به نام خدا

گزارش کار پروژه سوم آزمایشگاه سیستم عامل

" زمان بندی پردازه ها"

گروه 5 :

- سینا طبسی
810199554

- سید حامد میرامیرخانی
810199500

- فاطمه محمدی
810199489

repository link: https://github.com/HamedMiramirkhani/OS_Lab_CA3

❖ بخش اول - سوالات:

1) چرا فراخوانی تابع sched، منجر به فراخوانی تابع schedulerمی‌شود؟ (منظور توضیح شیوه اجرای فرایند است).

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

همانطور در کد بالا نشان داده شده است، این تابع از تابع switch که عملیات تعویض متن را انجام میدهد استفاده شده است که context پردازه فعلی ذخیره میشود و به scheduler سویچ میکند.
در ادامه به دلایل فراخوانی sched و شیوه اجرای فرایند میپردازیم:

فراخوانی تابع shed به سه دلیل انجام میشود (زمان ترک پردازنده توسط پردازه):

1. پردازه بخواهد پردازنده را ترک کند. (با استفاده از فراخوانی exit)
2. پردازه توسط وقفه ایجاد شده توسط timer مجبور به ترک پردازنده شود (تابع yield فراخوانی میشود)
3. پردازه با فراخوانی sleep به حالت sleeping در بیاید

پس فراخوانی تابع shed به هردلیل،باید عملیات تعویض متن صورت بگیرد (همانطور که در بالا به دلایل اشاره شده است، واضح است در هر کدام از حالات بالا پردازه پردازنده را ترک می کند در نتیجه پردازنده باید به پردازه ای دیگر اختصاص داده شود) ، در این عملیات context مربوط پردازه در حال اجرا ذخیره میشود و context مربوط به استراکت cpu بازیابی میشود؛ پس از بازیابی context مربوط به scheduler بازیابی میشود. (دراقع scheduler جایگزین پردازه فعلی میشود تا پردازه بعدی برای اجرا را انتخاب کند)

در ادامه به توضیح عملکرد خود تابع scheduler می پردازیم:
این تابع در تابع mpmain (که توسط هر هسته ای که شروع به کار کند صدا زده میشود) صدا زده میشود تا کار زمانبند هسته شروع شود. این تابع به دنبال پردازه ای قابل اجرا (runable)میگردد و اگر چنین پردازه ای یافت، پس از تخصیص حافظه(تغییر حافظه به حافظه پردازه)، عملیات ،، تغییر متن،، را انجام میدهد و در واقع مراحل لازم برای شروع کار پردازه جدید را انجام میدهد.

2) 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 پردازه‌ها بدین منظور استفاده می‌گردد. درزمان‌بند کاملاً منصف در لینوکس، صف اجرا چه ساختاری دارد؟

در زمانبند کاملا منصفانه لینوکس نیز از صفی استفاده نمیشود، در واقع از red-black tree استفاده میشود که براساس زمان اولویت دهی میکند (time- ordered).
حال task ها runable در این درخت قرار میگیرند و به علت وجود self-balance در این درخت، مناسب این کار است.
در چپ ترین نود در این درخت، پردازه با کمترین برش زمانی در حین اجرا داشته، قرار گرفته است. (این اطلاعات در vruntime اطلاعات پردازه ذخیره شده است.)

3) همان‌طور که در پروژه اول مشاهده شد، هر هسته پردازنده در xv6 یک زمان‌بند دارد. در لینوکس نیز به همین‌گونه است. این دو سیستم‌عامل 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.

- در xv6:
مشترک است؛ چراکه تمامی پردازه ها از ptable استفاده میکنند
- در لینوکس:
مجزا است؛ چون هر پردازنده صف مختص به خود را دارد تنها از صف خودش پردازه ها را انتخاب میکند، اما این طراحی نیازمند load balancing میباشد که در صف مشترک نیازی به این کار نمی باشد.

- مزیت صف مشترک: صف مشترک موجب میشود پردازه های به صورت مساوی بین پردازنده تقسیم شوند و در نتیجه این امکان که یک پردازنده صف پر داشته باشد و یکی بی کار باشد وجود نخواهد داشت.
- نقص صف مشترک: امکان دارد یک پردازه یک بار در پردازنده a باشد و بعد در پردارنده b اجرا شود و با توجه به اینکه هر پردازنده یک high-level cache مختص خود دارد این کار کارایی مکانیزم کش کردن را بی اثر میکند.

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

حالتی را در نظر بگیریم که پردازنده در حالت idle باشد، یا در حالتی که تعدادی پردازه در حالت I/O قرار بگیرد و پردازه ی دیگری نیز در حالت runable نباشد، در این حالت پردازه ای اجرا نمیشود درنتیجه اگر وقفه ها نیز فعال نگردد پس از اتمام عمل I/O مربوط به پردازه ها، نمی توانیم ان ها را در حالت runable قرار دهیم تا اجرا شوند و در این حالت سیستم فریز خواهد شد. (در واقع اگر هیچ وقفه ای فعال نشود، هیچ CPUعه دیگری که در حال اجرای یک پردازه بوده باشد، نمی تواند هیچ تعویض متنی (و یا سیستم کال های وابسته به پردازه) را اجرا کند.)

حال اگر قفل ptable فعال شود تمامی وقفه ها غیرفعال خواهند شد و طبق توضیحات بالا لازم است، جهت جلوگیری از فریز شدن سیستم وقفه ها فعال شوند تا اگر حالت پردازه ای به تغییر نیاز پیدا کرده است بتوان ان را تغییر داد. همچنین باتوجه به اینکه در زمان اجرای پردازه، هیچ Preemptiveی نباید رخ بدهد پس باید قبل از شروع کار پردازه باید اینکار انجام شود.

بله در سیستم های تک هسته ای نیز به این کار نیاز است. زیرا اگر پردازنده در حالت idle قرار بگیرد، به دلیل غیرفعال شدن وقفه ها، ممکن است ورودی/خروجی ها هرگز نرسند.(به طور مثال پردازهها منتظر یک عمل ورودی/خروجی باشند)

5)

وقفه‌ها اولویت بالاتری نسبت به پردازه‌ها 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؟

سیستم ها اغلب مدیریت وقفه را بین یک First-Level Interrupt Handler(FLIH) - کنترل کننده وقفه سطح اول - و یک Second-Level Interrupt Handler(SLIH) - کنترل کننده وقفه سطح دوم - تقسیم می کنند.

- در ابتدا باید فرایند سطح اول انجام شود و سپس فرایند سطح دوم چرا که FLIH مسئول مدیریت وقفه های ضروری است اما SLIH مسئول مدیریت بخش های زمان بر وقفه ها است.

توضیحات بیشتر:

- FLIH مسئولیت سوئیچ زمینه، ذخیره سازی حالت و صف بندی یک عملیات مدیریت را دارد؛ در حالی که SLIH برنامه ریزی شده جداگانه مدیریت عملیات درخواستی را انجام می دهد.
- FLIH : وظیفه ی مدیریت وقفه های ضروری را در کمترین زمان دارد؛در سرویس دهی به وقفه ها دو حالت وجود دارد:

-> به وقفه سرویس کامل می دهد (به طور کامل سرویس دهی میکند)
-> اطلاعات ضروری وقفه، که تنها در زمان وقوع وقفه در دسترس است،را ذخیره میکند و برای سرویس دهی کامل وقفه یک SLIH زمانبندی میکند.

FLIH میتواند باعث ایجاد لگ و یا همان jitter در پردازه ها بشود. همچنین FLIH میتواند باعث چشم پوشی از وقفه ها نیز بشود . در FLIH یک تعویض متن صورت میگیرد و کد مربوط به مدیریت وقفه (وقفه صورت گرفته)بارگذاری و اجرا شود.

همانطور که اشاره شد SLIH مسئولیت مدیریت بخش های زمانبر وقفه میباشد و این کار همانند یک پردازه انجام میشود. SLIHها یا یک ریسه مخصوص در سطح هسته برای هر handler دارد و یا توسط یک thread pool مدیریت میشوند.

به دلیل امکان طولانی شدن زمان اجرای SLIH ها معمولا مانند پردازه ها زمانبندی میشود؛و SLIHها در یک صف اجرا به انتظار پردازنده قرار میگیرند.

مدیریت وقفه‌ها 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؟
در سیستم های بی درنگ لازم است که تأخیر وقفه را به حداقل رساند تا وظایف مورد توجه فوری قرار بگیرند. در واقع، برای سیستمهای بی درنگ سخت، تأخیر وقفه نباید صرفاً به حداقل برسد، بلکه باید برای برآورده کردن الزامات سختگیرانه این سیستمها محدود شود. یکی از عوامل مهم در تاخیر وقفه، میزان وقفه های زمانی است که ممکن است در حین به روز رسانی ساختارهای داده هسته غیرفعال شوند.
سیستم عامل های بی درنگ نیاز دارند که وقفه ها فقط برای دوره های زمانی بسیار کوتاه غیرفعال شوند.
از جمله راهکار هایی که میتوان برای رفع گرسنگی در حالت کلی نام برد، در ادامه آمده است.

الف) aging
راهکار aging به این صورت میباشد که هرچه یک پردازه (عموما پردازه کم اولویت مد نظر ماست) در صف انتظار باقی بماند به مرور زمان اولویت آن افزایش پیدا میکند. (وابسته به چگونگی اولویت بندی روند افزایش اولویت بندی میتواند متفاوت باشد برای مثال اگر اولویت بندی به صورت یک شماره گذاری در بازه (0 - n) باشد و پردازه دارای شماره i باشد و اولویت پردازه با شماره بیشتر، کمتر باشد،پس مرور یک مدت زمان مشخص، شماره اولویت ان پردازه در صف مانده ۱ واحد کاهش پیدا میکند *اولویتش بیشتر میشود*) و این موضوع تضمین میکند حداکثر پس از یک مدت زمانی پردازه بیشتری اولویت را خواهد داشت و دچار گرسنگی نمی شود.

ب) کم کردن بدترین-حالت نرخ ایجاد وقفه ها
این راهکار باید در سطح دستگاه هایی که موجب ایجاد وقفه هستند،انجام شود.

ج) استفاده از polling به جای interrupt

د) سرعت بخشیدن به مدیریت وقفه ها و کوتاه کردن انها
- استفاده از FLIH , SLIH (در بالاتر توضیح داده شده است)

ه) محدود کردن نرخ ایجاد interrupt

❖ بخش دوم:

◆ سطح اول: زمانبندی گردشی

proc.c:

```
130     p->entered_queue = ticks;
131     p->queue = 2;
```

```
342  void
343  update_queues(void) {
344    struct proc *p;
345    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
346        // if (p->state == RUNNABLE)
347            if (ticks - p->entered_queue >= STARVING_THRESHOLD) {
348                p->queue = 1;
349                p->entered_queue = ticks;
350            }
351    }
352  }
```

```
354   struct proc* round_robin(void) {
355     // for queue 1 with the highest priority
356       struct proc *p;
357       struct proc *min_p = 0;
358       int time = ticks;
359       int starvation_time = 0;
360       for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
361           if (p->state != RUNNABLE || p->queue != 1)
362               continue;
363
364           if (p->state != RUNNABLE || p->queue != 1)
365               continue;
366
367           int starved_for = time - p->entered_queue;
368           if (starved_for > starvation_time) {
369               starvation_time = starved_for;
370               min_p = p;
371           }
372       }
373       return min_p;
374   }
```

```
436           // Loop over process table looking for process to run.
437           acquire(&ptable.lock);
438           update_queues();
439           p = round_robin();
```

proc.h:

```
55    int queue;                    // queue number
56    int entered_queue;            // time entered queue
```

- سطح دوم: زمان بند بخت آزمایی

proc.c:

```
23  int
24  generate_random_num(int min, int max)
25  {
26      if (min >= max)
27          return max > 0 ? max : -1 * max;
28      acquire(&tickslock);
29      int diff = max - min + 1, time = ticks;
30      release(&tickslock);
31      int rand_num = (1 + (1 + ((time + 2) % diff ) * \
32      (time + 1) * 132) % diff) * (1 + time % max) * \
33      (1 + 2 * max % diff);
34      rand_num = rand_num % diff + min;
35      return rand_num;
36  }
```

```
133     p->tickets = generate_random_num(1, DEFAULT_MAX_TICKETS);
```

```
376 struct proc*
377 lottery(void) {
378   // for queue #2 and entrance queue
379   struct proc *p;
380   int total_tickets = 0;
381   for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
382       if (p->state != RUNNABLE || p->queue != 2)
383           continue;
384       total_tickets += p->tickets;
385   }
386   int winning_ticket = generate_random_num(1, total_tickets);
387   for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
388       if (p->state != RUNNABLE || p->queue != 2)
389           continue;
390       winning_ticket -= p->tickets;
391       if (winning_ticket <= 0)
392           return p;
393   }
394   return 0;
395 }
```

```
440             if (p == 0)
441                 p = lottery();
```

proc.h:

```
57    int tickets;                  // number of lottery tickets
```

- سطح سوم:

زمانبند اولین ورود-اولین رسیدگی (FCFS)

ابتدا به فیلدهای استراکت proc در فایل proc.h ، یک arrival_time اضافه میکنیم تا با توجه به آن بتوانیم پردازه ای که اول آمده است را پیدا کنیم.

در ادامه مقدار arrival_time را باید هنگام ایجاد پردازه ست کنیم که این کار را در فایل proc.c در تابع allocproc به صورت زیر انجام می دهیم:

```
acquire(&tickslock);
p->arrival_time = ticks;
release(&tickslock);
```

سپس تابع fcfs برای این زمانبندی به صورت زیر مینویسم:

```
struct proc*
fcfs(void) {
  struct proc *p = ptable.proc;

  struct proc *first_p = p;
  int min_arrival_time = 2e9;
  int flag = 0;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if (p->state != RUNNABLE || p->queue != 2) {
          continue;
    }
    if (p->arrival_time <= min_arrival_time) {
      flag = 1;
      min_arrival_time = p->arrival_time;
      first_p = p;
    }
  }
  if (flag == 1)
    return first_p;
  return 0;
}
```

تست سیستم‌کال‌ها

```
Group #5 Members:
1- Fatemeh Mohammadi
2- Sina Tabasi
3- Hamed Miramirkhani
$ print_procs
Process_Name     PID    State    Queue   Arrival   Ticket
-------------------------------------------------------------------
init             1      sleeping 2       0         27
sh               2      sleeping 2       5        7
print_procs      3      running  2       5362     29
$ set_proc_queue 1 3
$ print_procs
Process_Name     PID    State    Queue   Arrival   Ticket
-------------------------------------------------------------------
init             1      sleeping 3       0         27
sh               2      sleeping 2       5        7
print_procs      5      running  2       10302    9
```

تست aging

```
Group #5 Members:
1- Fatemeh Mohammadi
2- Sina Tabasi
3- Hamed Miramirkhani
$ print_procs
Process_Name     PID    State    Queue   Arrival   Ticket
-------------------------------------------------------------------
init             1      sleeping 2       0         27
sh               2      sleeping 2       4        11
print_procs      3      running  2       613      29
$ print_procs
Process_Name     PID    State    Queue   Arrival   Ticket
-------------------------------------------------------------------
init             1      sleeping 1       0         27
sh               2      sleeping 2       4        11
print_procs      4      running  2       1268     19
$ print_procs
Process_Name     PID    State    Queue   Arrival   Ticket
-------------------------------------------------------------------
init             1      sleeping 1       0         27
sh               2      sleeping 2       4        11
print_procs      5      running  2       2238     9
$ print_procs
Process_Name     PID    State    Queue   Arrival   Ticket
-------------------------------------------------------------------
init             1      sleeping 1       0         27
sh               2      sleeping 1       4        11
print_procs      6      running  2       6531     9
```

چاپ اطلاعات

در فراخوانی سیستمی زیر، تمامی پردازه پیمایش شده و اطلاعات لازم این پردازه ها چاپ می شود:

```
689  void
690  print_process_info()
691  {
692    static char *states[] = {
693      [UNUSED] "unused",
694      [EMBRYO] "embryo",
695      [SLEEPING] "sleeping",
696      [RUNNABLE] "runnable",
697      [RUNNING] "running",
698      [ZOMBIE] "zombie"
699    };
700
701    static int columns[] = {16, 8, 9, 8, 8, 8};
702    cprintf("Process_Name     PID    State    Queue   Arrival   Ticket  \n"
703      "-------------------------------------------------------------------\n");
704
705    struct proc *p;
706    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
707      {
708        if(p->state  == UNUSED)
709        {
710          continue;
711        }
712
713      const char* state;
714      if(p->state >= 0 && p->state < NELEM(states)  && states[p->state])
715      {
716        state = states[p->state];
717      }
718      else
719      {
720        state = "???";
721      }
722      cprintf("%s", p->name);
723      printspaces(columns[0] - strlen(p->name));
724      cprintf("%d", p->pid);
725      printspaces(columns[1] - (log_10(p->pid)));
726      cprintf("%s", state);
727      printspaces(columns[2] - strlen(state));
728      cprintf("%d", p->queue);
729      printspaces(columns[3] - (log_10(p->queue)));
730      cprintf("%d", p->arrival_time);
731      printspaces(columns[4] - (log_10(p->arrival_time)));
732      cprintf("%d", p->tickets);
733      printspaces(columns[5] - (log_10(p->tickets)));
734      cprintf("\n");
735    }
```

برنامه سطح کاربر foo:

در این برنامه سطح کاربر، تعدادی پردازه می سازد (در این کد ۳ پردازه می سازد و می توان این عدد را تغییر داد) و سپس عملیاتی طولانی را انجام می دهد (for با تعداد iteration بالا):

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

int main(int argc, char *argv[])
{
    for (int i = 0 ; i < 3 ; i++)
    {
        int pid = fork();
        if (pid == 0)
        {
            for (long int j = 0 ; j < 3000000000 ; j++)
            {
                int temp = 3;
                temp*=100;
            }
            exit();
        }
    }
    while (wait());
    return 0;
}
```

تست سطح کاربر:

```
Group #5 Members:
1- Fatemeh Mohammadi
2- Sina Tabasi
3- Hamed Miramirkhani
$ foo&
$ print_procs
Process_Name     PID    State    Queue   Arrival   Ticket
-------------------------------------------------------------------
init             1      sleeping 2       0         27
sh               2      sleeping 2       4        11
foo              5      runnable 2       514      11
foo              4      sleeping 2       512      19
foo              6      running  2       514      11
foo              7      runnable 2       514      11
print_procs      8      running  2       1600     17
$ set_proc_queue 7 1
$ print_procs
Process_Name     PID    State    Queue   Arrival   Ticket
-------------------------------------------------------------------
init             1      sleeping 2       0         27
sh               2      sleeping 2       4        11
foo              5      runnable 2       514      11
foo              4      sleeping 2       512      19
foo              6      runnable 2       514      11
foo              7      running  1       514      11
print_procs      10     running  2       5233     29
```