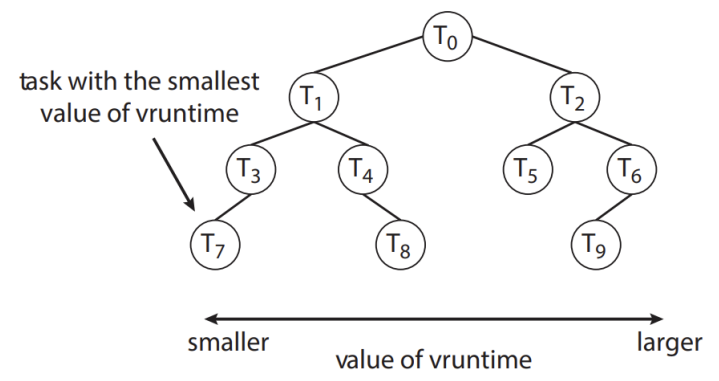
1) چرا فراخوانی sched منجر به فراخوانی scheduler می‌شود؟ (منظور، توضیح شیوه‌ی اجرای فرایند است)

```
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;
}
```

تعریف این تابع در فایل proc.c آمده است. این تابع در توابع exit() و yield() و sleep() فراخوانی می‌شود. همانطور که می‌بینیم در این تابع، تابع swtch() با پارامترهای context پردازه فعلی و scheduler برای پردازه فعلی، فراخوانی می‌شود؛ در واقع ابتدا context پردازه فعلی ذخیره می‌شود (تا بعدا هنگام بازگشت به این پردازه بتوان آن را بازیابی کرد) و سپس به scheduler سوییچ می‌کنیم و scheduler فعال می‌شود تا پردازه‌ی جدید را با توجه به الویت‌هانی تعریف شده، برای اجرا انتخاب کند؛ با اینکار وضعیت پردازه‌ی جدید از RUNNABLE به RUNNINGX تبدیل می‌شود.

2) صف پردازه‌هایی که تنها منبعی که برای اجرا کم دارندردازنده است، صف آماده یا صف اجرا نام دارد. در xv6 صف آماده مجزا وجود ندارد و از صف پردازه بدین منظور استفاده می‌شود. در زمان‌بند کاملا منصف در لینوکس، صف اجرا چه ساختاری دارد؟



زمان‌بند CFS در لینوکس از الگوریتمی کارآمد برای انتخاب تسکی که قرار است اجرا شود استفاده می‌کند؛ این سیستم‌عامل به جای استفاده از ساختمان داده‌ی صف استاندارد، هر تسک RUNNABLE را در یک red black tree قرار می‌دهد. این درخت، یک درخت دودویی متوازن است که مقدار key آن، مطابق با مقدار vruntime است. شکل این درخت (صفحه 237 کتاب) مطابق روبه‌رو است. وقتی وضعیت یک تسک RUNNABLE می‌شود، به درخت اضافه می‌شود و اگر یک تسک که جزء درخت است دیگر RUNNABLE نباشد، از درخت حذف می‌شود. تسک‌هایی که زمان پردازش کمتری دارند (مقادیر کمتر vruntime) سمت چپ درخت و تسک‌هایی که زمان پردازش بیشتری دارند، سمت راست درخت قرار دارند. طبق تعریف درخت دودویی چپ‌ترین نود، کمترین مقدار را دارد که در اینجا به معنای تسکی است که بالاترین اولویت را دارد. چون این درخت متوازن است، برای کشف این نود، o(logN) عملیات لازم است (N تعداد نودهای درخت است)؛ با این وجود زمان‌بند لینوکس به دلایل کارآمد بودن، این مقدار را در متغیر rb_leftmost نگهداری می‌کند و تشخیص این که چه تسکی باید اجرا شود نیازمند این است که این مقدار بازیابی شود.

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

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

در سیستم‌عامل xv6 یک صف مشترک وجود دارد که در فایل proc.c که در روبه‌رو آمده است تعریف شده است.

همانطور که می‌بینیم در این استراکت یک صف از پردازه‌ها و یک lock به منظور مدیریت دسترسی‌های هم زمان تعریف شده است که در هنگام استفاده از آن lock را فعال می‌کنیم و بعد از آن باید آن را آزاد کنیم. برخلاف xv6، برای هر پردازنده یک صف مخصوص به آن وجود دارد. مزیت صف مشترک این است که در این صورت نیازی به مدیریت لود پردازنده‌ها نیست و به دلیل یکسان بودن صف load balancing وجود ندارد. نقص صف مشترک نسبت به صف مجزا این است که دسترسی‌هایی که در یک زمان به صف انجام می‌شوند باید بررسی شوند که برای اینکار از قفل کردن استفاده می‌کنیم.

4) در اجرای حلقه برای مدتی وقفه فعال می‌شود. علت چیست؟ آیا در سیستم‌های تک هسته‌ای به آن نیاز است؟

برای مثال اگر در حالتی همه پردازه‌ها متوقف شده باشند (مثلا برای عملیات IO یعنی منتظر گرفتن ورودی یا نمایش خروجی باشند)، وضعیت هیچ پردازه‌ای RUNNABLE نیست و اگر وقفه‌ای وجود نداشته باشد، عملیات IO تمام نمی‌شود و نیاز است تا برای صحت عملکرد وقفه ایجاد شود. با استدلال گفته شده این مورد برای سیستم‌های تک هسته‌ای هم نیاز است و در صورت عدم وجود، عملکرد پردازه‌ها را مختل می‌کند.

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؟

در لینوکس مدیریت وقفه‌ها در دو سطح FLIH (first level interrupt handler) و SLIH (second level interrup handler) انجام می‌شود. (نیمه بالایی و نیمه پایینی)؛ در سطح اول (FLIH)،مدیریت وقفه‌های ضروری انجام می‌شود؛ اینکار ممکن است با انجام دو عمل همراه شود: ذخیره اطلاعات ضروری وقفه و زمانبندی یک SLIH دیگر برای رسیدگی به آن یا رسیدگی کامل به وقفه. توسط FLIH اولیه. در ادامه‌ی پاسخ به وقفه‌ها در این سطح مدیریت، context switch انجام می‌شود و کد مدیریت کننده‌ی وقفه در آن run می‌شود. FLIH ممکن است در پردازه‌ها lag تولید کند یا وقفه‌ای را نادیده بگیرد. FLIH مدیریت وقفه‌ها را در کمترین زمان انجام می‌دهد. برخلاف FLIH، SLIH قسمت‌هایی از پردازش وقفه که نیاز به زمان بیشتری دارند را انجام می‌دهند. برای این سطح مدیریت دو حالت وجود دارد:1) مدیریت هر handler توسط یک thread pool انجام می‌شود. 2) به ازای هر handler در سطح kernel یک thread به خصوص وجود دارد. SLIHها معمولا schedule می‌شوند چون ممکن است اجرای آنها زمان زیادی بگیرد.

مدیریت وقفه‌ها درصورتی 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 حل ها برای حل این مشکل aging است. به این معنا که بصورت دوره ایpriority (timer intrupt) تسک هارا یک level افزایش دهیم، به اینصورت کم اولویت ترین تسک ها، پس از مدتی اولویتشان افزایش می یابد و به بالاترین سطح اولویت سیستم می‌رسند و اجرا می‌شوند. همچنین برای جلوگیری از اختصاص بیش از حد پردازنده به وقفه‌ها در بعضی از سیستم‌ها بدترین حالت نرخ به وجود آمدن وقفه‌ها را کاهش می‌دهند، به جای وقفه از سر زدن دوره‌ای برای چک کردن اتفاقات استفاده می‌کنند و با از مدیریت وقفه سطح1 و سطح2 استفاده می‌کنند که در بالا توضیح داده شد.

.

طراحی Aging در xv6:

برای طراحی این بخش، در فایل trap.c و در تابع trap زمانی که به تعداد ticks ها اضافه می‌کنیم، تابعی را فراخوانی می‌کنیم و در آن به ازای هر پردازه، مقدار تعداد تیک‌هایی که تا زمان آخرین اجرای آن پردازه را از تعداد تیک‌های کنونی کم می‌کنیم و اگر این مقدار از 8000 بزرگتر بود، صف این پردازه را به 1 تغییر می‌دهیم.

```
switch(tf->trapno){
case T_IRQ0 + IRQ_TIMER:
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;
    ////////////////////////////////
    check_aging(ticks);
    ////////////////////////////////
    wakeup(&ticks);
    release(&tickslock);
  }
```

```
void check_aging(int tick)
{
  struct proc* p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
      if (tick - p->last_cpu_use >= 8000)
          p->queue = 1;
}
```

Scheduler در xv6:

برای اینکار به ازای هربار اجرای حلقه، ابتدا پردازه‌هایی که در صف 1 هستند را اولویت می‌دهیم و در بین آنها طبق round_robin زمانبندی را انجام می‌دهیم؛ سپس اگر در صف 1، پردازه‌ای در وضعیت RUNNABLE نبود به سراغ صف2 می‌رویم و بین پردازه‌های RUNNABLE آن صف، زمانبند lottery را اجرا می‌کنیم؛ درصورتی که در این صف هم پردازه‌ی RUNNABLE ای وجود نداشت، به سراغ صف 3 می‌رویم و طبق زمانبند bjf بین پردازه‌ها زمانبندی می‌کنیم و پس از آن، پردازه را در mycpu ذخیره می‌کنیم و تعداد تیک‌ها را در قسمت last_cpu_use ذخیره می‌کنیم. همچنین در استراکت proc فایل proc.h نیز متغییر هایی مانند queue, ticket_chance, last_cpu_use, priority, arrival_time, exec_cycle, priority_ratio, arrival_tima_ratio را اضافه میکنیم.

```
void
scheduler(void)
{
  struct proc *p = 0;
  struct cpu *c = mycpu();
  int found_index = -1;
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();
    acquire(&ptable.lock);

    found_index = round_robin(found_index);
    if (found_index != -1)
        p = &ptable.proc[found_index];

    if (p == 0)
        p = lottery();

    if(p == 0)
       p = bjf();

      if (p == 0){
      release(&ptable.lock);
      continue;
    }
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;
    acquire(&tickslock);
    p -> last_cpu_use = ticks;
    release(&tickslock);
    swtch(&(c->scheduler), p->context);
    switchkvm();
    c->proc = 0;
    release(&ptable.lock);

  }
```

زمانبند نوبت گردشی: در این زمانبند به صورت چرخشی پردازه‌های موجود در proc.table را چک می‌کنیم (از اندیس بعد از آخرین پردازه‌ی اجرا شده در این زمانبند آغاز می‌کنیم) درصورتی که وضعیت پردازه‌ای RUNNABLE بود و در صف 1 قرار داشت، به نوبت از ابتدا، آنها را اجرا می‌کنیم.

```
int
round_robin(int last_used)
{
  int fround_proc_index = -1;
  int i;
  for(i = 1; i <= NPROC; i++)
  {
      int index = (i + last_used) % NPROC;
      if(ptable.proc[index].state != RUNNABLE || ptable.proc[index].queue != 1)
        continue;

      fround_proc_index = index;
      break;
  }
  return fround_proc_index;
}
```

زمانبند بخت‌آزمایی:

در این زمانبند ابتدا به جمع تعداد بلیت شانس‌های تمامی پردازه‌هایی که RUNNABLE هستند و در صف 2 قرار دارند را مجاسبه می‌کنیم و سپس یک عدد رندوم بین 1 تا عدد یافته شده، تولید می‌کنیم. حال مجددا در حلقه‌ای دیگر جمع تعداد پردازه‌هایی که در تکرار حلقه، RUNNABLE و در صف 2 هستند را محاسبه می‌کنیم، در اولین باری که این جمع کوچکتر یا مساوی عدد رندوم تولید شده بود، آن پردازه را انتخاب می‌کنیم.

```
struct proc*
lottery(void)
{
  struct proc *p;
  int count = 0;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if (p->state != RUNNABLE || p->queue != 2)
        continue;
      count += p->ticket_chance;
  }
  int rand = get_random_number(1, count);
  count = 0;
  struct proc* chosen_proc = 0;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

      if (p->state != RUNNABLE || p->queue != 2 )
        continue;

      count += p->ticket_chance;
      if (rand <= count)
      {
        chosen_proc = p;
        break;
      }
  }
  return chosen_proc;
}
```

زمانبند اول بهترین کار:

در این زمانبند به ازای هر پردازه‌ای که وضعیت آن RUNNABLE است و در صف 3 قرار دارد، رنک را مطابق با فرمول داده شده در دستور گزارش، محاسبه می‌کنیم و پردازه‌ای که کمترین رنک را دارد، اجرا می‌کنیم.

```
struct proc*
bjf(void)
{
  struct proc* p;
  struct proc* first_rank = 0;
  float min_rank = 2e6;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if (p->state != RUNNABLE || p->queue != 3)
      continue;
    float rank = get_rank(p);
    if (rank< min_rank){
      first_rank = p;
      min_rank = rank;
    }
  }

  return first_rank;
}
```

فراخوانی‌های سیستمی:

```
void                    int queue
change_queue(int pid, int queue)
{
  struct proc *p;
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid == pid)
    {
      p->queue = queue;
      cprintf("Process %d is now in queue %d\n", pid, queue);
      break;
    }
  }
  release(&ptable.lock);
}
```

تغییر صف پردازه: طبق دستورالعمل اضافه کردن سیستم کال (تغییر در فایل‌های proc.c, sysproc.c ,syscall.c ,syscall.h defs.h, usys.s, user.h) انجام شد و برنامه سطح کاربر (change_queue.c) را نوشته و آن را در قسمت UPROGS در میک فایل اضافه کردیم.

```
$ change_queue 4 3
Process 4 is now in queue 3
```

مقداردهی بلیت بخت آزمایی: در این سیستم کال نیز تمامی مراحل فراخوانی سیستمی قبلی انجام شده است.

```
void
set_ticket_chance(int pid, int ticket_chance){
  struct proc *p;
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid == pid)
    {
        p->ticket_chance = ticket_chance;
        cprintf("Process %d has chance of:  %d\n", pid, ticket_chance);
    }
  }
  release(&ptable.lock);
}
```

```
$ set_ticket_chance 5 7
Process 5 has chance of:  7
```

مقدار دهی پارامتر BJF در سطح پردازه و در سطح سیستم:

```
void
set_bjf_s(int pr, int a_t_r, int e_c_r)
{
  struct proc *p;
  priority_ratio = pr;
  arrival_time_ratio = a_t_r;
  executed_cycle_ratio = e_c_r;
  acquire(&ptable.lock);
  cprintf("Bjf params set for all processes\n");
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    p->priority_ratio = priority_ratio;
    p->arrival_time_ratio = arrival_time_ratio;
    p->executed_cycle_ratio = executed_cycle_ratio;
  }
  release(&ptable.lock);
}
```

```
void
set_bjf_u(int pid, int priority_ratio, int arrival_time_ratio, int executed_cycle_ratio)
{
  struct proc *p;

  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid == pid)
    {
      p->priority_ratio = priority_ratio;
      p->arrival_time_ratio = arrival_time_ratio;
      p->executed_cycle_ratio = executed_cycle_ratio;
      cprintf("BJF params set for pid: %d\n", p->pid);
      break;
    }
  }
  release(&ptable.lock);
}
```

```
set_bjf_u 4 2 3 5
BJF params set for pid: 4
$ set_bjf_s 7 6 2
Bjf params set for all processes
```

چاپ اطلاعات:

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

    cprintf("Name: %s    ", p->name);

    cprintf("Pid: %d    ", p->pid);

    cprintf("State: %s    ", get_state_name(p->state));

    cprintf("Queue: %d    ", p->queue);

    cprintf("Ticket_Chance: %d    ", p->ticket_chance);

    cprintf("Priority_Ratio: %d    ", p->priority_ratio);

    cprintf("Arrival_time_Ratio: %d    ", p->arrival_time_ratio);

    cprintf("Executed_Cycle_Ratio: %d    ", p->executed_cycle_ratio);

    cprintf("Arrival_time: %d    ", p->arrival_time);

    cprintf("Exec_cycle * 10: %d    ", p->exec_cycle * 10);

    cprintf("-------------------------------------------------------------------------------\n");
  }
  release(&ptable.lock);
```

برنامه foo:

این برنامه چندین fork انجام می‌دهد و سپس برای عملیات محاسباتی در هر بار یک واحد به عدد temp اضافه می‌کند و به اندازه پایان آنها wait می‌کنیم.

```
int main(int argc, char *argv[])
{
    int pid[12];
    for (int i = 0 ; i < 6 ; i++)
    {
        pid[i] = fork();
        if (pid[i] == 0)
        {
            long long temp = 1;
            for (int j = 0 ; j < 4000000000 ; j++)
            {
                temp += 1;
            }
        }
    }
    while (wait());
    return 0;
}
```

چاپ اطلاعات پس از اجرای foo:

```
$ foo&
$ print_processes
Name: init    Pid: 1    State: SLEEPING    Queue: 1    Ticket_Chance: 2    Priority_Ratio: 0
  Arrival_time_Ratio: 0    Executed_Cycle_Ratio: 0    Arrival_time: 0    Exec_cycle * 10: 2
 rank: 0    ------------------------------------------------------------------------------

Name: sh    Pid: 2    State: SLEEPING    Queue: 2    Ticket_Chance: 22    Priority_Ratio: 0
 Arrival_time_Ratio: 0    Executed_Cycle_Ratio: 0    Arrival_time: 0    Exec_cycle * 10: 2
rank: 1    ------------------------------------------------------------------------------

Name: foo    Pid: 5    State: RUNNABLE    Queue: 2    Ticket_Chance: 22    Priority_Ratio: 0
  Arrival_time_Ratio: 0    Executed_Cycle_Ratio: 0    Arrival_time: 0    Exec_cycle * 10: 33
  rank: 35    ------------------------------------------------------------------------------


Name: foo    Pid: 4    State: SLEEPING    Queue: 2    Ticket_Chance: 22    Priority_Ratio: 0
  Arrival_time_Ratio: 0    Executed_Cycle_Ratio: 0    Arrival_time: 0    Exec_cycle * 10: 1
 rank: 31    ------------------------------------------------------------------------------

Name: foo    Pid: 6    State: RUNNABLE    Queue: 2    Ticket_Chance: 7    Priority_Ratio: 0
 Arrival_time_Ratio: 0    Executed_Cycle_Ratio: 0    Arrival_time: 0    Exec_cycle * 10: 5
rank: 32    ------------------------------------------------------------------------------

Name: foo    Pid: 7    State: RUNNABLE    Queue: 2    Ticket_Chance: 7    Priority_Ratio: 0
 Arrival_time_Ratio: 0    Executed_Cycle_Ratio: 0    Arrival_time: 0    Exec_cycle * 10: 5
rank: 32    ------------------------------------------------------------------------------

Name: foo    Pid: 8    State: RUNNABLE    Queue: 2    Ticket_Chance: 7    Priority_Ratio: 0
 Arrival_time_Ratio: 0    Executed_Cycle_Ratio: 0    Arrival_time: 0    Exec_cycle * 10: 1
```