گزارش پروژهٔ چهارم آزمایشگاه سیستم عامل

به تدریس دکتر کارگهی

محمد امین یوسفی

مبینا مهرآذر

متین نبی‌زاده

زمستان 1402

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

همگام‌سازی در xv6

1) علت غیرفعال کردن وقفه در هنگام استفاده از این نوع قفل چیست؟ چرا ممکن است CPU با مشکل deadlock رو به رو شود؟

وقفه[1]ها، حتی در سیستم‌های تک‌هسته‌ای، می‌توانند موجب هم‌روندی شوند؛ بدین صورت که ضمن ایجاد شدن و رسیدن هر کدام از وقفه‌ها، در هر آن، کُد هسته می‌بایست متوقف شده و این امکان فراهم شود تا فرآیند رسیدگی به وقفه، اجرا شود. به عنوان مثال، سناریویی را متصور می‌شویم که طی آن، تابع `iderw` از پیش، قفل idelock را اتخاذ کرده باشد و در ادامه، وقفه‌ای جهت اجرا شدن `ideintr` به آن، برسد. در این شرایط، خود ideintr نیز تلاش می‌کند تا قفل `idelock` را اتخاذ کند اما طبیعتا با از پیش قفل بودن آن، مواجه می‌شود؛ در نتیجه، مجبور می‌شود تا منتظر آزاد شدن آن، بماند. طی روند گفته شده، idelock هیچ‌گاه نمی‌تواند آزاد شود مگر توسط iderw که اجرای این تابع نیز از پیش، توسط وقفهٔ ایجاد شده در راستای رسیدگی به `ideintr`، متوقف شده است. واضح است که تحت این شرایط، سیستم دچار بن‌بست[2] شده است. به همین دلیل، در راستای اجتناب از رسیدن کلیّت سیستم به بن‌بست، اگر یک قفل چرخشی[3] توسط رسیدگی‌کننده به وقفه[4] مورد استفاده قرار گیرد، پردازنده موظف است تا از عدم اتخاذ آن قفل ضمن امکان رسیدن وقفه‌ها، اطمینان حاصل کند[5].

---

[1] Interrupt
[2] Deadlock
[3] Spin-lock
[4] Interrupt Handler
[5] R. Cox, F. Kaashoe and R. Morris, *xv6: a simple, Unix-like teaching operating system*, MIT, 2018, p. 56

2) توابع pushcli و popcli به چه منظور استفاده شده و چه تفاوتی با cli و sti دارند؟

- **از جمله وظایف تابع pushcli، می‌توان به غیرفعال کردن وقفه‌ها و یک واحد افزایش متغیرِ پردازنده-به‌ازای[6] ncli، اشاره کرد.**
- **همچنین، تابع release نیز popcli را در فرآیند اجرای خود صدا می‌زند که در ادامه، شمارنده را یک واحد کاسته و در صورت صفر شدن مقدار آن، امکان ایجاد وقفه را مجددا فعال می‌سازد.**

لازم به ذکر است که علت شمارش فراخوانی‌های تو در تو توسط این دو تابع، این است که اطمینان حاصل شود که وقفه‌ها تنها در صورتی غیرفعال می‌شوند که تمامی قفل‌ها، رها باشند.

کارکرد توابع `sti` و `cli` نیز غیرفعال کردن و فعال کردن وقفه‌ها است. کارکرد `pushcli` و `popcli` هم به طرز مشابه، با مکانیزم پشته مانند، سطوح تو در توی قفل‌ها را پشتیبانی می‌کنند. در تابع `pushcli`، فراخوانی تابع `cli` وجود دارد که در این نقطه از کد، وقفه‌ها غیر فعال می‌شوند. پس از این مرحله، مقدار متغیر `ncli` هم یک واحد اضافه می‌شود. در کد تابع `popcli` نیز یک واحد از متغیر `ncli` پردازهٔ مدنظر کم شده و به محض رسیدن این متغیر به صفر، تابع sti فراخوانده شده و وقفه‌ها فعال می‌شوند. در اصل، این متغیر کمک می‌کند تا به تعداد فرخوانده شدن تابع `pushcli`، تابع `popcli` نیز فراخوانی شود. این ساز و کار به طور خاص، در قفل‌های تو در تو کاربرد دارد.

[6] Per-CPU

3) چرا قفل مذکور (چرخشی) در سیستم‌های تک‌هسته‌ای مناسب نیست؟ روی کد توضیح دهید.

کد تابع `acquire` (که در صفحهٔ بعد، قابل مشاهده است) در فایل spinlock.c، در ابتدا وقفه‌های موجود را برای جلوگیری از به بن‌بست رسیدن سیستم، غیرفعال می‌سازد. سپس به کمک یک شرط، بررسی می‌شود که آیا هنوز هم قفل توسط پردازندهٔ کنونی گرفته شده است یا خیر. اگر پاسخ به این پرسش، آری باشد، سیستم بایستی به حالت اضطرار[7] برود؛ چرا که این امر، یک خطا[8]ی نرم‌افزاری تلقی می‌شود.

حلقهٔ اصلی در این بخش از برنامه قرار دارد که پردازنده نیز تا پیش از اتخاذ قفل، در آن قرار می‌گیرد. تابع xchg نیز به صورت اتمی[9] مقدار ذخیره شده در `lk -> locked` را با 1 جابه جا می‌کند. اگر پیش از اجرای تابع `xchg`، مقدار `lk->locked` برابر 0 باشد، یعنی قفل اتخاذ نشده باشد، مقدار صفر برگردانده شده و از حلقه خارج می‌شویم. ولی اگر مقدار اولیه `lk->locked` برابر 1 باشد، همچنان در حلقه، خواهیم ماند.

همان‌طور که گفته شد، در این روش دچار معطلی شلوغ[10] هستیم که این امر، به خصوص در سیستم‌های چندهسته‌ای[11]، زمان پردازنده را هدر داده و از بهینگی سیستم، به طور چشم‌گیری می‌کاهد. حتی در سیستم های تک‌هسته‌ای نیز بن‌بست صورت می‌گیرد؛ به عنوان مثال، اگر پردازه‌ای در این سیستم قفل را در دست گیرد و در این شرایط، پردازهٔ دیگری ضمن فرآیند ذکر شده سعی بر اتخاذ قفل داشته باشد، پردازهٔ دوم هرگز از حلقه خارج نشده و در نتیجه، پردازه‌های دیگر زمان‌بندی نخواهند شد و به وضوح، سیستم با بن‌بست مواجه شده است. خوانندهٔ گرامی می‌تواند در صفحهٔ بعد، قطعه کُد مربوط به تابع `acquire` را ملاحظه کند.

---

[7] Panic Mode
[8] Error
[9] Atomically
[10] Busy Waiting
[11] Multi-core

```
// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.

void acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move
loads or stores
  // past this point, to ensure that the critical
section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();

  // Record info about lock acquisition for debugging.
  lk->cpu = mycpu();
  getcallerpcs(&lk, lk->pcs);
}
```

4) در مجموعه دستورات RISC-V، دستوری با نام amoswap وجود دارد. دلیل تعریف و نحوهٔ کار آن را توضیح دهید.

به منظور درک کلیت این دستور، بایستی در ابتدا، بایستی کمی در تعریف فرآیندهای اتمی، عمیق‌تر شویم. می‌دانیم که به طور کلی، فرآیندهایی در دستهٔ اتمی قرار می‌گیرند که:

- شامل سه مرحلهٔ خواندن، تغییر و نوشتن باشند،
- مراحل مورد اول، به همین ترتیب (یعنی در ابتدا، خواندن[12]، سپس تغییر[13] و در نهایت، نوشتن[14]) اجرا شوند،
- در حین اجرای هر کدام از مراحل ذکر شده، هیچ وقفه‌ای رخ ندهد.

طبق نص صریح کتب و منابع مرجع مربوط به معماری RISC-V، دستورات AMO به منظور پیاده‌سازی یک عملیات پایهٔ اتمی، در مجموعه دستورات این معماری، گنجانده شده‌اند؛ به عنوان مثال:
دو دستور زیر، به ترتیب یک دادهٔ علامت‌دار 32-بیت / 64-بیت را از آدرس موجود در ثبّات[15] rs1 لود کرده، آن را درون ثبات rd قرار می‌دهد. در ادامه، یک عملیات باینری[16] بر روی مقدار موجود در این ثبات و همچنین ثبات rs2 (به عنوان دو عملوند موجود در این عملیات) انجام می‌دهد و در آخر، مقدار نهایی را در ثبات rs1 ذخیره می‌کند.

```
x[rd] = AMO32(M[x[rs1]] SWAP x[rs2])
```

```
x[rd] = AMO64(M[x[rs1]] SWAP x[rs2])
```

---

[12] Read (Load)
[13] Modify (Operate)
[14] Write (Store)
[15] Register
[16] در این دو دستور ذکر شده، عملیات باینری استفاده شده همان swap است که در صورت سؤال نیز ذکر شده است. از انواع دیگر عملیات‌های باینری مورد پشتیبانی این پیاده‌سازی می‌توان به موارد زیر اشاره نمود:

- Integer add, logical AND, logical OR, logical XOR, and signed and unsigned integer maximum and minimum

در مورد یکی از علل وجودی این نوع از عملیات‌ها نیز می‌توان گفت که در اصل، این مجموعه دستورات برای پشتیبانی کارآمد از عملیات اتمی C11/C++11 و همچنین برای پشتیبانی از کاهش موازی در حافظه برگزیده شده‌اند. یکی دیگر از کاربردهای این مجموعه دستورات، ارائهٔ به روزرسانی‌های اتمی برای ثبات‌های دستگاه دارای نقشه حافظه (به عنوان مثال، تنظیم، پاک کردن یا تغییر بیت ها) در فضای ورودی/خروجی است.

اما مزایای برنامه‌نویسی چندرشته‌ای درک نمی‌شود، مگر پس از آنکه هم‌روندی را به طور کامل، درک کرده باشیم. از خوانندهٔ محترم، درخواست می‌شود که توضیحاتی که در ادامه می‌آیند را این بار، از زاویه دید مفهوم هم‌زمانی بنگرد تا بتوانیم به کُنه فلسفهٔ وجود این مجموعه دستورات، پی ببریم. می‌دانیم که دو مشکل رایج همزمانی وجود دارد:

• **عدم انسجام حافظه پنهان**: هر ریسهٔ سخت‌افزاری، حافظهٔ نهان مخصوص به خود را دارد؛ از این رو، داده های اصلاح شده در یک ریسه ممکن است بلافاصله در ریسهٔ دیگر، منعکس نشوند. اغلب می‌توان با دور زدن حافظهٔ پنهان و نوشتن مستقیم در حافظه، از وقوع این مشکل، جلوگیری کرد. اما ذهن خود را درگیر این معضل، نکنید که لُب مطلب، تازه در پاراگراف بعد بر شما آشکار می‌شود!

• **چرخهٔ خواندن، تغییر، نوشتن:** یک الگوی بسیار رایج در برنامه نویسی است. در زمینهٔ برنامه‌نویسی چند ریسه‌ای، درهم آمیختن این سه مرحله اغلب مسائل (و بسا مشکلات) زیادی را به بار می‌آورد. برای حل مشکلات ناشی از اجرای این چرخه، بایستی به ایدهٔ اجرای بدون اخلال تکیه کنیم. در معماری RISC-V، ما دو دسته دستورات اتمی داریم که می‌توانند نیاز ما به دستورات اتمی را برآورده سازند؛ بالاخص، همین مجموعه دستوراتی که تا به الان، در حال توضیحشان بودیم:

- Load-reserve, store-conditional (اجرای اخلال-ناپذیر در سطح چندین دستور)
- Amo.swap[17] (عملیات واحد و اخلال-ناپذیر در سطح حافظه)

[17] و دیگر عملیات‌های این دسته که پیش‌تر، در همین بخش، به طور کلی بررسی شده بودند.

5) مختصری راجع به تعامل میان پردازه‌ها توسط دو تابع acquiresleep و releasesleep توضیح دهید.

پیش از هر چیز، خوب است تا نگاهی مختصر به پیاده‌سازی این دو تابع در سیستم‌عامل xv6، داشته باشیم تا بتوانیم ضمن بررسی بخش‌های مهم آن، نکاتی کوتاه اما تعیین‌کننده را بیان کنیم.
در فایل sleeplock.c، تعریف و پیاده‌سازی این دو تابع، به صورت زیر، مشاهده می‌شود:

```
void releasesleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  lk->locked = 0;
  lk->pid = 0;
  wakeup(lk);
  release(&lk->lk);

void acquiresleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  while (lk->locked) {
    sleep(lk, &lk->lk);
  }
  lk->locked = 1;
  lk->pid = myproc()->pid;
  release(&lk->lk);
}
}
```

پیاده‌سازی این دو تابع، چندان پیچیده نیست؛ به صورتی که در ابتدا، تابع aqcuiresleep، قفل چرخشی مورد نظر را اتخاذ کرده تا از دسترسی اختصاصی به این تابع، اطمینان حاصل شود و در واقع، شرط انحصار متقابل، رعایت شود.
در ادامه، درون یک حلقه، به طور مداوم، بررسی می‌شود که آیا این قفل، کماکان مورد اتخاذ هست یا خیر. در صورت ورود به حلقه، تابع sleep صدا زده می‌شود که این تابع، پردازهٔ مورد نظر را به خواب می‌برد. به خواب رفتن یک پردازه، به معنای آزاد شدن قفل چرخشی است که در این صورت، تا زمانی که تابع `wakeup` بر روی این پردازه صدا شود، در حالت خواب باقی می‌ماند.

در صورتی که پردازه بیدار شود، اجرای خود را از ادامهٔ جایی از حلقه که در آن به خواب رفته بود، ادامه می‌دهد تا قفل را اتخاذ کند. هنگامی که قفل خواب به دست آمد، تابع lk->locked بر روی 1 تنظیم می‌شود که نشان می‌دهد قفل اکنون نگه داشته شده است، و شناسهٔ پردازهٔ[18] فرآیند فعلی را در ساختار قفل خواب ثبت می کند. در نهایت، قفل چرخشی زیرین را آزاد می کند.

به همین طریق، می‌توان توضیحات متناظر تابع releasesleep را ارائه نمود؛ بدین صورت که برای اطمینان از دسترسی انحصاری به ساختار قفل خواب، ابتدا قفل چرخشی زیرین (lk->lk&) را دریافت کرده و lk->locked را روی 0 تنظیم می‌کند که نشان می‌دهد قفل اکنون آزاد شده است و شناسهٔ پردازهٔ مرتبط با قفل را پاک می کند.

سپس wakeup را در کانال خواب (lk) فراخوانی می کند تا هر پردازه‌ای که ممکن است روی این قفل خواب باشد را بیدار کند. عملکرد wakeup مسئول بیدار کردن فرآیندهای منتظر در کانال خواب مشخص شده است. در نهایت، قفل چرخشی زیرین نیز آزاد می شود.

[18] PID

6) حالات مختلف پردازه‌ها در xv6 را توضیح دهید. تابع sched چه وظیفه‌ای دارد؟

پیش از آن که به شرح و تفصیل انواع پردازه‌ها همت گماریم، می‌بایست به این پرسش بنیادین، پاسخ دهیم که اصلا حالت یک پردازه، در کجا و با چه ساختاری، نگهداری می‌شود؟ پاسخ، ساده است؛ در متغیر وضعیت آن که از جنس `enum procstate` است نگه داشته می‌شود:

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

حال، مانعی پیش روی نداریم که ما را از بررسی هر یک از انواع پردازه‌ها، باز دارد:

- **بی‌استفاده (UNUSED)**

از آنجا که پردازه‌ها در یک لیست 64 تایی نگه داشته می‌شوند، خانه‌هایی که در آن‌ها پردازهٔ حقیقی‌ای نیست با این وضعیت مشخص شده‌اند و در اصل، این وضعیت به منزلهٔ غیاب پردازه است.

- **جَنین (EMBRYO)**

وقتی که پردازهٔ جدیدی ساخته می‌شود (برای مثال، با استفاده از تابع fork)، این حالت، ابتدایی‌ترین وضعیت آن را توصیف می‌نماید. یعنی تابع `allocproc` از میان پردازه‌های UNUSED یکی را انتخاب کرده و آن را به وضعیت جنین، تغییر می‌دهد.

- **خُفته (SLEEPING)**

در این وضعیت، پردازه در میان انتخاب‌های زمان‌بند برای تخصیص پردازنده به آن قرار نمی‌گیرد و بدون هیچ فعالیتی می‌ماند. پردازه می‌تواند به صورت داوطلبانه یا توسط هسته به این حالت برود و در انتظار دسترسی به یک منبع بماند.

- **قابل اجرا (RUNNABLE)**

هنگامی که پردازه در این حالت است، یعنی در صف اجرای زمان‌بند قرار دارد و در یکی از راندهای زمان‌بندی، مجال تخصیص پردازندهٔ مرکزی به آن داده می‌شود و بدین وضعیت، در می‌آید. در اصل، می‌توان چند سناریو را که منجر به ورود به این وضعیت می‌شوند، متصور شد:

- پردازه به تازگی تشکیل شده باشد و از وضعیت جنین، بدین وضعیت رسد؛
- پردازه در حال اجرا بوده اما ضمن به پایان رسیدن برش زمانی خویش یا اجرای yield توسط هسته، پردازنده از آن گرفته شود؛
- پردازه، در حالت خواب بوده و با صدا شدن تابع `wakeup`، قابل اجرا شود؛
- پردازه‌ای که در حالت خواب بوده، کشته شده و پس از تغییر فیلد `killed` در آن پردازه به مقدار یک، بدین حالت درآید تا زمانی که دوباره اجرا شود و آن وقت، همان ابتدای امر، با توجه به پیشتر کشته شده بودن، خارج شود.

- **در حال اجرا (RUNNING)**

به وضوح، از این حالت، چنان برمی‌آید که پردازه در حال اجرا شدن توسط پردازنده است. همچنین، می‌دانیم که تعداد پردازه‌ها در یک زمان، حداکثر معادل تعداد پردازنده‌ها[19] است.

- **زامبی (ZOMBIE)**

هنگامی که پردازه، کارش تمام می‌شود و می‌خواهد خارج شود، ابتدا بدین حالت درمی‌آید؛ یعنی اینکه مستقیما به حالت بی‌استفاده نمی‌رود و در حالتی می‌ماند که پدرش بتواند با استفاده از تابع wait، از اتمام کار فرزندش مطلع شود.

در توضیح تابع `sched`، بایستی بگوییم که در اصل، برای تعویض متن[20] به زمان‌بند مورد استفاده قرار می‌گیرد. پردازه برای رها کردن پردازندهٔ مرکزی به این تابع می‌آید (که از قبل بایستی وضعیتش از در حال

---

[19] خوانندهٔ محترم، توجه دارد که برای ایجاب حداکثر دقت در این بخش، می‌بایست به واژه‌شناسی و تفاوت میان کلمهٔ "پردازنده" و "هسته"، دقت دوچندان صورت گیرد که تمایز میان استفاده از هر کدام از این دو کلمه در کلمات مربوط به برنامه‌نویسی چندریسه‌ای، نیاز به دانش فراتر پیرامون معماری پیاده‌سازی شده در سطح سخت‌افزار دارد.

[20] Context Switch

اجرا، تغییر کرده و قفل ptable را اتخاذ کرده باشد). در تابع فلگ interrupt enable ذخیره شده و پس از بازگشت برگردانده می‌شود.

این تابع با استفاده از swtch که در آزمایش سوم کامل توضیح داده شده و در اسمبلی نوشته شده، متن[21] را تغییر می‌دهد و ادامه تابع scheduler اجرا می‌شود که به متن پردازهٔ قابل اجرای دیگری تعویض می‌کند.

7) تغییری در توابع دستهٔ دوم دهید تا تنها پردازه صاحب قفل، قادر به آزادسازی آن باشد. قفل معادل در هستهٔ لینوکس را به طور مختصر معرفی نمایید.

کد تغییر یافته این تابع به صورت زیر است. در این تابع تنها شرط برابر بودن آیدی پردازه‌ها بررسی شده است.

```
void
releasesleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  if (lk->pid == myproc()->pid)
  {
    lk->locked = 0;
    lk->pid = 0;
    wakeup(lk);
  }
  release(&lk->lk);
}
```

به طور خلاصه، می‌توان بیان داشت که از mutex می‌توان به عنوان قفل معادل در سیستم‌عامل لینوکس، یاد کرد. پیاده‌سازی این قفل، در فایلی تحت عنوان mutex.h قرار داده شده است که محض اختصار این بخش از گزارش، از آوردن آن، صرف نظر شده است.

[21] Context

8) روشی دیگر برای نوشتن برنامه‌ها، استفاده از الگوریتم های بدون قفل[22] است. مختصری راجع به آن‌ها توضیح داده و از مزایا و معایب آن‌ها نسبت به برنامه نویسی با قفل بگویید.

به کار بستن از الگوریتم‌های بدون قفل، به طور خلاصه، به منزلهٔ استفادهٔ خاص‌منظوره از عملیات‌های اتمی پیاده‌سازی شده در سطح سخت‌افزار است؛ چنان که می‌دانستیم خود ساز و کار قفل‌ها (که در قسمت‌های پیشین، به طور مفصل توضیح داده شده است) نیز با استفاده از عملیات اتمی، پیاده‌سازی شده است. در مدح (!) برنامه‌نویسی بدون قفل، می‌توان گفت که به کار بستن این شیوه، می‌تواند تضمین کند که سیستم، هیچ‌گاه به واسطهٔ یک ریسهٔ منفرد، متوقف نمی‌شود و دیگر ریسه‌ها، می‌توانند پیش روند؛ برخلاف برنامه‌نویسی با قفل که می‌دانیم اگر پردازه‌ای، قفل را اتخاذ کند اما فرآیند اجرایش، به پایان نرسد، تمام برنامه بلوکه شده و در روند پیشرفت آن، تغییری حاصل نمی‌شود. قطعه کد سادهٔ زیر که در اصل، پیاده‌سازی یک شمارندهٔ هم‌روند است را در دو حالت، نظر می‌گیریم؛ به ترتیب، یکی پیاده‌سازی با قفل، و دیگری، پیاده‌سازی بدون قفل[23]:

```
int counter = 0;
std::mutex counter_mutex;

void increment_with_lock() {
std::lock_guard<std::mutex> _(counter_mutex);
++counter; }

std::atomic<int> counter(0);
void increment_lockfree() { ++counter; }
```

حال، خوانندهٔ محترم ممکن است حالتی را متصور شود که در آن، صدها ریسه، به صورت هم‌روند، یکی از توابع بالا را صدا کنند. در نسخهٔ با قفل، تا زمانی که ریسهٔ نگهدارندهٔ قفل آن را آزاد نسازد، هیچ ریسه‌ای نمی‌تواند پیشرفتی حاصل کند در صورتی که در نسخهٔ بدون قفل، تمامی ریسه‌ها می‌توانند

---

[22] Lock-free

[23] خوانندهٔ محترم، در نظر دارد که این مثال، تنها برای شبیه‌سازی وضعیت پیش آمده بر اثر دو پیاده‌سازی با قفل / بدون قفل، آورده شده است و به هیچ عنوان، مُعرّف یک پیاده‌سازی عملی با قفل / بدون قفل نیست. به عبارت دقیق‌تر، تابع بدون قفل موجود در این قطعه کد، در اصل بدون مانع (Block Free) است و نه بدون قفل.

آزادانه پیش روند و تنها ریسهٔ متوقف‌شده، نمی‌تواند سهم خود از کار را به انجام برساند. از آن‌جایی که عملیات اتمی، به طور ذاتی، کند است، در ذمّ پیاده‌سازی بدون قفل نیز بایستی این نکته را خاطر نشان کنیم که علیرغم ممکن بودن پیشروی برای هر پردازه در هر زمان، اقدام هر پردازه به اجرای عملیات اتمی ممکن است با اخلال روبرو شود که در این صورت، پردازه موظف است تمام فرآیند عملیات اتمی تا بدان نقطه را متوقف کرده و تغییرات اعمال شده تا بدان نقطه را بازگرداند. در عمل، غیرممکن است اگر ادعا کنیم که الگوریتم بدون قفلی طراحی کرده‌ایم که در عین صحّت، سریع و "منصفانه" نیز باشد؛ به دیگر سخن، برخی ریسه‌ها ممکن است خوش‌شانس بوده و ضمن اعمال عملیات اتمی، بدون مورد اخلال واقع شدن، کار خود را پیش ببرند اما طبیعتا، اطمینان داریم که ریسه‌های دیگری نیز وجود دارند که بخت از ایشان، برگشته و عملیات اتمی‌شان، به طور مداوم با اخلال روبرو می‌شود و نتیجتا، نمی‌توانند کاری از پیش ببرند[24].

بدیهی است که برنامه نویسی بدون قفل می‌تواند امری دشوار، زمان‌بر و پرهزینه باشد. با این حال، اگر پیش از این، داده‌ساختاری منطبق بر اصول برنامه‌نویسی بدون قفل تعبیه شده باشد (مانند صف، لیست، بردار و غیره)، استفاده از آن به وضوح می‌تواند بهینه‌سازی‌های مناسبی را نتیجه دهد.

[24] Anthony Williams, *C++ Concurrency in Action: Practical Multithreading*, Manning Publications, 1st edition, 2012

پیاده‌سازی متغیرهای مختص هر هستهٔ پردازنده

الف) روشی جهت حل این مشکل در سطح سخت‌افزار وجود دارد. مختصرا آن را توضیح دهید.

انسجام حافظهٔ نهان[25] در علوم رایانه به هم‌خوانی داده‌های ذخیره شده در حافظه‌های نهان محلی یک منبع مشترک گفته می‌شود. در شرایطی که در یک سامانهٔ دارای حافظه مشترک چند خدمت‌گیرنده حافظه‌های نهان خود را دارا باشند، امکان بروز مشکل به خاطر ناهم‌خوانی داده‌ها وجود دارد.
یک راه حل معمول در سطح سخت‌افزار برای این مشکل، استفاده از پروتکل‌های مرتبط با انسجام حافظهٔ نهان می‌باشد. این پروتکل‌ها شامل دسته‌ای از قوانین و دستورات بیان‌کنندهٔ نحوهٔ تعامل حافظه پنهان با یکدیگر برای حفظ ثبات داده‌ها می‌باشد.

- **برخی از این پروتکل‌ها، به شرح زیر می‌باشند:**

- **MESI[26]:**

یک پروتکل پرکاربرد است که انسجام کش را با ردیابی وضعیت هر خط کش حفظ می کند. هر خط کش می تواند در یکی از چهار حالت باشد: اصلاح شده، انحصاری، اشتراک گذاری شده یا نامعتبر. این پروتکل تضمین می‌کند که تنها یک کش حالت `Modified` یا `Exclusive` را برای یک خط کش معین داشته باشد و هر به‌روزرسانی در صورت نیاز به کش‌های دیگر منتشر می‌شود.

- **MOESI[27]:**

توسعه پروتکل MESI است که حالت "مالک" را معرفی می کند. حالت Owned به یک کش اجازه می دهد تا یک کپی از یک خط کش داشته باشد بدون اینکه لزوماً آن را تغییر داده باشد. این می تواند عملکرد را در برخی سناریوها افزایش دهد.

---

[25] Cache Coherence
[26] Modified, Exclusive, Shared, Invalid
[27] Modified, Owned, Exclusive, Shared, Invalid

- **MESIF[28]:**

این پروتکل توسعه داده شده‌ی MESI می‌باشد که حالت "Forward" را معرفی می کند.
حالت Forward به یک کش اجازه می دهد تا نشان دهد که دارای یک کپی از یک خط کش است، اما ممکن است این خط در حافظه پنهان دیگری در حالت اشتراکی وجود داشته باشد. حالت Forward نیاز به دسترسی های غیرضروری به حافظه را زمانی که چندین کش درخواست خط کش یکسانی دارند کاهش می دهد.

این پروتکل ها از مکانیسم های مختلفی مانند رویکردهای مبتنی بر عدم اعتبار[29] یا به مبتنی بر به روزرسانی[30] برای حفظ انسجام حافظه پنهان استفاده می کنند. پروتکل‌های مبتنی بر عدم اعتبار، کپی‌های خطوط موجود در حافظه‌های پنهان دیگر را هنگام نوشتن باطل می‌کنند و اطمینان حاصل می‌کنند که فقط از مقدار به‌روز شده استفاده می‌شود. پروتکل‌های مبتنی بر به‌روزرسانی، مقدار به‌روزشده را در حافظه‌های پنهان دیگر منتشر می‌کنند و از سازگاری همه نسخه‌ها اطمینان می‌دهند.

پیاده‌سازی این پروتکل‌های انسجام حافظه نهان مستلزم طراحی دقیق و هماهنگی بین حافظه‌های پنهان و کنترل‌کننده‌های حافظه است و بسته به معماری و انتخاب های طراحی ساخته شده توسط سازنده پردازنده می تواند متفاوت باشد.

ب) همان‌طور که در اسلایدهای معرفی پروژه ذکر شده است، یکی از روش‌های همگام‌سازی، استفاده از قفل‌هایی موسوم به قفل بلیت است. این قفل‌ها را از منظر مشکل مذکور، بررسی نمایید.

قفل‌های بلیت نیز می‌توانند با مشکل بی‌اعتباری دادهٔ تغییر یافته مواجه شوند، بدین صورت که اگر هنگامی که چندین ریسه یا پردازه برای قفل بلیت رقابت می‌کنند، هر کدام به شکل مکرر در یک حلقه در انتظار مشغول[31] به سر می‌برند و به شکل مکرر مقدار متغیر نشان دهنده‌ی نوبت را از حافظه می‌خوانند و در این مرحله ممکن است مشکل `cache coherence` رخ دهد. این می تواند منجر به

[28] Modified, Exclusive, Shared, Invalid, Forward
[29] Invalidation-based
[30] Update-based
[31] Busy Waiting

افزایش تأخیر و کشمکش میان رشته‌ها شود و بالاخص، مقیاس‌پذیری و عملکرد سیستم را کاهش دهد.

در سیستم‌های چندپردازنده‌ای[32]، هر پردازنده، حافظهٔ نهان منحصر به خود را دارد که هر ریسه پس از خواندن مقدار متغیر نشان دهنده‌ی نوبت می‌تواند یک کپی از آن را در حافظهٔ نهان خویش، ذخیره داشته باشد. حال اگر یک پردازه یا ریسهٔ دیگری پس از ورود و گرفتن نوبت و انجام فرایند مربوطه، مقدار متغیر نشان دهنده‌ی نوبت را تغییر دهد، مقدار موجود در حافظهٔ نهان برخی ریسه ها نامعتبر خواهند بود.
برای رفع این مشکل از پروتکل های ذکر شده در بخش های قبلی استفاده می‌شود.

برای رفع مشکلات انسجام حافظهٔ نهان با قفل بلیط، تکنیک‌های مختلفی می توانند مورد استفاده قرار گیرند؛ از جمله:

using cache-aware synchronization primitives
optimizing memory access patterns
employing alternative synchronization mechanisms

نباید از نظر خوانندهٔ محترم، مغفول بماند این نکته که علی‌رغم اینکه ساز و کار قفل بلیت، یک عدالت برای همگام‌سازی پدید می‌آورد (که پیش‌تر دیدیم در قفل چرخشی وجود ندارد)، کماکان مشکل انسجام حافظهٔ نهان وجود خواهد داشت.

ج) چگونه می‌توان در لینوکس، داده‌های مختص به هر هسته را در زمان کامپایل تعریف نمود؟

در سیستم‌عامل مبتنی بر لینوکس، معمولا داده‌های مختص به هر هسته در زمان کامپایل تعریف نمی‌شوند و دلیل این امر نیز آن است که این تعریف شدن در اکثر موارد، در زمان اجرا و به وسیله‌ی ساز و کار زمان‌بندی لینوکس انجام می‌شود.
ولی هستهٔ لینوکس، از مفهومی به نام per cpu variables پشتیبانی می‌کند که هسته را قادر می‌سازد تا متغیرهایی را تعریف کند که مختص هر هستهٔ پردازندهٔ مرکزی است.

[32] Multi-processor

- **در ادامه، خلاصه‌ای از نحوهٔ به کارگیری این مفهوم در لینوکس شرح داده شده است:**

1) تعریف متغیر:

متغیرهای هر پردازنده با استفاده از ماکروی `DEFINE_PER_CPU` تعریف می شوند. کارکرد این ماکرو بدین شکل است که یک نمونهٔ جداگانه از یک متغیر را به هر پردازنده اختصاص می دهد.

```
DEFINE_PER_CPU(type, name);
```

برای مثال، هنگام تعریف یک عدد صحیح به عنوان دادهٔ مختص به هسته مربوطه داریم:

```
DEFINE_PER_CPU(type, data_related_to_cpu_name);
```

2) مقداردهی اولیه به متغیر:

برای مقداردهی اولیه، هنگام راه‌اندازی سیستم، از `__init` و `__initdata` استفاده می‌شود. همچنین، می‌توان از ماکروی `per_cpu` در داخل تابعی استفاده کرد که در زمان اجرا فراخوانی می شود.

قطعه کد زیر، نمونه‌ای از مقداردهی اولیه به این متغیرها را نشان می‌دهد:

```
static int __init my_module_init(void) {
    int cpu;
    for_each_possible_cpu(cpu) {
        per_cpu(cpu_specific_data, cpu) =
initial_value_for_core(cpu);
    }
    return 0;
}
```

3) دسترسی به متغیر:

با ماکرو های `get_cpu_var` و `put_cpu_var` می‌توان به یک متغیر مختص هر CPU، دست یافت. در این فرآیند، Pre-emption غیرفعال می‌شود تا هرگاه یک پردازندهٔ مرکزی روی متغیر مختص به خود کار می‌کند، برنامه‌ریزی نشود.

```
int value = get_cpu_var(cpu_specific_data);
put_cpu_var(cpu_specific_data);
```

- همچنین، لازم به ذکر است که دسترسی به متغیرهای هر پردازندهٔ مرکزی، از طریق ماکروهای `this_cpu_read` یا `this_cpu_write` نیز ممکن است.

```
this_cpu_write(cpu_specific_data, new_value);
int value = this_cpu_read(cpu_specific_data);
```

پیاده‌سازی ساز و کار همگام‌سازی با قابلیت اولویت دادن

- **آیا این پیاده سازی ممکن است که دچار گرسنگی شود؟ راه حلی برای برطرف کردن این مشکل ارائه دهید.**

در صورتی که روند ورود پردازه‌ها به صف از شناسهٔ[33] بزرگ‌تر به شناسهٔ کوچک‌تر باشد، هر سری پردازه با اولویت بیشتر (بخوانید شناسهٔ بزرگ‌تر) برگزیده می‌شود و به این ترتیب نوبت انتخاب شدن به پردازه‌هایی با شناسهٔ کوچک‌تر که در این پیاده‌سازی اولویت کمتری دارند، نمی‌رسد.

یکی از ساز و کارهای مفید برای حل این مشکل، استفاده از مفهوم افزایش عمر[34] می‌باشد. با تعریف یک کوانتوم زمانی است که همگی پردازه‌ها در هر مرحله، چک میشوند که مقداری که در صف، انتظار کشیده‌اند، بیشتر از این مقدار هست یا خیر.
اگر بیشتر بود، اولویت آن را یک واحد کاهش می‌دهیم. محل جدید قرارگیری این پردازه بر حسب اولویت جدیدش است و در صورتی که پردازهٔ مورد بررسی با چندین پردازهٔ قبلی هم‌اولویت باشد، پردازهٔ مورد بررسی را پیش از این پردازه‌ها قرار می‌دهیم؛ چرا که مدت زمان طولانی‌تری را در صف، به انتظار گذرانده است.

یکی دیگر از ساز و کارهای قابل بررسی، هنگام اضافه کردن پردازهٔ جدید، محل قرارگیری آن طبق زمان ورود، که دیر تر از بقیه وارد شده است، بعد از بقیه پردازه های هم اولویت باشد.

- **یک نوع پیاده‌سازی همگام‌سازی توسط قفل بلیت انجام می‌شود. آن را بررسی کنید و تفاوت‌های آن با روش همگام‌سازی بالا را بیان کنید.**

قفل بلیت[35] برای هماهنگ‌سازی در ایجاد عدالت در زمینهٔ دسترسی به منابع، بدون ایجاد گرسنگی می‌باشد.
در این پیاده‌سازی، مشابه درخواست نوبت در بانک، به هر پردازه که سر می‌رسد و درخواست قفل می‌کند، یک شماره نسبت داده می‌شود و با استفاده از این شماره‌ها، به ترتیب، درخواست پردازه‌ها بررسی شده و قفل به همان ترتیب درخواست، تخصیص داده می‌شود.

---

[33] ID
[34] Aging
[35] Ticket Lock

- **عملکرد قفل بلیت به شرح زیر است:**

- دو متغیر به نام‌های بلیت (ticket) و نوبت (turn) را تعریف می‌کنیم.
- هنگام درخواست دریافت قفل توسط پردازه، مقدار متغیر بلیت را به شکل اتمی افزایش داده و در اختیار پردازه قرار می‌دهیم تا نوبتش مشخص شود.
- هر سری پردازه‌ای که نوبتش رسیده(یعنی متغیر مربوط به پردازه برابر نوبت است)، انتخاب می‌شود. پس در هر پردازه باید تا وقتی صبر کند که مقدار ترن برابر تیکتش شود.
- هر پردازه که مقدار بلیتش برابر با متغیر نوبت باشد، اجازه ورود به ناحیه بحرانی[36] را می‌گیرد.
- پردازه بعد از اتمام کارش در ناحیهٔ بحرانی، مقدار نوبت را یک واحد افزایش می‌دهد.

تفاوت ها:

۱) امکان گرسنگی در قفل با اولویت وجود دارد چراکه ترتیب اختصاص قفل در این دو مکانیزم متفاوت میباشد.

۲) از آنجایی که مکانیزم اولویت بندی تغییر یافته، نبازی به آیدی پردازه نیست.

۳) در این روش عدالت بیشتری وجود دارد.

۴) الگوریتم مورد استفاده در Ticket Lock، اولین ورودی - اولین خروجی، می‌باشد. قبلا دیدیم که priority lock، از الگویی برای رتبه بندی هر پردازه استفاده می‌کند.

[36] Critical Section

توضیحات مربوط به کد

- **کد همگام‌سازی در xv6**

کد تغییر یافته این تابع به صورت زیر است. در این تابع تنها شرط برابر بودن آیدی پردازه‌ها بررسی شده است.

```
void
releasesleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  if (lk->pid == myproc()->pid)
  {
    lk->locked = 0;
    lk->pid = 0;
    wakeup(lk);
  }
  release(&lk->lk);
}
```

- **کد پیاده‌سازی متغیرهای مختص هر هستهٔ پردازنده**

به داده ساختار cpu موجود در فایل proc.h، متغیر جدیدی به شرح زیر اضافه می‌کنیم.

```
uint syscall_counter;
```

از آنجایی که در سطح سیستم عامل، دو نوع(سطح) حافظه داریم. اولین نوع پیاده سازی حافظه نهان local پردازنده است که در struct cpu پیاده سازی شده است و نام آن **syscall_counter** است.

```
uint total_syscall_counter;

struct cpu cpus[NCPU];
int ncpu;
uchar ioapicid;
```

با تعریف متغیر `total_syscall_counter` از جنس uint در فایل mp.c (به این علت که این فایل مرتبط با امور multiprocessing است)، بین همگی cpu ها به شکل مشترک تعریف می‌شود (چراکه نقش پیاده‌سازی حافظه‌ی مشترک را داراست). همچنین این متغیر در فایل proc.h نیز extern می‌شود. با هر بار فراخوانی سیستمی، یک واحد به این متغیر جهانی موجود در حافظه‌ی مشترک و به شمارنده موجود در cpu مربوطه، اضافه می‌شود. این متغیر جهانی، تحت مراقبت قرار می‌گیرد تا از وقوع حالت رقابت[37]، جلوگیری شود. برای اینکار موقتا وقفه‌ها را غیر فعال می‌کنیم.

[37] Race Condition

فراخوانی سیستمی تعریف شده، مقدار شمارنده سیستم کال های هر cpu، همینطور شمارنده همگی سیستم کال های متعلق به همگی پردازنده‌ها را خروجی دهد.

```
int sys_print_cpu_syscalls_count(void)
{
    for(int i = 0; i < ncpu; i++)
    {
        cprintf("---CPU %d: %d\n", cpus[i].apicid,
cpus[i].syscall_counter);
    }
    cprintf("---Total: %d\n", total_syscall_counter);
    return 0;
}
```

از آنجایی که پس از اتمام هر بار کاری، متغیر ها باید ریست شوند، تابع سطح سیستمی set_zero_syscall_count در فایل sysextra.c تعریف شده‌است. محل فراخوانی آن هم در فایل sh.c و پس از هربار اجرای case مربوط به EXEC خواهد بود.

```
int sys_set_zero_syscall_count(void)
{
    pushcli();
    for(int i = 0; i < ncpu; i++)
    {
        cpus[i].syscall_counter = 0;
    }
    popcli();
    total_syscall_counter = 0;
    __sync_synchronize();
    return 0;
}
```

از آنجایی که در این پروژه از چهار پردازنده استفاده می‌کنیم، در makefile این تغییر را داریم:

```
ifndef CPUS
CPUS := 4
endif
```

برنامهٔ سطح کاربر نیز، حاوی چهار پردازندهٔ مرکزی درگیر که هر کدام متعلق به یک پردازه هستند، می‌باشد.

- **کد پیاده‌سازی ساز و کار همگام‌سازی با قابلیت اولویت دادن**

به علت پیاده‌سازی قفل جدید، فایل‌هایی به اسامی `prioritylock.c` و `prioritylock.h` تعریف می‌کنیم.

در بخش تعریف ساختار `lock`، مقدار `head` مربوط به صف پردازه‌های منتظر قفل (که پیاده‌سازی آن به شکل لیست پیوندی می‌باشد) را قرار دادیم.
در این پیاده‌سازی صف پردازه‌های منتظر، پردازه‌ای که `head` به آن اشاره می‌کند، بیشترین اولویت برای کسب قفل مربوطه را دارد. به همین علت، حین اضافه کردن یک عضو جدید به لیست، آن را در جای درست قرار می‌دهیم.

```
void add_to_queue(struct prioritylock *plk)
{
  struct node* temp = (struct node *)kalloc();
  temp->priority = myproc()->pid;
  temp->next = 0;
  if(plk->head == 0)
  {
    temp->next = plk->head;
    plk->head = temp;
  }
  else
  {
    struct node* trace = plk->head;
    while(trace->next != 0 && temp->priority <=
trace->next->priority)
    {
      trace = trace->next;
    }
    temp->next = trace->next;
    trace->next = temp;
  }
}
```

همین‌طور یک قفل چرخشی در آن قرار دادیم که از اطلاعات مربوط به قفل در زمان تغییر محافظت کند. از آنجایی که لیست پیوندی، از یک‌سری حلقه تشکیل است، داده‌ساختاری برای آن در این فایل تعریف شده که در آن مقدار اولویت پردازهٔ مدنظر، اشاره‌گری[38] به پردازه مدنظر، پوینتری به حلقهٔ بعدی وجود دارد.

```
struct node{
    int priority;
    struct node *next;
};

struct prioritylock {
  uint pid_locked;       // The process which helds the lock
  struct spinlock slk; // spinlock protecting this sleep lock
  char *name;        // Name of lock.
  struct node *head; // Head of the queue
};
```

در فایل `prioritylock.c` نیز توابعی برای اضافه کردن به صف، حذف از صف، چاپ اولویت های موجود در صف، init کردن قفل اولویت، acquire کردن و در نهایت release کردنش وجود دارد که هرکدام در زیر، توضیح داده شده‌اند:

در تابع `initprioritylock`، قفل مدنظر ساخته شده و متغیر های مربوطه مقداردهی می‌شوند.

```
void initprioritylock(struct prioritylock *plk, char *name)
{
  initlock(&plk->slk, "spin lock");
  plk->name = name;
  plk->pid_locked = 0;
  plk->head = 0;
}
```

تابع `add_to_queue`، قفل مدنظر را به صف پیوندی اضافه می‌کند. روش کار هم بدین شکل است که اگر صف خالی بود، به عنوان تنها حلقهٔ[39] موجود در صف اضافه می‌شود وگرنه، آنقدر جلو می‌رود که

[38] Pointer
[39] Node

به انتهایی ترین نقطه‌ای از صف برسد که اولویت پردازه‌های متعلق به حلقه‌های قبلی تا آن نقطه از صف، بیشتر یا مساوی اولویت پردازه مدنظر باشد. سپس حلقهٔ مربوطه را به صف می‌افزاید.

```
void add_to_queue(struct prioritylock *plk)
{
  struct node* temp = (struct node *)kalloc();
  temp->priority = myproc()->pid;
  temp->next = 0;
  if(plk->head == 0)
  {
    temp->next = plk->head;
    plk->head = temp;
  }
  else
  {
    struct node* trace = plk->head;
    while(trace->next != 0 && temp->priority <=
trace->next->priority)
    {
      trace = trace->next;
    }
    temp->next = trace->next;
    trace->next = temp;
  }
}
```

تابع `print_queue`، با پیمایش روی صف پیاده‌سازی‌شده به شکل لیست پیوندی، مقدار اولویت ها را به ترتیب چاپ می‌کند.

```
void print_queue(struct prioritylock *plk)
```

```
{
  struct node *temp = plk->head;
  cprintf("Queue: ");
  while (temp != 0)
  {
    cprintf("%d ", temp->priority);
    temp = temp->next;
  }
  cprintf("\n");
}
```

با فراخوانی تابع acquirepriority، اگر پردازه‌ای از قبل قفل را در اختیار نداشت، پردازهٔ خواهان قفل، آن را مستقیما دریافت می‌کند. در غیر این صورت، پردازه باید وارد صف پردازه‌های منتظر شود و تا هنگامی که نوبتش برسد، به حالت sleep رود.

```
void acquirepriority(struct prioritylock *plk)
{
  acquire(&plk->slk);
  add_to_queue(plk);
  if (plk->pid_locked != myproc()->pid && plk->pid_locked != 0)
  {
    sleep(myproc(), &plk->slk);
  }
  print_queue(plk);
  plk->pid_locked = myproc()->pid;
  release(&plk->slk);
}
```

در تابع releasepriority، قفل آزاد می‌شود. با آزاد شدن قفل، چک می‌شود که آیا پردازه ای در صف منتظر قفل هست یا خیر. اگر وجود داشت سیستم عامل باید قفل را به این پردازه انتقال دهد و تابع awaken_proc که در فایل proc.c تعریف شده است را فراخوانی کند. این تابع، پس از پیدا کردن پردازه مربوطه از جدول پردازه‌ها، حالت پردازه را به RUNNABLE تغییر داده و آن را بیدار کند.

توجه شود اگر پردازه ای منتظر نبود، قفل آزاد حساب می‌شود وگرنه قفل دوباره در اختیار پردازه‌ی جدید است و آزاد محسوب نمی‌شود.

```
void releasepriority(struct prioritylock *plk)
{
  if(myproc()->pid == plk->pid_locked)
  {
    acquire(&plk->slk);
    if (plk->head == 0 || plk->head->next == 0)
    {
      plk->pid_locked = 0;
      plk->head = 0;
    }
    else
    {
      plk->pid_locked = plk->head->next->priority;
      struct node *temp = plk->head->next;
      plk->head = temp;
      awaken_proc(temp->priority);
    }
    release(&plk->slk);
  }
}
```

```
void awaken_proc(int pid)
{
  struct proc *p;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid == pid)
    {
      p->state = RUNNABLE;
      wakeup(p);
    }
  }
}
```

همچنین برای استفاده از این قفل در سطح کاربر، در فایل sysextera.c ، تعدادی فراخوانی سیستمی پیاده‌سازی شده است.

```
struct prioritylock plk;
```

```
int sys_init_prioritylock(void)
{
    initprioritylock(&plk, "priority lock");
    return 0;
}

int sys_acquire_prioritylock(void)
{
    acquirepriority(&plk);
    return 0;
}

int sys_release_prioritylock(void)
{
    releasepriority(&plk);
    return 0;
}
```

- کد تست مربوط به syscall_count_test:

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

const char *fileNames[] = {
      "output0.txt",
      "output1.txt",
      "output2.txt",
      "output3.txt",
      "output4.txt"};

void write_file(const char *text, int i)
{
      int fd = open(fileNames[i], O_WRONLY | O_CREATE);
      if (fd < 0)
{
```

```
        printf(2, "Failed to open file for writing\n");
        exit();
        }
        write(fd, text, strlen(text));
        close(fd);
}
```

```
int main()
{
        for (int i = 0; i < 5; i++)
        {
        int pid = fork();
        if (pid < 0)
        {
                printf(2, "Fork failed\n");
                exit();
        }
        else if (pid == 0)
        {
                write_file("Dummy text for test!", i);
                exit();
        }
        }
        while (wait() != -1)
        ;
        print_cpu_syscalls_count();
        exit();
}
```

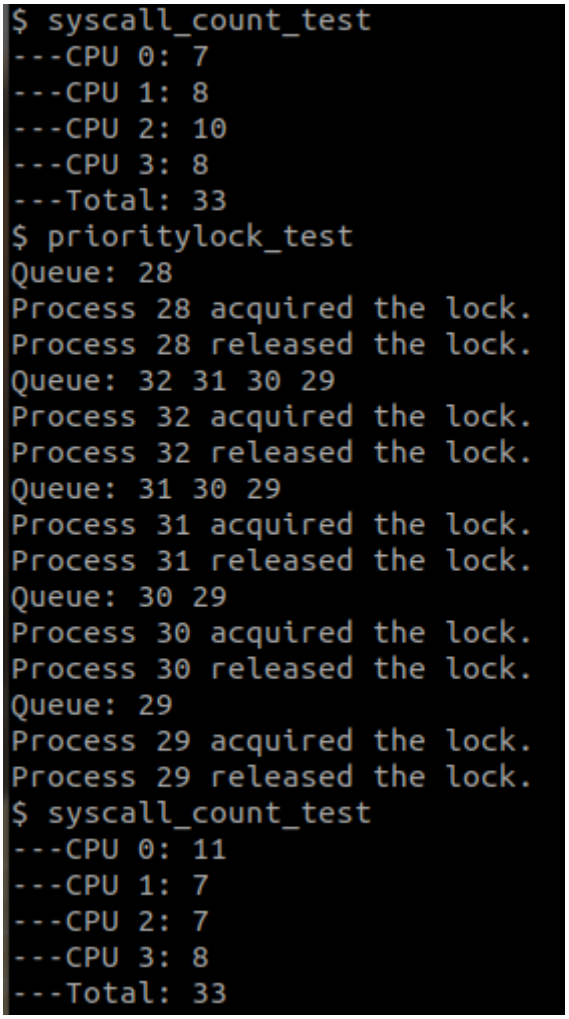
- کد تست مربوط به `prioritylock_test`

```
#include "types.h"
```

```c
#include "user.h"

#define NCHILD 10

void process_function(int i)
{
  acquire_prioritylock();
  printf(1, "Process %d acquired the lock.\n", getpid());
  sleep(500);
  release_prioritylock();
  printf(1, "Process %d released the lock.\n", getpid());
  exit();
}

int main()
{
  init_prioritylock();
  for (int i = 0; i < NCHILD; i++)
  {
      int pid = fork();
      if (pid < 0)
      {
      printf(1, "Fork failed.\n");
      exit();
      }
      else if (pid == 0)
      {
      process_function(i);
      }
  }
  for (int i = 0; i < 10; i++)
  {
      wait();
  }
  exit();
}
```

- خروجی برنامه‌های سطح کاربر، به صورت زیر خواهند بود:

```
init: starting sh
Group #10:
1. Mohamad Amin Yousefi
2. Mobina Mehrazar
3. Matin Nabizade
$ syscall_count_test
---CPU 0: 7
---CPU 1: 11
---CPU 2: 7
---CPU 3: 8
---Total: 33
$ syscall_count_test
---CPU 0: 6
---CPU 1: 7
---CPU 2: 13
---CPU 3: 7
---Total: 33
$ prioritylock_test
Queue: 16
Process 16 acquired the lock.
Process 16 released the lock.
Queue: 20 19 18 17
Process 20 acquired the lock.
Process 20 released the lock.
Queue: 19 18 17
Process 19 acquired the lock.
Process 19 released the lock.
Queue: 18 17
Process 18 acquired the lock.
Process 18 released the lock.
Queue: 17
Process 17 acquired the lock.
Process 17 released the lock.
```



- **Github Hash: 7e38956d9f8cd87c9e596c53020937c4c7ffb0fb**