به نام خدا

گزارش پروژه «۳» آزمایشگاه سیستم عامل

استاد: دکتر کارگهی

گروه ۱۱

امیرارسلان شهبازی ۸۱۰۱۰۱۴۵۱

سید محمدحسین مظهری ۸۱۰۱۰۱۵۲۰

محمدمهدی صمدی ۸۱۰۱۰۱۴۶۵

لینک مخزن https://github.com/AMIRSH1383/OS-SMS_LAB3.git

آخرین کامیت d937ba33775d174c0fd2d51879395c4da638e441

۱) ساختار PCB و همچنین وضعیت های تعریف شده برای هر پردازه را در xv6 پیدا کرده و گزارش کنید. آیا شباهتی میان داده های موجود در این ساختار و ساختار به تصویر کشیده شده در شکل ۳.۳ منبع درس وجود دارد؟

در فایل proc.h استراکت هایی که برای یک پراسس به کار می روند تعریف شده است.

همانطور که در شکل زیر مشاهده می کنید، داده ساختار زیر برای پراسس های ما وجود دارند.

توضیحات مربوط به هر مشخصه به صورت کامنت جلوی آن داده شده است و ما صرفا شباهت های آن نسبت به شکل منبع درس را گزارش خواهیم کرد.

```
43  // Per-process state
44  struct proc {
45    uint sz;                     // Size of process memory (bytes)
46    pde_t* pgdir;                // Page table
47    char *kstack;                // Bottom of kernel stack for this process
48    enum procstate state;        // Process state
49    int pid;                     // Process ID
50    struct proc *parent;         // Parent process
51    struct trapframe *tf;        // Trap frame for current syscall
52    struct context *context;     // swtch() here to run process
53    void *chan;                  // If non-zero, sleeping on chan
54    int killed;                  // If non-zero, have been killed
55    struct file *ofile[NOFILE];  // Open files
56    struct inode *cwd;           // Current directory
57    char name[16];               // Process name (debugging)
58    struct syscall_info syscalls[1000]; // List of system calls used in process
59    int syscall_count;           // Number of system calls used in process
60  };
```

شکل منبع درس به صورت زیر است.

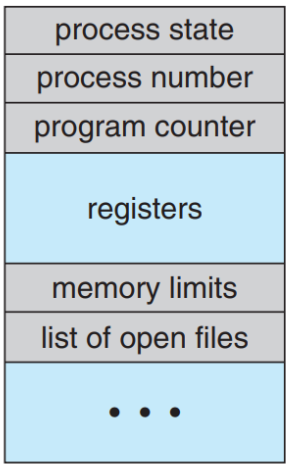


Figure 3.3 Process control block (PCB).

همانطور که می بینید شباهت هایی میان این دو ساختار وجود دارد.

در بخش زیر معادل هریک از ویژگی های جدول بالا در xv6 را می آوریم.

process state => enum procstate state

process number => int pid

registers => struct context *context;

در استراکتی که برای کانتکست تعریف می کنیم از اسامی رجیسترها استفاده و می کنیم به همین دلیل برای رجیسترها کانتکست را مشابه در نظر می گیریم.

Memory limits => uint sz

List of open file => struct file *ofile [*NOFILE*]

Memory management information => pde_t* pgdir

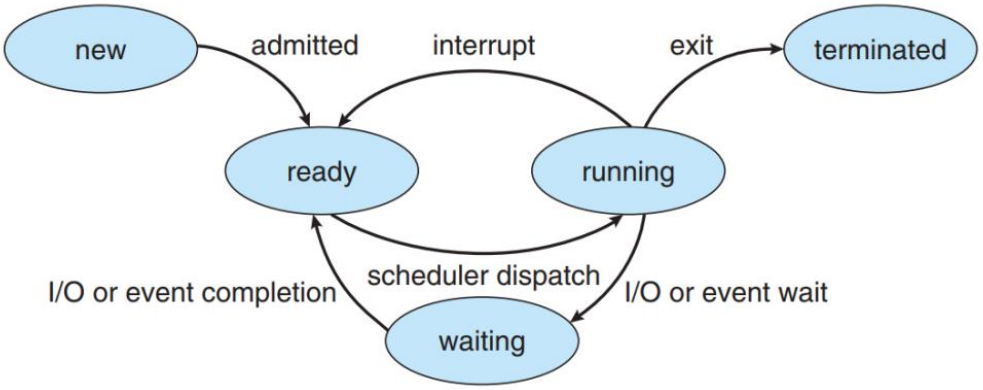
اما بخش دیگری که به آن باید بپردازیم وضعیت هایی است که هر استیت می تواند در آن قرار بگیرد.

این وضعیت ها مجدد در همان فایل چند خط بالاتر تعریف شده که به شرح زیر است.

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

در سوال بعد توضیحات مربوط به هریک را می دهیم.

۲) هرکدام از وضعیت های تعریف شده معادل کدام وضعیت در شکل ۱ می باشد؟



new => EMBRYO

ready => RUNNABLE

running => RUNNING

waiting => SLEEPING

terminated => ZOMBIE

یک استیت باقی ماند که در بالا تعریف نشده بود که آن استیت UNUSED است توضیحات این استیت به این صورت است که این وضعیت نشان‌دهنده این است که ورودی برای اختصاص به یک فرآیند جدید در دسترس است اما در حال حاضر در حال استفاده نیست.(به مرحله new نرسیده است)

۳) 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 تصویر کشیده شده، خواهد شد؟ وضعیت یک پردازه در xv6 در این گذار از چه حالتی به چه حالتی تغییر می کند؟ پاسخ خود را با پاسخ سوال ۲ مقایسه کنید.

همانطور که در شکل زیر مشاهده می کنید ما دوجا این کار را کردیم. اولی برای ساخت اولین پراسس کاربر است. و دومی که در بقیه موارد استفاده می شود در تابع fork است که برای ساخت پردازه جدید به کار می رود. کد به این صورت است که در پایان وضعیت پراسس در حال ساخت را به پراسس در حال اجرا تبدیل می کند.

```
122  userinit(void)
146    // this assignment to p->state lets other cores
148    // writes to be visible, and the lock is also needed
149    // because the assignment might not be atomic.
150    acquire(&ptable.lock);
151
152    p->state = RUNNABLE;
153
154    release(&ptable.lock);
155  }
```

```
182  fork(void)
213
214    pid = np->pid;
215
216    acquire(&ptable.lock);
217
218    np->state = RUNNABLE;
219
220    release(&ptable.lock);
221
222    return pid;
223  }
```

پس با توجه به این موارد جمع بندی ما این می شود که وضعیت ما از ابتدای کار که در EMBRYO یا همان نیو قرار داشتیم به وضعیت RUNNABLE یا همان ready تبدیل می شود که این قضیه با مواردی که از سوالات یک و دو دریافته بودیم همخوانی دارد.

۴) سقف تعداد پردازه های 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 می کند؟

سقف تعداد پردازه ها به صورت یک مقدار دیفاین شده در فایل param.h قرار گرفته است.

```
C param.h > NPROC
1   #define NPROC        64  // maximum number of processes
```

همانطور که مشاهده می کنید حداکثر ۶۴ پراسس می تواند در لحظه وجود داشته باشد.

با توجه به کدی که برای اختصاص دادن پراسس نوشته شده است کرنل نسبت به ساخت بیش از حد پراسس ها واکنش نشان می دهد. این عملیات به این صورت است که اگر ptable ما دیگر ظرفیت برای اختصاص دادن پراسس جدید را نداشت قفل آزاد شده و ریترن ۰ خواهیم داشت که این ریترن نشان دهنده این است که در اختصاص پراسس خطا رخ داده و نتوانستیم پراسس جدید ایجاد کنیم.

اما اگر جا برای این کار داشتیم همانطور که مشاهده می کنید از عملیات goto استفاده کردیم و ریترن ۰ را رد کردیم و پراسس را در جلوتر تشکیل داده و پوینتری به پراسس ساخته شده را برمی گردانیم.

```
73  static struct proc*
74  allocproc(void)
75  {
76    struct proc *p;
77    char *sp;
78
79    acquire(&ptable.lock);
80
81    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
82      if(p->state == UNUSED)
83        goto found;
84
85    release(&ptable.lock);
86    return 0;
87
88  found:
```

۵) چرا نیاز است در ابتدای هر حلقه تابع schedule، جدول پردازه ها قفل شود؟ آیا در سیستم های تک پردازه ای هم نیاز است این کار صورت بگیرد؟

این به این دلیل است که مطمئن باشیم در هرلحظه تنها یک پردازه یا ترد به جدول پردازه ها دسترسی دارد. اگر اینطور نبود یک رقابتی میان پردازه ها به وجود می آمد که اصطلاحا به آن race condition می گویند به این صورت که ممکن بود در لحظه چند پراسس به جدول پردازه ها دسترسی می داشتند که این منجر می شد که داده های ما خراب و ناپایدار باشند.

اگر سیستم تک پردازه ای بود از اینکه در هر لحظه در سیستم تنها یک پردازه وجود دارد مطمئن بودیم و شرایط رقابتی به وجود نمی آمد لذا نیازی نبود جدول پردازه ها را قفل کنیم.

۶) با فرض اینکه xv6 در حالت تک هسته ای در حال اجراست، اگر یک پردازه به حالت runnable برود و صف پردازه ها در حال طی شدن باشد، در مکانیزم زمان بندی xv6 نسبت به موقعیت پردازه در صف، در چه iteration ای امکان schedule پیدا می کند؟

در تابع scheduler طبق قطعه کد زیر مشاهده می کنیم که وقتی که روی صف پردازه ها پیمایش می کنیم دنبال این می گردیم که یک پراسسی که در وضعیت runnable باشد را پیدا کنیم.

همانطور که می بینید کد ما به این صورت است که اگر پراسسی قابل اجرا نباشد، از آن عبور می کند تا به یک پراسس با قابلیت اجرا برسد.

هر وقت که این پراسس را پیدا کرد، سپس context switch انجام داده تا پراسس runnable به پراسس running تبدیل بشود.

پس از همه ی این مراحل وقتی کارمان تمام می شود قفل جدول پردازه ها را رها می کنیم و مجدد این عملیات ها انجام می شوند.

```
324  scheduler(void)
329
330    for(;;){
331      // Enable interrupts on this processor.
332      sti();
333
334      // Loop over process table looking for process to run.
335      acquire(&ptable.lock);
336      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
337        if(p->state != RUNNABLE)
338          continue;
339
340        // Switch to chosen process.  It is the process's job
341        // to release ptable.lock and then reacquire it
342        // before jumping back to us.
343        c->proc = p;
344        switchuvm(p);
345        p->state = RUNNING;
346
347        swtch(&(c->scheduler), p->context);
348        switchkvm();
349
350        // Process is done running for now.
351        // It should have changed its p->state before coming back.
352        c->proc = 0;
353      }
354      release(&ptable.lock);
355
356    }
```

۷)

توضیحاتی درباره struct context:

- برای ذخیره حالت register ها در لحظه context switch استفاده می‌شود.
- تمام register ها ذخیره نمی‌شوند چرا که برخی از آن‌ها مقدار ثابتی در تمام موضوعات دارند.
- نیازی به ذخیره register هایی مثل EAX, ECX, EDX نداریم زیرا قرارداد طراحی XV6 تضمین می‌کند که که توسط صداکننده تابع ذخیره شده‌اند.
- برای هر استک، ساختار کانتکستش در انتهای پایین استک قرار دارد. پس ESP به مکان این ساختار اشاره می‌کند.

توضیحات هر register:

- EDI: مخفف extended destination index است. این رجیستر عمدتا برای عملیات‌های مربوط به string استفاده می‌شود. معمولا pointer به مقصد در هنگام عملیات‌هایی مثل MOVS, CMPS, SCAS و غیره را نگه می‌دارد.
- ESI: مخفف extended source index است. این register نیز برای عملیات‌های مربوط به string استفاده می‌شود. اما تفاوتش با EDI این است که pointer به مبدا را نگه می‌دارد. یعنی آدرس جایی که string از آن خوانده می‌شود را دارد.
- EBX: مخفف extended base register است. عموما برای نگه‌داری داده، شمارنده و آدرس‌ها استفاده می‌شود.
- EBP: مخفف extended base pointer است. برای نگه‌داری pointer به base استک کنونی استفاده می‌شود. کمک می‌کند که طبق قراردادهایی که داریم، به پارامترها و متغیرهای محلی تابع دسترسی پیدا کنیم. طبق قرارداد، این register در تمام مدت زمان اجرای یک تابع ثابت باقی می‌ماند و راحت‌تر می‌توانیم بدانیم هر متغیر کجا قرار دارد.
- EIP: مخفف extended instruction pointer است. به instruction بعدی که قرار است اجرا شود اشاره می‌کند. در هر بار اجرای instruction جدید این عدد افزایش میابد.

۸)

در struct context پردازنده XV6، مقدار program counter در EIP ذخیره می‌شود. در واقع این register آدرس دستور بعدی که باید اجرا شود را نگه می‌دارد.

چگونگی انجام این کار را باید در کد اسمبلی نوشته شده بررسی کنیم. کد به صورت زیر است:

swtch.S

```
1   # Context switch
2   #
3   #   void swtch(struct context **old, struct context *new);
4   #
5   # Save the current registers on the stack, creating
6   # a struct context, and save its address in *old.
7   # Switch stacks to new and pop previously-saved registers.
8
9   .globl swtch
10  swtch:
11    movl 4(%esp), %eax
12    movl 8(%esp), %edx
13
14    # Save old callee-saved registers
15    pushl %ebp
16    pushl %ebx
17    pushl %esi
18    pushl %edi
19
20    # Switch stacks
21    movl %esp, (%eax)
22    movl %edx, %esp
23
24    # Load new callee-saved registers
25    popl %edi
26    popl %esi
27    popl %ebx
28    popl %ebp
29    ret
```

- خط ۹ تا ۱۲: تابع دو آرگومان می‌گیرد. طبق قرارداد می‌دانیم آرگومان ها بلافاصله بعد از اشاره‌گر ESP قرار دارند. پس آدرس قدیمی در ESP+4 قرار دارد که به رجیستر EAX می‌رود. آدرس جدید هم که در ESP+8قرار دارد به رجیستر EDX می‌رود.
- خط ۱۵ تا ۱۸: مقدار کنونی رجیسترهای struct context (به جز EIP که برای PC است) به استک وارد می‌شوند.
- خط ۲۱ و ۲۲: آدرس استک کنونی که در ESP است، در آدرسی که رجیستر EAX اشاره می‌کند ذخیره می‌شود. در واقع این آدرس همان آدرس context قدیمی است که در آرگومان تابع داده شده بود. سپس محتوای جدید که در EDX ذخیره‌اش کرده بودیم، به ESP که استک کنونی را نشان می‌دهد انتقال میابد.
- خط ۲۵ تا ۲۹: حالا مقدار رجیسترهای struct context کنونی برعکس ترتیب پوش شدن‌شان از استک پاپ می‌شوند و در رجیسترهای EDI, ESI, EBX و EBP ذخیره می‌شوند. در آخر ret از استک return address را پاپ می‌کند و در EIP می‌گذارد. بدین ترتیب PC به درستی آپدیت می‌شود.

(۹

عواقب فعال نکردن interrupt ها:

- پاسخ ندادن به hardware interruptها: در این صورت کارهای مهمی مثل ورودی کیبورد، ماوس یا تایمر نادیده گرفته می‌شوند.
- نداشتن time sharing: در این حالت امکان اجرای preemptive را نداریم. پس هر process از زمانی که CPU را بگیرد تا زمانی که کارش تمام نشود، CPU را پس نمی‌دهد.

(۱۰

فاصله زمانی تو timer interrupt متوالی:

برای این سوال به توضیحات کتاب XV6 مراجعه می‌کنیم. طبق متن در هر ثانیه ۱۰۰ بار فعال می‌شود. پس فاصله زمانی دو وقفه متوالی ۱۰ میلی ثانیه است.

Code: Interrupts

Devices on the motherboard can generate interrupts, and xv6 must set up the hardware to handle these interrupts. Devices usually interrupt in order to tell the kernel that some hardware event has occured, such as I/O completion. Interrupts are usually optional in the sense that the kernel could instead periodically check (or "poll") the device hardware to check for new events. Interrupts are preferable to polling if the events are relatively rare, so that polling would waste CPU time. Interrupt handling shares some of the code already needed for system calls and exceptions.

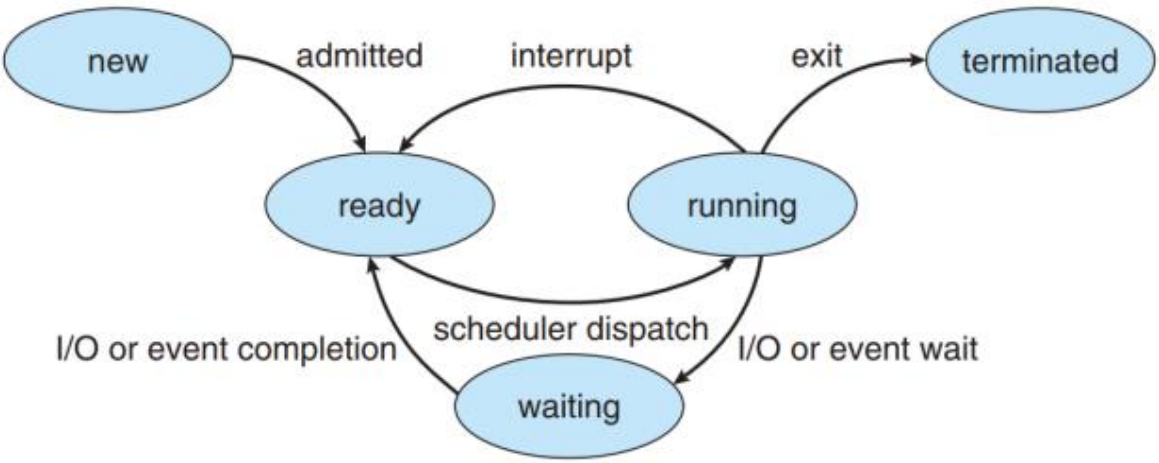
Interrupts are similar to system calls, except devices generate them at any time. There is hardware on the motherboard to signal the CPU when a device needs attention (e.g., the user has typed a character on the keyboard). We must program the device to generate an interrupt, and arrange that a CPU receives the interrupt.

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.

(۱۱

تابعی که منجر به تغییر حالت به علت وقفه می‌شود:

به شکل ۱ صورت پروژه مراجعه می‌کنیم. این تابع باید ما را از حالت running به حالت runnable ببرد. طبق توضیحات صورت پروژه تابع yield این وظیفه را به عهده دارد



۱۲)

مقدار کوانتوم زمانی الگوریتم Round Robin:

طبق توضیحات سوال ۱۰، هر ۱۰ میلی ثانیه وقفه زمان فعال می‌شود. از آن جایی که با هر بار فعال شدن وقفه زمانی، کوانتوم زمانی یک process به پایان می‌رسد می‌توان نتیجه گرفت که مقدار کوانتوم زمانی برابر ۱۰ میلی ثانیه است. اما در پیاده‌سازی که در سطح اول صف انجام می‌دهیم این مقدار برابر ۵۰ میلی ثانیه خواهد شد چرا که با هر ۵ بار فعال شدن وقفه زمانی کوانتوم گرفته می‌شود.

۱۳) تابع wait در نهایت از چه تابعی برای منتظر ماندن برای اتمام کار یک پردازه استفاده می کند ؟

- این تابع wait است. این تابع به پردازه والد امکان می‌دهد برای اتمام پردازه‌های فرزند خود منتظر بماند.
- عملکرد تابع به این صورت است که والد منتظر می‌ماند تا یکی از فرزندانش به حالت ZOMBIE برسد. پس از آن، اطلاعات مربوط به پردازه فرزند (مانند PID و کد خروج) به والد بازگردانده شده و منابع فرزند از سیستم حذف می‌شوند.
- اگر فرزندی وجود نداشته باشد یا همه فرزندان قبلاً جمع‌آوری شده باشند، تابع wait مقدار -1 بازمی‌گرداند.

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

تابع wait علاوه بر آزادسازی منابع پردازه‌های فرزند که به پایان رسیده‌اند، یک کاربرد مهم دیگر نیز دارد:

جلوگیری از پردازه‌های زامبی

- زمانی که یک پردازه فرزند به پایان می‌رسد، وضعیت آن به ZOMBIE تغییر می‌کند.
- پردازه در حالتZOMBIE باقی می‌ماند تا زمانی که پردازه والد اطلاعات وضعیت نهایی آن (مثل exit code) را دریافت کند.
- اگر پردازه والد از wait استفاده نکند، پردازه فرزند به حالت زامبی باقی می‌ماند و منابع آن (مانند جدول پردازه‌ها) آزاد نمی‌شوند.

چگونه wait از پردازه‌های زامبی جلوگیری می‌کند؟

۱. وقتی والد تابع waitرا فراخوانی می‌کند:

- وضعیت پردازه فرزند را دریافت می‌کند.
- منابع آن پردازه در کرنل آزاد می‌شوند.

۲. این کار تضمین می‌کند که هیچ پردازه زامبی‌ای در سیستم باقی نماند.

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

- درxv6 ، تابعی که برای آگاه‌سازی پردازه از وقوع رویدادها استفاده می‌شود، wakeup است.
- وقتی پردازه‌ای منتظر یک رویداد) مانند اتمام I/O) است، در حالت SLEEPING قرار می‌گیرد. زمانی که رویداد مورد نظر رخ دهد، wakeup همه پردازه‌هایی را که در انتظار آن رویداد خاص بوده‌اند، به حالت RUNNABLE منتقل می‌کند.

۱۶) با توجه به پاسخ سوال ۹ ، این تابع منجر به گذار از چه وضعیتی به چه وضعیتی در شکل ۱ خواهد شد ؟

- تابع wakeup باعث تغییر وضعیت پردازه از waiting به ready می‌شود.
- به این ترتیب: پردازه‌ای که در حالت waiting بوده (منتظر رویدادی مانند اتمام خواندن فایل)، پس از وقوع رویداد مورد نظر به حالت ready تغییر وضعیت داده و آماده اجرا می‌شود.

۱۷) آیا تابع دیگری وجود دارد که منجر به انجام این گذار شود؟ نام ببرید .

علاوه بر wakeup، تابع wakeنیز وجود دارد که در xv6 برای بیدار کردن یک پردازه خاص استفاده می‌شود.

تفاوت بین این دو تابع:

- wakeupهمه پردازه‌هایی که منتظر یک رویداد خاص هستند را بیدار می‌کند.
- wakeیک پردازه خاص (براساس ساختار داده struct proc) را از حالت SLEEPING به RUNNABLE تغییر می‌دهد.

۱۸) در بخش ۳.۳.۲ منبع درس با پردازه های Orphan آشنا شدید ، رویکرد xv6 در رابطه با این گونه پردازه ها چیست ؟

تعریف پردازه orphan (یتیم):

یک پردازه یتیم (Orphan Process) پردازه‌ای است که والد آن (Parent) پیش از اتمام اجرای پردازه فرزند، خاتمه یافته باشد. در این شرایط، پردازه یتیم بدون والد باقی می‌ماند و نیازمند مدیریتی خاص توسط سیستم‌عامل است.

رویکرد xv6 نسبت به پردازه‌های Orphan :

درxv6 ، وقتی یک پردازه والد خاتمه پیدا می‌کند:

۱. تمام پردازه‌های فرزند آن (اعم از پردازه‌های فعال، معلق، یا زامبی) به پردازه initproc منتقل می‌شوند.

۲. Initproc به عنوان والد جدید این پردازه‌های یتیم عمل می‌کند و مسئولیت مدیریت آن‌ها را بر عهده می‌گیرد.

جزئیات عملکرد:

- زمانی که والد پردازه‌ای حذف می‌شود، کرنل xv6 حلقه‌ای در جدول پردازه‌ها اجرا می‌کند. این حلقه هر پردازه‌ای که به پردازه خاتمه‌یافته به‌عنوان والد وابسته باشد، پیدا کرده و والد آن را به initproc تغییر می‌دهد.
- اگر پردازه یتیم به حالت ZOMBIE برسد، initproc وظیفه جمع‌آوری و آزادسازی منابع آن را دارد.

چرا initproc؟

- Initproc اولین پردازه‌ای است که در سیستم توسط کرنل اجرا می‌شود و همیشه فعال باقی می‌ماند.
- به همین دلیل، از این پردازه به‌عنوان والد جایگزین برای مدیریت پردازه‌های یتیم استفاده می‌شود.

زمان بندی بازخوردی چندسطحی

در ابتدا ما می بایست یکسری ساختار جدید به ساختار پراسس اضافه کنیم که مربوط به کارهای زمان بندی است.

با توجه به اینکه سه سطح الگوریتم داریم لذا یک enum در فایل proc.h برای این سه الگوریتم داریم که به شکل زیر است.

```
37    enum schedule_algorithms {NOTHING , ROUND_ROBIN , SJF , FCFS};
```

سپس یک استراکت جدید به نام schedule_info می سازیم که اطلاعاتی که در زمان بندی برای ما مهم هستند را در آن قرار می دهیم. این استراکت به شرح زیر است.

```
39    typedef struct schedule_info
40    {
41        int queue_level;
42        int burst_time;
43        int confidence;
44        int consecutive_run;
45        int arrival_time;
46        int last_running_time;
47        int wait_time;
48
49    }schedule_info;
```

گام بعدی این است که نمونه ای از این استراکت را در استراکت پراسس قرار می دهیم. اینگونه هر پراسسی که ساخته می شود اطلاعاتی برای زمان بندی را نیز درون خود نگه می دارد. مقادیر اولیه ی این خصیصه ها به شرح زیر است.

```
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    p->syscall_count = 0;
93    p->sched_info.queue_level = 3; // Default queue is the third queue
94    p->sched_info.burst_time = 2;
95    p->sched_info.confidence = 50;
96    p->sched_info.wait_time = 0;
97    p->sched_info.consecutive_run = 0;
98    p->sched_info.arrival_time = ticks;
99    p->sched_info.last_running_time = ticks;
100   release(&ptable.lock);
```

این مقادیر اولیه هنگام تخصیص پراسس به خصیصه ها تعلق می گیرد.

بخش عمومی به اتمام رسید و از الان به پیاده سازی هر الگوریتم می پردازیم.

سطح اول: زمان بند نوبت گردشی با کوانتوم زمانی

```
331  scheduler(void)
336  {
337    for(;;){
338      // Enable interrupts on this processor.
339      sti();
340
341      // Loop over process table looking for process to run.
342      acquire(&ptable.lock);
343      struct proc *rr_process = 0;
344      struct proc *shortest_proc = 0;
345      struct proc *last_proc = 0;
346      long long min_burst_time = 1e9;
347      long long min_last_running_time_fcfs = 1e9;
348      long long min_last_running_time_rr = 1e9;
349      int found_first_priority_process = 0;
350      int found_second_priority_process = 0;
351      int found_third_priority_process = 0;
352
353      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
354        if(p->state != RUNNABLE)
355          continue;
356
```

در ابتدای حلقه بی نهایتی که در فایل زمان بند وجود دارد یکسری متغیرهایی که قرار است استفاده کنیم را تعریف می کنیم. سه تا پراسس تعریف شده که هر کدام برای خود پراسس مربوط به همان سطح را اخذ کند و بسته به وزن هرکدام از پراسس مربوطه استفاده کنیم.

یکسری متغیرهایی مانند برخی زمان ها که در الگوریتم های اول و سوم به کار می رود نیز تعریف شده اند.

در نهایت هم برای هر پراسسی که از هر سطح پیدا می کنیم مشخص می کنیم اگر پراسسی در آن سطح وجود دارد یا نه.

```
// A
if (p->sched_info.queue_level == ROUND_ROBIN) {
  found_first_priority_process = 1;
  if(p->sched_info.last_running_time < min_last_running_time_rr)
  {
      min_last_running_time_rr = p->sched_info.last_running_time;
      rr_process = p;
  }
}
```

در ادامه حلقه بالا ما می بایست برای هریک از الگوریتم هایی که داریم یک شرط بگذاریم و ببینیم که آیا پراسسی که داریم پیمایش می کنیم عضو این الگوریتم هست یا خیر.

در سطح اول اگر پراسس جزو سطح اول بود، می گوییم یک پراسس سطح اول پیدا کردیم و در ادامه چک می کنیم اگر بیشتر از بقیه از مدت زمان اجرا شدنش گذشته است این نشان می دهد که بیشتر از بقیه از زمان اجرا شدنش گذشته و الان وقت اجرای آن است. پس مینیمم اخرین زمان اجرا را مجدد مقدار دهی می کنیم و پراسس نوبت گردشی را ست می کنیم.

در نهایت از انجایی که هر تیک به اندازه ۱۰ میلی ثانیه در نظر گرفته شده است و ما می خواهیم که کوانتوم زمانی ما ۵۰ میلی ثانیه باشد، لذا می بایست هر ۵ اینتراپت یک بار تابع yield صدا زده شود که این امر را در فایل trap.c اعمال می کنیم که به شکل زیر می شود.

```
127   if(myproc() && myproc()->state == RUNNING &&
128     tf->trapno == T_IRQ0+IRQ_TIMER)
129     {
130       myproc()->sched_info.consecutive_run++;
131       if (myproc()->sched_info.queue_level == ROUND_ROBIN)
132       {
133         if(myproc()->sched_info.consecutive_run == 5)
134         {
135             myproc()->sched_info.consecutive_run = 0;
136             cprintf("Tick: Process with id %d is running in level %d\n", myproc()->pid, myproc()->sched_info.queue_level);
137             cprintf("Cpu num: %d\n",cpuid());
138             yield();
139         }
140       }
```

همانطور که مشاهده می کنید ما یک چیزی تعریف کردیم به عنوان تعداد دفعات متوالی اجرا که به این صورت است که اگر در نوبت گردشی به پنج رسید یعنی انگار پنج تیک خورده است و ما در این زمان yield را صدا می زنیم.

در نهایت هم به کمک تابع سطح کاربری که نوشتیم تست می کنیم که هر یک از پراسس ها به چه صورت اجرا می شوند.

تابع سطح کاربر را در فایل schedule_test نوشتیم که به شکل زیر است. یکسری پراسس به تعدادی که مشخص می کنیم می سازد و یک حلقه طولانی اجرا می کند.(به جای sleep از حلقه طولانی استفاده کردیم)

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

#define NUM_OF_PROCESSES 4
#define job_ITERATIONS 10000000

void job(int id)
{
    volatile long long x = 0;
    for (long long i = 0; i < job_ITERATIONS; i++)
        x += i;
}

int main(void)
{
    int pid;
    for (int i = 0; i < NUM_OF_PROCESSES; i++)
    {
        pid = fork();
        if (pid < 0)
        {
            printf(1, "Fork failed\n");
            exit();
        }
        else if (pid == 0)
        {
            // Child process
            job(i);
            exit();
        }
    }

    while (wait() > 0);

    printf(1, "All processes done\n\n");
    exit();
}
```

در انتها هم این فایل را برای دو تعداد پردازنده یک و دو اجرا می کنیم و نتیجه را مشاهده می کنیم.

برای تک پردازنده ای داریم:

```
Tick: Process 4 is running in 1 level
Cpu num: 0
Tick: Process 5 is running in 1 level
Cpu num: 0
Tick: Process 6 is running in 1 level
Cpu num: 0
Tick: Process 7 is running in 1 level
Cpu num: 0
Tick: Process 4 is running in 1 level
Cpu num: 0
Tick: Process 5 is running in 1 level
Cpu num: 0
Tick: Process 6 is running in 1 level
Cpu num: 0
Tick: Process 7 is running in 1 level
Cpu num: 0
Tick: Process 4 is running in 1 level
Cpu num: 0
```

همانطور که می بینید الگوی تکراری نوبت گردشی دارد رعایت می شود و همه پراسس ها روی پردازنده صفرم اجرا می شوند.

۱۹)

اما اگر تعداد پردازنده ها را به دو برگردانیم دو اتفاق می افتد. اول اینکه پراسس ها بین دو پردازنده تقسیم می شوند.(مانند شکل پایین) و همچنین سرعت پردازش بیشتر شد و کار ما زودتر برای همان پراسس ها به اتمام رسید.

لذا اگر هرپراسس را بخواهیم بررسی کنیم انگار ۴و ۶ و بعد ۵ و ۷ اجرا می شوند. اما جایی که ما پرینت می کنیم به ازای هر تیک کلاک است که مشخص می کند هر پردازنده در چه وضعیتی است.

```
Tick: Process 4 is running in 1 level
Cpu num: 1
Tick: Process 5 is running in 1 level
Cpu num: 0
Tick: Process 6 is running in 1 level
Cpu num: 1
Tick: Process 7 is running in 1 level
Cpu num: 0
Tick: Process 4 is running in 1 level
Cpu num: 1
Tick: Process 5 is running in 1 level
Cpu num: 0
Tick: Process 6 is running in 1 level
Cpu num: 1
Tick: Process 7 is running in 1 level
Cpu num: 0
```

پس مشخص کردیم که سطح اول که نوبت گردشی بود به درستی کار می کند.

نکته پایانی: تست دیگری هم برای دو پردازنده انجام دادیم به این صورت که این بار فرد تا پراسس تولید کردیم مثلا ۵ تا. وقتی اجرا کردیم مشاهده کردیم که یک ترتیب اجرا بین پردازنده ها می بینیم. یعنی فرض کنید سه پردازه و دو پردازنده داشته باشیم ترتیب پراسس و پردازنده به شرح زیر بود.

(1,1) (2,2) (3,1) (1,2) (2,1) (3,2), …

همانطور که می بینید در هر دور هر پراسس یکبار روی پردازنده یک و بار دیگر روی پردازنده دو اجرا می شود.

سطح دوم: اول، کوتاه ترین کار

```
418         if (p->sched_info.queue_level == SJF){
419           found_second_priority_process = 1;
420           shortest_proc_not_confident = p;
421           if(p->sched_info.burst_time < min_burst_time_sjf){
422               if(rand_value <= p->sched_info.confidence)
423               {
424                 shortest_proc = p;
425                 min_burst_time_sjf = p->sched_info.burst_time;
426               }
427           }
428         }
```

در ادامه ایف مرحله قبل، این دفعه با توجه به الگوریتم کوتاه ترین کار چک می کنیم که آیا مقدار تصادفی ای که ساختیم از اطمینان ما کمتر است یا خیر. اگر بود آن پراسس را انتخاب می کنیم در غیر این صورت سراغ پراسس بعدی می رویم.

مقدار تصادفی هم برگرفته از یک تابع دستی است که ما نوشتیم. این تابع براساس تیکی که در آن لحظه در آن قرار داریم کار می کند لذا تصادفی است.

سطح سوم: اولین ورود-اولین رسیدگی

```
430         if (p->sched_info.queue_level == FCFS){
431           found_third_priority_process = 1;
432           if(p->sched_info.arrival_time < min_last_arrival_time_fcfs){
433             last_proc = p;
434             min_last_arrival_time_fcfs = p->sched_info.arrival_time;
435           }
436         }
437       }
```

در این بخش ما یک خصیصه ای داریم به نام آخرین زمانی که یک پراسس به صف ready رسیده است. پس به ترتیبی که پراسس ها رسیده اند پراسس انتخاب شده و اجرا می شود که این همان الگوریتم اولین ورود اولین رسیدگی است.

برش دهی زمانی و ساز و کار افزایش سن

در ابتدا struct schedule info را تعریف می‌کنیم و به عنوان فیلد جدید struct proc قرار می‌دهیم.

- Queue level: صفی که در آن قرار دارد.
- Burst time: زمان اجرا که به طور پیش فرض برای همه ۲ در نظر گرفته شده.
- Confidence: اطمینان از اجرا که به طور پیش فرض ۵۰ است.
- Consecutive run: تعداد کوانتوم‌های اجراهای متوالی.
- Arrival time: کلاک در زمان اولین بار رسیدن این پردازه به ready queue.
- Last running time: کلاک در زمان آخرین بار رسیدن این پردازه به ready queue.
- Wait time: تعداد کلاک‌هایی که این پردازه در ready queue مانده.
- Last tick: آخرین کلاکی که wait time این پردازه آپدیت شده است.

همچنین نیاز به تغییراتی در struct cpu داریم که در تصویر می‌بینیم.

- Slice count: تعداد کلاک‌هایی که در صف فعالی ماندیم.
- Current queue: صف فعلی.

```c
// Per-CPU state
struct cpu {
  int slice_count;
  int current_queue;
```

```c
typedef struct schedule_info
{
    int queue_level;
    int burst_time;
    int confidence;
    int consecutive_run;
    int arrival_time;
    int last_running_time;
    int wait_time;
    int last_tick;
} schedule_info;
```

مقداردهی اولیه به فیلدهای جدید پردازه در تابع allocproc فایل proc.c انجام شده است.

```c
allocproc(void)
{
  if ((p->pid != 1) && (p->pid != 2))
  {
    p->sched_info.queue_level = FCFS; // Default queue is the third queue
  }
  else
  {
    p->sched_info.queue_level = ROUND_ROBIN; // For shel and init processes, Default queue is the first queue
  }
  p->sched_info.burst_time = 2;
  p->sched_info.confidence = 50;
  p->sched_info.wait_time = 0;
  p->sched_info.consecutive_run = 0;
  p->sched_info.arrival_time = ticks;
  p->sched_info.last_running_time = ticks;
  p->sched_info.last_tick = ticks;
```

برای مقداردهی فیلدهای جدید cpu به فایل main.c مراجعه می‌کنیم. از اینجا متوجه می‌شویم برخی فیلدهای cpu در تابع mpinit مقداردهی شده است.

```c
// main.c > main(void)
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // detect other processors
  lapicinit();     // interrupt controller
  seginit();       // segment descriptors
  picinit();       // disable pic
  ioapicinit();    // another interrupt controller
  consoleinit();   // console hardware
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  ideinit();       // disk
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  mpmain();        // finish this processor's setup
}
```

پس ما هم فیلدهای جدید را آنجا می‌بریم.

```c
// mp.c > ...
mpinit(void)
  for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
    switch(*p){
    if(ncpu < NCPU) {
      cpus[ncpu].apicid = proc->apicid;  // apicid may differ from ncpu

      // Added in Ex-3 to initialize new fields of cpu struct
      cpus[ncpu].slice_count = 0;
      cpus[ncpu].current_queue = ROUND_ROBIN;

      ncpu++;
    }
```

تغییرات تابع scheduler فایل proc.c:

در ابتدای حلقه بی‌نهایت، متغیرهایی برای انتخاب پردازه مطابق سیاست خواسته شده اضافه شده‌اند.

```c
C proc.c > scheduler(void)
367  scheduler(void)
375    for(;;) {
382      // variables to hold the chosen process based on each of the three scheduling algorithms
383      struct proc *rr_process = 0;
384      struct proc *shortest_proc = 0;
385      struct proc *shortest_proc_not_confident = 0;
386      struct proc *last_proc = 0;
387
388      long long min_burst_time_sjf = 1e9;
389      long long min_last_arrival_time_fcfs = 1e9;
390      long long min_last_running_time_rr = 1e9;
391
392      int found_first_priority_process = 0;
393      int found_second_priority_process = 0;
394      int found_third_priority_process = 0;
```

سپس روی تمام پردازه‌ها حلقه زده شده. در ابتدای این حلقه، شرط تمام شدن حد صبر یک پردازه را چک می‌کنیم و اگر سر رسیده باشد صفش را عوض می‌کنیم تا از starvation جلوگیری کنیم.

```c
C proc.c > scheduler(void)
367  scheduler(void)
375    for(;;) {
396      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
400        if (p->sched_info.wait_time >= WAITLIMIT) {
401          goto_lower_priority_queue(p);
402        }
403
404        if (ticks > p->sched_info.last_tick) {
405          p->sched_info.last_tick = ticks;
406          p->sched_info.wait_time++;
407        }
```

در این تابع اگر صف کنونی RR نباشد به صف پایین‌تر می‌رویم و مقداردهی متغیرهای مربوطه را نیز انجام می‌دهیم.

```c
C proc.c > ...
344  void
345  goto_lower_priority_queue(struct proc* p)
346  {
347    if (p->sched_info.queue_level != ROUND_ROBIN)
348    {
349      release(&ptable.lock);
350      setqueue(p->pid, p->sched_info.queue_level-1);
351      acquire(&ptable.lock);
352      p->sched_info.arrival_time = ticks;
353      p->sched_info.last_running_time = ticks;
354      p->sched_info.wait_time = 0;
355    }
356  }
```

بعد از آن در ادامه حلقه روی پردازه‌ها، برای هر سه سیاست گفته شده پردازه انتخاب می‌کنیم. تا بعدا بر اسا صف کنونی cpu تصمیم به اجرای یک کدام از آن‌ها بگیریم.

```
C proc.c > scheduler(void)
367  scheduler(void)
375    for(;;) {
396      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
409        if (p->sched_info.queue_level == ROUND_ROBIN) {
410          found_first_priority_process = 1;
411          if(p->sched_info.last_running_time < min_last_running_time_rr)
412          {
413            min_last_running_time_rr = p->sched_info.last_running_time;
414            rr_process = p;
415          }
416        }
417
418        if (p->sched_info.queue_level == SJF){
419          found_second_priority_process = 1;
420          shortest_proc_not_confident = p;
421          if(p->sched_info.burst_time < min_burst_time_sjf){
422              if(rand_value <= p->sched_info.confidence)
423              {
424                shortest_proc = p;
425                min_burst_time_sjf = p->sched_info.burst_time;
426              }
427          }
428        }
429
430        if (p->sched_info.queue_level == FCFS){
431          found_third_priority_process = 1;
432          if(p->sched_info.arrival_time < min_last_arrival_time_fcfs){
433            last_proc = p;
434            min_last_arrival_time_fcfs = p->sched_info.arrival_time;
435          }
436        }
437      }
```

بعد از حلقه روی پردازه‌ها، باید بر اساس صف کنونی cpu یمی را انتخب کنیم. در ابتدا چک می‌کنیم آیا اصلا پردازه‌ای در ready queue قرار دارد یا خیر. اگر قرار داشت به ادامه عملیات می‌پردازیم. حالا یکی از پراسس‌ها را برای اجرا انتخاب می‌کنیم.

```
C proc.c > scheduler(void)
367  scheduler(void)
375    for(;;) {
439      if (!found_first_priority_process  &&
440          !found_second_priority_process &&
441          !found_third_priority_process) {
442        // Did not found any runnable process
443      }
444      else {
445        p = 0;
446        if (mycpu()->current_queue == ROUND_ROBIN) {
447          if (found_first_priority_process) p = rr_process;
448        }
449
450        else if (mycpu()->current_queue == SJF) {
451            if (found_second_priority_process) {
452                if (shortest_proc == 0) {
453                  p = shortest_proc_not_confident;
454                }
455                else {
456                  p = shortest_proc;
457                }
458            }
459        }
460        else if (mycpu()->current_queue == FCFS) {
461            if (found_third_priority_process) p = last_proc;
462        }
```

پردازه انتخاب شده اجرا می‌شود و فیلدهای مورد نیاز آپدیت می‌شوند. همچنین اگر در صف SJF بوده باشیم و پردازه کنونی تمام شود نیاز به تولید عدد رندوم جدیدی داریم.

```
C proc.c > scheduler(void)
367   scheduler(void)
375     for(;;) {
444       else {
464         if (p != 0)
465         {
466           c->proc = p;
467           switchuvm(p);
468           p->state = RUNNING;
469
470           swtch(&(c->scheduler),p->context);
471           // record the last tick this process was RUNNING
472           p->sched_info.wait_time = 0;
473           p->sched_info.last_running_time = ticks;
474           if (p->sched_info.queue_level == SJF && p->state == ZOMBIE)
475             rand_value = generate_random_integer(2, 100);
476
477           switchkvm();
478           c->proc = 0;
479         }
480       }
481       release(&ptable.lock);
482
483     }
484   }
```

تابع تولید عدد رندوم بر اساس کلاک سیستم یک خروجی چندجمله‌ای می‌سازد که ضرایب آن اعداد اول هستند.

```
C proc.c > wait(void)
330   int
331   generate_random_integer(int degree, int max_num) {
332     int rand = 0;
333     int pw = ticks % max_num;
334     int coeff[8] = {1, 17, 7};
335     for (int i=1; i<=degree; i++) {
336       rand += coeff[i] * pw;
337       rand %= max_num;
338       pw *= ticks;
339       pw %= max_num;
340     }
341     return rand;
342   }
```

فرآیند time slicingدر فایل trap.c انجام گرفته. اگر شمارنده هر صف به حداکثر مقدار خود برسد آن را ریست کرده و به صف پایین‌تر می‌رویم، اگر نه شمارنده را به علاوه یک می‌کنیم.

```
C trap.c > trap(trapframe *)
37    trap(struct trapframe *tf)
49      switch(tf->trapno){
59        struct cpu* c = mycpu();
60        if (c->current_queue == ROUND_ROBIN && c->slice_count == 30)
61        {
62          c->current_queue = SJF;
63          c->slice_count = 0;
64        }
65        else if (c->current_queue == SJF && c->slice_count == 20)
66        {
67          c->current_queue = FCFS;
68          c->slice_count = 0;
69        }
70        else if (c->current_queue == FCFS && c->slice_count == 10)
71        {
72          c->current_queue = ROUND_ROBIN;
73          c->slice_count = 0;
74        }
75        else
76        {
77          c->slice_count++;
78        }
```

```
17  int
18  main(void)
19  {
20    kinit1(end, P2V(4*1024*1024)); // phys page allocator
21    kvmalloc();      // kernel page table
22    mpinit();        // detect other processors
23    lapicinit();     // interrupt controller
24    seginit();       // segment descriptors
25    picinit();       // disable pic
26    ioapicinit();    // another interrupt controller
27    consoleinit();   // console hardware
28    uartinit();      // serial port
29    pinit();         // process table
30    tvinit();        // trap vectors
31    binit();         // buffer cache
32    fileinit();      // file table
33    ideinit();       // disk
34    startothers();   // start other processors
35    kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
```

با توجه به راهنمایی سوال به فایل main.c می رویم و مشاهده می کنیم که تابعی به نام mpinit صدا زده شده است که در آن کار initialize کردن انجام می گیرد.

این تابع در فایل mp.c قرار دارد لذا به آن فایل رفته و در تابع مربوط مقادیر جدید را اضافه می کنیم.

```
 92  mpinit(void)
105    for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
106      switch(*p){
109        if(ncpu < NCPU) {
111
112          // Added in Ex-3 to initialize new fields of cpu struct
113          cpus[ncpu].slice_count = 0;
114          cpus[ncpu].current_queue = ROUND_ROBIN;
115
116          ncpu++;
117        }
```

همان مواردی که به استراکت پردازنده اضافه کرده بودیم را اینجا مقدار اولیه می دهیم.

(۲۱

به کمک بخش برش دهی زمانی ما کاری کردیم که لزوما تا وقتی صف اول پراسس دارد پردازنده روی آن نماند و به صف های دیگر هم اجازه اجرا بدهد.

یعنی در واقع ما گرسنگی بین صف ها را از بین بردیم اما مورد دیگری که با آن رو به رو هستیم گرسنگی بیت پراسس های هم صف است. یعنی مثلا یک پراسس در صف دو که کوتاه ترین کار اجرا می شود ممکن است هیچوقت نتواند پردازنده را بگیرد و اجرا شود لذا دچار گرسنگی می شود.

راه حل این موضوع ساز و کار سن است که در بخش بعد به آن می پردازیم و این مشکل را حل می کنیم.

(۲۲

ممکن است یک process مدت زیادی منتظر عمل I/O و یا کار دیگری باشد که در این زمان حالت sleeping قرار دارد و منطقی نیست که جزو زمان‌های انتظار در صف ready در نظرش بگیریم. حتی ممکن است زمان انتظارش برای I/O از ۸۰۰ کلاک بیشتر شود و ما در حالی اولویتش را بهبود بخشیم که کار قبلی‌اش تمام نشده.

فراخوانی های سیستمی مورد نیاز

Printinfo

```
193  int sys_printinfo(void) {
195      char name_title [100];
196      char pid_title [100];
197      char state_title [100];
198      char queue_title [100];
199      char wait_title [100];
200      char confidence_title [100];
201      char burst_title [100];
202      char consecutive_title [100];
203      char arrival_title [100];
204      align_word("Name", 16, name_title);
205      align_word("PID", 4, pid_title);
206      align_word("State", 11, stat
207      align_word("Queue", 8, queue
208      align_word("Wait-time", 11, wait_title);
209      align_word("Confidence", 12, confidence_title);
210      align_word("Burst-time", 12, burst_title);
211      align_word("Consecutive-run", 19, consecutive_title);
212      align_word("Arrival", 9, arrival_title);
213      cprintf("%s %s %s %s %s %s %s %s %s \n",
214                  name_title,
215                  pid_title,
216                  state_title,
217                  queue_title,
218                  wait_title,
219                  confidence_title,
220                  burst_title,
221                  consecutive_title,
222                  arrival_title);
223      cprintf("-----------------------------------------------------------------------------
224
```

char wait_title[100]

برای چاپ اطلاعات هم طبق خواسته صورت پروژه ، اطلاعات مربوطه را به صورت جدول پرینت میکنیم.

```
193  int sys_printinfo(void) {
227      for (p = ptable_array; p < &ptable_array[NPROC]; p++) {
228          if (p->state != UNUSED) {
229              char _name[100];
230              align_word(p->name, 16, _name);
231
232              char __pid[100];
233              myitos(p->pid, __pid, 10);
234              char _pid[100];
235              align_word(__pid, 4, _pid);
236
237              char _state[100];
238              align_word(state_str[p->state], 11, _state);
239
240              char __queue[100];
241              myitos(p->sched_info.queue_level, __queue, 10);
242              char _queue[100];
243              align_word(__queue, (int)100, _queue);
244
245              char __wait[100];
246              myitos(p->sched_info.wait_time, __wait, 10);
247              char _wait[100];
248              align_word(__wait, 11, _wait);
249
250              char __conf[100];
251              myitos(p->sched_info.confidence, __conf, 10);
252              char _conf[100];
253              align_word(__conf, 12, _conf);
254
255              char __burst[100];
256              myitos(p->sched_info.burst_time, __burst, 10);
257              char _burst[100];
258              align_word(__burst, 12, _burst);
259
260              char __cons[100];
261              myitos(p->sched_info.consecutive_run, __cons, 10);
262              char _cons[100];
263              align_word(__cons, 19, _cons);
264
265              char __arr[100];
266              myitos(p->sched_info.arrival_time, __arr, 10);
267              char _arr[100];
268              align_word(__arr, 9, _arr);
269
270              cprintf("%s %s %s %s %s %s %s %s %s \n",
271                _name,
272                _pid,
273                _state,
274                _queue,
275                _wait,
276                _conf,
277                _burst,
278                _cons,
279                _arr);
280          }
281      }
282      cprintf("----------------------------------------------------------------------------
283      releaseptable();
```

SetBurstConf

در این تابع هم pid مورد نظر را گرفته و confidence و burst time آن را به عدد های خواسته شده تغییر می دهیم .

```
int sys_setburstconf(void) {
    int pid, burst_time, confidence;

    if (argint(0, &pid) < 0 || argint(1, &burst_time) < 0 || argint(2, &confidence) < 0) {
        cprintf("Kernel: Could not extract all of the arguments for setburstconf\n");
        return -1;
    }

    struct proc *p;
    struct proc *ptable_array = getptable();

    for (p = ptable_array; p < &ptable_array[NPROC]; p++) {
        if (p->pid == pid) {
            p->sched_info.burst_time = burst_time;
            p->sched_info.confidence = confidence;
            releaseptable();
            return 0;
        }
    }

    releaseptable();
    return -1;
}
```

SetQueue

```
int sys_setqueue(void) {
    int pid, level;

    if (argint(0, &pid) < 0 || argint(1, &level) < 0) {
        cprintf("Kernel: Could not extract all of the arguments for setqueue\n");
        return -1;
    }

    struct proc *ptable_array = getptable();

    struct proc *p;
    int success = -1;

    for (p = ptable_array; p < &ptable_array[NPROC]; p++) {
        if (p->pid == pid) {
            p->sched_info.queue_level = level;
            success = 0;
            break;
        }
    }

    releaseptable();
    return success;
}
```

در تابع SetQueue هم pid مورد نظر را گرفته و صف پردازه را به صف مورد نظر تغییر میدهیم .

Test_systemcalls code

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

#define NUM_OF_PROCESSES 10
#define job_ITERATIONS 100000000

void job(int id)
{
    volatile long long x = 0;
    for (long long i = 0; i < job_ITERATIONS; i++)
    {
        if(i == 100000 && id == 8)
        {
            printinfo();
        }
        x += i;
    }
}
int main(void)
{
    int pid;
    for (int i = 0; i < NUM_OF_PROCESSES; i++)
    {
        pid = fork();
        if (pid < 0)
        {
            printf(1, "Fork failed\n");
            exit();
        }
        else if (pid == 0)
```

```c
        else if (pid == 0)
        {
            if (getpid() % 2 == 0)
            {
                printf(1, "Set queue called for process: %d\n" , getpid());
                setqueue(getpid(), 2);
            }
            if (getpid() % 4 == 0)
            {
                printf(1, "Set burst confidence called for process: %d\n" , getpid());
                setburstconf(getpid(),1,70);
            }
            // Child process
            job(getpid());
            exit();
        }


    }
    while (wait() > 0);

    printf(1, "All processes done\n\n");
    exit();
}
```

این کد در یک به تعداد دلخواه (اینجا ۱۰) بار اجرا میشود و هر سری fork را صدا میزند و یک فعالیت طولانی برای هر پردازه در نظر میگیرد تا بتوانیم اعمال صف ها را به دقت بررسی کنیم . برای پردازه های زوج صف آنها را تغییر داده و به ۲ منتقل میکنیم . برای پردازه های مضرب ۴ burst time انها را از ۲(دیفالت) به ۱ تغییر داده و confidence آنها را نیز از ۵۰ (دیفالت) به ۷۰ تغییر میدهیم .

Test_systemcalls result

همان طور که توضیح داده شد ، پردازه های زوج از صف ۳ به صف ۲ منتقل شدند . و پردازه های مضرب ۴ نیز burst time و confidence آنها دچار تغییر شد .(در جدول نهایی هم تغییر این مقادیر مشهود است .)

State, wait time , consecutive run , arrival time هم میتواند در هر بار اجرای برنامه متفاوت باشد ، پس صحبت خاصی درباره ی آنها نمی توان کرد . ولی مثلا مشاهده میکنیم که پردازه های در صف های مختلفی قرار دارند که نشان میدهد الگوریتم aging درست کار میکند و انتقال پردازه ها از صف با اولویت کمتر به صف با اولویت بیشتر به درستی انجام شده است .

```
$ test_systemcalls
Set queue called for process: 4
Set burst confidence called for process: 4
proccess 6 goes to level 2 from level 3
proccess 7 goes to level 2 from level 3
proccess 8 goes to level 2 from level 3
proccess 9 goes to level 2 from level 3
proccess 10 goes to level 2 from level 3
proccess 11 goes to level 2 from level 3
proccess 12 goes to level 2 from level 3
proccess 13 goes to level 2 from level 3
proccess 6 goes to level 1 from level 2
Set queue called for process: 6
proccess 7 goes to level 1 from level 2
proccess 8 goes to level 1 from level 2
proccess 9 goes to level 1 from level 2
proccess 10 goes to level 1 from level 2
proccess 11 goes to level 1 from level 2
proccess 12 goes to level 1 from level 2
Set queue called for process: 8
SetSet queue called for process: 10
Set queue called for process: 12
Set burst confidence called for process: 12
proccess 6 goes to level 1 from level 2
proccess 8 goes to level 1 from level 2
 burst confidence called for process: 8
```

```
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            2                   13
test_systemcall  3    SLEEPING    3        0           50           2            0                   4158
test_systemcall  6    RUNNABLE    1        32          50           2            0                   7243
test_systemcall  7    RUNNABLE    1        1           50           2            0                   5823
test_systemcall  8    RUNNING     1        33          70           1            0                   7301
test_systemcall  9    RUNNABLE    1        31          50           2            0                   5829
test_systemcall  10   RUNNABLE    2        14          50           2            0                   5830
test_systemcall  11   RUNNABLE    1        2           50           2            0                   5831
-------------------------------------------------------------------------------------------------------------
All processes done
```