آزمایشگاه سیستم عامل

گزارش کار پروژه سوم

اعضای گروه:

امیرحسین عارف زاده 810101604

مهدی نایینی 810101536

کیارش خراسانی 810101413

Repository : https://github.com/Amir-rfz/OS-Lab

Latest Commit : c824ba8c99d8f98b3c5270a072dd78c4a5dcb5b8

شرح پروژه

1. ساختار PCB و همچنین وضعیت‌های تعریف شده برای هر پردازه را در xv6 پیدا کرده و گزارش کنید. آیا شباهتی میان داده‌های موجود در ان ساختار و ساختار به تصویر کشیده شده در شکل 3.3 منبع درس وجود دارد؟ (ذکر حداقل ۵ مورد و معادل آن‌ها در xv6)

```
37   enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

همانطور که در تصویر بالا قابل مشاهده است۶ وضعیت برای پردازه ها در xv6 وجود دارد.
.UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE
UNUSED: فرآیند استفاده نمی‌شود و آماده تخصیص است.
EMBRYO: فرآیند در حال ایجاد شدن و در مراحل ابتدایی تخصیص است.
SLEEPING: فرآیند در حالت خواب است و منتظر وقوع یک رویداد است.
RUNNABLE: فرآیند آماده اجرا است و منتظر تخصیص CPU است.
RUNNING: فرآیند در حال اجرا توسط CPU است.
ZOMBIE: فرآیند اجرای خود را به پایان رسانده اما هنوز در جدول فرآیندها باقی مانده تا وضعیت خروج آن خوانده شود.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  struct syscall_info syscall_data[MAX_SYSCALLS];
  int syscall_counts;
};
```

unit sz: سایز پردازه ایجاد شده به byte.

pgdir *pde_t: یک پونتر به دایرکتوری page است که کاربرد آن برای کنترل حافظه مجازی پردازه است.

char *kstack: هر پردازه در کرنل استک خود را دارد که هنگام اجرا در سطح کرنل استفاده میشود و این پونتری به انتهای آن است.

enum procstate state: به حالت در حال حاضر برنامه اشاره میکند و در بالا اشاره شد چه حالاتی میتواند داشته باشد.

int pid: همان id پردازه که عددی یکتا برای شناسایی پردازه است.

struct proc *parent: به پردازه پدر اشاره میکند (به طور مثال در هنگام استفاده از fork()) این فیلد به کنترل روابط پردازه کمک میکند.

struct trapframe *tf: به trap frame پردازه برای system call کنونی اشاره میکند و درواقع دارای حالت هسته در زمان اجرای تله است.

struct context *context: به اطلاعات پردازه اشاره میکند که هنگام جابه جایی پردازه استفاده میشود. شامل اطلاعات رجیستر ها و اطلاعات حالت برنامه که لازم برای contex switch است میشود.

void *chan: اگر صفر نباشد یعنی پردازه در حالت خواب است و در این چنل منتظر است.

int killed: اگر صفر نباشد یعنی پردازه kill شده است و باید خارج شود.

struct file *ofile[NOFILE]: آرایه ای از پوینتر ها است که به فایل های باز شده توسط این پردازه اشاره میکند. که حداکثر به تعداد NOFILE میتواند فایل باز کند.

struct inode *cwd: یک پوینتر به دایرکتوری حال حاضر پردازه است که در حال اجرا در آن است.

char name[16]: است پردازه است که برای دیباگ کردن ساخته شده است.

struct syscall_info syscall_data[MAX_SYSCALLS]: یه آرایه از اطلاعات مربوط به systemcall های درست شده توسط پردازه است که هر کدام شامل تعداد دفعات صدا شدن است و اسم syscall.
int syscall_counts: تعداد کل systemcall های ایجاد شده توسط پردازه.
شباهات:
در منبع کتاب نیز process state که توضیح داده شد وجود دارد. -> enum procstate state
Process number که در کتاب است همان process id است که اشاره شد. ->int pid
Registers و program counter که در منبع کتاب وجود دارد همان اطلاعات داخل رجیستر ها است که توضیح داده شد در اینجا هم وجود دارد. ->struct context *context
Memory limits: همان شروع و پایان استک ها و محدودیت های حافظه برنامه است که در اینجا نیز اشاره شد. ->char *kstack و ->unit sz
List of open files همانطور که اشاره شد در struct file *ofile[NOFILE] فایل های باز شده توسط برنامه وجود دارد که در اینجا هم همین است. ->struct file *ofile[NOFILE]

2. هر کدام از وضعیت‌های تعریف شده معادل کدام وضعیت در شکل ۱ می‌باشند؟

ZOMBIE را میتوان معادل terminated دانست زیرا در آن اجرا تمام شده و برنامه منتظر دیده شدن توسط syscall wait است.
RUNNING را میتوان معادل running در شکل دانست.
RUNNABLE را میتوان معادل ready دانست.
UNUSED را میتوان معادل new دانست.
SLEEPING را میتوان معادل waiting دانست.
EMBRYO در واقع حالت میانی بین ready و new است زیرا پردازه ایجاد شده ولی منابع لازم به آن تخصیص پیدا نکرده است.
RUNNABLE را نیز میتوان معادل ready از جهتی دیگر داست.

تغییر وضعیت در xv6

Admitted

3. با توجه به توضیحات گفته شده، کدام یک از توابع موجود در proc.c منجر به انجام گذار از حالت new به حالت ready که در شکل ۱ به تصور کشیده شده، خواهد شد؟ وضعیت یک پردازه در xv6 در این گذار از چه حالتی/حالت‌های به چه حالت/حالت‌های تغییر می‌کند؟ پاسخ خود را با پاسخ سوال ۲ مقایسه کنید.

```
69  //PAGEBREAK: 32
70  // Look in the process table for an UNUSED proc.
71  // If found, change state to EMBRYO and initialize
72  // state required to run in the kernel.
73  // Otherwise return 0.
74  static struct proc*
75  allocproc(void)
76  {
77    struct proc *p;
78    char *sp;
79
80    acquire(&ptable.lock);
81
82    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
83      if(p->state == UNUSED)
84        goto found;
85
86    release(&ptable.lock);
87    return 0;
88
89  found:
90    p->state = EMBRYO;
91    p->pid = nextpid++;
92
93    release(&ptable.lock);
94
95    // Allocate kernel stack.
```

```
96    if((p->kstack = kalloc()) == 0){
97      p->state = UNUSED;
98      return 0;
99    }
100   sp = p->kstack + KSTACKSIZE;
101
102   // Leave room for trap frame.
103   sp -= sizeof *p->tf;
104   p->tf = (struct trapframe*)sp;
105
106   // Set up new context to start executing at forkret,
107   // which returns to trapret.
108   sp -= 4;
109   *(uint*)sp = (uint)trapret;
110
111   sp -= sizeof *p->context;
112   p->context = (struct context*)sp;
113   memset(p->context, 0, sizeof *p->context);
114   p->context->eip = (uint)forkret;
115
116   // Initialize syscall_data to zero
117   for (int i = 0; i < MAX_SYSCALLS; i++) {
118     p->syscall_data[i].number = MAX_SYSCALLS + 1;
119     p->syscall_data[i].count = 0;
120   }
121   p->syscall_counts = 0;
```

همانطور که قابل مشاهده است در تابع allocproc توابعی که در حالت UNSUED هستند که همان معادل حالت new هستند(همانطور که در سوال ۲ اشاره کردیم.) از جدول پردازه های UNUSED پیدا میشوند و به حالت EMBRYO تغییر وضعیت میدهند و سپس وارد تابع fork میشود و به حالت RUNABLE تغییر وضعیت میدهد و initialize میشوند(که معادل ready است). این وضعیت وضعیتی است که یک پردازه نیاز دارد تا قابلیت اجرا شدن در کرنل را داشته باشد.

البته تغییر وضعیت از EMBRYO به RUNABLE در تابع userinit نیز رخ میدهد.

یافته ها پاسخ سوال ۲ را تایید میکنند.

Scheduler Dispatch

4. سقف تعداد پردازه‌های ممکن در xv6 چه عددی است؟ در صورتی که یک پردازه تعداد زیادی پردازه فرزند ایجاد کند و از این سقف عبور کند، کرنل چه واکنشی نشان داده و برنامه سطح کاربر چه بازخوردی دریافت می‌کند؟

سقف پردازه هایی که میتوان ایجاد کرد برای یک پدر برابر با NPROC است که در فایل params.h تعریف شده و مقدار ۶۴ دارد پس حداکثر ۶۴ فرزند میتوان ایجاد کرد.
در صورتی که بیش از آن ایجاد کنیم با توجه به کدی که از allocproc در سوال قبل آپلود شد در خط ۸۲ به داخل حلقه for نمیرویم و 0 return دیده میشود که به معنی اجرای ناموفقیت برنامه بوده است.
این status توسط fork در خطی با کد زیر:

```
if((np = allcoproc()) == 0){
        return -1;
}
```

این کد خطای allocproc را میبیند و fork ناموفق خواهد بود و -1 status به برنامه سطح کاربر برگردانده میشود که نشان دهنده اجرای ناموفق است و کاربر میتواند با آن تصمیم گیری کند.

5. چرا نیاز است در ابتدای هر حلقه تابع scheduler، جدول پردازه‌ها قفل شود؟ آیا در سیستم‌های تک پردازه‌ای هم نیاز است این کار صورت بگیرد؟

به طور کلی سه دلیل وجود دارد:
یکپارچگی: وقتی چند cpu یا core میتوانند به ptable دسترسی پیدا کنند قفل کردن ptable باعث میشود که در آن واحد تنها یک cpu میتواند ptable را تغییر دهد.
جلوگیری از race conditions: وقتی ptable قفل نشود چند cpu میتوانند در آن واحد یک پردازه را انتخاب کنند و این منجر به رفتار های تعریف نشده مانند crash های غیر منتظره میشود. و قفل کردن باعث میشود هر کدام به ترتیب بتوانند به آن دسترسی پیدا کنند.
Atomic operations: همانطور که در کلاس توضیح داده شد بعضی از پردازه ها atomic هستند و تخصیص یا آزاد سازی پردازه ها که این قفل ها باعث میشوند که این کار ها بدون دخالت انجام گیرند.
در سیستم های تک پردازه دسترسی ها به صورت همزمان نیست و به همین دلیل میتوان مطمئن بود پردازنده دیگری در کار دخالت نمیکند و ptable را تغییر نمیدهد یا باعث اینجاد race condition نمیشود.
به همین دلیل در سیستم های تک پردازه نیازی به lock کردن نیست.

6. با فرض اینکه xv6 در حالت تک هسته‌ای در حال اجراست، اگر یک پردازه به حالت RUNNABLE برود و صف پردازه‌ها در حال طی شدن باشد (proc:335)، در مکانیزم زمان‌بندی xv6 نسبت به موقعیت پردازه در صف، در چه iteration ای امکان schedule پیدا می‌کند؟ (در همان iteration یا در iteration بعدی)

بستگی دارد در چه زمانی پردازه به حالت RUNNABLE تغییر وضعیت بدهد.
اگر قبل از اینکه scheduler اطلاعات آن پردازه را در ptable چک کند تغییر وضعیت داده شود، در همان iteration دیده میشود و در غیر این صورت در iteraion بعدی.
مثال زیر را در نظر بگیرید:
اگر پردازه های p1,p2,p3,p4,p5,...pn را داشته باشیم و scheduler در حال schedule کردن p2 باشد و حالت p4 به RUNNABLE تغییر کند در همان iteration این پردازه schedule میشود زیرا وقتی schduler اطلاعات آن را برسی میکند حالت آن RUNNABLE است.
اما اگر scheduler در حال scheduleکردن پردازه p7 باشد و اطلاعات p4 را قبلا چک کرده باشیم. Scheduler در این iteration تغییر وضعیت p4 را نخواهد دید و در iteraion بعدی schedule میشود.

Context Switch

7. رجیسترهای موجود در ساختار context را نام ببرید.

edi: EXtended destination index register**:** معمولا توسط عملیات های آرایه ها یا رشته ها به عنوان پوینتری به مقصد استفاده میشود.
esi: Extended Source Index register: معمولا توسط عملیات های آرایه ها یا رشته ها به عنوان پوینتری به مبدا استفاده میشود.
ebx: Extended Base register: یه رجیستر است که برای کار محاسبات ریاضی و آدرس دهی استفاده میشود.
ebp: Extended Base pointer register: معمولا اشاره گری به شروع frame پشته است که به دسترسی به پارامتر های توابع و مقدایر محلی استفاده میشود.
eip: Extended Instruction Pointer register: آدرس دستور بعدی که باید اجرا شود را نگه میدارد و به همین دلیل برای کنترل روند اجرای برنامه لازم است.

8. همانطور که می‌دانید یکی از مهم‌ترین رجیسترها قبل از هر تعویض متن program counter است که نشان می‌دهد روند اجرای برنامه تا کجا پیش رفته است. با ذخیره‌سازی این رجیستر می‌توان محل ادامهٔ برنامه را بازیابی کرد. این رجیستر در ساختار context چه نام دارد؟
این رجیستر چگونه قبل از انجام تعویض متن ذخیره می‌شود؟
همانطور که در سوال قبل اشاره شد نام آن eip است.

وقتی تابع swtch صدا زده میشود context در حال حاضر که شامل return address(یا همان eip) است را روی stack ذخیره میکند. بعد از جابه جایی پوینتر ها به stack برای context جدید تابع ret صدا زده میشود و این تابع دستور بعدی را از روی stack از داخل eip میخواند و به همین ترتیب eip جدید خوانده شده و به ادامه دستورات پردازه جدید اشاره میکند.
بنابراین به طور کل با تابع ret آن را آپدیت میکنیم.

Interrupt

9. همانطور که در قسمت قبل مشاهده کردید، ابتدای ابع scheduler، ایجاد وقفه به کمک ابع sti، فعال می‌شود. با توجه به توضیحات این قسمت، اگر وقفه‌ها فعال نمی‌شد چه مشکلی به وجود می‌آمد؟

میتواند مانع preemption شود. از آنجا که در xv6 سیستم RR برقرار است اگر interupt ها فعال نشود پردازه ها دیگر اگر بیش از تایم مجاز خود استفاده کنند متوقف نمیشوند و کنترل cpu را به طور کامل به دست میگیرند و time-sharing انجام نمیشود.
چون که interupt ها فعال نیستند interupt های timer دیده نمیشود که باعث میشود context switch انجام نشود و از طرفی clock سیستم از بین میرود و از طرفی چون scheduler صدا نمیشود و برنامه عوض نمیشود سیستم قفل میکند.
از طرفی حتی interrupt های سخت افزار مثل I/O از دست میرود و ممکن است باعث از دست داده داده یا کارایی شود.
بعضی از سرویس های سیستمی که بر interrupt ها سوار هستند درست کار نمیکنند.
خیلی از systemcall ها با استفاده از interrupt ها کار میکنند که اگر آنها غیر فعال باشند باعث عملکرد نادرست یا کار نکردن systemcall ها میشود.

10. بنظر شما وقفهٔ تایمر هر چه مدت یک بار صادر می‌شود؟ (راهنمای: می‌توانید با اضافه کردن یک cprintf پس از *ticks++* این موضوع را مشاهده کنید.)
در xv6 تنظیم شده که timer interrupt ها تقریبا هر ۱۰ میلی ثانیه یک بار فرستاده شوند .
از داکیومنت آن میتوان استفاده کرد و به آن استناد کرد.

> Let's look at the timer device and timer interrupts. We would like the timer hardware to generate an interrupt, say, 100 times per second so that the kernel can track the passage of time and so the kernel can time-slice among multiple running processes. The choice of 100 times per second allows for decent interactive performance while not swamping the processor with handling interrupts.

11. با توجه به توضیحات داده شده، چه تابعی منجر به انجام شدن گذار interrupt در شکل 1 خواهد شد؟

تابع yield مسئولیت انجام این کار را دارد
با تغییر وضعیت پردازه از running به runnable این اجازه را میدهد که scheduler دوباره پردازه را انتخاب کند.
این کار باعث میشود که زمان cpu به طور عادلانه بین پردازه ها تقسیم شود.

12. با توجه به توضیحات قسمت scheduler dispatch می‌دانیم زمان‌بندی در xv6 به شکل نوبت گردشی است. حال با توجه به مشاهدات خود در این قسمت، استدلال کنید، کوانتوم زمانی این پیاده‌سازی از زمان‌بندی نوبت گردشی چند میلی ثانیه است؟

اگر در فایل trap.c را نگاه کنیم خواهیم دید که در هر تیک یک بار تابع yield صدا میشود که باعث میشود پردازه عوض شود. از آنجا که مقدار تیک در فایل lapic.c برابر با ۱۰ میلی ثانیه تعریف شده است پس در هر 10 میلی ثانیه پردازه عوض میشود و عملیات context switch انجام میشود.

```
69      lapicw(TICR, 10000000);
```

Wait

13. تابع wait در نهایت از چه تابعی برای منتظر ماندن برای اتمام کار یک پردازه استفاده می‌کند؟

در نهایت از تابع sleep() استفاده میشود که تا وقتی که روی کانال curproc هیچ event ای اتفاق نیفتد منتظر میماند. در نهایت توابع wakeup1 یا wakeup صدا میشوند (به وسیله proc_exit به طور معمول) که به معنی این است(در واقع برای termination یک پردازه فرزند) توسط sleep دیده میشود و دوباره وارد حلقه شده و چک میکند آیا فرزندی در حالت zmoblie قرار دارد یا نه اگر قرار داشت آن را حذف میکند.

```
315     }
316
317     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
318     sleep(curproc, &ptable.lock);  //DOC: wait-sleep
319   }
320 }
```

14. با توجه به پاسخ سوال قبل، استفاده(های) دیگر این تابع چیست؟(ذکر یک نمونه)

از تابع sleep استفاده های متنوعی میتوان کرد به طور مثال میتوان منتظر رخداد های خاص به طور مثال I/O یا منتظر باز شدن فایل ها یا منتظر وجود داشتن دیتا روی یک socket در یک شبکه بود.
از طرفی میتوان برای استفاده بهینه تر از پردازنده از آن استفاده کرد. زیرا پردازنده به جای اینکه به طور کامل در حلقه for گردش کند و کار انجام دهد، تنها زمانی cpu کار میکند که تابع sleep بیدار شود utilization بالاتر میرود.
از طرفی میتوان همانند lock ها و semaphore ها از آن برای ایجاد قفل استفاده کرد.

15. با این تفاسیر، چه تابعی در سطح کرنل، منجر به آگاه سازی پردازه از رویدادی است که برای آن منتظر بوده است؟

تابع wakeup ابتدا صدا میشود و ptable را قفل میکند. سپس wakeput1 را صدا میکند که باعث میشود پردازه از حالت SLEEPING به حالت RUNNABLE تغییر کند و بعد از این تابع scheduler میتواند پردازه را دوباره انتخاب کند و در iteration پردازنده را به آن اختصاص دهد. در انتها تابع wakeup قفل ptable باز میشود.

16. با توجه به پاسخ سوال 9، این تابع منجر به گذار از چه وضعیتی به چه وضعیتی در شکل 1 خواهد شد؟

همانطور که گفته شد پردازه از حالت SLEEPING به حالت RUNNABLE تغییر میکند که معادل آن در شکل این است که از حالت waitingبه حالت ready برویم که در transition I/O or event completion اتفاق می افتد.

17. آیا تابع دیگری وجود دارد که منجر به انجام این گذار شود؟ نام ببرید.
یک سری از interrupt handler ها و تابع proc_exit میتواند تابع wakeup یا wakeup1 را صدا کنند که منجر به این گذار میشود.
از طرف دیگر تابع kill نیز توانایی انجام این کار را دارد.
همانطور که در تصویر زیر قابل مشاهده است تابع kill بعد از اینکه پردازه را میکشد چک میکند آیا حالت آن SLEEPING است و اگر بود آن را به حالت RUNNABLE تغییر میدهد.

```
487  int
488  kill(int pid)
489  {
490    struct proc *p;
491
492    acquire(&ptable.lock);
493    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
494      if(p->pid == pid){
495        p->killed = 1;
496        // Wake process from sleep if necessary.
497        if(p->state == SLEEPING)
498          p->state = RUNNABLE;
499        release(&ptable.lock);
500        return 0;
501      }
502    }
503    release(&ptable.lock);
504    return -1;
505  }
506
```

Exit

18. در بخش 3.3.2 منبع درس با پردازه های orphan آشنا شدید، رویکرد xv6 در رابطه با این گونه پردازه ها چیست؟

```
232  // Exit the current process.  Does not return.
233  // An exited process remains in the zombie state
234  // until its parent calls wait() to find out it exited.
235  void
236  exit(void)
237  {
238    struct proc *curproc = myproc();
239    struct proc *p;
240    int fd;
241
242    if(curproc == initproc)
243      panic("init exiting");
244
245    // Close all open files.
246    for(fd = 0; fd < NOFILE; fd++){
247      if(curproc->ofile[fd]){
248        fileclose(curproc->ofile[fd]);
249        curproc->ofile[fd] = 0;
250      }
251    }
252
253    begin_op();
254    iput(curproc->cwd);
255    end_op();
256    curproc->cwd = 0;
257
258    acquire(&ptable.lock);
259
```

```
260     // Parent might be sleeping in wait().
261     wakeup1(curproc->parent);
262
263     // Pass abandoned children to init.
264     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
265       if(p->parent == curproc){
266         p->parent = initproc;
267         if(p->state == ZOMBIE)
268           wakeup1(initproc);
269       }
270     }
271
272     // Jump into the scheduler, never to return.
273     curproc->state = ZOMBIE;
274     sched();
275     panic("zombie exit");
276   }
```

همانطور که در عکس قابل مشاهده است هر پردازه قبل از خروج ابتدا پدر خود را بیدار میکند و سپس فرزندان خود را به پردازه اصلی که init است میدهد. به همین ترتیب پردازه های orphan به init نسبت داده میشوند و init پدر آنها میشود.

زمانبندی بازخوردی چند سطحی:

سطح اول: زمان‌بند نوبت گردشی با کوانتوم زمانی

```
struct proc *
round_robin(struct proc *last_scheduled)
{
  struct proc *p = last_scheduled;
  for (;;)
  {
    p++;
    if (p >= &ptable.proc[NPROC])
      p = ptable.proc;

    if (p->state == RUNNABLE && p->sched_info.queue == ROUND_ROBIN)
      return p;

    if (p == last_scheduled)
      return 0;
  }
}
```

```
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER) {
  myproc()->consecutive_time += 1;
  myproc()->sched_info.last_run = ticks;
  struct cpu *c = mycpu();
  int time_periode = (c->cpu_ticks % 60) + 1;
  c->cpu_ticks++;
  if((myproc()->sched_info.queue == ROUND_ROBIN && myproc()->consecutive_time >= 5) ||
    (time_periode == 30 || time_periode == 50 || time_periode == 60)) {
    // cprintf("tick = %d cpu_get_time = %d process pid = %d queue = %s\n",ticks, myproc()->sc
    yield();
  }
}
```

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

#define PROCESSS_NUM 5

int main()
{
    for (int i = 0; i < PROCESSS_NUM; i++)
    {
        int pid = fork();
        if (pid == 0) {
            for (int j = 0; j < 1000; j++) {
                for (int k = 0; k < 1000; k++) {
                    getpid();
                }
            }
            exit();
        }
    }
    for (int i = 0; i < PROCESSS_NUM; i++)
        wait();
    exit();
}
```

نتیجه:

```
$ schedule info
name            pid   state      queue   wait_time   confidence   burst_time   consecutive_run   Arrival
-------------------------------------------------------------------------------------------------------
init            1     sleeping   1       0           50           2            2                 0
sh              2     sleeping   1       0           50           2            1                 9
schedule        3     running    1       0           50           2            3                 872
$ schedule_test&
$ schedule info
name            pid   state      queue   wait_time   confidence   burst_time   consecutive_run   Arrival
-------------------------------------------------------------------------------------------------------
init            1     sleeping   1       0           50           2            2                 0
sh              2     sleeping   1       0           50           2            0                 9
schedule_test   6     runnable   1       10          50           2            5                 1593
schedule_test   5     sleeping   1       0           50           2            4                 1589
schedule_test   7     runnable   1       7           50           2            5                 1593
schedule_test   8     runnable   1       10          50           2            5                 1595
schedule_test   9     runnable   1       5           50           2            5                 1595
schedule_test   10    runnable   1       1           50           2            5                 1596
schedule        11    running    1       0           50           2            3                 1912
$
```

19. مقدار cpu را مجددا به عدد 2 برگردانید. آیا همچنان ترتیبی که قبلا مشاهده می کردید پا برجاست؟ علت این امر چیست؟

اگر مقدار CPU را دوباره به 2 برگردانیم، ترتیب اجرای فرآیندها ممکن است تغییر کند. این به دلیل تفاوت در رفتار چند پردازشی (Multithreading) است. در حالت تک‌پردازنده (CPUS=1)، تنها یک CPU در حال اجرای فرآیندها است و برنامه‌ریزی (Scheduling) به صورت ترتیبی و کنترل‌شده توسط Scheduler انجام می‌شود. وقتی مقدار CPUS به 2 افزایش می‌یابد، دو CPU می‌توانند همزمان فرآیندها را اجرا کنند. به دلیل رفتار غیرقطعی (Nondeterministic) در برنامه‌ریزی و رقابت بین CPUها، ممکن است فرآیندها به شکل متفاوتی زمان‌بندی شوند. از دلایل این اتفاق می توان به رقابت بر سر منابع مشترک (مانند قفل‌ها یا حافظه مشترک) و تاخیرهای ناشی از هماهنگی بین CPUها اشاره کرد.

در تصویر زیر ما مقدار CPUS را برابر با 4 قرار دادیم:

```
mahdi  main -  make qemu CC=gcc-11 CPUS=4
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,
 -m 512
xv6...
cpu1: starting 1
cpu2: starting 2
cpu3: starting 3
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
Group members:
1. Amirhossein Arefzadeh
2. Mahdi Naieni
3. kiarash Khorasani
$ schedule info
name            pid   state      queue   wait_time   confidence   burst_time   consecutive_run   Arrival
-------------------------------------------------------------------------------------------------------
init            1     sleeping   1       0           50           2            0                 0
sh              2     sleeping   1       0           50           2            0                 64
schedule        3     running    1       0           50           2            1                 508
$ schedule_test&
$ schedule info
name            pid   state      queue   wait_time   confidence   burst_time   consecutive_run   Arrival
-------------------------------------------------------------------------------------------------------
init            1     sleeping   1       0           50           2            0                 0
sh              2     sleeping   1       0           50           2            0                 64
schedule_test   6     running    1       0           50           2            4                 1366
schedule_test   5     sleeping   1       0           50           2            3                 1358
schedule_test   7     runnable   1       2           50           2            5                 1367
schedule_test   8     runnable   1       2           50           2            5                 1369
schedule_test   9     running    1       0           50           2            0                 1371
schedule_test   10    running    1       0           50           2            1                 1372
schedule        11    running    1       0           50           2            2                 1609
$
```

سطح دوم: اول(،) کوتاه‌ترین کار

```
static unsigned int seed = 1;

void srand(unsigned int s) {
  seed = s;
}

int rand(void) {
  seed = (1103515245 * seed + 12345) & 0x7fffffff;
  return seed;
}
```

```
struct proc * shortest_job_first(void){
  struct proc *p;
  struct proc *sjf_process[NPROC];
  int count = 0;

  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if (p->state == RUNNABLE && p->sched_info.queue == SJF){
      sjf_process[count++] = p;
    }
  }

  if (count == 0)
    return 0;

  for (int i = 0; i < count - 1; i++){
    for (int j = i + 1; j < count; j++){
      if (sjf_process[i]->sched_info.sjf.BurstTime > sjf_process[j]->sched_info.sjf.BurstTime){
        struct proc *temp = sjf_process[i];
        sjf_process[i] = sjf_process[j];
        sjf_process[j] = temp;
      }
    }
  }

  for (int i = 0; i < count; i++){
    srand(i+1);
    int rand_num = rand() % 100;
    if (rand_num < sjf_process[i]->sched_info.sjf.Confidence){
      return sjf_process[i];
    }
  }

  return sjf_process[count - 1];
}
```

```
p->sched_info.queue = UNSET;
p->sched_info.get_cpu_time = ticks;
p->sched_info.sjf.arrival_time = ticks;
p->sched_info.sjf.Confidence = 50;
p->sched_info.sjf.BurstTime = 2;
p->consecutive_time= 0;
```

سطح سوم: اولین ورود-اولین رسیدگی

```
struct proc * first_come_first_serve(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 != FCFS)
      continue;
    if (result != 0){
      if (result->sched_info.arrival_queue_time > p->sched_info.arrival_queue_time)
        result = p;
    }
    else
      result = p;
  }
  return result;
}
```

برش‌دهی زمانی :

```
void scheduler(void){
  struct proc *p;
  struct proc *last_scheduled_RR = &ptable.proc[NPROC - 1];
  struct cpu *c = mycpu();
  c->proc = 0;

  for (;;){
    sti();
    acquire(&ptable.lock);

    int time_period = (c->cpu_ticks % 60) + 1;
    if (time_period <= 30){
      p = round_robin(last_scheduled_RR);
      if(p)
        last_scheduled_RR = p;
      else
        time_period = 31;
    }

    if (time_period >= 31 && time_period <= 50){
      p = shortest_job_first();
      if (!p)
        time_period = 51;
    }

    if (time_period >= 51){
      p = first_come_first_serve();
      if (!p){
        c->cpu_ticks = 0;
        release(&ptable.lock);
        continue;
      }
    }
    c->cpu_ticks = time_period - 1;
    if (c->proc)
      c->proc->sched_info.last_run = ticks;
    c->proc = p;
    switchuvm(p);

    p->state = RUNNING;
    p->sched_info.get_cpu_time = ticks;

    p->consecutive_time= 0;

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

    switchkvm();

    c->proc = 0;
    release(&ptable.lock);
  }
}
```

```
struct cpu {
  uchar apicid;
  struct context *scheduler;
  struct taskstate ts;
  struct segdesc gdt[NSEGS];
  volatile uint started;
  int ncli;
  int intena;
  struct proc *proc;
  int cpu_ticks;
};
```

20. در صورت نیاز به مقدار دهی اولیه به فیلد های اضافه شده در ساختار cpu، در چه تابعی از xv6 بهنر است این کار انجام گیرد؟

ما متغیر cpu_ticks را در struct cpu تعریف می کنیم.

```
// Per-CPU state
struct cpu {
  uchar apicid;                // Local APIC ID
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint started;       // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?
  struct proc *proc;           // The process running on this cpu or null
  int cpu_ticks;
};
```

بهتر است مقدار دهی اولیه را در تابع static void mpmain(void) یا static void startothers(void) انجام دهیم.

```
// Common CPU setup code.
static void
mpmain(void)
{
  cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
  idtinit();       // load idt register
  xchg(&(mycpu()->started), 1); // tell startothers() we're up
  mycpu()->cpu_ticks = 0;
  scheduler();     // start running processes
}
```

```
// Start the non-boot (AP) processors.
static void
startothers(void)
{
  extern uchar _binary_entryother_start[], _binary_entryother_size[];
  uchar *code;
  struct cpu *c;
  char *stack;

  // Write entry code to unused memory at 0x7000.
  // The linker has placed the image of entryother.S in
  // _binary_entryother_start.
  code = P2V(0x7000);
  memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);

  for(c = cpus; c < cpus+ncpu; c++){
    if(c == mycpu())  // We've started already.
      continue;

    c->cpu_ticks = 0;

    // Tell entryother.S what stack to use, where to enter, and what
    // pgdir to use. We cannot use kpgdir yet, because the AP processor
    // is running in low  memory, so we use entrypgdir for the APs too.
    stack = kalloc();
    *(void**)(code-4) = stack + KSTACKSIZE;
    *(void(**)(void))(code-8) = mpenter;
    *(int**)(code-12) = (void *) V2P(entrypgdir);

    lapicstartap(c->apicid, V2P(code));

    // wait for cpu to finish mpmain()
    while(c->started == 0)
      ;
  }
}
```

21. با توجه به سیاست های پیاده سازی شده در سطح های دوم و سوم و همچنین استفاده از روش time-slicing توجیه کنید چرا همچنان مشکل starvation امکان رخ دادن دارد؟

حتی با وجود الگوریتم WRR (Time Slicing) در Multilevel Feedback Queue، همچنان احتمال بروز Starvation وجود دارد. به عنوان مثال:

در صف دوم، اگر از الگوریتم Shortest Job First استفاده شود، ممکن است یک فرآیند به دلیل طولانی بودن مدت اجرای خود، همواره نادیده گرفته شود و هرگز فرصت اجرا پیدا نکند.

در صف سوم که از الگوریتم First Come First Serve استفاده می‌کند، ممکن است یک فرآیند به دلیل دارا بودن چرخه بی‌نهایت در کد خود باعث ایجاد Starvation برای سایر فرایندهای درون این صف شود.

برای رفع این مشکل، می‌توان از مکانیسم‌هایی مانند سازوکار افزایش سن (Aging) استفاده کرد.

سازوکار افزایش سن

```
#define AGING_THRESHOLD 800
```

```
void aging_process(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){
        if (p->sched_info.queue == FCFS){
            release(&ptable.lock);
            change_queue(p->pid, SJF);
            p->sched_info.last_run = ticks;
            acquire(&ptable.lock);
        }else{
          release(&ptable.lock);
          change_queue(p->pid, ROUND_ROBIN);
            p->sched_info.last_run = ticks;
          acquire(&ptable.lock);
        }
      }
    }
  }
  release(&ptable.lock);
}
```

22. به چه علت مدت زمانی که که پر دارد در وضعیت sleeping می باشد به عنوان رمان انتظار در نظر گرفته نمی شود؟

مدت زمانی که پردازش در وضعیت Sleeping قرار دارد، به عنوان زمان انتظار (Waiting Time) در نظر گرفته نمی‌شود زیرا در این حالت، پردازش عملاً در حال انتظار برای اجرای CPU نیست. وقتی یک پردازش در حالت Sleeping است، معمولاً منتظر رویدادی خاص (مانند تکمیل عملیات I/O یا دریافت سیگنال) است. در این وضعیت، پردازش از Ready Queue خارج شده و CPU به دیگر پردازش‌ها اختصاص داده می‌شود. در واقع، این حالت نشان‌دهنده انتظار پردازش برای دسترسی به منابع خارجی است، نه CPU. از طرف دیگر زمان انتظار به مدت زمانی گفته می‌شود که یک پردازش در Ready Queue حضور دارد و منتظر تخصیص CPU است. این زمان شامل زمانی است که پردازش آماده اجرا است اما CPU هنوز به آن اختصاص داده نشده است.

فراخوانی‌های سیستمی مورد نیاز:

مقداردهی اولیه زمان اجرای تخمینی و سطح اطمینان :

```
int sys_set_sjf_params(void){
  int pid;
  int priority_ratio, arrival_time_ratio;
  if(argint(0, &pid) < 0 || argint(1, &priority_ratio) < 0 || argint(2, &arrival_time_ratio) < 0){
    return -1;
  }

  return set_sjf_params(pid, priority_ratio, arrival_time_ratio);
}
```

```
int set_sjf_params(int pid, int burstTime, int confidence){
  struct proc *p;

  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if (p->pid == pid){
      p->sched_info.sjf.BurstTime = burstTime;
      p->sched_info.sjf.Confidence = confidence;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

تغییر صف :

```
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 || pid == 2)
      new_queue = ROUND_ROBIN;
    else if (pid > 2)
      new_queue = FCFS;
    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;
      if (compare_string(p->name, "sh"))
        p->sched_info.queue = ROUND_ROBIN;
      else
        p->sched_info.queue = new_queue;
      p->sched_info.arrival_queue_time = ticks;
    }
  }
  release(&ptable.lock);
  return old_queue;
}
```

چاپ اطلاعات :

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

  static int columns[] = {16, 6, 11, 8, 12, 13, 13, 18, 11};
  cprintf("name");
  print_blank(columns[0] - strlen("name"));
  cprintf("pid");
  print_blank(columns[1] - strlen("pid"));
  cprintf("state");
  print_blank(columns[2] - strlen("state"));
  cprintf("queue");
  print_blank(columns[3] - strlen("queue"));
  cprintf("wait_time");
  print_blank(columns[4] - strlen("wait_time"));
  cprintf("confidence");
  print_blank(columns[5] - strlen("confidence"));
  cprintf("burst_time");
  print_blank(columns[6] - strlen("burst_time"));
  cprintf("consecutive_run");
  print_blank(columns[7] - strlen("consecutive_run"));
  cprintf("Arrival");
  print_blank(columns[8] - strlen("Arrival"));
  cprintf("\n");
  cprintf("-------------------------------------------------------------------------------------------------------------\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);
  print_blank(columns[0] - strlen(p->name));

  cprintf("%d", p->pid);
  print_blank(columns[1] - find_length(p->pid));

  cprintf("%s", state);
  print_blank(columns[2] - strlen(state));

  cprintf("%d", p->sched_info.queue);
  print_blank(columns[3] - find_length(p->sched_info.queue));

  int wait_time = 0;
  if (p->state == RUNNABLE){
    wait_time = ticks - p->sched_info.last_run;
  }
  cprintf("%d", (int)(wait_time));
  print_blank(columns[4] - find_length((int)(wait_time)));

  cprintf("%d", (int)p->sched_info.sjf.Confidence);
  print_blank(columns[5] - find_length((int)p->sched_info.sjf.Confidence));
```

```
    cprintf("%d", (int)p->sched_info.sjf.BurstTime);
    print_blank(columns[6] - find_length((int)p->sched_info.sjf.BurstTime));

    cprintf("%d", (int)p->consecutive_time);
    print_blank(columns[7] - find_length((int)p->consecutive_time));

    cprintf("%d", p->sched_info.sjf.arrival_time);
    print_blank(columns[8] - find_length(p->sched_info.sjf.arrival_time));

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

خروجی برنامه :

در تصویر زیر چند schedule_test را اجرا میکنیم و با استفاده از دستور info می توانیم روند اجرا شدن آن ها را در بازه های زمانی متفاوت مشاهده کنیم.

```
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
Group members:
1. Amirhossein Arefzadeh
2. Mahdi Naieni
3. kiarash Khorasani
$ schedule info
name            pid   state      queue   wait_time   confidence   burst_time   consecutive_run   Arrival
-------------------------------------------------------------------------------------------------------
init            1     sleeping   1       0           50           2            4                 0
sh              2     sleeping   1       0           50           2            1                 11
schedule        3     running    1       0           50           2            3                 1233
$
```

```
$ schedule_test&
$ schedule info
name            pid   state      queue   wait_time   confidence   burst_time   consecutive_run   Arrival
-------------------------------------------------------------------------------------------------------
init            1     sleeping   1       0           50           2            4                 0
sh              2     sleeping   1       0           50           2            0                 11
schedule_test   7     running    3       0           50           2            3                 2619
schedule_test   6     sleeping   1       0           50           2            3                 2614
schedule_test   8     runnable   3       4           50           2            10                2620
schedule_test   9     runnable   3       364         50           2            0                 2620
schedule_test   10    runnable   3       362         50           2            0                 2622
schedule_test   11    runnable   3       362         50           2            0                 2622
schedule        12    running    1       0           50           2            1                 2979
$
```

```
$ schedule info
name            pid   state      queue   wait_time   confidence   burst_time   consecutive_run   Arrival
-------------------------------------------------------------------------------------------------------
init            1     sleeping   1       0           50           2            4                 0
sh              2     sleeping   1       0           50           2            1                 11
schedule_test   7     runnable   3       7           50           2            10                2619
schedule_test   6     sleeping   1       0           50           2            3                 2614
schedule_test   8     runnable   3       3           50           2            10                2620
schedule_test   9     runnable   2       455         50           2            20                2620
schedule_test   10    runnable   2       15          50           2            20                2622
schedule_test   11    running    2       0           50           2            4                 2622
schedule        14    running    1       0           50           2            1                 3882
$
```

```
$ schedule info
name            pid   state      queue   wait_time   confidence   burst_time   consecutive_run   Arrival
-------------------------------------------------------------------------------------------------------
init            1     sleeping   1       0           50           2            4                 0
sh              2     sleeping   1       0           50           2            1                 11
schedule_test   7     running    3       0           50           2            3                 2619
schedule_test   6     sleeping   1       0           50           2            3                 2614
schedule_test   8     runnable   3       386         50           2            10                2620
schedule_test   9     runnable   1       2           50           2            5                 2620
schedule_test   10    runnable   2       3           50           2            20                2622
schedule_test   11    runnable   2       19          50           2            20                2622
schedule        21    running    1       0           50           2            0                 7777
```

حال دستورات change_queue را برای پردازه 10 استفاده می کنیم و آن را از صف 3 به صف 1 انتقال می دهیم. سپس دستور set_sjf_param را برای همین پردازه صدا می کنیم و مقادیر مربوط به burst time و confidence را برای این پردازه تغییر می دهیم. در نهایت می توانیم خروجی برنامه را در زیر مشاهده کنیم.

```
$ schedule_test&
$ schedule info
name            pid   state      queue   wait_time   confidence   burst_time   consecutive_run   Arrival
-------------------------------------------------------------------------------------------------------
init            1     sleeping   1       0           50           2            0                 0
sh              2     sleeping   1       0           50           2            1                 12
schedule_test   9     running    3       0           50           2            4                 7751
schedule_test   8     sleeping   1       0           50           2            4                 7748
schedule_test   10    runnable   3       6           50           2            10                7753
schedule_test   11    runnable   3       260         50           2            0                 7753
schedule_test   12    runnable   3       261         50           2            0                 7754
schedule_test   13    runnable   3       262         50           2            0                 7755
schedule        14    running    1       0           50           2            2                 8007
$ schedule change_queue 1 10
Queue changed successfully
$ schedule info
name            pid   state      queue   wait_time   confidence   burst_time   consecutive_run   Arrival
-------------------------------------------------------------------------------------------------------
init            1     sleeping   1       0           50           2            0                 0
sh              2     sleeping   1       0           50           2            0                 12
schedule_test   9     runnable   3       12          50           2            10                7751
schedule_test   8     sleeping   1       0           50           2            4                 7748
schedule_test   10    runnable   1       3           50           2            0                 7753
schedule_test   11    runnable   1       2           50           2            5                 7753
schedule_test   12    runnable   2       26          50           2            20                7754
schedule_test   13    runnable   2       35          50           2            20                7755
schedule        16    running    1       0           50           2            3                 10256
$ schedule set_sfj_param 1 2  100
SJF params has been set successfully
$ schedule info
name            pid   state      queue   wait_time   confidence   burst_time   consecutive_run   Arrival
-------------------------------------------------------------------------------------------------------
init            1     sleeping   1       0           50           2            0                 0
sh              2     sleeping   1       0           50           2            1                 12
schedule_test   9     runnable   3       4           50           2            10                 7751
schedule_test   8     sleeping   1       0           50           2            4                 7748
schedule_test   10    running    1       0           100          4            1                 7753
schedule_test   11    runnable   1       3           50           2            5                 7753
schedule_test   12    runnable   2       11          50           2            20                7754
schedule_test   13    runnable   2       31          50           2            20                7755
schedule        18    running    1       0           50           2            2                 13267
```