گزارش آزمایشگاه سیستم عامل: پروژه شماره ۳
گروه ۲۲

پردیس زندکریمی - محمد امانلو - شهزاد ممیز
۸۱۰۱۰۱۰۸۱ - ۸۱۰۱۰۰۰۸۴ - ۸۱۰۱۰۰۲۷۲

Last commit code:
`461dfaf1b42feca7dfc3a2fb1e575ce3b11a3ae2`
Github repo:
https://github.com/MohammadAmanlou/OS_lab

---

1. هر هسته ای که شروع به کار می کند تابع mpmain را صدا میکند. در انتهای این تابع scheduler صدا زده میشود و بنابراین زمان بند مربوط به هر هسته شروع به کار میکند. همچنین این تابع در بدنه خود جدول پردازه ها را به دنبال پردازه ای که در state RUNNABLE باشد میگردد. در صورت وجود پردازه ای با چنین استیتی آن پردازه انتخاب شده و پس از تغییر حافظه به حافظه پردازه توسط تابع switchuvm با استفاده از تابع swtch عملیات تعویض متن را انجام میدهد. این تابع رجیستر های context قدیمی را در آدرس مربوط به همان context ذخیره کرده و رجیستر های مربوط به context جدید را از آدرس مربوط به همان context بازیابی میکند که با این کار program counter تغییر میکند و به این ترتیب پردازه جدید شروع به کار میکند. در حالتی که پردازه با استفاده سیستم کال exit پردازنده را ترک کند یا با استفاده از سیستم کال sleep به استیت SLEEPING در آید یا در حالتی که پس از interrupt ایجاد شده توسط تایمر پردازه مجبور به خروج از پردازنده شود تابع sched فراخوانی میشود. (در مورد آخر ابتدا yield فراخوانی میشود و درون آن تابع sched فراخوانی میشود.) در نهایت در این تابع مجددا عملیات تعویض متن صورت میگیرد و در این حالت context ای که در استراکت cpu بازیابی میشود و context مربوط به پردازه در حال اجرا ذخیره میشود. پس از بازیابی context مربوط به زمان بند program counter باعث ادامه کار scheduler می شود. پروگرم کانتر به صورت مستقیم ذخیره نمیشود و همان return adr تابع است که در زمان فراخوانی swtch در استک push می شود. این آدرس پس از دستور ret در خط 3078 از استک pop شده و در رجیستر مربوط به program counter قرار میگیرد.
2. صف اجرا در لینوکس توسط یک red black tree پیاده سازی میشود. در چپ ترین گره این درخت پردازه ای قرار گرفته که کمترین برش زمانی در حین اجرا را داشته است

3. در xv6 فقط از یک صف زمان بندی برای همه پردازنده ها به طور مشترک استفاده میشود. در ptable که یک struct است و در آن spinlock که وظیفه آن جلوگیری از بروز خرابی در نتیجه ی دسترسی همزمان چند پردازنده روی این صف است و لیستی از پردازه ها به صورت زیر تعریف شده:

   struct proc proc[NPROC]

   این صف میتواند ۶۴ پردازه را در خود نگه دارد. داشتن تنها یک صف پیاده سازی را ساده تر میکند اما باید درنظر داشته باشیم که حتما در جاهای مورد نیاز lock داشته باشیم که میتواند کمی بر روی performance اثر بگذارد.

4. با استفاده از قفل ptable تمامی interrupt ها توسط pushcli غیرفعال می شوند و در این صورت اگریک پردازه ای منتظر عملیات io باشد و هیچ یک از پردازه ها در RUNNABLE state نباشند هیچ پردازه دیگری اجرا نمیشود و اگر interrupt ها هرگز فعال نشوند پس از پایان عملیان io نمیتوانیم پردازه های مربوطه را به RUNNABLE تغییر دهیم و سیستم به صورت کلی فریز میشود. پس در این حلقه به مدت کوتاهی وقفه ها فعال میشوند تا بتوانیم در صورت لزوم state پردازه ها را تغییر دهیم.
5. دو سطح FLIH و SLIH داریم. که به اولی نیمه بالایی و به دومی نیمه پایینی گفته میشود. وظیفه اولی مدیریت وقفه های ضروری در کمترین زمان ممکن است و دومی وظیفه پردازش وقفه هایی را بر عهده دارد که زمان برهستند و مانند یک پردازه این کار را انجام میدهد. آنها یک ریسه مخصوص در سطح کرنل برای هر هندلر دارند، یا توسط یک thread pool مدیریت میشوند. آنها در صف قرار میگیرند و منتظر پردازنده میمانند و از جایی که ممکن است اجرایشان طول بکشد مانند ترد ها و پردازه ها زمان بندی میشوند
6. گرسنگی پردازه یا اشاره به مسئله "starvation" در زمینه‌ی هوش مصنوعی و سیستم‌های برنامه‌ریزی شده است. این مسئله زمانی به وجود می‌آید که یک یا چند پردازه یا منبع منابع مورد نیاز خود را در اختیار نمی‌گیرند و محروم می‌شوند. یکی از روش‌های حل این مسئله، استفاده از الگوریتم‌های برنامه‌ریزی منابع است که به توزیع منابع به پردازه‌ها و منابع به نحوی که هیچ کدام احساس محرومیت نکنند و تضمین کند که تمام پردازه‌ها و منابع به منظور ادامه فعالیت خود دسترسی داشته باشند.

   الگوریتم aging همچنین به عنوان یک روش رفع گرسنگی پردازه‌ها مورد استفاده قرار می‌گیرد. این الگوریتم از یک رویکرد اولویت‌دهی با استفاده از سن پردازه‌ها برای تخصیص منابع استفاده می‌کند. هرگاه یک پردازه مدت زمان طولانی‌تری منتظر منابع بوده باشد، سن آن پردازه افزایش می‌یابد. وقتی یک منبع در دسترس قرار گیرد، پردازه با کمترین سن (یا گران‌ترین پردازه) اولویت دارد و از منبع استفاده می‌کند.

   به این ترتیب، احتمال اینکه یک پردازه مدت طولانی‌تری در صف منتظری به سادگی منابع را بدست آورد و به دلیل این مدت زمان بیشتر منتظری، دچار گرسنگی شود، کاهش می‌یابد. این الگوریتم قادر است به گونه‌ای منطقی اولویت‌بندی کند که حتی پردازه‌هایی که مدت زمان زیادی

را منتظر می‌مانند، همچنان فرصت به تعامل با منابع را داشته باشند و از گرسنگی جلوگیری کنند.
راهکار aging معروف و شناخته شده ترین راه حل برای این مشکل است. بدین معنی است که اولویت هر پردازه که در صف قرار گرفته پس از مدتی یکی اضافه میشود. این کار تا زمانی که پردازنده به پردازه اختصاص یابد ادامه پیدا میکند.

زمان بندی بازخوردی چندسطحی

با استفاده از یک استراکت ageproc که تعریف شده و به فیلد های استراکت پردازه اضافه کردیم توانستیم اطلاعات زمان بندی پردازه را ذخیره کنیم که در ادامه مسئله از آن استفاده شود. یکی از فیلد های این استراکت نشان دهنده صفی است که پردازه در آن قرار دارد.
برای افزایش سن در زمان بندی تابع ageprocs نوشته شد که در آن پس از هر tick در interrupt مربوط به تایمر در فایل trap.c صدا زده میشود.

```
1  void
2  ageprocs (int osTicks)
3  {
4    struct proc *p;
5    acquire (&ptable.lock);
6    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
7      {
8        if (p->state == RUNNABLE && p->sched_info.queue != ROUND_ROBIN)
9      {
10       if (osTicks - p->sched_info.last_run > AGING_THRESHOLD)
11         {
12           release (&ptable.lock);
13           change_queue (p->pid, ROUND_ROBIN);
14           acquire (&ptable.lock);
15         }
16     }
17     }
18   release (&ptable.lock);
19 }
```

ساز و کار افزایش سن

برای ساز و کار افزایش سن تابع زیر نوشته شده است. در استراکت اضافه شده یک فیلد last_runتعریف شده است که آخرین تیک سیستم که این پردازه اجرا شده است را ذخیره می کند. با محاسبه تفاوت last_runو تیک سیستم در هر واحد زمان تصمیم می گیریم که پردازه به صف اول انتقال پیدا کند یا نه. این تابع پس از هر بار افزایش مقدار ticksدر فایل trap.cصدا زده می شود

```
void ageprocs(int os_ticks)
{
  struct proc *p;
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state == RUNNABLE && p->sched_info.queue != ROUND_ROBIN)
    {
      if (os_ticks - p->sched_info.last_run > AGING_THRESHOLD)
      {
        release(&ptable.lock);
        change_Q(p->pid, ROUND_ROBIN);
        acquire(&ptable.lock);
      }
    }
  }
  release(&ptable.lock);
}
```

زمان بند نوبت گردشی

در این زمان بند تابع roundrobin اضافه شد. در این تابع در میان پردازه ها با استفاده از یک لوپ میچرخیم تا بتوانیم پردازه ای را جهت زمان بندی پیدا کنیم که در وضعیت RUNNABLE باشد.

```
struct proc *
roundrobin (struct proc *lastScheduled)
{
  struct proc *p = lastScheduled;
  for (;;)
    {
      p++;
      if (p >= &ptable.proc[NPROC])
    p = ptable.proc;
      if (p->state == RUNNABLE && p->sched_info.queue == ROUND_ROBIN)
    return p;
      if (p == *lastScheduled)
    return 0;
    }
}
```

سطح دوم: زمانبند آخرین ورود - اولین رسیدگی (LCFS)

الگوریتم زمانبندی LCFS (که مخفف Last Come, First Served است) در واقع یکی از الگوریتم‌های زمانبندی پردازه‌ها است که در سیستم عامل‌ها استفاده می‌شود. در این الگوریتم، آخرین پردازه‌ای که وارد حالت آماده (READY) می‌شود، اولویت بیشتری برای اجرا دارد.

به عبارت دیگر، زمانبندی LCFS به این معنی است که پردازه‌ها براساس زمان ورود به حالت آماده مورد زمان‌بندی قرار می‌گیرند. پردازه‌ای که آخرین بار به حالت آماده وارد شده است، اولویت بالاتری برای اجرا دارد و به عبارت دیگر، الگوریتم LCFS زمانبندی بر اساس ترتیب معکوس ورود به حالت آماده را اعمال می‌کند.

مزیت این الگوریتم این است که پردازه‌هایی که به تازگی به حالت آماده وارد شده‌اند و نیاز به اجرا دارند، اولویت بیشتری دارند تا بتوانند به سرعت اجرا شوند. اما یکی از مشکلات معروف این الگوریتم، مسئله inversion اشتغال (وارونگی اشتغال) است که می‌تواند به کاهش کارایی سیستم منتهی شود.

متغیر تاثیرگذار که برای مدیریت این صف به استراکت schedinfo اضافه کردیم، time_queue_arrival است. این متغیر را هربار که یک پردازه وارد یک صف میشود آپدیت میکنیم و به این نحو برای هر پردازه در هر صف زمانی که پردازه وارد آن صف شده را خواهیم داشت. آپدیت کردن مقدار این متغیر در تابع queue_change انجام میشود. از آنجا که این تابع هنگام ساخت هر پردازهای حتما فراخوانی میشود مطمئن هستیم که این متغیر مقداردهی خواهد شد.

```
struct proc*
lcfs(void){
    struct proc *result = 0;
    struct proc *p;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p -> state != RUNNABLE || p -> sched_info.queue != LCFS)
            continue;
        if(!result){
            if (result->sched_info.arrival_queue_time <
            result-> sched_info.arrival_queue_time){
                result = p;
            }

        }
        else{
            result = p;
        }
    }
    return result;
}
```

زمان بند اول بهترین کار

در این تابع برای مشخص کردن زمان بندی پردازه با استفاده از priority پنج اولویت در نظر گرفته شد که از ۱ (بالاترین اولویت) تا ۵ (پایینترین اولویت) مشخص شده اند. در این تابع همه پردازه هایی که در وضعیت RUNNABLE قرار دارند رنک میشوند ودر هر مرحله پردازه ای با اولویت بالا تر به عنوان پردازه ای که باید اجرا شود انتخاب میشود.

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

struct proc *
bestjobfirst (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_info.queue != BJF)
    continue;
      float rank = bjfrank (p);
      if (result == 0 || rank < minrank)
    {
      result = p;
      minrank = rank;
    }
    }
  return result;
}
```

- **فراخوانی های سیستمی:**

تغییر صف پردازه: با استفاده از تابع زیر که دو آرگومان pid و صف مقصد را میگیرد صف پردازه ای با شماره pid به صف مقصد انتقال داده میشود.

```
int
change_queue (int pid, int new_queue)
{
  struct proc *p;
  int old_queue = -1;
  if (new_queue == UNSET)
    {
      if (pid == 1)
    new_queue = ROUND_ROBIN;
      else if (pid > 1)
    new_queue = LOTTERY;
      else
    return -1;
    }
  acquire (&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if (p->pid == pid)
    {
      old_queue = p->sched_info.queue;
      p->sched_info.queue = new_queue;
      release (&ptable.lock);
      return old_queue;
    }
    }
  release (&ptable.lock);
  return old_queue;
}
```

مقدار دهی پارامتر BJF در سطح پردازه:

در این تابع که به عنوان ورودی pid پردازه و ضریب BJF را میگیرد این کار صورت میگیرد.

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

مقداردهی پارامتر BJF در سطح سیستم:

تابعی که در سطح سیستم برای این کار تعریف شد بعنوان ورودی سه ضریب BJF را میگیرد و آنها برای همه پردازه ها تنظیم میکند.

```
void
set_bjf_params_system (float priority_ratio, float
              arrival_time_ratio, float executed_cycles_ratio)
{
  acquire (&ptable.lock);
  struct proc *p;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      p->sched_info.bjf.priority_ratio = priority_ratio;
      p->sched_info.bjf.arrival_time_ratio = arrival_time_ratio;
      p->sched_info.bjf.executed_cycle_ratio = executed_cycles_ratio;
    }
  release (&ptable.lock);
}
```

چاپ اطلاعات:

در این سیستم کال همه پردازه ها پیمایش میشوند و هریک از فیلد های مورد انتظار جهت به نمایش گذاشتن پرینت می شوند.

```
void
print_process_info ()
{
  static char *states[] = {
    [UNUSED] "unused",
    [EMBRYO] "embryo",
    [SLEEPING] "sleeping",
    [RUNNABLE] "runnable",
    [RUNNING] "running",
    [ZOMBIE] "zombie"
  };
  static int columns[] = { 16, 8, 9, 8, 8, 8, 8, 9, 8, 8, 8, 8 };
  cprintf ("Process_Name PID State Queue Cycle
Arrival Ticket Priority R_Prty R_Arvl R_Exec Rank\n" "----------------------
-----------------------------------------------\n");
  struct proc *p;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if (p->state == UNUSED)
    continue;
      const char *state;
      if (p->state >= 0 && p->state < NELEM (states) && states[p - >state])
    state = states[p->state];
      else
    state = "???";
      cprintf ("%s", p->name);
      printspaces (columns[0] - strlen (p->name));
```

برنامه سطح کاربر:

در این قسمت دو برنامه سطح کاربر نوشته شد. برنامه سطح کاربر به نام foo که در آن چند پردازه ساخته می شوند و شامل عملیات پراسس و sleep هستند.

```
int
main ()
{
  for (int i = 0; i < 5; ++i)
    {
      int pid = fork ();
      if (pid > 0)
    continue;
      if (pid == 0)
    {
      sleep (5000);
      for (int j = 0; j < 100 * i; ++j)
        {
          int x = 1;
          for (long k = 0; k < 1000000000000; ++k)
        x++;
        }
      exit ();
    }
    }
  while (wait () != -1)
    ;
  exit ();
}
```

همچنین برنامه ای به نام schedule ساخته شد که نتیجه آن به صورت زیر است:

```
$ schedule info
Process_Name    PID     State    Queue   Cycle   Arrival Ticket  Priority R_Prty  R_Arvl  R_Exec  Rank
-------------------------------------------------------------------------------------------------------
init            1       sleeping 1       1       0       0       3        1       1       1       4
sh              2       sleeping 2       1       4       6       3        1       1       1       8
schedule        3       running  2       1       254     10      3        1       1       1       258
$ foo&
$ schedule info
Process_Name    PID     State    Queue   Cycle   Arrival Ticket  Priority R_Prty  R_Arvl  R_Exec  Rank
-------------------------------------------------------------------------------------------------------
init            1       sleeping 1       1       0       0       3        1       1       1       4
sh              2       sleeping 2       2       4       6       3        1       1       1       9
foo             6       sleeping 2       44      473     8       3        1       1       1       520
foo             5       sleeping 2       1       472     3       3        1       1       1       476
foo             7       sleeping 2       44      473     6       3        1       1       1       520
foo             8       sleeping 2       44      473     4       3        1       1       1       520
foo             9       sleeping 2       44      474     3       3        1       1       1       521
foo             10      sleeping 2       44      474     1       3        1       1       1       521
schedule        11      running  2       0       912     4       3        1       1       1       915
$ schedule info
Process_Name    PID     State    Queue   Cycle   Arrival Ticket  Priority R_Prty  R_Arvl  R_Exec  Rank
-------------------------------------------------------------------------------------------------------
init            1       sleeping 1       1       0       0       3        1       1       1       4
sh              2       sleeping 2       2       4       6       3        1       1       1       9
schedule        12      running  2       0       2728    8       3        1       1       1       2731
foo             5       sleeping 2       1       472     3       3        1       1       1       476
foo             8       running  2       193     473     4       3        1       1       1       669
foo             9       runnable 2       180     474     3       3        1       1       1       657
foo             10      runnable 2       99      474     1       3        1       1       1       576
```