# زمان بندی در xv6

## 1 ) چرا فراخوانی تابع sched() منجر به فراخوانی تابع scheduler() میشود؟

هر هسته هنگام شروع به کار تابع `mpmain` را کال میکند که خود منتهی به تابع scheduler میشود و زمان بند مربوط به هسته را اجرا میکند. زمان بند با دسترسی به `ptable` که حاوی اطلاعات هر پردازه است به دنبال پردازه قابل اجرا می گردد و حافظه پردازنده را با تابع `switchuvm` به حافظه پردازه مد نظر تغییر میدهد. سپس با استفاده از تابع دیگری بنام `swtch` که کد اسمبلی است عملیات context switch را انجام میدهد و رجیسترهای متن[1] قبلی را در آدرس خود ذخیره کرده و رجیسترهای متن جدید را از آدرس آن بازیابی میکند. بدین صورت شمارنده برنامه [2] مربوط به متن جدید در دسترس بوده و پردازه جدید آماده اجراست.

### *در حالات زیر نیاز به فراخوانی زمان بند یا همان تابع sched میشود:*

1 – با تابع `exit` خود پردازه خاتمه یابد.

2 – با تابع `sleep` تابع موقتا متوقف شود.

3 – با رخداد یک وقفه پردازه مجبور به خروج از پردازنده شود که درین حالت با اجرای yield در بدنه آن تابع sched صدا زده میشود.

درین حالات با فراخوانی تابع `sched` بار دیگر عملیات تعویض متن صورت گرفته و رجیستر های مربوط به پردازه ذخیره میشوند و رجیستر های زمان بند بازیابی شده و شمارنده به جایی میرود که باعث اجرای تابع scheduler میشود. لازم بذکرست پردازه مربوط به تابع زمان بندد همیشه در این تابع باقی می ماند و فقط هنگام تعویض متن از پردازنده خارج میشود. شمارنده مربوط به پردازه ها مستقیما ذخیره نشده و در آدرس بازگشت تابع است و هنگام ذخیره رجیستر ها در استک push میشود و در هنگام برگشت از استک توسط تابع `scheduler` پاپ میشود و در رجیستر مربوطه اش ذخیره میشود.

[1] context
[2] Program counter

## زمان بندی

### 2) در لینوکس صف اجرا چه ساختاری دارد؟

در لینوکس صف اجرا یک درخت سیاه قرمز است که در چپ ترین گره این درخت پردازه ای قرار دارد که کمترین برش زمانی در هنگام اجرا را داشته است.

### بررسی لینوکس و xv6 از منظر مشترک یا جدا بودن صف های زمان بندی

هر پردازنده زمان بند خودش را دارد ولی در xv6 تنها از یک صف استفاده میشود که برای همه پردازه ها مشترک است

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

درین صف می توان حداکثر به تعداد 64 پردازه را نگه داشت که همان عدد ثابت NPROC است.

برای جلوگیری از اثرات ناشی از عمل چند پردازنده بر روی صف از spinlock استفاده کرده و هنگام دسترسی به پردازه ای در ptable ابتدا lock آن را `acquire` می کنیم و پس از اتمام فرآیند آن را `release` می کنیم تا جدول مجدد قابل دسترسی برای سایر پردازنده ها باشد.

داشتن یک صف پیاده سازی را تسهیل کرده اما نیاز به عملیات lock دارد که بازدهی را تحت تاثیر قرار خواهد داد.
همچنین چون هر پردازه بین پردازنده های مختلف منتقل میشود و هر پردازنده حافظه نهان خود را دارد لذا کارایی cache بسیار کاهش می یابد.

در لینوکس هر پردازنده صف خودش را  دارد و پردازه ها بصورت جداگانه در پردازنده ها قرار می گیرند. این پیاده سازی نیازمند آن است که مدیریت صف ها بگونه ای باشد که بعضی صف ها خالی و بعضی دیگر پر باشند.

### اجرای وقفه در ابتدای حلقه

هنگام قفل ptable تمامی وقفه ها غیرفعال میشوند. حال اگر پردازه یا پردازه هایی موجود باشند که منتظر عملیات I/O هستند و پردازه قابل اجرای دیگری وجود نداشته باشد ، هیچ پردازه ای اجرا نمی شود. از طرفی اگر وقفه ها فعال نشوند پس از اتمام عملیات I/O امکان تغییر حالت پردازه ها به قابل اجرا نیست و سیستم فریز میشود. به همین دلیل است که درین حلقه و قبل قفل کردن ptable برای مدت کوتاهی وقفه ها فعال میشوند تا بتوان حالت پردازه ها را تغییر داد.

## سطوح مدیریت وقفه ها در لینوکس

مدیریت وقفه ها اصولا در دو سطح اول و دوم انجام میشود. به این دو سطح FLIH[3] و SLIH[4] گفته میشود. در لینوکس به آنها نیمه بالایی و پایینی نیز گفته میشود. وظیفه سطح اول مدیریت وقفه های ضروری در کمترین زمان ممکن است. یا بصورت کامل آن را سرویس دهی میکند یا اطلاعات ضروری آن را ذخیره کرده و یک کنترل کننده سطح دو را برای این کار زمان بندی می کند. در فرایند مدیریت وقفه در سطح اول ابتدا یک تعویض متن صورت می گیرد و کد مربوط به مدیریت کننده وقفه اجرا میشود. درین سطح می توان شاهد lag در پردازه ها بود .

سطح دوم اما به سرویس دهی بخش هایی از وقفه ها می پردازد که زمان بر هستند و این کار مانند یک پردازه انجام میشود یعنی یا یک ریسه [5]در سطح هسته برای هر کنترل کننده وجود دارد یا توسط یک thread pool این ریسه ها از سطح کاربر مدیریت میشوند. سپس تمام این SLIH ها در یک صف قرار گرفته و منتظر پردازنده می شوند و چون ممکن است این هندلر ها طولانی شوند لذا مانند ریسه و پردازه ها زمان بندی میشوند

## حل مشکل گرسنگی در سیستم های بی درنگ

ممکن است پردازه ای بدلیل اولویت کمتر مدت طولانی در صف انتظار بماند درین حالت می گوییم گرسنگی رخ داده است. برای حل این مشکل از aging استفاده می کنیم بدین صورت که هر چه پردازه ای با اولویت کمتر بیشتر در انتظار بماند اولویتش به مرور زمان بیشتر میشود. این کار تا زمانی ادامه می باید که پردازنده به آن پردازه اختصاص یابد.
بعلاوه می توان با انجام بعضی راهکار ها از اختصاص بیش از حد پردازنده به وقفه ها جلوگیری کرد:

- ✓ کوتاه و سریع نگه داشتن وقفه مثلا با استفاده از مدیریت دوسطحی ذکرشده
- ✓ محدود کردن نرخ ایجاد وقف ها
- ✓ کم کردن بدترین حالتی که وقفه ها ایجاد میشوند
- ✓ استفاده از روش سرکشی بجای مدیریت وقفه ها

[3] First Level Interrupt Handler
[4] Second Level Interrupt Handler
[5] Thread

## پیاده سازی زمان بند با MLFQ[6]

در ابتدای کار در ساختار proc یک فیلد جدید برای ذخیره سازی اطلاعات مربوط به زمان بندی هر پردازه ایجاد می کنیم. در بدنه این ساختار age ، یک enum برای مشخص کردن شماره صف هر پردازه و همچنین یک سختار داده برای ذخیره سازی اطلاعات مربوط به هر پردازه است وقتی که در صف BJF قرار دارد از جمله سیکل های اجرا و ضرایب مربوطه برای محاسبه rank

```
struct scheduling
{
  enum shced_queues queue; // Process queue
  uint age;                // Last time process was run
  struct bjf_data bjf;     // Best-Job-First scheduling data
};
```

سپس تابع `scheduler` را در فایل proc.c پیاده سازی میکنیم:

```
void scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  // cprintf("Here-1\n");
  c->proc = 0;

  for (;;)
  {
    // cprintf("Here-2\n");

    refresh_queue();
    // Enable interrupts on this processor.
    sti();

    while (1)
    {
      acquire(&ptable.lock);
      p = get_rr_proc();
      if (p == 0)
        p = get_lcfs_proc();
      if (p == 0)
        p = get_bjf_proc();
      release(&ptable.lock);
      if (p != 0)
        break;
    }
    run_proc(p, c);

    acquire(&ptable.lock);
    acquire(&tickslock);
    p->scheduling_data.age = ticks;
    release(&tickslock);
    release(&ptable.lock);
  }
}
```

[6] Multi Level Feedback Queue

درین تابع ابتدا پردازنده را خالی میکنیم و سپس در یک حلقه ابتدا شماره صف تمام پردازه ها با تابع refresh_queue که توضیح آن جلوتر آورده شده است شماره صفوف پردازه ها را به روز می کنیم. در ادامه در یک حلقه دیگر به ترتیب با قفل کردن ptable به سراغ یک پردازه قابل اجرا در صفوف RR ، LCFS و BJF می رویم و پس از یافت شدن پردازه مجدد Ptable برای مدیریت باقی پردازه ها رها میشود. حال پردازه مد نظر با تابع run_proc اجرا می شود. سپس در انتها برای به روز رسانی عمر پردازه مجدد ptable و هم ساختار مربوط به ticks را قفل و پس از ست شدن آخرین زمان اجرای پردازه به ticks حال حاضر مجدد دو ساختار داده را رها می کنیم.

تابع change_queue آیدی یک پردازه و شماره یک صف را گرفته و چک میکند اگر آن پردازه در صفی قرار نداشت با توجه به آیدی آن پردازه آن را در صف مناسب قرار می دهد.

```
int change_queue(int pid, int new_queue)
{
  struct proc *p;
  int newq = new_queue;

  if (newq == NO_QUEUE)
  {
    if (pid == 1 || pid == 2)
      newq = ROUND_ROBIN;
    else if (pid > 1)
      newq = LCFS;
    else
      return -1;
  }
```

اما در تابع اصلی scheduler پنج تابع زیر استفاده شده اند:

- refresh_queue
- get_rr_proc
- get_bjf_proc
- get_lcfs_proc
- run_proc

در ادامه به توضیح هر کدام خواهیم پرداخت.

### Refresh_queue()

درین تابع در یک حلقه برای تک تک پردازه های موجود که در صفی قرار ندارد شماره صف ها را با توجه به pid آنها اصلاح میکنیم

```
void refresh_queue()
{
  struct proc *temp;
  for (temp = ptable.proc; temp < &ptable.proc[NPROC]; temp++)
  {
    if (temp->scheduling_data.queue == NO_QUEUE)
      change_queue(temp->pid, NO_QUEUE);
  }
}
```

### Run_proc()

درین تابع ابتدا ptable را قفل می کنیم و بعد پردازنده را به پردازه آماده اجرا اختصاص می دهیم. در ادامه با تابع switchuvm حافظه پردازنده را به حافظه پردازه کنونی تبدیل کرده حالت پردازه موجود هم به Running تغییر یافته و عملیات تعویض متن با تابع swtch صورت گرفته و طبق دستور کار تعداد سیکل اجرای پردازه بعلاوه 0.1 میشود و در آخر بعد از تغییر حافظه مجدد پردازنده با تابع switchkvm پردازنده مجدد خالی شده و ptable رها میشود.

```
void run_proc(struct proc *p, struct cpu *c)
{
  acquire(&ptable.lock);
  // Switch to chosen process.  It is the process's job
  // to release ptable.lock and then reacquire it
  // before jumping back to us.
  c->proc = p;
  switchuvm(p);
  p->state = RUNNING;

  swtch(&(c->scheduler), p->context);

  p->scheduling_data.bjf.executed_cycle += 0.1;

  switchkvm();

  // Process is done running for now.
  // It should have changed its p->state before coming back.
  c->proc = 0;
  release(&ptable.lock);
}
```

### Get_rr_proc()

درین تابع برای تمام پردازه ها بررسی میشود و بین تمام پردازه هایی که در صف RR هستند و وضعیت آنها Runnable است رتبه آنها را در متغیر rank ریخته با هم مقایسه می کنیم و پردازه با آخرین زمان اجرای عقب تر اجرا میشود.

```
struct proc *get_rr_proc()
{
  struct proc *res = 0, *temp;
  float min;
  for (temp = ptable.proc; temp < &ptable.proc[NPROC]; temp++)
  {
    if (temp->state != RUNNABLE || temp->scheduling_data.queue != ROUND_ROBIN)
      continue;
    // cprintf("found round robin: %d\n",temp->pid);
    float rank = temp->scheduling_data.age;
    if (res == 0 || rank < min)
    {
      res = temp;
      min = rank;
    }
  }
  return res;
}
```

## Get_lcfs_proc

این تابع مشابه تابع قبل است با این تفاوت که پردازه ها از نظر دیرتر آمدن از نظر زمانی مقایسه می شوند و هر پردازه ای آخر آمده باشد پردازنده به آن اختصاص خواهد یافت.

```
struct proc *get_lcfs_proc()
{
  struct proc *res = 0, *temp;
  float max;
  for (temp = ptable.proc; temp < &ptable.proc[NPROC]; temp++)
  {
    if (temp->state != RUNNABLE || temp->scheduling_data.queue != LCFS)
      continue;
    // cprintf("found lcfs\n");

    float rank = temp->scheduling_data.age;
    if (res == 0 || rank > max)
    {
      res = temp;
      max = rank;
    }
  }
  return res;
}
```

### Get_bjf_proc

درین تابع نیز ماشبه دو تابع قبلی از بین پردازه هایی که در صف bjf قرار دارند و حالت Runnable دارند رتبه ها را که بر اساس فرمول داده شده در دستور پروژه آورده شده و مقدارش برای هر پردازه توسط تابع `get_bjf_rank` محاسبه میشود مقایسه کرده و هر پردازه ای که شاخص کمتری دارد را به پردازنده ارسال می کند.

```
struct proc *get_bjf_proc()
{
  struct proc *res = 0, *temp;
  float min;
  for (temp = ptable.proc; temp < &ptable.proc[NPROC]; temp++)
  {
    if (temp->state != RUNNABLE || temp->scheduling_data.queue != BJF)
      continue;
    float rank = get_bjf_rank(temp);
    if (res == 0 || rank < min)
    {
      res = temp;
      min = rank;
    }
  }
  return res;
}
```

## فرایند aging

```
void do_aging(int tiks)
{      nima8tajik, 3 days ago • do_aging and change_queue added
  struct proc *p;
  acquire(&ptable.lock);

  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if (p->state == RUNNABLE && p->scheduling_data.queue != ROUND_ROBIN)
    {
      // cprintf("aging %d:%d \n", tiks, p->scheduling_data.age);
      if (tiks - p->scheduling_data.age > AGED_OUT)
      {
        // cprintf("%d\t%d\t%d\t%d\n", tiks, p->scheduling_data.age, p->pid, p->scheduling_data.queue);
        release(&ptable.lock);
        change_queue(p->pid, p->scheduling_data.queue == LCFS ? ROUND_ROBIN : LCFS);
        acquire(&ptable.lock);
        p->scheduling_data.age = ticks;
      }
    }

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

درین تابع طبق تصویر فوق برای افزایش اولویت پردازه ها ابتدا ptable را قفل می کنیم و سپس در تمام پردازه ها اگر پردازه ای در صف RR نباشد و ضمنا Runnable باشد ، اگر سن آن از حد بالایی بیشتر شود بدین معناست که باید به اولویت بالاتری برود. برای این کار ptable مجدد باز شده و پردازه مورد نظر با تابع change_queue صف خود را تغییر می دهد.

سپس ptable برای آپدیت سن مجددا قفل شده و با استفاده از ticks زمان آخرین اجرای پردازه را ذخیره میکنیم و در نهایت مجدد ptable قفل خود را release میکند.

## برنامه سطح کاربر

درین بخش در انتهای فایل proc.c یک سیستم کال بنام sys_ps برای چاپ اطلاعات پردازه هایی که خودمان در فایل foo.c ایجادشان کرده ایم نوشته شده است.

```
int sys_ps(void)      AliGhAliGh, 2 days ago • some bug fixed
{
  cprintf("proc name\tPID\tState\t\tQueue\tCycle\tArrival\tPriority\tR_Party\tR_arvl\tR_Exec\tR_Size\tRank\n");
  cprintf("-------------------------------------------------------------------------------------------------------------------------------\n");
  acquire(&ptable.lock);
  struct proc *p;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state == RUNNABLE || p->state == RUNNING || p->state == SLEEPING)
    {
      cprintf(p->name);
      cprintf("\t\t");
      cprintf("%d", p->pid);
      cprintf("\t");
      cprintf(p->state == RUNNABLE ? "RUNNABLE" : p->state == RUNNING ? "RUNNING\t"
                                              : p->state == SLEEPING  ? "SLEEPING"
                                                                      : "ZOMBIE");
      cprintf("\t");
      cprintf("%d\t", p->scheduling_data.queue);
      cprintf("%d\t%d\t%d\t\t%d\t%d\t%d\t%d\t%d\n", (int)p->scheduling_data.bjf.executed_cycle, (int)p->xticks,
              (int)p->scheduling_data.bjf.priority, (int)p->scheduling_data.bjf.priority_ratio, (int)p->scheduling_data.bjf.arrival_time_ratio,
              (int)p->scheduling_data.bjf.executed_cycle_ratio, (int)p->scheduling_data.bjf.process_size_ratio, (int)get_bjf_rank(p));
    }
  }
  release(&ptable.lock);
  return 0;
}
```

```c
#define PROCS_NUM 5

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

foo.c

خروجی این سیستم کال در دو حالتی که گرسنگی داریم و نداریم در تصاویر زیر قابل مشاهده است:

| $ proc name | PID | State | Queue | Cycle | Arrival | Priority | R_Party | R_arvl | R_Exec | R_Size | Rank |
|---|---|---|---|---|---|---|---|---|---|---|---|
| init | 1 | SLEEPING | 1 | 1 | 0 | 3 | 1 | 1 | 1 | 1 | 12292 |
| sh | 2 | SLEEPING | 1 | 0 | 6 | 3 | 1 | 1 | 1 | 1 | 16393 |
| foo | 5 | RUNNABLE | 2 | 0 | 793 | 3 | 1 | 1 | 1 | 1 | 13084 |
| foo | 4 | SLEEPING | 2 | 0 | 792 | 3 | 1 | 1 | 1 | 1 | 13083 |
| foo | 6 | RUNNABLE | 2 | 0 | 793 | 3 | 1 | 1 | 1 | 1 | 13084 |
| foo | 7 | RUNNING | 2 | 0 | 794 | 3 | 1 | 1 | 1 | 1 | 13085 |
| foo | 8 | RUNNABLE | 2 | 0 | 794 | 3 | 1 | 1 | 1 | 1 | 13085 |
| foo | 9 | RUNNABLE | 2 | 0 | 794 | 3 | 1 | 1 | 1 | 1 | 13085 |

درین حالت قبل از رخ دادن گرسنگی است و پردازه هایی که در صف اول هستند چون در حالت Sleeping می باشند و در انتظار اجرا نیستد تخصیص نیافتن پردازنده به آنها باعث رخ دادن گرسنگی نمی شود.

$ ps

| proc name | PID | State | Queue | Cycle | Arrival | Priority | R_Party | R_arvl | R_Exec | R_Size | Rank |
|---|---|---|---|---|---|---|---|---|---|---|---|
| init | 1 | SLEEPING | 1 | 1 | 0 | 3 | 1 | 1 | 1 | 1 | 12292 |
| sh | 2 | SLEEPING | 1 | 0 | 7 | 3 | 1 | 1 | 1 | 1 | 16394 |
| ps | 10 | RUNNING | 1 | 0 | 428 | 3 | 1 | 1 | 1 | 1 | 12719 |
| foo | 4 | SLEEPING | 2 | 0 | 303 | 3 | 1 | 1 | 1 | 1 | 12594 |
| foo | 6 | RUNNABLE | 1 | 9 | 304 | 3 | 1 | 1 | 1 | 1 | 12604 |
| foo | 7 | RUNNABLE | 1 | 4 | 304 | 3 | 1 | 1 | 1 | 1 | 12599 |
| foo | 8 | RUNNABLE | 1 | 4 | 304 | 3 | 1 | 1 | 1 | 1 | 12599 |
| foo | 9 | RUNNABLE | 1 | 4 | 304 | 3 | 1 | 1 | 1 | 1 | 12599 |

درین حالت اما مشهود است که پردازه هایی با حالت Runnable در صف یک منتظر پردازنده هستند و این موضوع می تواند با گذشت زمان طولانی منجر به این شود که پردازه های با اولویت بالاتر بدلیل انتظار بیش از حد برای تخصیص پردازنده دچار گرسنگی یا starvation شوند.