPU: 57.9%
PID: 8  TICKETS: 50     TICKS: 0        CPU: 0.0%
PID: 9  TICKETS: 150    TICKS: 1        CPU: 0.1%
```