به نام خدا

آزمایشگاه سیستم عامل - پاییز ۱۴۰۴

استاد: دکتر مهدی کارگهی

پروژه چهارم : همگام‌سازی در xv6

طراحان: محمد امانلو - مصطفی کرمانی‌نیا

تاکنون در دنیای xv6، با پروسه‌ها، حافظه و فایل‌ها آشنا شده‌اید. اما تمام این‌ها در یک دنیای ایزوله ساده بوده‌اند. دنیای واقعی سیستم‌عامل‌ها، دنیای «همروندی» (Concurrency) است. جایی که چندین پردازنده همزمان تلاش می‌کنند به یک لیست، یک بافر یا یک متغیر دسترسی پیدا کنند. بدون "همگام‌سازی"، سیستم‌عامل شبیه به چهارراهی بدون چراغ راهنمایی است که تصادف در آن قطعی است. در این پروژه، ما کالبدشکافی دقیقی روی مکانیزم‌های قفل‌گذاری انجام می‌دهیم. شما یاد می‌گیرید که چگونه قفل‌ها می‌توانند گلوگاه کارایی شوند و چگونه می‌توان آن‌ها را بهینه کرد (Per-CPU). همچنین یاد می‌گیرید که عدالت در سیستم‌عامل همیشه برقرار نیست و گاهی باید با پیاده‌سازی مکانیزم‌های اولویت‌دار، جلوی گرسنگی (Starvation) پروسه‌های مهم‌تر را گرفت.

بخش اول: تحلیل هسته و قفل‌های پیشرفته

در این بخش، با کالبدشکافی رفتار سطح پایین هسته، تفاوت حیاتی میان بستر وقفه و بستر پروسه را درک خواهید کرد و می‌آموزید چرا برای جلوگیری از بن‌بست در یک هسته با قابلیت ورود مجدد، ترکیب قفل‌های چرخشی با غیرفعال‌سازی وقفه‌ها الزامی است.

سوالات تئوری:

1. مفهوم مسیر کنترلی (Control Path) را تعریف کنید. تفاوت بین مسیرهای کنترلی ایجاد شده توسط «فراخوانی سیستمی» (Syscall)، «وقفه» (Interrupt) و «استثنا» (Exception) چیست؟

2. اصطلاح Reentrant Kernel به چه معناست؟ سناریویی را در xv6 شرح دهید که یک پروسه در حال اجرای یک Syscall است، ناگهان وقفه دیسک رخ می‌دهد و CPU مجبور می‌شود مجدداً وارد کد هسته شود. در این حالت اگر همگام‌سازی نباشد، چه خطری داده‌های کرنل را تهدید می‌کند؟

3. تفاوت Process Context و Interrupt Context را شرح دهید.

4. چرا کدی که در Interrupt Context اجرا می‌شود (مثلاً هندلر تایمر)، به هیچ وجه نباید مسدود (Block) شود یا به خواب (Sleep) برود؟

5. چرا برای محافظت از داده‌ها در Interrupt Context، معمولاً از ترکیب Spinlock و Disable Interrupts استفاده می‌شود؟

6. سه رویکرد کلی برای همگام‌سازی وجود دارد. هر یک را در یک پاراگراف تعریف کرده و دو مزیت/معایب برای آن ذکر کنید:

- Spinlock (قفل چرخشی): مناسب برای انتظارهای کوتاه.
- Sleep lock (قفل خواب): مناسب برای انتظارهای طولانی (مثل I/O).
- Lock-free Programming: روشی پیشرفته بدون استفاده از قفل .

7. در تابع acquire، قبل از گرفتن قفل، وقفه‌ها غیرفعال می‌شوند (cli)، چرا؟ فرض کنید وقفه‌ها باز باشند و یک پردازنده قفلی را بگیرد، سپس یک وقفه رخ دهد و هندلر وقفه بخواهد همان قفل را بگیرد. چه اتفاقی می‌افتد؟ (شرح سناریوی Deadlock در سیستم تک‌هسته‌ای).

8. چرا xv6 به جای استفاده مستقیم از cli و sti، از توابع pushcli و popcli استفاده می‌کند؟ این توابع چگونه مشکل "تودرتو بودن نواحی بحرانی" (Nested Critical Sections) را حل می‌کنند؟

سوالات عملی:

قفل‌های خواب (sleeplock) در xv6 یک نقص طراحی دارند: آنها نمی‌دانند چه کسی قفل را گرفته است! این یعنی یک پروسه می‌تواند قفل را بگیرد و پروسه دیگری (حتی به اشتباه) آن را آزاد کند. برای بهبود این وضع:

- در struct sleeplock، یک فیلد برای ذخیره مشخصات مالک (مثلاً pid یا اشاره‌گر به proc) اضافه کنید.
- تابع acquiresleep را طوری تغییر دهید که پس از گرفتن قفل، مالک را ثبت کند.
- تابع releasesleep را ایمن کنید: اگر کسی غیر از مالک قصد آزادسازی قفل را داشت، سیستم باید واکنش نشان دهد (مثلاً panic کند یا پیغام خطا دهد). سیاست خود را در گزارش توجیه کنید .

نهایتا یک برنامه بنویسید که در آن پدر قفل را می‌گیرد و فرزند تلاش می‌کند آن را آزاد کند. سیستم باید جلوی این کار را بگیرد.

طراحی قفل جدید **(انجام این بخش اختیاری بوده و شامل نمره امتیازی است)**

بسیاری از داده‌ها در سیستم‌عامل (مثل لیست فایل‌ها) بیشتر "خوانده" می‌شوند تا "نوشته". استفاده از قفل‌های انحصاری (Mutual Exclusion) برای خوانندگان کارایی را پایین می‌آورد، چون چند نفر می‌توانند همزمان یک متن را بخوانند بدون اینکه مشکلی پیش بیاید.

با الگوبرداری از sleeplock، یک قفل جدید به نام Reader-Writer Lock پیاده‌سازی کنید که قوانین زیر را رعایت کند:

1. چند خواننده همزمان: اگر قفل دست خواننده‌هاست، خوانندگان جدید هم بتوانند وارد شوند.
2. نویسنده انحصاری: اگر یک نویسنده قفل را گرفت، هیچکس (نه خواننده، نه نویسنده دیگر) حق ورود ندارد.
3. پیاده‌سازی: برای ساخت این قفل، شما نیاز به ترکیبی از ابزارهای زیر دارید:
    - یک spinlock برای محافظت از متغیرهای داخلی خودِ قفل.

- یک شمارنده برای تعداد خوانندگان فعال (read_count).
- مکانیزم sleep/wakeup برای منتظر نگه داشتن نویسندگان یا خوانندگان در صورت اشغال بودن منبع.

رابط پیشنهادی (Interface):

```
void rwlock_init(struct rwlock *rw, char *name);
void rwlock_acquire_read(struct rwlock *rw);  // قفل خواندن
void rwlock_release_read(struct rwlock *rw);  // آزادسازی خواندن
void rwlock_acquire_write(struct rwlock *rw); // قفل نوشتن
void rwlock_release_write(struct rwlock *rw); // آزادسازی نوشتن
```

همچنین یک برنامه سطح کاربر بنویسید که سناریوی زیر را شبیه‌سازی کند:

- چندین پروسس "خواننده" که همزمان وارد ناحیه بحرانی می‌شوند (و پیامی چاپ می‌کنند تا همزمانی دیده شود).
- یک یا چند پروسس "نویسنده" که وقتی وارد می‌شوند، هیچکس دیگر در ناحیه بحرانی نیست.

بخش دوم: مقیاس‌پذیری، داده‌های Per-CPU و پروفایلینگ قفل‌ها در xv6

در این بخش، از سطح مفهومی فراتر رفته و با پیاده‌سازی ابزار سنجش رقابت روی قفل‌های xv6، تأثیر حیاتی مقیاس‌پذیری، تداخل حافظه نهان و داده‌های محلی (Per-CPU) بر کارایی و رفع گلوگاه‌های سیستم را به صورت عملی تجربه خواهید کرد.

سوالات تئوری:

1. چرا زمانی که یک پردازنده یک spinlock را در اختیار دارد، حتماً باید وقفه‌ها روی آن پردازنده غیرفعال باشند؟
2. اگر وقفه فعال بماند و هندلر وقفه (Interrupt Handler) بخواهد همان قفل را بگیرد، چه نوع بن‌بستی (Deadlock) رخ می‌دهد؟ (با رسم شکل یا مثال توضیح دهید)
3. با توجه به اسلایدهای درس، اگر چندین هسته پردازنده بخواهند به طور مداوم یک متغیر مشترک سراسری (مثلاً GlobalCounter) را تغییر دهند، پدیده‌ای رخ می‌دهد که باعث کندی

سیستم می‌شود. پروتکل‌های همگام‌سازی حافظه نهان (مانند MESI) در این شرایط چه می‌کنند؟

4. چرا استفاده از متغیرهای محلی (Per-CPU) این مشکل را تا حد زیادی کاهش می‌دهد؟ به‌طور خلاصه توضیح دهید چگونه نگه‌داشتن شمارنده‌ها به‌صورت per-CPU می‌تواند تعداد invalidation های کش را کم کند.
5. تفاوت اصلی بین spinlock (دسته اول توابع در xv6) و sleeplock (دسته دوم توابع) در چیست؟ کدام یک باعث «انتظار مشغول» (Busy Waiting) می‌شود و کدام یک پردازنده را به پروسس دیگری واگذار می‌کند؟

سوالات عملی:

در این بخش برای یک قفل واقعی موجود در xv6 آمار per-CPU جمع‌آوری می‌کنید و از آن یک نمره‌ی رقابت per-CPU می‌سازید. این نمره نشان می‌دهد روی هر هسته‌ی پردازنده، این قفل چقدر پررقابت بوده است. یک قفل دلخواه انتخاب کنید. (مثلاً قفل مربوط به تیک‌های زمانی، قفل صف آماده‌ها، یا قفل دیگری که در کتاب و کد xv6 دیده‌اید).

1. دو آرایه‌ی جدید به ساختار spinlock اضافه کنید تا برای هر CPU آمار جداگانه داشته باشیم.

```
2. uint64 acq_count[NCPU]; // تعداد دفعاتی که این قفل روی هر CPU گرفته شده
3. uint64 total_spins[NCPU]; // مجموع دورهای spin روی هر CPU قبل از گرفتن این قفل
```

4. در تابعی که قفل‌ها مقداردهی اولیه می‌شوند، مطمئن شوید این دو آرایه برای همه‌ی اندیس‌ها از ۰ تا NCPU-1 صفر می‌شوند.
5. در تابعی که مسئول گرفتن قفل است، یک متغیر محلی برای شمارش تعداد دورهای انتظار تعریف و مقداردهی اولیه به صفر کنید. و حلقه‌ای که منتظر آزاد شدن قفل می‌ماند، هر بار که می‌بینید قفل هنوز آزاد نشده است، متغیر محلی تعریف شده را یکی افزایش دهید. به محض این‌که قفل با موفقیت گرفته شد، شناسه‌ی CPU جاری را به‌دست آورید مقدار مربوط به این هسته CPU را در acq_count یکی افزایش داده و total_spins مربوط به این هسته را با مقدار متغیر محلی که تعریف کرده بودید جمع کنید.

به این فکر کنید که چرا معمولاً فرض می‌کنیم نوشتن روی متغیرهای uint64 روی این معماری به اندازه‌ای امن است که برای خودِ آمار نیازی به قفل جداگانه نباشد، و در چه شرایطی این فرض ممکن است مشکل‌ساز شود. (نیازی به نوشتن این بخش در گزارش نیست؛ فقط آن را درک کنید.)

6. در این مرحله، می‌خواهیم آمار جمع‌آوری‌شده را به فضای کاربر منتقل کنیم، آن هم به صورت یک شاخص ترکیبی per-CPU. برای این منظور سیستم کال زیر را تعریف کنید:

```
int sys_getlockstat(uint64 *score);
```

در این سیستم کال در هسته، به قفل انتخاب‌شده دسترسی پیدا کنید (مثلاً با یک اشاره‌گر global به آن قفل). برای هر CPU (از ۰ تا NCPU-1) مقادیر acq_count و total_spins را از ساختار قفل بخوانید و یک نمره‌ی رقابت برای آن CPU محاسبه کنید (فرمول پیشنهادی حاصل نسبت مقدار acq_count مربوط به CPU به مقدار total_spins آن هسته CPU است). این مقادیر را در آرایه‌ای موقت در هسته با طول NCPU ذخیره کنید و در نهایت این آرایه را به آدرس score در فضای کاربر منتقل کنید.

7. در فایل‌های user.h و usys.S تابع زیر را اضافه کنید. در برنامه‌های کاربری، این تابع مثل یک تابع عادی C فراخوانی می‌شود.

```
int getlockstat(uint64 *score);
```

8. یک برنامه‌ی سطح کاربر بنویسید که با استفاده از getlockstat رفتار قفل انتخابی را زیر بار (load) بررسی کند. در ابتدای برنامه، یک آرایه‌ی scores تعریف کنید. قبل از ایجاد بار، یک بار getlockstat را صدا بزنید و مقادیر اولیه‌ی هر CPU را چاپ کنید. چندین فرایند فرزند با fork ایجاد کنید (مثلاً ۴ فرایند). هر فرزند در یک حلقه‌ی نسبتاً طولانی (مثلاً ۱۰۰۰ یا بیشتر تکرار) یک sleep کوتاه انجام دهد (برای توزیع بهتر بار روی CPUها و ایجاد تداخل زمانی) و چند سیستم‌کال ساده اجرا کند که می‌دانید در مسیر آن‌ها قفل انتخابی شما گرفته می‌شود. هدف این است که در طول اجرای این حلقه‌ها، روی قفل انتخابی در هسته رقابت واقعی ایجاد شود تا مقادیر acq_count و total_spins معنی‌دار شوند. فرایند والد با استفاده از wait منتظر اتمام همه‌ی فرزندان بماند. بعد از پایان کار همه‌ی فرزندها، دوباره getlockstat را صدا بزنید و برای هر CPU مقدار نهایی score را چاپ کنید.

9. تحلیل کنید خروجی‌های تولید شده چه برداشتی از نظر رقابت روی قفل و رفتار زمان‌بند (scheduler) به شما می‌دهد؟ و به‌طور خلاصه بیان کنید که چگونه این مکانیزم ساده‌ی پروفایلینگ می‌تواند به پیدا کردن گلوگاه‌های همگام‌سازی در یک سیستم واقعی کمک کند.

بخش سوم: قفل‌های اولویت‌دار و مسئله گرسنگی

در این بخش، با عبور از سیاست‌های همگام‌سازی شانسی، نحوه پیاده‌سازی قفل‌های اولویت‌محور برای سیستم‌های حساس را می‌آموزید و با چالش‌های کلاسیک مدیریت منابع نظیر وارونگی اولویت و گرسنگی به صورت عملی دست‌وپنج نرم خواهید کرد.

در قفل‌های استاندارد spinlock و sleeplock در xv6، سیاست اعطای قفل FIFO یا اولویت‌محور نیست؛ بلکه وقتی قفل آزاد می‌شود، هر پردازنده‌ای که زودتر دستور acquire را اجرا کند (یا زودتر از خواب بیدار شود) برنده است. این رفتار باعث می‌شود پردازه‌های مهم‌تر (با اولویت بالا) ممکن است مدت‌ها پشت قفل بمانند. در این بخش، شما باید ساختار قفل‌گذاری کرنل را تغییر دهید تا به جای شانس، اولویت (Priority) تعیین‌کننده باشد. همچنین با چالش گرسنگی (Starvation) که نتیجه‌ی مستقیم اولویت‌دهی است، آشنا خواهید شد.

سوالات تئوری:

1. آیا در طراحی plock (قفلی که در ادامه پیاده‌سازی می‌کنید)، امکان گرسنگی برای پردازه‌های با اولویت پایین وجود دارد؟ یک سناریوی دقیق (Scenario) بنویسید که در آن یک پردازه با اولویت پایین (Low Priority) درخواست قفل می‌کند، اما با وجود اینکه قفل بارها توسط دیگران گرفته و آزاد می‌شود، این پردازه هرگز موفق به دریافت قفل نمی‌شود .

2. یک مکانیزم دیگر برای مدیریت صف، استفاده از قفل بلیطی (Ticket Lock) است که شبیه سیستم نوبت‌دهی نانوایی عمل می‌کند (FIFO). نحوه کارکرد Ticket Lock را به صورت مفهومی توضیح دهید .

3. قفل اولویت‌دار (plock) را با قفل بلیطی (ticket lock) از نظر انصاف[1] (کدام‌یک عادلانه‌تر رفتار می‌کند؟) پیچیدگی[2] (پیاده‌سازی کدام‌یک سربار محاسباتی کمتری دارد؟) احتمال گرسنگی: کدام‌یک در برابر گرسنگی ایمن است؟

[1] Fairness

[2] Complexity

سوالات عملی:

در قفل‌های استاندارد xv6 (مانند spinlock و sleeplock)، ترتیب اعطای قفل به پردازه‌ها معمولاً شانسی یا وابسته به زمان‌بندی سخت‌افزار است. این موضوع در سیستم‌های بلادرنگ یا حساس، می‌تواند منجر به مشکلاتی نظیر وارونگی اولویت (Priority Inversion) شود. هدف ما اینجا طراحی و پیاده‌سازی یک قفل جدید به نام plock در سطح کرنل است که تضمین کند در شرایط رقابتی، قفل همواره به پردازه‌ای که بالاترین اولویت را دارد اعطا می‌شود.

1. شما باید فایل‌های هدر مناسب (مانند plock.h یا درون spinlock.h) را ویرایش کرده و ساختار داده‌ای برای مدیریت صف انتظار طراحی کنید. در این تمرین، صف لازم نیست مرتب باشد، فقط هنگام release گره max را پیدا کنید. (به برتری‌های صف مرتب شده به حالت فعلی فکر کنید.)

   ساختار گره (Node): هر پردازه‌ای که منتظر قفل است، باید در قالب یک گره در صف ذخیره شود. این ساختار باید حداقل شامل اطلاعات پردازه (یا اشاره‌گر به struct proc)، مقدار اولویت (priority) و اشاره‌گری به نفر بعدی در صف باشد.

   ساختار قفل (plock): این ساختار باید وضعیت کلی قفل را مدیریت کند. اجزای پیشنهادی عبارتند از: یک spinlock داخلی (برای محافظت از داده‌های خودِ قفل)، وضعیت قفل (locked) که مشخص کند قفل آزاد است یا اشغال و اشاره‌گری به ابتدای صف انتظار (Head of Queue).

2. توابع زیر را پیاده‌سازی کنید:

   تابع void plock_init(struct plock *pl): قفل را مقداردهی اولیه کنید؛ وضعیت قفل باید آزاد باشد و صف انتظار خالی. یک نمونه‌ی global از این قفل را در کرنل تعریف کنید (مثلاً global_plock) و در زمان بوت سیستم (در کدی شبیه main یا userinit) همین تابع plock_init را روی آن صدا بزنید. توجه کنید که این مقداردهی اولیه در خودِ کرنل و هنگام boot انجام می‌شود و نیازی به system call جداگانه برای init از سمت کاربر نیست. برای این تمرین، همین یک قفل سراسری کافی است و نیازی به چند plock مختلف نیست.

   تابع void plock_acquire(struct plock *pl, int priority): این تابع درخواست دریافت قفل با یک اولویت مشخص است. اگر قفل آزاد است، بلافاصله آن را بگیرید (locked = 1) و برگردید. اگر قفل اشغال است، یک گره‌ی جدید حاوی مشخصات پردازه‌ی جاری و اولویت آن بسازید، گره

را به صف انتظار اضافه کنید (می‌توانید گره‌های جدید را در ابتدای لیست قرار دهید و بعداً هنگام آزادکردن قفل، گره با بالاترین اولویت را پیدا کنید) و پردازه را به حالت خواب (sleep) ببرید. طراحی باید به گونه‌ای باشد که وقتی پردازه از خواب بیدار می‌شود، به این معنی باشد که قفل به او واگذار شده است (Handoff) و نیازی به رقابت مجدد برای گرفتن قفل نداشته باشد.

تابع void plock_release(struct plock *pl) این تابع قفل را آزاد می‌کند و تصمیم می‌گیرد نفر بعدی کیست. اگر صف انتظار خالی است، قفل را آزاد کنید (locked = 0) و برگردید. اگر صف انتظار پر است، کل لیست را پیمایش کنید و گره‌ای که بالاترین اولویت را دارد پیدا کنید، آن گره را از لیست حذف کنید، مالکیت قفل را مستقیماً به آن پردازه منتقل کنید (یعنی می‌توانید locked را همچنان ۱ در نظر بگیرید و فقط مالک را عوض کنید) و فقط همان پردازه‌ی منتخب را با مکانیزم wakeup بیدار کنید. به این ترتیب، قفل به صورت مستقیم از صاحب قبلی به صاحب جدید تحویل داده می‌شود و لازم نیست قفل آزاد شود تا همه برای آن هجوم بیاورند.

3. برای تست این قابلیت در فضای کاربر، دو فراخوانی سیستمی جدید sys_plock_acquire(int priority) و sys_plock_release را پیاده‌سازی کنید. که در نهایت همان توابع plock_acquire و plock_release را روی plock سراسری کرنل (مثلاً global_plock) فراخوانی می‌کنند. (در صورت تمایل می‌توانید به جای استفاده از global_plock یک سیستم‌کال نیز برای plock_init تعریف کنید تا از مشکلات احتمالی جلوگیری کنید. به مشکلات احتمالی مربوط به پیاده‌سازی فعلی فکر کنید.)
4. حالا برنامه‌ای بنویسید که صحت عملکرد اولویت‌دهی را اثبات کند. در این برنامه، چند پردازه فرزند با اولویت‌های مختلف (مثلاً ۱۰، ۲۰، ۳۰، ۴۰، ۵۰) ایجاد کنید. سناریو را طوری طراحی کنید که یک پردازه‌ی با اولویت پایین‌تر زودتر درخواست قفل دهد و قفل را بگیرد، سپس در همین حین پردازه‌های با اولویت بالاتر نیز درخواست plock_acquire بدهند و در صف منتظر بمانند. پس از آزادسازی قفل توسط پردازه‌ی اول، سیستم باید قفل را به ترتیب اولویت نزولی (از بالاترین به پایین‌ترین) به پردازه‌ها بدهد، فارغ از این‌که کدام یک زودتر درخواست داده است. در هر مرحله، PID و اولویت پردازه‌ای که قفل را می‌گیرد روی خروجی چاپ کنید تا بتوانید مشاهده کنید که قفل واقعاً بر اساس اولویت عمل می‌کند، نه فقط بر اساس ترتیب ورود.

سایر نکات

- پروژه خود را در یک مخزن خصوص در Github پیش برده و در نهایت یک نفر از اعضای گروه کدها را به همراه پوشهٔ .git زیپ کرده و در سامانه با فرمت OS-Lab4-<SID1>-<SID2>-<SID3>.zip آپلود نمایید.
- بخشی از نمرهٔ شما را پاسخ دهی به سوالات تئوری تشکیل می‌دهد. (نیازی به پاسخ‌دهی به سوالات سایر بخش‌ها یا گزارش کار نیست) لطفا پاسخ سوال‌ها را مختصر و مفید بنویسید.
- در پیاده‌سازی خود در مواردی که ذکر نشده‌اند فرض منطقی و دلخواه خود را داشته باشید.
- تابع printf در سطح برنامه‌های کاربر و تابع cprintf در سطح کرنل و همچنین ابزار gdb، امکانات خوبی در هنگام عیب‌یابی می‌باشند، در صورت نیاز آن‌ها را به کار بگیرید.
- همه افراد می‌بایست به پروژه مسلط باشند و نمره تمامی اعضای گروه لزوما یکسان نیست.
- تمامی مواردی که در جلسه توجیهی، گروه تیمز و فروم درس مطرح می‌شوند، جزئی از پروژه خواهند بود. در صورت وجود هرگونه سوال یا ابهام میتوانید با ایمیل دستیاران مربوطه یا گروه اسکایپی درس در ارتباط باشید.
- این تمرین صرفا برای یادگیری شما طرح شده است. در صورت محرز شدن تقلب در تمرین، مطابق با قوانین درس برخورد خواهد شد.

موفق باشید!