گزارش پروژه سوم آزمایشگاه سیستم عامل

کیاوش جمشیدی ۸۱۰۱۹۷۴۸۶

درین مسلمان یزدی ۸۱۰۱۹۷۵۸۳

مرضیه باقری نیا ۸۱۰۱۹۷۶۸۲

۱) چرا فراخوانی ()sched منجر به فراخوانی ()scheduler می شود؟

در ابتدا شرایط اولیه در تابع ()sched بررسی می‌شود. با استفاده از تابع holding چک می‌کند که ptable، lock باشد و current state of process به صورت running نبوده و flagها هم بررسی می‌شود. بعد از این، تابع swtch عمل context switch را به پردازه‌ای که توسط mycpu()->scheduler() جایگزین می‌شود انجام می‌دهد که یعنی دو آرگومان می‌گیرد.

۲) صف پردازه‌هایی که تنها منبعی که برای اجرا کم دارند پردازنده است، صف آماده یا صف اجرا نام دارد. در xv6 صف آماده مجزا وجود نداشته و از صف پردازه ها بدین منظور استفاده می‌گردد. در زمان‌بند کاملاً منصف در لینوکس، صف اجرا چه ساختاری دارد؟

هدف زمان بندی لینوکس منصفانه بودن است که run queue ها را به صورت درخت red black tree می‌گذارد که Self-balancing است. Processهایی که زمان کمتری از cpu می‌گیرند تا کارشان تمام شود در سمت راست و processهایی که کمتر interactive هستند، در سمت چپ قرار می‌گیرند. سیاست انتخاب processها از چپ ترین node است (انتخاب از چپ). معیار کلی درخت vruntime آن کار است.

۳) همان طور که در پروژه یک مشاهده شد، هر هسته پردازنده در xv6 یک زمان بند دارد. در لینوکس نیز به همین گونه است. این دو سیستم عامل را از منظر مشترک یا مجزا بودن صف های زمانبندی بررسی نمایید.

Xv6 از ptable استفاده می‌کند و هربار همه آن را iterate می‌کند و در آن صف زمانبندی به صورت جدا وجود ندارد. همه core ها از همان صف استفاده می‌کند و بین همه هسته‌ها مشترک است. هر logical process در Linux یک صف زمان بندی مجزا وجود دارد که اگر یک صف شلوغ باشد امکان thread migration در بین run queues وجود دارد.

۴) در هر اجرای حلقه ابتدا برای مدتی وقفه فعال می گردد. علت چیست؟ آیا در سیستم تک هسته‌ای به آن نیاز است؟

با گرفتن Lock، Interrupt ها غیر فعال می‌شوند و با رها کردن آن بعضی وقت‌ها دوباره فعال می‌شوند. اگر دوباره فعال نشد، interruptها را مجزا فعال می‌کنیم تا بتوانند taskهای دیگر مثل IO را انجام دهد. اگر فعال نشود ممکن است همه Process ها متوقف شوند. این اتفاق برای سیستم‌های single-core مهم است چون در آن صورت کل سیستم متوقف می‌شود.

۵) تابع معادل ()scheduler را در هسته لینوکس بیابید. جهت حفظ اعتبار اطلاعات جدول پردازه ها، از قفل‌گذاری استفاده می‌شود. این قفل در لینوکس چه نام دارد؟

تابع معادل ()scheduler است که در فایل c.sched/kernel است. برای هر run queue یک قفل (lock) مجزا وجود دارد که با توابع مخصوص خود گرفته شده و آزاد می‌شود.

۶) وصله PREEMPT_RT در سیستم عامل لینوکس چگونه پیشبینی پذیری اجرا را افزایش می‌دهد؟ آیا سیستم کاملا پیش بینی پذیر می‌شود؟ چرا؟

Kernel توسط این وصله PREEEMT می‌شود و باعث می‌شود یک Process از زمان مخصوص خودش بیشتر نشود و در زمان معینی انجام شود که باعث می شود predictability اجرا بیشتر شود.

Kernel کامل Preemptable شده پس کل سیستم نیز predictable می شود.

زمان‌بند بازخورد چند سطحی

پردازه‌ها را در سه دسته Level بندی می‌کنیم؛ پردازه‌های دسته اول را با الگوریتم Round-Robin زمانبندی می‌کنیم، پردازه‌های دسته دوم را با الگوریتم Lottery زمانبندی می‌کنیم و پردازه‌های دسته اول را با الگوریتم BJF زمانبندی می‌کنیم. هربار در میان صف پردازه‌ها می‌چرخیم؛ اگر پردازه‌ای در سطح یک موجود بود آن را زمانبندی می‌کنیم اگر نه به سراغ سطح دوم رفته و بازهم اگر پردازه ای در سطح دوم نبود در انتها به سطح سوم می‌رویم.

برای اجرای الگوریتم‌های زمانبندی چند ویژگی به ساختار پردازه‌ها اضافه می‌کنیم:

```
struct proc {
  uint 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
  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)
  // char* syscalls_name[NPROC];
  int syscalls_count [NSYSCALLS];
  int sys_count_stat;
  int level;                   // 1 for RR, 2 for lottery, 3 for BJF
  int tickets;                 // for lottery scheduling
  int waiting_cycles;          // cycles waiting for turn - for aging
  int arrival_time;
  float arrival_time_ratio;
  float executes_cycle;
  float executes_cycle_ratio;
  float priority_ratio;
};
```

به منظور مقداردهی اولیه این پارامترها، از تابع allocproc در فایل proc.c استفاده می‌کنیم:

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->level = 2;   //setting level default to 2
  p->tickets = 10;  //tickets default to 10
  p->waiting_cycles = 0;
  acquire(&tickslock);
  p->arrival_time = ticks;
  release(&tickslock);
  p->executes_cycle = 0;
  p->arrival_time_ratio = rand_zero_to_one();
  p->executes_cycle_ratio = rand_zero_to_one();
  p->priority_ratio = rand_zero_to_one();
```

همانطور که در صورت پروژه به آن اشاره شد، پردازه‌هایی که در Shell اجرا می‌شودند نیاز به اولویت بالاتری برای اجرا دارند تا قفل نشوند؛ بنابراین در هنگام exec شدن باید اولویت بالاتری به آن ها اختصاص دهیم؛ بدین منظور در فایل exec.c مقدار ticket پردازه‌ای که از طریق exec شدن اجرا می‌شود را افزایش می‌دهیم تا شانس اجرای آن بالا برود:

```
  oldpgdir = curproc->pgdir;
  curproc->pgdir = pgdir;
  curproc->sz = sz;
  curproc->tf->eip = elf.entry;  // main
  curproc->tf->esp = sp;
  curproc->tickets = 10000;
```

همچنین پیاده‌سازی هر سه نوع زمانبندی ذکر شده در انتهای فایل Proc.c قابل مشاهده است.

برای حل مشکل گرسنگی نیز از مکانیزم افزایش سن استفاده می‌کنیم:

```
void
age_procs(int running_proc){
  struct proc *p;
  for(int i =0; i < NPROC; i++){
    p = &ptable.proc[i];
    if(p->state == UNUSED)
      continue;
    if(p->pid != running_proc){
      p->waiting_cycles += 1;
      if(p->waiting_cycles >= 10000)
        set_level(p->pid, 1);

    }
  }
}
```

پیاده‌سازی فراخوانی‌های سیستمی

برای اضافه کردن یک فراخوانی سیستمی تمامی مراحل آموخته شده در پروژه قبل اعم از اضافه کردن آن به فایل های syscall.h، syscall.c، user.h، def.h، usys.S ، sysproc.c و proc.c را تکرار می‌کنیم:

```
void
print_procs(void){
  struct proc *p;
  cprintf("Name \t PID \t State \t Level \t Tickets \t Waited\n");
  for(int i = 0; i < NPROC; i++){
    p = &ptable.proc[i];
    if(p->state == UNUSED)
      continue;
    cprintf("%s \t %d \t %s \t %d \t %d \t %d \n",
      p->name,
      p->pid,
      states_string[p->state],
      p->level,
      p->tickets,
      p->waiting_cycles);
  }
}

void
set_tickets(int pid, int tickets){
  struct proc *p;
  for(int i = 0; i < NPROC; i++){
    p = &ptable.proc[i];
    if(p->pid == pid){
      p->tickets = tickets;
      break;
    }
  }
}
```

```
void
set_level(int pid, int level){
  struct proc *p;
  for(int i = 0; i < NPROC; i++){
    p = &ptable.proc[i];
    if(p->pid == pid){
      p->level = level;
      p->waiting_cycles = 0;  //reseting waited cycles
      break;
    }

  }
}
```

```
void
set_bjf_params_proc(int pid, int pr, int atr, int ecr){
  float priority_ratio_ = itof(pr);
  float arrival_time_ratio_ = itof(atr);
  float executes_cycle_ratio_ = itof(ecr);

  struct proc *p;
  for(int i = 0; i < NPROC; i++){
    p = &ptable.proc[i];
    if(p->pid == pid){
      p->priority_ratio = priority_ratio_;
      p->arrival_time_ratio = arrival_time_ratio_;
      p->executes_cycle_ratio = executes_cycle_ratio_;
      break;
    }
  }
}
```

```
void
set_bjf_params_system(int pr, int atr, int ecr){

  float priority_ratio_ = itof(pr);
  float arrival_time_ratio_ = itof(atr);
  float executes_cycle_ratio_ = itof(ecr);
  struct proc *p;
  for(int i = 0; i < NPROC; i++){
    p = &ptable.proc[i];
    p->priority_ratio = priority_ratio_;
    p->arrival_time_ratio = arrival_time_ratio_;
    p->executes_cycle_ratio = executes_cycle_ratio_;
  }
}
```

آدرس مخزن Gitlab:

https://gitlab.com/dmosalman/operatingsystemlab3.git