گزارش پروژهٔ سوم آزمایشگاه سیستم عامل

به تدریس دکتر کارگهی

محمد امین یوسفی

مبینا مهرآذر

متین نبی‌زاده

پاییز 1402

اطلاعات مربوط به گیت‌هاب، در آخرین صفحهٔ گزارش، درج شده است.

زمان‌بندی در xv6

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

پیش از هر چیز، بایستی در مورد ساز و کار اجرا شدن فرآیند گفته شده در صورت این سؤال، توضیحاتی را ارائه کنیم. می‌دانیم که هر هسته‌ای که شروع به کار می‌کند، تابع mpmain را صدا می‌زند. این تابع نیز در انتها تابع scheduler را صدا می‌زند که باعث شروع به کار کردن زمان‌بند مربوط به هر هسته می‌شود. تابع ذکر شده در ptable نیز به دنبال پردازهٔ قابل اجرا می‌گردد و در صورت یافتن چنین پردازه‌ای (که RUNNABLE باشد)، پس از تغییر حافظه به حافظهٔ پردازه توسط تابع switchuvm، با استفاده از تابع swtch که در زبان اسمبلی پیاده‌سازی شده، عملیات تعویض متن صورت می‌گیرد. این تابع، رجیسترهای متن[1] قدیمی (cpu::struct context *scheduler) را در آدرس مربوط به همان context ذخیره می‌کند و رجیسترهای مربوط به context جدید را از آدرس مربوط به همان متن، بازیابی می‌کند که با این کار، شمارندهٔ برنامه نیز به مقدار متناظر آن در متن جدید تبدیل می‌شود و به این ترتیب، اجرای پردازهٔ جدید، آغاز می‌شود.

- در سه حالت، ممکن است که یک پردازهٔ در حال اجرا، تابع ()sched را فراخوانی کند:

- پردازه با استفاده از فراخوانی سیستمی exit، پردازنده را ترک کند.
- پردازه با استفاده از فراخوانی سیستمی sleep، به حالت SLEEPING درآید.
- پس از وقفهٔ ایجاد شده توسط تایمر، پردازه مجبور به خروج از پردازنده شود که در این حالت، تابع yield فراخوانی شده و در آن تابع نیز تابع sched فراخوانی می‌شود.

در نهایت، در تابع sched، مجددا عملیات تعویض متن صورت می‌پذیرد و در این حالت متنی که در ساختار cpu[2] است، بازیابی شده و متن مربوط پردازه در حال اجرا ذخیره می‌شود. پس از بازیابی متن مربوط به program counter، scheduler به خط 2782 اشاره می‌کند و باعث ادامه کار scheduler می‌شود. در واقع پردازه‌ای که هر هسته را آماده به کار می‌کند، هیچ وقت از تابع scheduler خارج نمی‌شود و فقط با عملیات تعویض متن از پردازنده خارج می‌شود و با اجرای تابع sched، دوباره به ادامه کار خود می‌پردازد.

---

[1] Context
[2] struct context *scheduler

زمان‌بندی

2) صف پردازه‌هایی که تنها منبعی که برای اجرا کم دارند پردازنده است، صف آماده یا صف اجرا نام دارد. در xv6، صف آمادهٔ مجزا وجود نداشته و از صف پردازه‌ها بدین منظور استفاده می‌گردد.

- در زمان‌بند کاملاً منصف در لینوکس، صف اجرا چه ساختاری دارد؟

در لینوکس، با توجه به استفاده از الگوریتم CFS، ساختار صف اجرا به صورت یک درخت قرمز - سیاه است؛ برخلاف الگوریتم‌های دیگر که از داده‌ساختارهایی به صورت FIFO بهره می‌برند. از آن‌جایی که درخت قرمز - سیاه یک درخت جستجوی دودویی خودتراز[3] با عملیات های درج و استخراج بهینه است (و می‌دانیم که این دو عملیات، پیچیدگی زمانی‌ای از مرتبهٔ زمانی O(logN) دارند)، به عنوان یک داده‌ساختار برای مورد استفاده قرار گرفتن به عنوان صف آماده در یک الگوریتم زمان‌بند[4]، بسیار مناسب است. در این الگوریتم، هر رأس درخت مذکور نشانگر یک کار[5] است که می‌بایست زمان‌بندی شود و نتیجتا، کلیت درخت، مانند هر صف اجرای دیگری، نشانگر یک تسلسل زمانی[6] است برای اجرای کار یا پردازه‌ها.

همان‌طور که گفته شد، رئوس این درخت بر اساس مدت زمانی که در پردازنده وجود داشته اند (متغیر vruntime)، اندیس‌گذاری می‌شوند. به عنوان یک نتیجهٔ (تقریبا) بدیهی، می‌توان گفت که کار یا پردازه‌های موجود در زیردرخت‌های سمتِ چپ هر رأس، مدت زمان کمتری موجود بوده اند (و مقدار متغیر vruntime برای آنها، کمتر است). پس، خوانندهٔ محترم از پیش، نتیجه‌گیری کرده است که کار یا پردازهٔ موجود به عنوان چپ‌ترین رأس این درخت، کمترین مقدار vruntime را به خود اختصاص داده و در ابتدای صف اولویت، قرار دارند.

---

[3] Self-balancing
[4] Scheduler
[5] Task
[6] Timeline

در مورد پیاده‌سازی آن در لینوکس نیز می‌توان به نمونه‌تعریف[7] task_struct اشاره کرد، که درون خود، یک ساختار دیگر به نام sched_entity دارد که وظیفهٔ آن، نگهداری اطلاعات مربوط به زمان‌بندی پردازه‌ها (یا به عبارت دقیق‌تر، ذخیره‌سازی متغیر vruntime به ازای هر کار یا پردازه) است. همچنین، خود درخت قرمز-سیاه نیز یک نمونه‌تعریف به نام cfs_rq دارد که با استفاده از متغیر rb_root، به ریشهٔ این درخت اشاره می‌کند. همچنین، هر یک از رئوس نیز اشاره‌گری به ریشه و دو فرزند آن دارند اما خوانندهٔ محترم، در نظر دارد که نظر به شیوهٔ پیاده‌سازی عملیات‌های مختص به درخت قرمز-سیاه، تمامی برگ‌های این درخت، nil هستند!

3) همانطور که در پروژه اول مشاهده شد، هر هستهٔ پردازنده در xv6 یک زمان‌بند دارد. در لینوکس نیز به همین شکل است.

- این دو سیستم‌عامل را از منظر مشترک یا مجزا بودن صف‌های زمان‌بندی بررسی نمایید و یک مزیت و یک نقص صف مشترک نسبت به صف مجزا را بیان کنید.

در سیستم‌عامل xv6، فقط از یک صف زمان‌بندی برای همه پردازنده‌ها به طور مشترک استفاده می‌شود؛ گواهمان برای این گفته نیز ساختار ptable است که در آن، یک صف از structها proc نگه‌داری می‌شود. همچنین، برای جلوگیری از خرابی حاصل از تغییرات هم‌زمان چند پردازنده بر روی این صف، از یک spinlock استفاده می‌شود؛ بدین صورت که هنگام دسترسی به ptable.proc، ابتدا بایستی ptable.lock را اخذ[8] کنیم و پس از انجام تغییرات در آن، آن را رها[9] کنیم. اما همانطور که از پیش می‌دانیم، در لینوکس، هر پردازنده صف زمان‌بندی مخصوص خودش را دارد و پردازه‌ها به صورت مجزا در آنها قرار می‌گیرند.

[7] Instance
[8] Acquire
[9] Release

- **از مزایا و معایب صف‌های زمان‌بندی اشتراکی، می‌توان به ترتیب، به موارد زیر اشاره کرد:**

- **فراهم آوردن امکان توزین بار**؛ چه بر خوانندهٔ گرامی، مبرهن است که صف زمان‌بندی اشتراکی، به سیستم‌عامل این امکان را می‌دهد که بار پردازشی را به سهولت بیشتری بتواند در میان چند هسته، تقسیم کند؛ به عبارت دیگر، این قابلیت، ممکن می‌شود که پردازه‌ها، به طور پویا در میان هسته‌ها تقسیم شوند که این فرآیند تقسیم نیز طبیعتا بایستی بر اساس بار فعلی و آزاد بودن یا نبودن منابع، صورت گیرد.

- **محتمل شدن وقوع اختلاط**[10]؛ به ویژه در محیط‌های به شدت چندریسه‌ای / چندهسته‌ای. این اختلاط بالقوه، ممکن است سربار همگامی به بار آورده و نتیجتا، موجب کاهش کارایی سیستم شود.

- **همچنین، از مزایا و معایب صف‌های زمان‌بندی مجزا، می‌توان به ترتیب، به موارد زیر اشاره کرد:**

- **اختلاط کمتر**؛ از بیان چگونگی کاهش اختلاط ضمن پیاده‌سازی صف‌های زمان‌بندی به صورت مجزا، سخن کوتاه کرده و تنها به ذکر این نکته، بسنده می‌کنیم که در این شیوهٔ پیاده‌سازی، از آن‌جایی که هر هسته به صورت مستقل عمل می‌کند، کمتر نیاز به ایجاد مداوم همگامی میان هسته‌ها وجود دارد و در نتیجه، همین امر، موجب بهبود کارایی می‌گردد.

- **عدم توازن بار**؛ به صورتی که ممکن است یک / چند هسته، مورد فرواستفادگی[11] قرار گیرد / گیرند اما بالعکس، یک / چند هستهٔ دیگر، مورد فرااستفادگی[12] واقع شوند.

---

[10] Contention
[11] Underutilization
[12] Overutilization

4) در هر اجرای حلقه، ابتدا برای مدتی وقفه فعال می‌گردد. علت این امر چیست؟ آیا در سیستم تک‌هسته‌ای به آن نیاز است؟

زمانی که قفل ptable فعال می‌شود، تمامی وقفه‌ها به وسیله تابع pushcli غیرفعال می‌شوند. نمی‌بایست غافل شد از احتمال اینکه پردازنده، در حالتی قرار بگیرد که تعدادی از پردازه‌های آن، در انتظار پایان عملیات I/O باشند و هیچ کدام از پردازه‌های دیگر نیز در حالت RUNNABLE نباشند. در این حالت، هیچ پردازهٔ دیگری قابلیت اجرا ندارند و اگر وقفه‌ها نیز هیچ وقت فعال نشوند، پس از پایان عملیات ورودی/خروجی، نمی‌توان پردازه‌های مربوطه را به حالت RUNNABLE تغییر داد که بتوانند اجرا شوند؛ در نتیجه، کل سیستم فریز می‌شود. به همین دلیل است که در این حلقه برای مدت کوتاهی (تا پیش از قفل کردن ptable)، وقفه‌ها فعال می‌شوند تا در صورت نیاز، بتوانیم حالت پردازه‌ها را تغییر دهیم.

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 برخی از سیستم‌عامل‌های مدرن (از جمله لینوکس، ویندوز، مک‌اواس و...)، مدیریت وقفه‌ها در دو سطح صورت می‌گیرد:

- سطح اول مدیریت وقفه‌ها (FLIH[13])؛ که از آن، تحت عنوان رسیدگی‌کنندهٔ سخت (سریع) نیز یاد می‌شود. در این سطح مدیریت، در پاسخ به هر وقفه، یک تعویض متن[14] رخ می‌دهد و سپس، کد وقفهٔ مربوطه، بارگذاری شده و به آن، رسیدگی می‌شود. وظیفهٔ اصلی این سطح مدیریتی، رسیدگی **سریع** به وقفه‌ها، در کنار ذخیره‌سازی اطلاعات مختص به پلتفرم (که تنها در زمان رسیدن وقفه‌ها در دسترس هستند) است که در نتیجه، بتواند سطح مدیریتی دوم (که در قسمت بعدی معرفی می‌شود) را زمان‌بندی کند. لازم به ذکر است که این سطح مدیریتی، ممکن است باعث به وجود آمدن جیتر شده، یا حتی یک وقفه را نادیده گرفته و موجب پوشاندن آن شود[15]. در سیستم‌های بی‌درنگ[16]، کاهش دادن مقدار این جیتر، امری حیاتی است؛ زیرا واضح است که در مورد اینگونه سیستم‌ها، بایستی تضمین شود که یک قطعه کد مشخص، در مدت زمانی مشخص و از پیش تعیین شده، اجرا می‌شود. به همین دلیل و به منظور کاهش جیتر در این سطح مدیریتی، یک راهکار مفید آن است که زمان اجرای آن تا حد ممکن، کمینه شود و این امر، میسر نمی‌شود مگر آنکه محتوای اجرا شونده، هر چه بیشتر، به سطح بعدی مدیریت منتقل شود.

- سطح دوم مدیریت وقفه‌ها (SLIH[17])؛ که از آن، تحت عنوان رسیدگی‌کنندهٔ نرم (کند) نیز یاد می‌شود. از آنجایی که اجرای این سطح مدیریتی ممکن است زمان‌بر باشد، معمولا فرآیند زمان‌بندی آن، مشابه پردازه‌ها و ریسه‌ها صورت می‌گیرد. لازم به ذکر است که در این سطح مدیریتی، به ازای هر رسیدگی‌کننده، بایستی یک ریسه ساخته شود.

---

[13] First-Level Interrupt Handlers
[14] Context Switch
[15] Mask
[16] Real-time
[17] Second-Level Interrupt Handlers

- مدیریت وقفه‌ها در صورتی 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؟

در ابتدا، بایستی به طور مختصر، در این مورد سخن بگوییم که اصلا چگونه یک‌سری وقفه، می‌توانند موجب گرسنگی برخی پردازه‌ها شوند و سپس در ادامه، راه‌حل‌هایی را بررسی کنیم و ببینیم که آیا برای برون‌رفت از وضعیتِ ناگوار وجود پردازه‌های گرسنه، راهکاری وجود دارد؟

به طور کلی، اگر قرار باشد علت گرسنگی یک (یا چند) پردازه، رسیدن وقفه(ها) باشد، می‌بایست یکی از سناریوهای زیر اتفاق افتاده باشد:

1. **وارونگی اولویت**[18] پدید آمده باشد؛ به صورتی که پردازهٔ پُراولویتی مانند روتین رسیدگی به وقفه‌ها[19]، منابع مورد نیاز پردازه‌های کم‌اولویت‌تر را برای مدتی طولانی، به خود اختصاص داده باشد.

2. **طوفان وقفه**[20] رخ داده باشد؛ یعنی اینکه نرخ وقوع وقفه‌ها در واحد زمانی معین، بیش از مقداری مشخص باشد.

3. **زمان‌بندی وقوع وقفه**، معیوب باشد؛ به طوری که اگر تایمر رسیدگی‌کننده به وقفه‌ها به طور مناسبی تنظیم نشده باشد، طبیعی است که احتمال گرسنه ماندن پردازه‌ها، بالا خواهد بود.

4. **وقفه‌های سنگین** ایجاد شده باشند؛ یعنی آنکه سربار رسیدگی به وقفه‌ها، بالا باشد یا تاخیر چشم‌گیری حین سرویس‌دهی به وقفه‌ها، وجود داشته باشد.

---

[18] Priority Inversion
[19] Interrupt Service Routine
[20] Interrupt Storm

در تمامی حالات فوق‌الذکر، احتمال گرسنه ماندن پردازه‌های کم‌اولویت‌تر، بسیار زیاد است. برای رفع مشکل ناشی از این گرسنگی پردازه‌ها، راهکارهای بسیار زیادی وجود دارد که در این قسمت، به برخی از آنها به صورت خلاصه، اشاره می‌شود:

1. **برقراری سیاست‌های زمان‌بندی منصفانه**؛ همانند سیاست CFS که پیش‌تر، به آن اشاره شده بود.

2. **افزایش عمر پردازه‌ها**؛ که موجب حصول اطمینان از نماندن پردازهٔ کم‌اولویت در همان اولویت کم می‌شود.

3. **تخصیص منابع تضمین شده**؛ که بدین شیوه، از تخصیص یک سهم حداقلی و بدون کم و کاست از منبعی مشخص، اطمینان حاصل می‌شود.

4. **برقراری زمان‌بندی بازخورد-محور**؛ که در آن، شیوه و سیاست زمان‌بندی بر اساس تاریخچهٔ تمامی وقایع و زمان پردازش‌های پیشین پردازه‌های موجود در صف، تعیین می‌شود.

5. **توزین بار**؛ به طوری که در سیستم‌های چندهسته‌ای، بار پردازشی میان هسته‌های جداگانه، تقسیم شود تا پردازه‌ای از مورد پردازش گرفتن، مغفول نماند.

زمانبندی بازخوردی چند سطحی

داده ساختاری به نام MFQ_info در proc.h قرار داده شده است که همهٔ اطلاعات لازم برای انجام این الگوریتم زمان‌بندی توسط سیستم‌عامل را ذخیره کند.

```
struct MFQ_info  {
  enum MFQ_Types queue_type; // Process queue
  int last_exec_time;        // Last time process was run
  struct p_info bjf;         // Best-Job-First scheduling info
  int arrive_lcfs_queue_time;
};
```

- **در ادامه، هر یک از متغیرهای این قطعه کد، به طور مختصر، معرفی و ارائه می‌شوند:**

همان‌طور که نشان داده شده است، متغیر اولیه، نشان می‌دهد که صف پردازه از چه نوع است. همچنین، متغیر last_exec_time زمان اخرین پردازش را ذخیره می‌کند. این متغیر در موارد افزایش عمر پردازه، برای چک کردن شرط لزوم این افزایش عمر به کار می‌رود و مقدار آن، اولین بار هنگام fork تعیین می‌شود و همواره در تابع scheduler مقدار آن آپدیت می‌شود. متغیر بعدی (یعنی bjf) نیز اطلاعات لازم برای محاسبه رتبه[21] در الگوریتم BJF را ذخیره می‌کند. متغیر arrive_lcfs_queue_time نیز هنگام fork شدن، مقداردهی می‌شود. توجه شود که این متغیر، زمان ورود به صف دوم را ذخیره می‌کند؛ بنابراین در صورتی که از دستور transfer_queue استفاده شده باشد و صف به صف دوم تغییر یافته باشد، این متغیر مقداردهی می‌شود.

کد تابع fork:

[21] Rank

```
190  int fork(void)
191  {
192    int i, pid;
193    struct proc *np;
194    struct proc *curproc = myproc();
195    // Allocate process.
196    if ((np = allocproc()) == 0)
197    {
198      return -1;
199    }
200    // Copy process state from proc.
201    if ((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0)
202    {
203      kfree(np->kstack);
204      np->kstack = 0;
205      np->state = UNUSED;
206      return -1;
207    }
208    np->sz = curproc->sz;
209    np->parent = curproc;
210    *np->tf = *curproc->tf;
211    // Added by me
212    np->generated_time = ticks / 100;
213    // Clear %eax so that fork returns 0 in the child.
214    np->tf->eax = 0;
215    for (i = 0; i < NOFILE; i++)
216      if (curproc->ofile[i])
217        np->ofile[i] = filedup(curproc->ofile[i]);
218    np->cwd = idup(curproc->cwd);
219    safestrcpy(np->name, curproc->name, sizeof(curproc->name));
220    pid = np->pid;
221    acquire(&ptable.lock);
222    np->state = RUNNABLE;
223    acquire(&tickslock);
224    np->mfq_info.last_exec_time = ticks;
225    np->mfq_info.bjf.arrival_time = ticks;
226    np->mfq_info.bjf.process_size = np->sz;
227    np->mfq_info.arrive_lcfs_queue_time = ticks;
228    release(&tickslock);
229    release(&ptable.lock);
230    transfer_process_queue(np->pid, LCFS);
231    return pid;
232  }
```

کد تابع scheduler:

```c
void scheduler(void)
{
  struct proc *p;
  struct proc *last_proc_scheduled_RR = &ptable.proc[NPROC - 1];
  struct proc *last_proc_scheduled_LCFS = 0;
  struct cpu *c = mycpu();
  c->proc = 0;
  for (;;)
  {
    // Enable interrupts on this processor.
    struct proc *temp_p;
    for (temp_p = ptable.proc; temp_p < &ptable.proc[NPROC]; temp_p++)
    {
      if (strcmp(temp_p->name, "sh"))
      {
        transfer_process_queue(temp_p->pid, RR);
      }
    }

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

    p = roundrobin(last_proc_scheduled_RR);
    if (p)
    {
      last_proc_scheduled_RR = p;
    }
    else
    {
      if (last_proc_scheduled_LCFS != 0 && last_proc_scheduled_LCFS->state == RUNNABLE)
        p = last_proc_scheduled_LCFS;
      else if (last_proc_scheduled_LCFS == 0 || last_proc_scheduled_LCFS->state != RUNNING)
      {
        p = lcfs();
      }
      if (p)
      {
        last_proc_scheduled_LCFS = p;
      }
      else
      {
        p = best_job_first();
        if (!p)
        {
          release(&ptable.lock);
          continue;
        }
      }
```

0 ⚠ 0 0

```
435         {
436           p = best_job_first();
437           if (!p)
438           {
439             release(&ptable.lock);
440             continue;
441           }
442         }
443       }
444
445       // Switch to chosen process.  It is the process's job
446       // to release ptable.lock and then reacquire it
447       // before jumping back to us.
448       c->proc = p;
449       switchuvm(p);
450       p->state = RUNNING;
451
452       p->mfq_info.last_exec_time = ticks;
453       p->mfq_info.bjf.executed_cycle += 0.1f;
454
455       swtch(&(c->scheduler), p->context);
456       switchkvm();
457
458       // Process is done running for now.
459       // It should have changed its p->state before coming back.
460       c->proc = 0;
461       release(&ptable.lock);
462     }
463   }
```

تابع ageproc، به فایل proc افزوده شده است که با اضافه شدن مقدار ticks در وقفهٔ مربوط به تایمر موجود در فایل trap.c، فراخوانی می‌شود. ساز و کار افزایش عمر به این نحو است که پردازه‌هایی که بعد از گذشت 8000 سیکل زمانی، همچنان در صف دوم باشند، وارد صف اول می‌شوند و transfer_queue روی آن‌ها فراخوانی می‌شود. در فایل proc.c d، تابع scheduler وجود دارد که بعد از هربار انجام شدن زمان‌بندی سیستم‌عامل توسط آن، مقدار p->mfq_info.last_run آپدیت می‌شود.

خالی از لطف نیست ذکر این مطلب که در این فرآیند، چندین تابع xv6 نیز آپدیت شده‌اند؛ از جمله توابع scheduler و allocproc که برای تغییر الگوریتم زمان‌بندی، دچار تغییراتی شده‌اند.

تابع ageproc:

```
void ageproc(int ticks)
{
  acquire(&ptable.lock);
  for (int i = 0; i < NPROC; i++)
  {
    if (ptable.proc[i].state == RUNNABLE && ptable.proc[i].mfq_info.queue_type != RR)
    {
      if (WAITING_CYCLES_THRESHOLD < ticks - ptable.proc[i].mfq_info.last_exec_time)
      {
        release(&ptable.lock);
        transfer_process_queue(ptable.proc[i].pid, RR);
        acquire(&ptable.lock);
      }
    }
  }
  release(&ptable.lock);
}
```

در این تابع ابتدا RUNNABLE بودن و از صف RR نبودن را چک کردیم و سپس شرط اعمال aging و در نهایت عملیات مورد نیاز برای انجام aging را انجام می‌دهیم.

از جمله تغییرات تابع allocproc، می‌توان به تعیین ضریب مربوط به محاسبه رتبه در bjf (به مقدار یک)، اشاره کرد. این رتبه، تا وقتی که کاربر این ضرایب را تغییر دهد و به مقدار جدید آپدیت شوند، ثابت می‌ماند.

تابع allocproc:

```
84  static struct proc *
85  allocproc(void)
86  {
87    struct proc *p;
88    char *sp;
89    acquire(&ptable.lock);
90    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
91      if (p->state == UNUSED)
92        goto found;
93    release(&ptable.lock);
94    return 0;
95
96  found:
97    p->state = EMBRYO;
98    p->pid = nextpid++;
99    release(&ptable.lock);
100   // Allocate kernel stack.
101   if ((p->kstack = kalloc()) == 0)
102   {
103     p->state = UNUSED;
104     return 0;
105   }
106   sp = p->kstack + KSTACKSIZE;
107   // Leave room for trap frame.
108   sp -= sizeof *p->tf;
109   p->tf = (struct trapframe *)sp;
110   // Set up new context to start executing at forkret,
111   // which returns to trapret.
112   sp -= 4;
113   *(uint *)sp = (uint)trapret;
114   sp -= sizeof *p->context;
115   p->context = (struct context *)sp;
116   memset(p->context, 0, sizeof *p->context);
117   p->context->eip = (uint)forkret;
118
119   memset(&p->mfq_info, 0, sizeof(p->mfq_info));
120
121   p->mfq_info.bjf.priority = BJF_PRIORITY_DEF;
122   p->mfq_info.bjf.priority_ratio = 1;
123   p->mfq_info.bjf.arrival_time_ratio = 1;
124   p->mfq_info.bjf.executed_cycle_ratio = 1;
125   p->mfq_info.bjf.process_size_ratio = 1;
126
127   return p;
128 }
```

- زمان‌بند نوبت-گردشی[22] (RR)

[22] Round-Robin

تابعی به اسم roundrobin نوشته شده که به عنوان ورودی، آخرین پردازهٔ زمان‌بندی‌شده توسط این الگوریتم را می‌گیرد تا سراغ پردازه‌های بعد این پردازه در صف برود و اگر پردازه RUNNABLE را یافت، این پردازه را برای زمان‌بندی، به عنوان خروجی بدهد.

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

    if (p->state == RUNNABLE && p->mfq_info.queue_type == RR)
      return p;

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

- **زمان‌بند آخرین کار، اولین سرویس[23] (LCFS)**

این تابع به نام lcfs در proc.c قرار دارد که برحسب زمان ورود پردازه‌ها، از بین پردازه‌هایی که در وضعیت RUNNABLE قرار دارند(یعنی هنوز کار این پردازه تمام نشده است) و در صف مربوط به سیاستِ آخرین کار، اولین سرویس قرار ندارند، پردازه‌ای را می‌یابد که زمان ورودی آن، دیرتر از بقیه باشد و آن را برای زمان‌بندی، به عنوان خروجی می‌دهد.

[23] Last Come First Served

```
struct proc *lcfs()
{
  struct proc *result = 0;
  int max_arrival_time = -1;

  struct proc *p;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state != RUNNABLE || p->mfq_info.queue_type != LCFS)
      continue;
    if (max_arrival_time < p->mfq_info.arrive_lcfs_queue_time)
    {
      max_arrival_time = p->mfq_info.arrive_lcfs_queue_time;
      result = p;
    }
  }
  return result;
}
```

- **زمان‌بند ابتدائاً، بهترین کار[24] (BJF)**

برای این منظور، تابع best_job_first در proc.c پیاده‌سازی شده است. این تابع با زدن یک حلقه بر روی تمام پردازه‌ها، آن دسته از پردازه‌هایی را که از نوع RUNNABLE هستند و مربوط به صف شمارهٔ دو (یعنی صف BJF) در پردازنده هستند را پیدا می‌کند و از بین تمام این پردازه‌ها، آن پردازه‌ای را که رتبهٔ کمتری دارد را پیدا کرده و در نهایت، به عنوان خروجی می‌دهد. اولویت‌های مربوط به پردازه‌ها از 1 تا 5 محاسبه می‌شود و هرچه عدد کمتر باشد یعنی اولویت بالاتری دارد. همچنین، یک تابع کمکی هم برای محاسبه رتبهٔ پردازه نوشته شده است.

[24] Best Job First

```
struct proc *
best_job_first(void)
{
  float min_bjf_rank;
  struct proc *best_job = 0;

  for (int i = 0; i < NPROC; i++)
  {
    struct proc *process = &ptable.proc[i];

    if (process->mfq_info.queue_type != BJF)
      continue;
    if (process->state != RUNNABLE)
      continue;

    float curr_proc_rank = calc_bjf_rank(process);
    if (best_job == 0 || curr_proc_rank < min_bjf_rank)
    {
      best_job = process;
      min_bjf_rank = curr_proc_rank;
    }
  }
  return best_job;
}
```

که برای محاسبه rank از یک تابع کمکی به شرح زیر استفاده شده است:

```
static float
calc_bjf_rank(struct proc *p)
{
  return p->mfq_info.bjf.priority * p->mfq_info.bjf.priority_ratio +
         p->mfq_info.bjf.arrival_time * p->mfq_info.bjf.arrival_time_ratio +
         p->mfq_info.bjf.executed_cycle * p->mfq_info.bjf.executed_cycle_ratio +
         p->sz * p->mfq_info.bjf.process_size_ratio;
}
```

فراخوانی های سیستمی

- **تغییر صف پردازه**

یک تابع سیستمی به اسم transfer_process_queue در کد قرار دارد و صف پردازهٔ مربوطه با شناسهٔ پردازهٔ ورودی (pid) را به صف مدنظر تغییر می‌دهد. در حلقه‌ای که روی پراسس ها میزنیم به دنبال پردازه‌ای با pid مدنظر می‌گردیم. به محض یافتن این پردازه، حالت صف آن را به new_queue آپدیت می‌کنیم. حالتی که صف جدید lcfs باشد نیاز دارد زمان ورود پردازه به این صف ثبت شود.

```
int transfer_process_queue(int pid, int new_queue)
{
  int old_queue = -1;

  if (pid < 1)
    return old_queue;

  acquire(&ptable.lock);
  for (int i = 0; i < NPROC; i++)
  {
    if (ptable.proc[i].pid != pid)
      continue;

    if (ptable.proc[i].pid == 1 || ptable.proc[i].pid == 2)
      new_queue = RR;

    if (new_queue == LCFS)
      ptable.proc[i].mfq_info.arrive_lcfs_queue_time = ticks;

    old_queue = ptable.proc[i].mfq_info.queue_type;
    if (old_queue == new_queue)
    {
      release(&ptable.lock);
      return -1;
    }

    ptable.proc[i].mfq_info.queue_type = new_queue;
    release(&ptable.lock);
    return old_queue;
  }
  release(&ptable.lock);
  return old_queue;
}
```

- **مقدار دهی پارامتر BJF در سطح پردازه**

یک تابع سیستمی به اسم set_bjs_process_parameters در کد قرار دارد که شناسهٔ پردازه و چهار ضریب مربوط به پردازه را به عنوان ورودی دریافت کرده و تغییرات را روی پردازهٔ مربوطه، اعمال می‌کند.

```
int set_bjs_process_parameters(int pid, float priority_ratio, float arrival_time_ratio, float executed_cycles_ratio,
                               float process_size_ratio)
{
  acquire(&ptable.lock);
  for (int i = 0; i < NPROC; i++)
  {
    if (ptable.proc[i].pid != pid)
      continue;

    ptable.proc[i].mfq_info.bjf.priority_ratio = priority_ratio;
    ptable.proc[i].mfq_info.bjf.arrival_time_ratio = arrival_time_ratio;
    ptable.proc[i].mfq_info.bjf.executed_cycle_ratio = executed_cycles_ratio;
    ptable.proc[i].mfq_info.bjf.process_size_ratio = process_size_ratio;
    release(&ptable.lock);
    return 0;
  }
  release(&ptable.lock);
  return -1;
}
```

• مقدار دهی پارامتر BJF در سطح سیستم

یک تابع سیستمی به اسم set_bjf_system_parameters در کد قرار دارد که چهار ضریب مربوط به پردازه را به عنوان ورودی دریافت کرده و تغییرات را روی پراسس اعمال می‌کند.

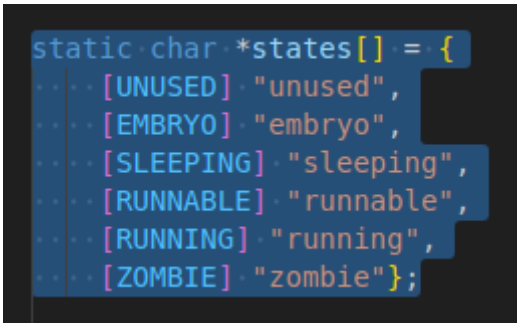
```
void set_bjf_system_parameters(float priority_ratio, float arrival_time_ratio, float executed_cycles_ratio, float process_size_ratio)
{
  acquire(&ptable.lock);
  for (int i = 0; i < NPROC; i++)
  {
    ptable.proc[i].mfq_info.bjf.priority_ratio = priority_ratio;
    ptable.proc[i].mfq_info.bjf.arrival_time_ratio = arrival_time_ratio;
    ptable.proc[i].mfq_info.bjf.executed_cycle_ratio = executed_cycles_ratio;
    ptable.proc[i].mfq_info.bjf.process_size_ratio = process_size_ratio;
  }
  release(&ptable.lock);
}
```

• چاپ اطلاعات

این فراخوانی سیستمی روی تابع سیستمی به اسم print_process_info_table اطلاعات مربوط به همگی پردازه ها را طبق موارد خواسته شده در صورت پروژه چاپ می‌کند.

```c
void print_process_info_table()
{
  print_header();
  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 = "idk mannn";
    cprintf("%s", p->name);
    print_spaces(16 - strlen(p->name));
    cprintf("%d", p->pid);
    print_spaces(8 - digitcount(p->pid));
    cprintf("%s", state);
    print_spaces(9 - strlen(state));
    cprintf("%d", p->mfq_info.queue_type);
    print_spaces(8 - digitcount(p->mfq_info.queue_type));
    cprintf("%d", (int)p->mfq_info.bjf.executed_cycle);
    print_spaces(8 - digitcount((int)p->mfq_info.bjf.executed_cycle));
    cprintf("%d", p->mfq_info.bjf.arrival_time);
    print_spaces(8 - digitcount(p->mfq_info.bjf.arrival_time));
    cprintf("%d", p->mfq_info.bjf.priority);
    print_spaces(8 - digitcount(p->mfq_info.bjf.priority));
    cprintf("%d", (int)p->mfq_info.bjf.priority_ratio);
    print_spaces(9 - digitcount((int)p->mfq_info.bjf.priority_ratio));
    cprintf("%d", (int)p->mfq_info.bjf.arrival_time_ratio);
    print_spaces(8 - digitcount((int)p->mfq_info.bjf.arrival_time_ratio));
    cprintf("%d", (int)p->mfq_info.bjf.executed_cycle_ratio);
    print_spaces(8 - digitcount((int)p->mfq_info.bjf.executed_cycle_ratio));
    cprintf("%d", (int)p->mfq_info.bjf.process_size);
    print_spaces(8 - digitcount((int)p->mfq_info.bjf.process_size_ratio));
    cprintf("%d", (int)calc_bjf_rank(p));
    cprintf("\n");
  }
}
```

استیت‌های مورد استفاده برای حالت های مختلف پردازه به شرح زیر میباشد.



• برنامه سطح کاربر

برنامه سطح کاربر برای اجرای فراخوانی های سیستمی بالا را در mfq_info.c نوشته‌ایم.
برای درک نحوهٔ استفاده از این برنامه، برنامه را بدون آرگومان اجرا کرد تا دستورات help چاپ شوند.

برنامه سطح کاربر foo هم به تعداد دیفاین شده پردازه میسازد و این پردازه ها بعد از مدتی در حالت sleep ماندن، محاسباتی نسبتا طولانی میکنند تا بتوانیم عملکرد کد را روی این پردازه ها چک کنیم.

توجه شود برای اجرا شدن برنامه های سطح کاربر تغییرات لازمه در میک فایل به شرح زیر انجام شده است:

```
UPROGS=\
    cat\
    echo\
    fork test\
    grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    stress\
    _usertests\
    wc\
```

```
    _zombie\
    streiff\
    _find_digital_root\
    _get_uncle_count_test\
    _get_process_lifetime_test\
    _gdb_test\
    foo\
    _mfq_info\
```

• خروجی برنامه

- در ابتدا، اطلاعات پردازه‌ها را نمایش می‌دهیم:

```
Group #10:
1. Mohamad Amin Yousefi
2. Mobina Mehrazar
3. Matin Nabizade
$ mfq_info get_info
Process_Name    PID     State    Queue   Cycle   Arrival Priority R_Prty  R_Arvl  R_Exec  R_Size  Rank
-------------------------------------------------------------------------------------------------------
init            1       sleeping 1       2       0       3       1       1       1       0       12293
sh              2       sleeping 1       1       4       3       1       1       1       12288       16392
mfq_info        3       running  1       1       1900    3       1       1       1       16384       18288
$
```

- سپس، پارامترهای پردازهٔ 2 (که پردازهٔ پوسته می‌باشد) را تغییر می‌دهیم:

```
$ mfq_info set_bjf_process 2 9 9 9 9
the bjf parameters related to process(id:1091567616) changed successfully.
mfq_info get_info
Process_Name    PID     State    Queue   Cycle   Arrival Priority R_Prty  R_Arvl  R_Exec  R_Size  Rank
-------------------------------------------------------------------------------------------------------
init            1       sleeping 1       2       0       3       1       1       1       0       12293
sh              2       sleeping 1       2       4       3       9       9       9       12288       147541
mfq_info        5       running  1       0       13625   3       1       1       1       16384       30012
$
```

- حال، پارامترهای تمام پردازه‌ها را تغییر می‌دهیم. توجه شود که پردازهٔ آخر چون در حال اجرا می‌باشد، پارامترهایش تغییر نمی‌کنند.

```
$ mfq_info set_bjf_system 2 2 2 2
the bjf parameters related to system changed successfully.
$ mfq_info get_info
Process_Name    PID     State    Queue   Cycle   Arrival Priority R_Prty  R_Arvl  R_Exec  R_Size  Rank
-------------------------------------------------------------------------------------------------
init            1       sleeping 1       2       0       3       2       2       2       0       11
sh              2       sleeping 1       2       4       3       2       2       2       12288       19
mfq_info        7       running  1       0       22280   3       1       1       1       16384       38667
$
```

- حال، پردازهٔ پوسته را به صف‌های مختلف منتقل می‌کنیم. ابتدا به صف سوم، سپس به صف دوم و در نهایت، به صف اول.

```
Group #10:
1. Mohamad Amin Yousefi
2. Mobina Mehrazar
3. Matin Nabizade
$ mfq_info get_info
Process_Name    PID     State    Queue   Cycle   Arrival Priority R_Prty  R_Arvl  R_Exec  R_Size  Rank
-------------------------------------------------------------------------------------------------
init            1       sleeping 1       2       0       3       1       1       1       0       12293
sh              2       sleeping 1       1       4       3       1       1       1       12288       16392
mfq_info        3       running  1       1       1631    3       1       1       1       16384       18019
$ mfq_info transfer_queue 2 3
Queue of proc with pid = 2 changed successfully from 1 to 3
$ mfq_info get_info
Process_Name    PID     State    Queue   Cycle   Arrival Priority R_Prty  R_Arvl  R_Exec  R_Size  Rank
-------------------------------------------------------------------------------------------------
init            1       sleeping 1       2       0       3       1       1       1       0       12293
sh              2       sleeping 3       2       4       3       1       1       1       12288       16393
mfq_info        5       running  1       0       4563    3       1       1       1       16384       20950
$ mfq_info transfer_queue 2 2
Queue of proc with pid = 2 changed successfully from 3 to 2
$ mfq_info get_info
Process_Name    PID     State    Queue   Cycle   Arrival Priority R_Prty  R_Arvl  R_Exec  R_Size  Rank
-------------------------------------------------------------------------------------------------
init            1       sleeping 1       2       0       3       1       1       1       0       12293
sh              2       sleeping 2       2       4       3       1       1       1       12288       16393
mfq_info        7       running  1       0       6834    3       1       1       1       16384       23221
$ mfq_info transfer_queue 2 1
Queue of proc with pid = 2 changed successfully from 2 to 1
$ mfq_info get_info
Process_Name    PID     State    Queue   Cycle   Arrival Priority R_Prty  R_Arvl  R_Exec  R_Size  Rank
-------------------------------------------------------------------------------------------------
init            1       sleeping 1       2       0       3       1       1       1       0       12293
sh              2       sleeping 1       3       4       3       1       1       1       12288       16394
mfq_info        9       running  1       0       7467    3       1       1       1       16384       23854
$
```

- در نهایت نیز برنامهٔ foo را در پس‌زمینه اجرا می‌کنیم. مشاهده می‌شود که با گذشت زمان، حالت پردازه‌ها تغییر می‌کند.

```
$ foo&
$ mfq_info get_info
Process_Name    PID     State    Queue   Cycle   Arrival Priority R_Prty  R_Arvl  R_Exec  R_Size  Rank
-------------------------------------------------------------------------------------------------
init            1       sleeping 1       2       0       3       1       1       1       0       12293
sh              2       sleeping 1       3       4       3       1       1       1       12288       16394
foo             12      sleeping 2       30      17528   3       1       1       1       12288       29849
foo             11      sleeping 2       1       17527   3       1       1       1       49152       29819
foo             13      sleeping 2       30      17528   3       1       1       1       12288       29849
foo             14      sleeping 2       30      17528   3       1       1       1       12288       29849
foo             15      sleeping 2       30      17529   3       1       1       1       12288       29850
foo             16      sleeping 2       30      17529   3       1       1       1       12288       29850
mfq_info        17      running  1       1       17828   3       1       1       1       16384       34216
$
```

```
Process_Name    PID     State    Queue   Cycle   Arrival Priority R_Prty  R_Arvl  R_Exec  R_Size  Rank
-------------------------------------------------------------------------------------------------
init            1       sleeping 1       2       0       3       1       1       1       0       12293
sh              2       sleeping 1       4       4       3       1       1       1       12288       16395
mfq_info        19      running  1       0       36797   3       1       1       1       16384       53184
foo             11      sleeping 1       1       17527   3       1       1       1       49152       29819
foo             13      runnable 1       1614    17528   3       1       1       1       12288       31433
foo             14      runnable 1       1614    17528   3       1       1       1       12288       31433
foo             15      running  1       1614    17529   3       1       1       1       12288       31434
foo             16      runnable 1       1613    17529   3       1       1       1       12288       31433
$
```

کد مربوط به mft info:

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

void get_info()...

void help()...

void transfer_queue(int pid, int new_queue_id)...

void set_bjf_process_params(int pid, float priority_ratio, float arrival_time_ratio, float executed_cycles_ratio, float process_size_rati

void set_bjf_system_params(float priority_ratio, float arrival_time_ratio, float executed_cycles_ratio, float process_size_ratio)...

int main(int argc, char *argv[])
{
    if (argc == 2 && !strcmp(argv[1], "get_info"))
        get_info();
    else if (argc == 4 && !strcmp(argv[1], "transfer_queue"))
        transfer_queue(atoi(argv[2]), atoi(argv[3]));
    else if (argc == 7 && !strcmp(argv[1], "set_bjf_process"))
        set_bjf_process_params(atoi(argv[2]), atoi(argv[3]), atoi(argv[4]), atoi(argv[5]), atoi(argv[6]));
    else if (argc == 6 && !strcmp(argv[1], "set_bjf_system"))
        set_bjf_system_params(atoi(argv[2]), atoi(argv[3]), atoi(argv[4]), atoi(argv[5]));
    else
        help();
    exit();
}
```

کد مربوط به foo:

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

#define SOME_BIG_NUM 1000000000000
#define NUM_FORKS 5

int main()
{
    transfer_process_queue(getpid(), 2);
    for (int i = 0; i < NUM_FORKS; ++i)
    {
        int pid = fork();
        if (pid > 0)
            continue;
        if (pid == 0)
        {
            int y;
            sleep(5000);
            y = 432;
            for (int j = 0; j < 100 * i; ++j)
            {
                y *= y;
                int x = 1;
                for (long k = 0; k < SOME_BIG_NUM; ++k)
                {
                    x++;
                    x /= y;
                }

            }
            exit();
        }
    }
    while (wait() != -1)
        ;
    exit();
}
```

- Github Hash: 90a7480092ab87c90850bc06c8598fd712080b65