گزارش کار پروژه سوم آزمایشگاه سیستم عامل

گروه ۲۷
ارشیا عطایی نائینی | ۸۱۰۱۰۰۲۵۲
فاطمه کرمی محمدی | ۸۱۰۱۰۰۲۵۶
امیر پارسا موبد | ۸۱۰۱۰۰۲۷۱

آدرس مخزن گیتهاب :

https://github.com/phatemek/OS-Lab-Projects

شناسه آخرین کامیت :

a992b6741bf273af6300a570f843ae7e8bb91f89

زمان بندی در xv6

۱) چرا فراخوانی تابع ()sched، منجر به فراخوانی تابع ()scheduler می‌شود؟

تابع ()sched به صورت رو به رو در proc.c تعریف شده است.

```
2807 void
2808 sched(void)
2809 {
2810   int intena;
2811   struct proc *p = myproc();
2812
2813   if(!holding(&ptable.lock))
2814     panic("sched ptable.lock");
2815   if(mycpu()->ncli != 1)
2816     panic("sched locks");
2817   if(p->state == RUNNING)
2818     panic("sched running");
2819   if(readeflags()&FL_IF)
2820     panic("sched interruptible");
2821   intena = mycpu()->intena;
2822   swtch(&p->context, mycpu()->scheduler);
2823   mycpu()->intena = intena;
2824 }
2825
```

این تابع در یکی از سه حالت زیر صدا می‌شود.

1. پردازه با فراخوانی سیستمی exit، پردازنده را ترک کند.
2. پردازه با فراخوانی سیستمی sleep، به حالت SLEEPING برود.
3. پس از یک interrupt ایجاد شده، پردازه مجبور به خروج از پردازنده شود. در این صورت تابع yield فراخوانده می‌شود.

همانطور که می‌بینید در خط 2823 کانتکست سوییچینگ انجام می‌شود. که در آن تابع پیاده‌سازی شده در زبان Assembly، swtch فراخوانده می‌شود و در انتهای swtch مطابق آنچه در booklet، xv6 نوشته شده است در انتهای swtch، program counter به ابتدای ()scheduler می‌رود.

```
2750 // Per-CPU process scheduler.
2751 // Each CPU calls scheduler() after setting itself up.
2752 // Scheduler never returns.  It loops, doing:
2753 //  - choose a process to run
2754 //  - swtch to start running that process
2755 //  - eventually that process transfers control
2756 //      via swtch back to the scheduler.
2757 void
2758 scheduler(void)
2759 {
```

زمان بندی

۲) در زمان بندی کاملا منصف در لینوکس، صف اجرا چه ساختاری دارد؟

صف اجرا در Linux توسط داده‌ساختار Red-Black Tree پیاده سازی شده است که مقدار key، گره‌ها vruntime است و چپ‌ترین گره در این داده‌ساختار، پردازه ای قرار می‌گیرد که کمترین برش زمانی در حین اجرا را داشته باشد.

۳) همانطور که در پروژه اول مشاهده شد، هر هسته پردازنده در xv6 یک زمان‌بند دارد. در لینوکس نیز به همین‌گونه است. این دو سیستم عامل را از منظر مشترک یا مجزا بودن صف‌های زمان‌بندی بررسی نمایید. و یک مزیت و یک نقص صف مشترک نسبت به صف مجزا را بیان کنید.

هر پردازه یک زمان‌بند برای خودش دارد اما xv6 همانطور که در تصویر دیده می‌شود. فقط از یک صف زمان‌بندی برای همه پردازه ها به طور مشترک استفاده می‌کند.

```
2409 struct {
2410   struct spinlock lock;
2411   struct proc proc[NPROC];
2412 } ptable;
2413
```

۴) در هر اجرای حلقه، ابتدا برای مدتی وقفه فعال می‌گردد. علت چیست؟ آیا در سیستم تک‌هسته‌ای به آن نیاز است؟

وقتی قفل ptable فعال می‌شود، تمام وقفه‌ها به وسیله تابع pushcli غیرفعال می‌شوند.

در این صورت ممکن است تعدادی از پردازه‌ها در این حالت قرار بگیرند که منتظر پایان عملیات I/O باشند و هیچ پردازه دیگری نیز در حالت RUNNABLE موجود نباشد. در این صورت هیچ پردازه دیگری اجرا نخواهد شد و اگر وقفه‌ها نیز غیر فعال نشوند، پس از پایان I/O نمی‌توانیم پردازه‌های موردنظر را به حالت

قابل اجرا تغییر دهیم؛ در نتیجه سیستم فریز می‌شود. به همین دلیل در این حلقه برای مدتی کوتاه پیش از قفل کردن ptable، وقفه‌ها فعال می‌شوند تا در صورت نیاز بتوانیم حالت پردازه‌ها را تغییر دهیم. با توجه به حالتی که ممکن است پیش بیاید این مورد برای سیستم‌های تک هسته‌ای هم نیاز است.

۵) وقفه‌ها اولویت بالاتری نسبت به پردازه‌هل دارند. به طور کلی مدیریت وقفه‌ها در لینوکس در دو سطح صورت می‌گیرد. آن‌ها را نام برده و به اختصار توضیح دهید. اولویت این دو سطح مدیریت نسبت به هم و نسبت به پردازه‌ها چگونه است؟

در لینوکس وقفه‌ها در دو سطح FLIH(First Level Interrupt Handler) و SLIH(Second Level Interrupt Handler) انجام می‌پذیرد.در سطح FLIH وقفه‌های مدیریت وقفه‌های ضروری انجام می‌پذیرد. این کار به یکی از این دو صورت انجام می‌شود. یا رسیدگی کامل آن توسط خود FLIH انجام می‌شود یا اطلاعات وقفه و زمان‌بندی آن به SLIH داده می‌شود تا به رسیدگی آن بپردازد. SLIH اما قسمت‌هایی از پردازش وقفه که نیاز به زمان بیشتری دارد را انجام می‌دهد.

برای این سطح مدیریت دو حالت وجود دارد که به این ترتیب است: ۱. مدیریت هر handler توسط یک thread pool انجام شود. ۲. به ازای هر handler در سطح kernel یک ریسه به خصوص وجود دارد. SLIH ها به طور معمول زمان‌بندی می‌شوند چون ممکن است اجرای آنها به طول بیانجامد.

مدیریت وقفه ها در صورتی که بیش از حد زمان‌بر شود، می‌تواند منجر به گرسنگی پردازه‌ها گردد. این می‌تواند به خصوص در سیستم‌های بی‌درنگ مشکل‌ساز باشد. چگونه این مشکل حل شده است؟

یکی از راه حل‌های موجود در این مشکل aging است. این کار به این صورت انجام می‌شود که به صورت دوره‌ای اولویت تسک ها را یک level افزایش می‌دهیم. در این ترتیب تسک‌های کم اولویت پس از مدتی که اولویتشان افزایش پیدا می‌کند تا اجرا شوند. علاوه بر این برای جلوگیری از اختصاص یافتن بیش از اندازه پردازنده به وقفه‌ها راهکار های زیر هم انجام می‌پذیرد:

1 . کوتاه نگهداشتن مدیریت وقفه‌ها که با استفاده از همان FLIH و SLIH صورت می‌پذیرد.
2 . محدود کردن نرخ ایجاد وقفه‌ها.
3 .کم کردن بدترین حالت نرخ ایجاد وقفه‌ها
4 . چک کردن دوره‌ای اتفاقات به جای استفاده از وقفه‌ها

زمان بندی بازخوردی چند سطحی:

ابتدا با اضافه کردن استراکت sche به proc ها اطلاعات مورد نیاز برای زمان بندی آن‌ها را نگه می‌داریم. حال با استفاده از آن تابع handle_ages به صورت زیر پیاده سازی می‌شود:

```
void
handle_ages(int tmpticks) {
  struct proc *p;

  acquire(&ptable.lock);

  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state == RUNNABLE && p->sched.queue != RR)
    {
      if (tmpticks - p->sched.last_run > AGING_TIME)
      {
        release(&ptable.lock);
        switch_queue(p->pid, RR);
        acquire(&ptable.lock);
      }
    }
  }

  release(&ptable.lock);
}
```

که در اینتراپت مربوطه در trap.c با افزایش ticks صدا زده می‌شود.

1. زمان‌بند نوبت گردشی:

این زمان‌بند به صورت زیر برنامه نویسی شده است که از پردازه قبلی‌ای که اجرا شده است به دنبال پردازه قابل اجرای بعدی می‌گردد تا آن را اجرا کند:

```
struct proc*
find_rr_proc(struct proc* prv) {
  struct proc* p = prv;
  for(;;) {
    p++;
    if (p >= &ptable.proc[NPROC]) { p = ptable.proc; }
    if (p -> state == RUNNABLE && p -> sched.queue == RR) { return p; }
    if (p == prv) { return 0; }
  }
}
```

2. زمان‌بند آخرین ورود، اولین رسیدگی(LCFS):

ابتدا در آرگومان st زمان شروع هر پردازه نگهداری شده است و هردفعه پردازه‌ای از این صف که قابل اجرا باشد و بیشترین زمان شروع را داشته باشد، اجرا می‌شود:

```
struct proc*
find_lcfs_proc() {
  struct proc init_proc;
  struct proc* resp = &init_proc;
  resp->st = 0;
  int found = 0;
  for(struct proc* p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->state == RUNNABLE && p->sched.queue == LCFS && p->st > resp->st) {
      resp = p;
      found = 1;
    }
  }
  if (found) {
    return resp;
  } else {
    return 0;
  }
}
```

3. زمان‌بند اول بهترین کار(BJF):

در این زمان‌بندی ابتدا بر اساس پارامترهای مشخص شده، برای هر کدام از پردازه‌های این صف یک اولویت بدست می‌آید و سپس پردازه با کمترین اولویت، اجرا می‌شود:

```
static float
bjfrank(struct proc* p){
  return p->sched.bjf.priority *
  p->sched.bjf.priority_ratio +
  p->sched.bjf.arrival_time *
  p->sched.bjf.arrival_time_ratio +
  p->sched.bjf.executed_cycle *
  p->sched.bjf.executed_cycle_ratio +
  p->sched.bjf.process_size *
  p->sched.bjf.process_size_ratio;
  ;
}
```

```
struct proc*
find_bestjobfirst_proc(void)
{
  struct proc* result = 0;
  float minrank;
  struct proc* p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE || p->sched.queue != BJF)
     continue;
    float rank = bjfrank(p);
    if(result == 0 || rank < minrank){
      result = p;
      minrank = rank;
    }
  }
  return result;
}
```

فراخوانی‌های سیستمی موردنیاز:

1. تغییر صف پردازه:

به ازای این دستور یک سیستم کال به نام switch_queue ساخته شده است که مطابق مراحل اضافه کردن سیستم کال‌ها در آز قبلی اضافه می‌شود:

```
int
switch_queue(int pid, int num) {
  acquire(&ptable.lock);
  int res = -1;
  for(struct proc* p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if (p->pid == pid) {
      res = p->sched.queue;
      p->sched.queue = num;
      release(&ptable.lock);
      return res;
      break;
    }
  }
  release(&ptable.lock);
  return res;
}
```

2. مقداردهی پارامتر BJF در سطح پردازنده:

مطابق قسمت قبل سیستم کالی با نام bjf_parameters_pl را به سیستم عامل اضافه می‌کنیم:

```
int
bjf_parameters_pl(
  int pid,
  int priority_ratio,
  int arrival_time_ratio,
  int executed_cycle_ratio,
  int process_size_ratio
  ) {
  acquire(&ptable.lock);
  for(struct proc* p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->pid == pid) {
      p->sched.bjf.priority_ratio = priority_ratio;
      p->sched.bjf.arrival_time_ratio = arrival_time_ratio;
      p->sched.bjf.executed_cycle_ratio = executed_cycle_ratio;
      p->sched.bjf.process_size_ratio = process_size_ratio;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

3. مقداردهی پارامترهای BJF در سطح سیستم:

مطابق بخش قبل سیستم کال bjf_parameters_sl را اضافه می‌کنیم:

```
int
bjf_parameters_sl(
  int priority_ratio,
  int arrival_time_ratio,
  int executed_cycle_ratio,
  int process_size_ratio
) {
  acquire(&ptable.lock);
  for(struct proc* p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    p->sched.bjf.priority_ratio = priority_ratio;
    p->sched.bjf.arrival_time_ratio = arrival_time_ratio;
    p->sched.bjf.executed_cycle_ratio = executed_cycle_ratio;
    p->sched.bjf.process_size_ratio = process_size_ratio;
  }
  release(&ptable.lock);
  return 0;
}
```

4. چاپ اطلاعات:

برای این بخش با استفاده از سیستم کال show_info اطلاعات را در ترمینال سیستم‌عامل نمایش می‌دهیم.(نمونه‌ای از آن جلوتر نشان داده شده است.)

برنامه‌های سطح کاربر:

برنامه‌های سطح کاربری foo, switch_queue, set_system_bjf, set_process_bjf, show_info نوشته شده و در قسمت UPROGS و EXTRA اضافه شده‌اند که کد آن‌ها در فایل آپلود شده موجود است. نتیجه صحت‌آزمایی سیستم کال‌ها با استفاده از برنامه‌های سطح کاربری به صورت زیر است:

```
Group #27:
1.Arshia Ataei
2.Fateme karami
3.AmirParsa Mobed
$ foo&
$ show_info
Process_Name  |  PID   |  State  |  Queue  |  Cycle  |  Arrival  |  Priority  |  R_Prty  |  R_Arvl  |  R_Exec  |  R_Size  |  Rank
-------------------------------------------------------------------------------------------------------------------------------------
init          1        sleeping  1         1         0           1            1          1          1          1          2
sh            2        sleeping  2         1         2           1            1          1          1          1          4
foo           5        runnable  2         24        248         1            1          1          1          1          49425
foo           4        sleeping  2         0         247         1            1          1          1          1          248
foo           6        runnable  2         24        248         1            1          1          1          1          273
foo           7        runnable  2         24        248         1            1          1          1          1          273
foo           8        runnable  2         24        248         1            1          1          1          1          273
foo           9        runnable  2         24        248         1            1          1          1          1          273
show_info     10       running   2         0         491         1            1          1          1          1          492
$ switch_queue 5 3
Queue changed successfully
$ show_info
Process_Name  |  PID   |  State  |  Queue  |  Cycle  |  Arrival  |  Priority  |  R_Prty  |  R_Arvl  |  R_Exec  |  R_Size  |  Rank
-------------------------------------------------------------------------------------------------------------------------------------
init          1        sleeping  1         1         0           1            1          1          1          1          2
sh            2        sleeping  2         1         2           1            1          1          1          1          4
foo           5        runnable  3         159       248         1            1          1          1          1          49560
foo           4        sleeping  2         0         247         1            1          1          1          1          248
foo           6        runnable  2         159       248         1            1          1          1          1          408
foo           7        runnable  2         159       248         1            1          1          1          1          408
foo           8        runnable  2         159       248         1            1          1          1          1          408
foo           9        runnable  2         159       248         1            1          1          1          1          408
show_info     12       running   2         0         1841        1            1          1          1          1          14130
$ set_system_bjf 2 2 2 2
BJF params set successfully
$ show_info
Process_Name  |  PID   |  State  |  Queue  |  Cycle  |  Arrival  |  Priority  |  R_Prty  |  R_Arvl  |  R_Exec  |  R_Size  |  Rank
-------------------------------------------------------------------------------------------------------------------------------------
init          1        sleeping  1         1         0           1            2          2          2          2          4
sh            2        sleeping  2         2         2           1            2          2          2          2          10
foo           5        runnable  3         343       248         1            2          2          2          2          99488
foo           4        sleeping  2         0         247         1            2          2          2          2          497
foo           6        runnable  2         343       248         1            2          2          2          2          1184
foo           7        runnable  2         343       248         1            2          2          2          2          1184
foo           8        runnable  2         343       248         1            2          2          2          2          1184
foo           9        runnable  2         343       248         1            2          2          2          2          1184
show_info     14       running   2         0         3684        1            1          1          1          1          15973
$
```