آزمایشگاه سیستم عامل

پروژه شماره ۳

علی پادیاو – کسری حاجی حیدری – اولدوز نیساری

بهار ۱۴۰۲

Repository Link: https://github.com/alumpish/OS-Lab-Projects

Latest Commit Hash: d1ee5c59c94aff422997b7bf955563a2fd43ea85

مقدمه

-۱

اگر به بدنه تابع sched در فایل proc.c دقت کنیم، مشاهده می‌کنیم که درون این تابع ، تابع swtch صدا شده است. این تابع در حقیقت تابعی است که برای ما عمل context switch را انجام میدهد. برای فراخوانی swtch کانتکست فعلی پردازه به همراه scheduler برنامه فعلی به تابع داده می‌شود تا بتواند اطلاعاتش را با context در آن ذخیره کند تا بتواند بعد از بازگشت بتواند پردازه را بازیابی کند و با scheduler عمل سوییچ را انجام دهد.

پس از این کار وضعیت پردازه از RUNNABLE به RUNNING تبدیل می‌شود.

لازم به ذکر است که تابع sched درون تابع‌های exit(), yield(), sleep() در زمان‌هایی که وضعیت پردازه به RUNNABLE تغییر پیدا میکند صدا زده میشود.

در نهایت در تابع sched، دوباره context switch صورت می‌گیرد و به کانتکست پردازنده‌ای که ذخیره کرده بودیم برمی‌گردیم. در واقع پردازه هیچ گاه از تابع scheduler خارج نمی‌شود و تنها با تعویض متن از پردازنده خارج می‌شود و با فراخوانی swtch دوباره شروع به کار می‌کند.

-۲

در حقیقت صف اجرا در لینوکس دارای ساختار red black tree است . هر وقت وضعیت یک پردازه RUNNABLE می‌شود به این درخت اضافه می‌شود و هر پردازه‌ای هم که وضعیت آن از RUNNABLE تغییر می‌کند از درخت حذف می‌شود.

بنا به طراحی این نوع درخت، نودهایی که در سمت چپ قرار می‌گیرند زمان پردازش کمتری دارند و عملا چپ ترین نود دارای کم‌ترین زمان است. و در اصل دارای بیشترین اولویت هم هست. در حقیقت در طراحی این صف اجرا vruntime به عنوان کلید استفاده شده است.

-۳

در لینوکس هر پردازنده صف جداگانه ای دارد در حالی که در xv6 یک صف مشترک (در اصل همان صف پردازه‌ها) را داریم.

```
struct
{
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

اگر به ساختار استراکت ptable توجه کنیم متوجه می‌شویم که دارای یک آرایه و یک قفل است. آرایه برای ذخیره سازی پردازه‌ها و قفل برای مدیریت دسترسی‌ها است. در واقع قفل به این صورت عمل می‌کند که هنگام استفاده lock فعال می‌شود و پس از آن قفل آزاد می‌شود.

مزیت صف مشترک این است که نیازی به برقراری توازن بین صفوف مختلف نیست در حالی که وقتی چند صف داریم ، وقتی و انرژی زیادی صرف تنظیم کردن صف‌های مختلف می‌شود. در مقابل مزیت داشتن چند صف این است که هر پردازنده فقط به صف خودش دسترسی دارد و نیازی به قفل کردن در هنگام استفاده نیست. در حالی که وقتی از صف مشترک استفاده می‌کنیم در زمان استفاده یک پردازنده از صف، صف را برای بقیه پردازنده‌ها باید قفل کنیم و آن‌ها باید منتظر بمانند تا کار آن پردازنده با صف تمام شود.

-۴

اگر در وضعیتی باشیم که هیچ پردازه ای در حالت آماده اجرا (RUNNABLE) نباشد و همه پردازه‌ها در حال اجرا برنامه دیگری باشند، مثلا در حال عملیات I/O باشند. در این زمان اگر وقفه برای مدتی فعال شود، پردازه‌ها زمان می‌یابند تا فرآیند I/O خاتمه یابد و به حالت RUNNABLE در بیایند و روند ادامه یابد. در نتیجه این ساز و کار برای سیستم‌های تک‌هسته‌ای هم نیاز است.

-۵

این دو سطح top-half و bottom-half نام دارند.

top-half روالی است که به وقفه‌ها پاسخ می‌دهد و bottom-half روالی است که در اصل زمان‌بندی آن بر عهده top-half است. top-half زمانی که پردازنده scheduler را دریافت

می‌کند اجرا می‌شود. در این بخش وقفه و زمان بند هر دو غیرفعال هستند. در این بخش در اصل تنها بخشی که ضروری است اجرا می‌شود. bottom-half برای اجرا کارهای باقی مانده و به تعویق افتاده است. در این بخش وقفه‌ها فعال هستند ولی scheduler غیر فعال است.

مشکل گرسنگی توسط الگوریتم aging هندل می‌شود. این کار به این صورت انجام می‌شود که با مرور زمان تسک‌های کم اولویت، اولویتشان یک پله افزایش می‌یابد و پله پله به اولین سطح اولویت سیستم می‌رسند و اجرا می‌شوند.

زمان‌بندی بازخوردی چندسطحی

به دلیل مشکل در lock کردن، از همان یک صف xv6 استفاده می‌کنیم. منتها با labelدار کردن processها به نوعی سطح‌ها را ایجاد می‌کنیم. برای این کار، یک استراکت را به ساختار پردازه اضافه می‌کنیم.

```
struct schedinfo
{
  enum schedlevel queue; // Process queue
  int last_run;          // Last time process was run
  int arrival_time;      // Time process arrived in queue
  int tickets_count;     // Number of tickets for lottery scheduler
};
```

درواقع هر پردازه می‌تواند در یکی از سطح‌های زیر باشد:

```
enum schedlevel
{
  UNSET,
  ROUND_ROBIN,
  LOTTERY,
  FCFS
};
```

۱- زمان‌بند نوبت گردشی

```
1  struct proc *
2  roundrobin(struct proc *lastScheduled)
3  {
4    struct proc *p = lastScheduled;
5    for (;;)
6    {
7      p++;
8
9      // Reached the end of the process table, wrap around to the beginning
10     if (p >= &ptable.proc[NPROC])
11       p = ptable.proc;
12
13     // Check if the process is runnable and belongs to the round robin scheduling queue
14     if (p->state == RUNNABLE && p->sched.queue == ROUND_ROBIN)
15       return p;
16
17     // It means we've checked all processes and didn't find any eligible ones
18     if (p == lastScheduled)
19       return 0;
20   }
21 }
```

برای این الگوریتم، نیاز داریم که بدانیم آخرین پردازه‌ای که در این سطح زمان‌بندی شده کدام است، تا از بعد آن شروع به اختصاص دادن پردازنده کنیم. برای همین آن پردازه را به تابع `roundrobin` پاس می‌دهیم. اگر پردازه‌ای پیدا نشود، ۰ را برمی‌گردانیم تا در `scheduler` سراغ سطح بعدی برویم.

۲- زمان‌بند بخت‌آزمایی

```
struct proc *
lottery()
{
  struct proc *result = 0;

  uint total_tickets = 0;
  for (int i = 0; i < NPROC; ++i)
  {
    if ((ptable.proc[i].state == RUNNABLE) && (ptable.proc[i].sched.queue == LOTTERY))
    {
      total_tickets += ptable.proc[i].sched.tickets_count;
    }
  }
  if (total_tickets == 0)
  {
    return result;
  }

  // Choose a random ticket between 0 and the total number of tickets
  uint winning_ticket = rand() % total_tickets;

  // Iterate through all runnable processes and check which one holds the winning ticket
  uint prev_tickets_sum = 0;
  for (int i = 0; i < NPROC; ++i)
  {
    if (ptable.proc[i].state != RUNNABLE)
      continue;
    if (ptable.proc[i].sched.queue != LOTTERY)
      continue;
    if ((winning_ticket >= prev_tickets_sum) && (winning_ticket <= prev_tickets_sum + ptable.proc[i].sched.tickets_count))
    {
      // This process holds the winning ticket
      result = &ptable.proc[i];
      break;
    }
    prev_tickets_sum += ptable.proc[i].sched.tickets_count;
  }

  return result;
}
```

ما در هر پردازه که در سطح `lottery` است، یک عدد به `tickets_count` از ۱ تا ۱۰ اختصاص دادیم. در این تابع همه این مقادیر را جمع می‌کنیم و با ایجاد یک عدد رندوم می‌بینیم که در بازه کدام پردازه قرار می‌گیرد. اگر پردازه‌ای انتخاب شد، آن را به `scheduler` ریترن می‌کنیم. در غیر این صورت، ۰ را برمی‌گردانیم تا جستجو در سطح بعدی ادامه پیدا کند.

۳- زمان‌بند اولین ورود-اولین رسیدگی (FCFS)

```
1  struct proc *
2  fcfs(void)
3  {
4    struct proc *result = 0;
5    int min_arrival = INT_MAX;
6
7    for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++)
8    {
9      if (p->state != RUNNABLE || p->sched.queue != FCFS)
10       continue;
11     if (result == 0 || p->sched.arrival_time < min_arrival)
12     {
13       result = p;
14       min_arrival = p->sched.arrival_time;
15     }
16   }
17
18   return result;
19 }
```

در همه پردازه های این سطح پیمایش می‌کنیم و پردازه‌ای که کمترین `arrival_time` را دارد را برمی‌گردانیم.

فراخوانی‌های سیستمی

۱- تغییر صف پردازه

```
int
sys_change_scheduling_queue(void)
{
  int queue_number, pid;
  if(argint(0, &pid) < 0 || argint(1, &queue_number) < 0)
    return -1;

  if(queue_number < ROUND_ROBIN || queue_number > FCFS)
    return -1;

  return change_queue(pid, queue_number);
}
```

```
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.queue;
      p->sched.queue = new_queue;
      if (new_queue == LOTTERY && p->sched.tickets_count <= 0)
      {
        p->sched.tickets_count = (rand() % MAX_RANDOM_TICKETS) + 1;
      }
      break;
    }
  }
  release(&ptable.lock);
  return old_queue;
}
```

۲- مقداردهی بلیت بخت‌آزمایی

```
int
sys_set_lottery_ticket(void) {
  int pid, tickets;
  if(argint(0, &pid) < 0 || argint(1, &tickets) < 0)
    return -1;

  if (tickets < 0)
    return -1;

  return set_lottery_ticket(pid, tickets);
}
```

```
int set_lottery_ticket(int pid, int tickets)
{
  acquire(&ptable.lock);
  struct proc *p;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid == pid && p->sched.queue == LOTTERY)
    {
      p->sched.tickets_count = tickets;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

۳- چاپ اطلاعات

```
int
sys_print_process_info(void)
{
  print_process_info();
  return 0;
}
```

```
void print_process_info()
{
  static char *states[] = {
      [UNUSED] "unused",
      [EMBRYO] "embryo",
      [SLEEPING] "sleeping",
      [RUNNABLE] "runnable",
      [RUNNING] "running",
      [ZOMBIE] "zombie"};

  cprintf("name    pid     state    Queue   arrive_time ticket\n"
          ".....................................................\n");

  struct proc *p;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state == UNUSED)
      continue;

    cprintf("%s", p->name);
    cprintf("%d", p->pid);
    cprintf("%s", states[p->state]);
    cprintf("%d", p->sched.queue);
    cprintf("%d", p->sched.arrival_time);
    cprintf("%d", p->sched.tickets_count);

    cprintf("\n");
  }
```

برنامه سطح کاربر

برنامه‌ای به نام sched ایجاد کردیم که در آن از ۳ سیستم‌کالی که تعریف کردیم استفاده می‌کنیم.

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

void display_help()···

void display_process_info()···

void set_process_queue(int pid, int new_queue)···

void set_process_tickets(int pid, int tickets)···

int main(int argc, char *argv[])
{
    if (argc < 2)
    {
        display_help();
        exit();
    }
    if (!strcmp(argv[1], "info"))
        display_process_info();
    else if (!strcmp(argv[1], "set_queue"))
    {
        if (argc < 4)
        {
            display_help();
            exit();
        }
        set_process_queue(atoi(argv[2]), atoi(argv[3]));
    }       You, 53 minutes ago • Fix some minor issues
    else if (!strcmp(argv[1], "set_tickets"))
    {
        if (argc < 4)
        {
            display_help();
            exit();
        }
        set_process_tickets(atoi(argv[2]), atoi(argv[3]));
    }
    else
        display_help();
    exit();
}
```

برنامه دیگری به نام foo نیز داریم که ۴ پردازه در خودش fork می‌کند تا اطلاعات آن را با سیستم‌کال چاپ اطلاعات مشاهده کنیم.

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

#define FORK_COUNT 4
#define LOOP_COUNT 1000000000

int main()
{
    for (int i = 0; i < FORK_COUNT; ++i)
    {
        int pid = fork();
        if (pid > 0)
            continue;
        if (pid == 0)
        {
            printf(1, "Process %d started\n", getpid());
            sleep(1000);

            for (int j = 0; j < 2 * i; ++j)
            {
                int x = 1;
                for (long k = 0; k < LOOP_COUNT; k++)
                    x += 1;
            }

            printf(1, "Process %d finished\n", getpid());
            exit();
        }
    }

    for (int i = 0; i < FORK_COUNT; ++i)
        wait();

    exit();
}
```

اجرای برنامه سطح کاربر

```
$ sched info
name       pid     state          queue   arrive_time   ticket
.....................................................
init       1       sleeping        1       0             0
sh         2       sleeping        2       7             6
sched      3       running         2       306           10
$ foo&
$ Process 6 started
Process 7 started
Process 8 started
Process 9 started
sched info
name       pid     state          queue   arrive_time   ticket
.....................................................
init       1       sleeping        1       0             0
sh         2       sleeping        2       7             6
foo        6       sleeping        2       572           8
foo        5       sleeping        2       570           10
foo        7       sleeping        2       572           6
foo        8       sleeping        2       573           9
foo        9       sleeping        2       574           8
sched      10      running         2       788           5
$ Process 6 finished
sched info
name       pid     state          queue   arrive_time   ticket
.....................................................
init       1       sleeping        1       0             0
sh         2       sleeping        2       7             6
sched      11      running         2       2026          2
foo        5       sleeping        2       570           10
foo        7       runnable        2       572           6
foo        8       runnable        2       573           9
foo        9       running         2       574           8
```

```
$ foo&
$ sched info
name       pid     state          queue   arrive_time   ticket
.....................................................
init       1       sleeping        1       0             0
sh         2       sleeping        2       9             6
foo        5       sleeping        2       373           6
foo        4       sleeping        2       371           5
foo        6       sleeping        2       374           10
foo        7       sleeping        2       374           8
foo        8       sleeping        2       375           7
sched      9       running         2       624           5
$ sched set_queue 8 3
Queue changed successfully
$ sched info
name       pid     state          queue   arrive_time   ticket
.....................................................
init       1       sleeping        1       0             0
sh         2       sleeping        2       9             6
sched      11      running         2       1916          8
foo        4       sleeping        2       371           5
foo        6       running         2       374           10
foo        7       running         2       374           8
foo        8       runnable        3       375           7
```