

Project 5 report

رضا چهرقانی	810101401
امیر نداف فهمیده	810101540
مصطفی کرمانی نیا	810101575

مخزن گیتهاب این پروژه:

<https://github.com/mostafa-kermaninia/OS-LAB-5>

آخرین کامیت:

aa32aa625a939bfa2688ffdeb067b4d03a09bf9a

پرسش 1) علت غیرفعال کردن وقفه چیست؟ توابع popcli و pushcli به چه منظور استفاده شده و چه تفاوتی با cli و sti دارند؟

تضمين اجرای اتميک در بخش‌های بحراني (Critical Sections)

با غيرفعال کردن وقفه‌ها، اطمینان حاصل می‌شود که کدی که در حال اجرا است به صورت اتمیک انجام شده و وقفه‌ها باعث ناتمام ماندن عملیات یا شرایط ناسازگار نمی‌شوند.

جلوگیری از تغییر ناگهانی زمینه (Context Switch)

اگر وقفه‌ای در هنگام اجرای کد در بخش بحرانی رخ دهد، ممکن است پردازنده وظیفه دیگری را اجرا کند که به داده‌های مشابه دسترسی داشته باشد و باعث ناسازگاری داده‌ها شود.

پیشگیری از بن‌بست (Deadlock)

اگر در هنگام اجرای بخش بحرانی، وقفه‌ای رخ دهد و سرویس‌دهنده وقفه سعی کند به قفلی که در دسترس نیست دسترسی پیدا کند، ممکن است بن‌بست ایجاد شود.

حفظ یکپارچگی داده‌ها:

در بخش‌های بحرانی معمولاً داده‌های مشترک به روزرسانی می‌شوند. غیرفعال کردن وقفه‌ها از بروز شرایط رفاقتی (Race Conditions) جلوگیری می‌کند.

توابع `popcli` و `pushcli` به چه منظور استفاده شده‌اند؟

در XV6، توابع `popcli` و `pushcli` برای مدیریت وضعیت وقفه‌ها به صورت لایه‌ای (nested) استفاده می‌شوند و این توابع در فایل `spinlock.c` پیاده‌سازی شده‌اند.

```
// Pushcli/popcli are like cli/sti except that they are matched:  
// it takes two popcli to undo two pushcli. Also, if interrupts  
// are off, then pushcli, popcli leaves them off.  
  
void pushcli(void)  
{  
    int eflags;  
  
    eflags = readeflags();  
    cli();  
    if (mycpu()>ncli == 0)  
        mycpu()>intena = eflags & FL_IF;  
    mycpu()>ncli += 1;  
}  
  
void popcli(void)  
{  
    if (readeflags() & FL_IF)  
        panic("popcli - interruptible");  
    if (--mycpu()>ncli < 0)  
        panic("popcli");  
    if (mycpu()>ncli == 0 && mycpu()>intena)  
        sti();  
}
```

:pushcli •

- ❖ این تابع وقفه‌ها را در قسمت بحرانی کد غیرفعال میکند و از deadlock جلوگیری میکند. به این شکل که وضعیت فعلی وقفه‌ها را با استفاده از `readeflags()` می‌خواند.
- ❖ سپس با `cli()` وقفه‌ها را غیرفعال می‌کند.
- ❖ اگر این اولین بار باشد که وقفه‌ها غیرفعال می‌شوند (یعنی شمارنده `ncli` صفر است)، وضعیت فعلی وقفه‌ها در متغیری به نام `intena` ذخیره می‌شود.
- ❖ شمارنده `ncli` (تعداد دفعات غیرفعال شدن وقفه‌ها در سطح تودرتو) افزایش می‌یابد.

:popcli •

- ❖ بررسی می‌کند که وقفه‌ها فعال نباشند. اگر فعال باشند، خطای سیستمی (panic) ایجاد می‌کند.
- ❖ شمارنده `ncli` کاهش می‌یابد و بررسی می‌شود که از صفر کمتر نشود.
- ❖ اگر شمارنده به صفر برسد و وضعیت ذخیره شده قبلی نشان‌دهنده فعال بودن وقفه‌ها باشد، وقفه‌ها با دستور `sti()` فعال می‌شوند.

این توابع زمانی استفاده می‌شوند که ممکن است چندین بخش کد به صورت تودرتو وقفه‌ها را غیرفعال کنند، و نیاز است که فقط در زمانی که همه بخش‌ها تمام شدند، وقفه‌ها مجددًا فعال شوند.

تفاوت `sti` و `cli` و `popcli` و `pushcli`

طبق توضیح خود MIT در کدها:

```
// Pushcli/popcli are like cli/sti except that they are
matched:

// it takes two popcli to undo two pushcli. Also, if
interrupts

// are off, then pushcli, popcli leaves them off.
```

با جزئیات بیشتر:

:sti و cli .1

```
static inline void
cli(void)
{
    __asm volatile("cli");
}

static inline void
sti(void)
{
    __asm volatile("sti");
}
```

- دستورات سطح پایین پردازنده هستند.
- وقفه‌ها را غیرفعال می‌کند.
- وقفه‌ها را فعال می‌کند.
- این دستورات ساده هستند و قابلیت مدیریت چندلایه‌ای (Nested Interrupt) را ندارند.

:popcli و pushcli .2

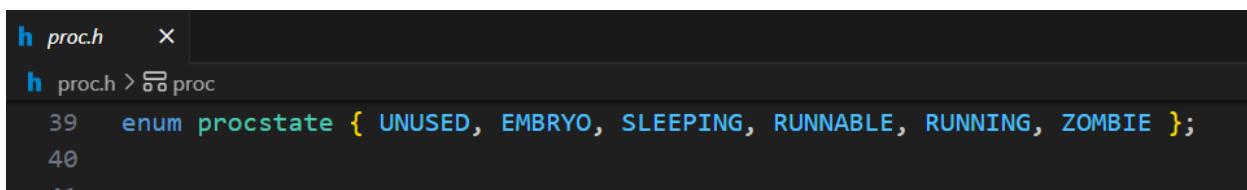
- این توابع در لایه‌ی سیستم‌عامل تعریف شده‌اند و از cli و sti برای مدیریت وقفه‌ها استفاده می‌کنند.
- قابلیت مدیریت چندلایه‌ای دارند، به طوری که اگر وقفه‌ها چندین بار غیرفعال شوند، فقط پس از اتمام تمام لایه‌ها، وقفه‌ها فعال می‌شوند.

نهایتاً در XV6، توابع popcli و pushcli با استفاده از دستورات cli و sti، مدیریت پیشرفته‌تری برای وضعیت وقفه‌ها فراهم می‌کنند. این کار برای جلوگیری از شرایط رقابتی، بن‌بست (اگر وقفه‌ها در هنگام

اجرای کد بحرانی فعال بمانند، امکان دارد یک وقفه به همان قفل دسترسی پیدا کند و باعث بنبست شود.) و اطمینان از اجرای اتمیک عملیات بسیار حیاتی است.

پرسش 2) حالات مختلف پردازه‌ها در XV6 را توضیح دهید. تابع `sched` چه وظیفه‌ای دارد؟

در سیستم‌عامل XV6، پردازه‌ها در طول چرخه حیات خود می‌توانند در یکی از حالات زیر قرار گیرند. این حالات در ساختار `enum procstate` تعریف شده‌اند:



```
h proc.h x
h proc.h > proc
39 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
40
```

:UNUSED .1

- پردازه‌ای که از جدول پردازه‌ها (process table) استفاده نمی‌کند.
- هیچ منبعی به این پردازه اختصاص داده نشده است.

:EMBRYO .2

- پردازه‌ای که در حال ایجاد شدن است.
- منابع اولیه مانند حافظه و توصیف‌گر فایل به آن تخصیص داده می‌شود.

:SLEEPING .3

- پردازه‌ای که منتظر وقوع یک رویداد یا آزاد شدن یک منبع است.
- در این حالت، پردازه توسط زمان‌بند انتخاب نمی‌شود.

:RUNNABLE .4

- پردازه‌ای که آماده اجراست و همه منابع لازم برای اجرا را دارد.
- منتظر زمان‌بند است تا به آن CPU اختصاص دهد.

- انتقال به حالت RUNNABLE: یک پردازه می‌تواند از حالات مختلف به حالت RUNNABLE منتقل شود. مهم‌ترین این مسیرها عبارتند از:
 1. پردازه جدید: هنگامی که یک پردازه جدید با موفقیت ایجاد می‌شود و مراحل اولیه مانند تخصیص حافظه و تنظیمات پایه انجام می‌شود، به حالت RUNNABLE منتقل می‌شود. مثلاً پس از فراخوانی تابع `fork()`، فرزند جدید در این حالت قرار می‌گیرد.
 2. بازگشت از حالت SLEEPING: اگر پردازه‌ای منتظر یک رویداد یا منبع باشد (در حالت SLEEPING) و آن رویداد رخ دهد یا منبع آزاد شود، پردازه به حالت RUNNABLE برمی‌گردد. مثال: پس از اتمام `O/I` یا آزاد شدن یک قفل.
 3. پیش‌دستی توسط زمان‌بند (Preempted): اگر پردازه‌ای در حال اجرا (RUNNING) باشد اما به دلیل اتمام زمان تخصیص داده شده (Time Slice) توسط زمان‌بند متوقف شود، به حالت RUNNABLE بازی‌گردد. این رفتار به منظور پشتیبانی از چندوظیفه‌ای (Multitasking) انجام می‌شود.
 4. تکمیل عملیات `O/I`: پردازه‌ای که منتظر تکمیل عملیات ورودی/خروجی (`O/I`) بوده، پس از اتمام این عملیات، به حالت RUNNABLE منتقل می‌شود.
 5. آزاد شدن قفل (Released Lock): اگر پردازه‌ای در حالت SLEEPING منتظر یک قفل باشد، با آزاد شدن آن قفل، به حالت RUNNABLE بازی‌گردد.
 6. دریافت سیگنال (Signal Handling): پردازه‌ای که در حالت SLEEPING قرار دارد و سیگنالی دریافت می‌کند که نیاز به پردازش دارد، به حالت RUNNABLE منتقل می‌شود. این انتقال به دلیل اهمیت پاسخ‌دهی سریع به سیگنال‌ها انجام می‌شود.
- :RUNNING .5**
- پردازه‌ای که در حال حاضر روی CPU اجرا می‌شود.
 - در هر لحظه، تنها یک پردازه در هر CPU می‌تواند در این حالت باشد.
- :ZOMBIE .6**
- پردازه‌ای که اجرای آن تمام شده است، اما والد آن هنوز وضعیت خروج (exit status) آن را دریافت نکرده است.

- جدول پردازه‌ها باید تا زمان فراخوانی تابع wait() توسط والد، اطلاعات این پردازه را نگه دارد.

تابع sched بخشی از زمان‌بند (scheduler) سیستم‌عامل است که وظیفه مدیریت پردازه‌ها و تخصیص CPU به آن‌ها را دارد.

```
// Enter scheduler. Must hold only ptable.lock
// and have changed proc->state. Saves and restores
// intena because intena is a property of this
// kernel thread, not this CPU. It should
// be proc->intena and proc->ncli, but that would
// break in the few places where a lock is held but
// there's no process.
void sched(void)
{
    int intena;
    struct proc *p = myproc();

    if (!holding(&ptable.lock))
        panic("sched ptable.lock");
    if (mycpu()->ncli != 1)
        panic("sched locks");
    if (p->state == RUNNING)
        panic("sched running");
    if (readeflags() & FL_IF)
        panic("sched interruptible");
    intena = mycpu()->intena;
    swtch(&p->context, mycpu()->scheduler);
    mycpu()->intena = intena;
}
```

وظایف اصلی این تابع عبارتند از:

1. انتخاب پردازه بعدی برای اجرا:

پردازه‌ای را از حالت RUNNING انتخاب کرده و آن را به حالت RUNNING منتقل می‌کند.

2. تغییر زمینه (Context Switching):

زمانی که یک پردازه به حالت SLEEPING، ZOMBIE یا UNUSED منتقل می‌شود، sched تغییر زمینه را انجام می‌دهد و پردازه دیگری را اجرا می‌کند. تغییر زمینه شامل ذخیره وضعیت پردازه فعلی و بازیابی وضعیت پردازه جدید است.

3. مکانیزم خواب و بیداری (Sleep and Wakeup):

از توابع sleep و wakeup برای مدیریت شرایطی که پردازه‌ها باید منتظر یک رویداد باشند، استفاده می‌کند. این مکانیزم شبیه به wait و signal در متغیرهای شرطی است.

4. چندوظیفه‌ای (Multiplexing):

اگر دو یا چند پردازه بخواهند از یک CPU استفاده کنند، sched با تغییر سریع بین آنها، توهیمی ایجاد می‌کند که هر پردازه به صورت همزمان اجرا می‌شود.

5. بررسی شرایط ایمنی:

- بررسی می‌کند که آیا قفل‌های لازم در هنگام اجرای تابع sched نگه داشته شده‌اند.
- اطمینان از عدم وقوع شرایط بحرانی (Race Condition) یا وضعیت نامعتبر.
- توضیح مراحل کار تابع sched

بررسی شرایط قبل از اجرا:

- ابتدا بررسی می‌کند که آیا قفل پردازه فعلی نگه داشته شده است یا خیر. اگر این قفل در اختیار نباشد، سیستم متوقف می‌شود. این تضمین می‌کند که هیچ پردازه دیگری نتواند وضعیت این پردازه را تغییر دهد.
- سپس بررسی می‌کند که تعداد قفل‌ها روی CPU برابر یک باشد. این شرط از وقوع شرایط بحرانی جلوگیری می‌کند.
- حالا اگر پردازه هنوز در حالت RUNNING باشد (در حال اجرا)، فراخوانی sched منطقی نیست و سیستم متوقف می‌شود.
- همچنان بررسی می‌کند که وقفه‌ها غیرفعال باشند. اگر وقفه‌ها فعال باشند، ممکن است پردازه در میانه تغییر زمینه متوقف شود که خطرناک است.

ذخیره وضعیت وقفه‌ها:

- وضعیت فعلی وقفه‌ها ذخیره می‌شود تا بعداً بازیابی شود: `;intena = mycpu()->intena`

تغییر زمینه:

تغییر زمینه انجام می‌شود:

- وضعیت پردازه فعلی (`p->context`) ذخیره می‌شود.
- کنترل به زمانبند (`mycpu()->scheduler`) منتقل می‌شود.

وضعیت وقفه‌ها به حالت قبلی بازگردانده می‌شود.

پس نهایتاً تابع sched ابتدا تضمین می‌کند که قبل از اجرای تغییر زمینه، تمام شرایط بحرانی کنترل شده باشند، سپس پردازه فعلی از حالت اجرا خارج می‌شود و زمانبند کنترل CPU را بر عهده می‌گیرد. نهایتاً زمانبند پردازه جدیدی را انتخاب می‌کند و کنترل را به آن منتقل می‌کند. این انتقال از طریق تابع swtch انجام می‌شود.

پرسش 3) یکی از روش‌های سینک کردن این حافظه‌های نهان با یکدیگر روش مخصوص است. آن را به اختصار توضیح دهید.

روش MSI (Modified-Shared-Invalid) یک cache coherence protocol است که در سیستم‌های چندپردازنده استفاده می‌شود تا اطمینان حاصل شود که داده‌های ذخیره‌شده در حافظه نهان (cache) پردازنده‌ها و حافظه اصلی (main memory) همگام و سازگار هستند با این حال، به دلیل مشکلات عملکردی در شرایط خاص، پروتکل‌های پیشرفته‌تری مثل MESI و MOESI در سیستم‌های مدرن استفاده می‌شوند.

این پروتکل هر بلوک داده (cache line) را به یکی از سه حالت زیر تقسیم می‌کند:

:(Modified) M . 1

داده در حافظه نهان تغییر یافته است (dirty data) این داده فقط در حافظه نهان یک پردازنده وجود دارد و نسخه موجود در حافظه اصلی نامعتبر است. اگر این داده باید با سایر پردازنده‌ها به اشتراک گذاشته شود یا از حافظه نهان حذف شود، ابتدا باید آن را در حافظه اصلی ذخیره کند (write-back).

:(Shared) S . 2

داده در حافظه نهان بیش از یک پردازنده وجود دارد و در حافظه اصلی نیز معتبر است. این حالت به معنای "فقط خواندن" است. هیچ پردازنده‌ای نمی‌تواند داده را تغییر دهد مگر اینکه به حالت Modified منتقل شود.

:(Invalid) I . 3

داده در حافظه نهان پردازنده معتبر نیست. این حالت زمانی رخ می‌دهد که داده تغییر کرده باشد و نسخه جدید آن در حافظه نهان پردازنده دیگری یا در حافظه اصلی ذخیره شده باشد.

انتقال بین حالات در پروتکل MSI

پروتکل MSI از پیام‌های بین پردازنده‌ها (Bus Messages) استفاده می‌کند تا انتقال بین حالات مختلف داده را مدیریت کند. در ادامه، انتقال‌های رایج بین حالات توضیح داده می‌شود:

: $(M \rightarrow S)$. 1

زمانی که پردازنده دیگری درخواست داده‌ای را می‌دهد که در حالت Modified است، داده ابتدا به حافظه اصلی نوشته می‌شود (write-back) و سپس به حالت Shared تبدیل می‌شود.

: $(S \rightarrow M)$. 2

زمانی که پردازنده نیاز به تغییر داده‌ای دارد که در حالت Shared است، درخواست بهروزرسانی (invalidate) برای سایر پردازنده‌ها ارسال می‌شود سپس نسخه‌های داده در حافظه‌های نهان دیگر پردازنده‌ها به حالت Invalid تغییر می‌کنند. و حالت داده در حافظه نهان پردازنده محلی به Modified تغییر می‌کند.

: $(S \rightarrow I)$. 3

اگر پردازنده دیگری داده‌ای را که در حالت Shared قرار دارد تغییر دهد، نسخه‌های دیگر داده در حافظه نهان به Invalid تغییر می‌کنند. یعنی داده در حافظه نهان پردازنده محلی دیگر معتبر نیست و باید در صورت نیاز، مجدداً از حافظه اصلی یا پردازنده دیگر خوانده شود.

: $(I \rightarrow S)$. 4

اگر پردازنده‌ای بخواهد داده‌ای را که در حالت Invalid است بخواند، نسخه معتبر داده از حافظه اصلی یا حافظه نهان پردازنده دیگری که آن را در حالت Shared یا Modified دارد، دریافت می‌شود. سپس حالت داده در حافظه نهان پردازنده محلی به Shared تغییر می‌کند.

زمانی که پردازنده نیاز به نوشتن (write) روی داده‌ای دارد که در حالت Invalid است. ابتدا باید نسخه معتبر آن را از حافظه اصلی یا پردازنده دیگری که مالک آن است دریافت کند و سپس حالت داده در حافظه نهان پردازنده محلی به **Modified** تغییر می‌کند.

مشکلاتی که MSI حل می‌کند:

1. اطمینان می‌دهد که هر پردازنده به نسخه معتبر داده دسترسی دارد.
2. جلوگیری از ناسازگاری: پردازنده‌ها نمی‌توانند داده‌های مشترک را به طور همزمان تغییر دهند.
3. داده‌ها تا جای ممکن در حافظه نهان ذخیره می‌شوند و از انتقال‌های غیرضروری جلوگیری می‌شود.

معایب پروتکل MSI

1. کارایی در خواندن مشترک:

داده‌های مشترک اغلب به حالت Invalid منتقل می‌شوند، حتی اگر فقط نیاز به خواندن باشند. این امر منجر به کاهش کارایی می‌شود. این مشکل در پروتکل‌های پیشرفته‌تر مثل MESI (Modified-Exclusive-Shared-Invalid) برطرف شده است.

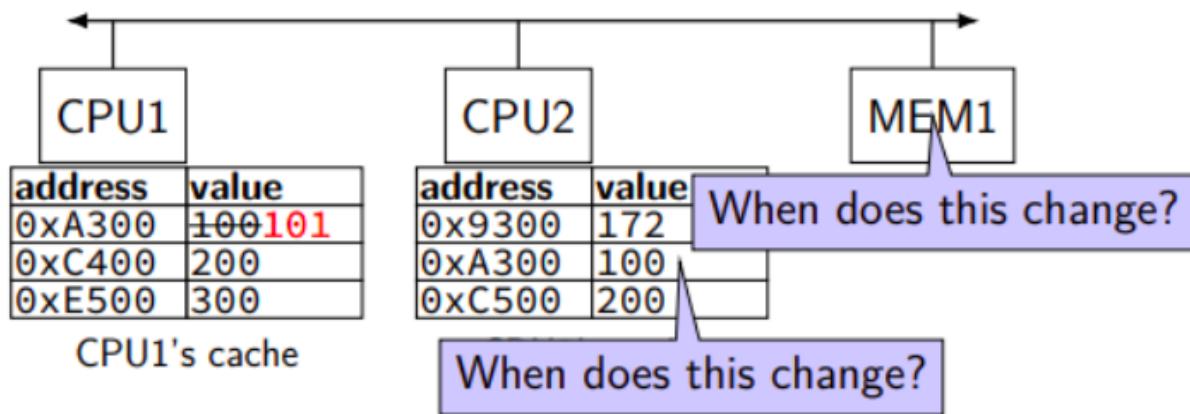
2. پیام‌های اضافی روی گذرگاه (Bus):

به دلیل استفاده از پیام‌های invalidate، بار روی گذرگاه سیستم ممکن است افزایش یابد.

پرسش 4) یکی از روش‌های همگام‌سازی استفاده از قفل‌های معروف به قفل بلیت است. این قفل‌ها را از منظر مشکل مذکور در بالا بررسی نمایید.

قفل بلیت (Ticket Lock) یک الگوریتم همگام‌سازی است که مانند صف بلیت در فروشگاهها یا نانوایی‌ها عمل می‌کند. در این روش، به هر پردازنده (یا رشته) که قصد دسترسی به بخش بحرانی (Critical Section) را دارد، یک شماره بلیت اختصاص داده می‌شود. این پردازنده‌ها باید به ترتیب شماره بلیت خود وارد بخش بحرانی شوند.

ساختار قفل بلیت شامل یک Queue Ticket است که نشان‌دهنده شماره بلیتی است که به آخرین پردازنده در صف اختصاص داده شده است و یک Dequeue Ticket که نشان‌دهنده شماره بلیتی است که در حال حاضر پردازنده دارای آن می‌تواند وارد بخش بحرانی شود و هر دو شمارنده در ابتدا مقدار 0 دارند.



نکته این است که با افزایش تعداد پردازه‌ها کارایی این روش به صورت نمایی افت پیدا می‌کند. همچنین مشکل تصویر بالا که گفته شد، برای این قفل وجود دارد. بدین صورت که تمام CPU‌ها در cache خود متغیر مشترک بلیت در حال اجرا را دارند و وقتی که قفل رها می‌شود، مقدار این متغیر در تمام CPU‌ها نادرست می‌شود و بدین ترتیب تمام CPU‌ها باید مقدار جدید متغیر را از حافظه بخوانند.

پس از منظر هماهنگی حافظه نهان، قفل بلیت از مزایای مکانیزم‌های موجود در پروتکل‌های هماهنگی حافظه نهان مانند MSI یا MSI بهره می‌برد:

- هنگامی که یک پردازنده مقدار `dequeue_ticket` را به روزرسانی می‌کند، این تغییر از طریق پروتکل هماهنگی حافظه نهان به دیگر پردازنده‌ها منتقل می‌شود.
- زمانی که یک پردازنده قفل را به دست می‌آورد، این قفل در حالت Exclusive قرار می‌گیرد. این امر تضمین می‌کند که هیچ پردازنده دیگری نمی‌تواند همزمان به این قفل دسترسی پیدا کند.
- با به روزرسانی قفل، تمامی پردازنده‌ها از تغییرات مطلع می‌شوند و می‌توانند رفتار خود را متناسب با آن تنظیم کنند.

مزایای استفاده از قفل بلیت:

1. از لحاظ هزینه حافظه مناسب است زیرا تمام CPU‌ها در حال بررسی یک متغیر global مشترک هستند بعنوان شماره بلیط در حال اجرا و اخرين بلیط داده شده.
2. منصفانه بودن: دسترسی پردازنده‌ها به منابع مشترک به ترتیب درخواست (FCFS) انجام می‌شود. و گرسنگی رخ نمیدهد. از طرفی در این مدل همگام سازی، رفتار مشخص و قبل پیشبینی است.
3. هماهنگی خودکار با مکانیزم Cache Coherence: هرگونه تغییر در قفل و داده‌ها از طریق پروتکل‌های هماهنگی حافظه نهان (مانند MSI) به سایر پردازنده‌ها منتقل می‌شود.

پرسش 5) دو مورد از معایب استفاده از قفل با امکان ورود مجدد را بیان نمایید.

```

130 struct reentrantlock {
131     struct spinlock lock;    // Underlying spinlock for atomicity
132     struct proc *owner;      // Current owner of the lock
133     int recursion;           // Recursion depth for reentrancy
134 };
135

```

قفل‌های بازگشتشی (Reentrant Locks) به برنامه‌ها اجازه می‌دهند که یک رشته (Thread) بتواند چندین بار متوالی بدون ایجاد بن‌بست (Deadlock) یک قفل را به دست آورده و آزاد کند. با این حال، استفاده از این نوع قفل‌ها می‌تواند معایبی نیز داشته باشد:

1. پیچیدگی در پیاده‌سازی و نگهداری:

پیاده‌سازی قفل‌های بازگشتشی در XV6 نیازمند مکانیزم‌هایی مانند شمارنده عمق بازگشتشی (recursion depth) و مدیریت مالکان قفل (owner) است. این موارد می‌توانند پیاده‌سازی را پیچیده کرده و احتمال بروز خطا در کد را افزایش دهند. در XV6، قفل‌های معمولی (مانند spinlock) ساده‌تر هستند، زیرا نیازی به مدیریت مالکان قفل یا عمق بازگشتشی ندارند. این پیچیدگی می‌تواند در سیستم‌هایی که بهینه‌سازی و کارایی اهمیت دارند، به یک ضعف تبدیل شود.

2. کاهش کارایی:

قفل‌های بازگشتشی به دلیل نیاز به مدیریت اضافه، مانند شمارش عمق بازگشتشی (recursion depth)، و بررسی‌های مکرر برای مالکیت قفل، سریار بیشتری نسبت به قفل‌های ساده ایجاد می‌کنند. در سیستم‌های کم منبع مانند XV6، این افزایش سریار می‌تواند عملکرد کلی سیستم را کاهش دهد.

3. ایجاد وابستگی‌های غیرضروری

در XV6، طراحی مبتنی بر قفل‌های ساده‌تر (spinlock) منجر به کاهش وابستگی بین بخش‌های مختلف سیستم می‌شود. استفاده از قفل‌های بازگشتشی می‌تواند این سادگی را مختل کند مثلاً پردازه یا رشته‌ای که وارد قفل شده است ممکن است ناخواسته در چرخه‌های بی‌پایان بازگشتشی قرار گیرد. این وابستگی‌های پیچیده مدیریت کد را دشوارتر می‌کند.

4. افزایش احتمال بن‌بست‌های پیچیده:

- در شرایطی که چندین قفل بازگشتشی به صورت تو در تو استفاده شوند، احتمال وقوع بن‌بست‌های پیچیده افزایش می‌یابد که تشخیص و رفع آن‌ها دشوار است.

5. سازگاری محدود با سیستم‌های چندپردازنده‌ای:

-
- در سیستم‌های چندپردازنده‌ای، استفاده نادرست از قفل‌های بازگشتی می‌تواند به مشکلات هماهنگی حافظه نهان (Cache Coherence) و کاهش کارایی منجر شود.

با توجه به این معایب، استفاده از قفل‌های بازگشتی باید با دقت و در موارد ضروری انجام شود تا از بروز مشکلات عملکردی و پیچیدگی‌های غیرضروری در سیستم جلوگیری شود.

پرسش 6) یکی دیگر از ابزارهای همگام‌سازی قفل Read-Write lock است. نحوه کارکرد این قفل را توضیح دهید و در چه مواردی این قفل نسبت به قفل با امکان ورود مجدد برتری دارد.

قفل‌های خواندن-نوشتن (Read-Write Locks) ابزارهای همگام‌سازی هستند که امکان دسترسی همزمان چندین رشته (Thread) به منابع مشترک را فراهم می‌کنند، به‌طوری‌که چندین رشته می‌توانند به‌طور همزمان عملیات خواندن را انجام دهند، اما عملیات نوشتن به صورت انحصاری و تنها توسط یک رشته در هر لحظه صورت می‌گیرد. این قفل برای شرایطی که خواندن داده بیشتر از نوشتن آن رخ می‌دهد، طراحی شده است.

حالات قفل:

(قفل اشتراکی): برای عملیات خواندن استفاده می‌شود. در این حالت، چندین رشته می‌توانند به‌طور همزمان قفل را به‌دست آورند و داده‌ها را بخوانند، مشروط بر اینکه هیچ رشته‌ای در حال نوشتن (Exclusive Lock) نباشد.

(قفل انحصاری): برای عملیات نوشتن استفاده می‌شود. در این حالت، تنها یک رشته می‌تواند قفل را به‌دست آورد و عملیات نوشتن را انجام دهد. در هنگام نوشتن، هیچ رشته دیگری اجازه خواندن یا نوشتن را ندارد.

مکانیزم عملکرد:

کسب قفل برای خواندن: رشته‌ای که قصد خواندن دارد، درخواست قفل اشتراکی می‌کند. اگر هیچ رشته‌ای در حال نوشتن نباشد، قفل به آن داده می‌شود و شمارنده خوانندگان افزایش می‌یابد.

کسب قفل برای نوشتمن: رشته‌ای که قصد نوشتمن دارد، درخواست قفل انحصاری می‌کند. این درخواست تنها زمانی پذیرفته می‌شود که هیچ خواننده یا نویسنده دیگری در حال دسترسی به منبع نباشد.

آزادسازی قفل: پس از اتمام عملیات خواندن یا نوشتمن، رشته مربوطه قفل را آزاد می‌کند. در صورت آزادسازی قفل اشتراکی، شمارنده خوانندگان کاهش می‌یابد؛ و در صورت آزادسازی قفل انحصاری، نویسنده اجازه دسترسی به دیگران را می‌دهد.

مزایای قفل‌های خواندن-نوشتمن نسبت به :Reentrant Locks

افزایش همزمانی در عملیات خواندن: قفل‌های خواندن-نوشتمن امکان دسترسی همزمان چندین خواننده را فراهم می‌کنند، در حالی که قفل‌های بازگشتی چنین قابلیتی ندارند و تنها یک رشته می‌تواند در هر لحظه قفل را به دست آورد. پس در برنامه‌هایی که عملیات خواندن بیشتر از نوشتمن است، قفل‌های خواندن-نوشتمن با اجازه دسترسی همزمان به چندین خواننده، کارایی بهتری نسبت به قفل‌های بازگشتی ارائه می‌دهند.

جلوگیری از بنبست‌های پیچیده: قفل‌های خواندن-نوشتمن با مدیریت مناسب دسترسی‌ها، احتمال وقوع بنبست‌ها را کاهش می‌دهند، در حالی که استفاده نادرست از قفل‌های Reentrant می‌تواند به بنبست‌های پیچیده منجر شود.

اما این قفل محدودیت‌ها و چالش‌هایی هم دارد مثلا:

پیچیدگی در پیاده‌سازی: مدیریت همزمان خوانندگان و نویسنندگان نیازمند پیاده‌سازی دقیق و مکانیزم‌های مناسب برای جلوگیری از شرایط رقابتی است.

احتمال گرسنگ نویسندها: در صورتی که تعداد زیادی خواننده به طور مداوم در حال دسترسی باشند، ممکن است نویسندها برای مدت طولانی منتظر بمانند و به منابع دسترسی پیدا نکنند.

بخش عملی:

اضافه کردن سیستم کالی که تعداد سیستم کال های هر پردازنده را به علاوه کل سیستم کال ها در یک بار کاری سیستم نمایش می دهد.

ابتدا تعداد cpu ها در param.h و در makefile تغییر می دهیم.

```
3 #define NCPU          4 // maximum number of CPUs
```



```
226 // Local APIC ID
227 ifndef CPUS
228 CPUS := 4
229 endif
```

حال که تعداد پردازنده ها به 4 تا است شد، ابتدا به استراکت cpu، یک متغیر به اسم syscallnum اضافه می کنیم.

```
/ 
8 // Per-CPU state
9 struct cpu
10 {
11     uchar apicid;           // Local APIC ID
12     struct context *scheduler; // swtch() here to enter scheduler
13     struct taskstate ts;    // Used by x86 to find stack for interrupt
14     struct segdesc gdt[NSEGS]; // x86 global descriptor table
15     volatile uint started;  // Has the CPU started?
16     int ncli;               // Depth of pushcli nesting.
17     int intena;             // Were interrupts enabled before pushcli?
18     struct proc *proc;      // The process running on this cpu or null
19     enum schedqueue schedqueue; // Current scheduling queue
20     int queueticks;        // Number of ticks in the current queue
21     int syscallnum;         // for counting cpu syscalls
22 };
23 extern struct cpu ncpus;
```

این متغیر مقدار فراخوانی های سیستمی را که توسط یک پردازنده فراخوانی شده را در خود ذخیره می کند.

حال یک متغیر global برای ذخیره کل سیستم کال ها در trap.c تعريف می کنیم و در defs.h نیز به صورت زیر تعريف می کنیم همچنین نیازمند یک lock هستیم که مقدار total_syscall تنها زمانی بتواند تغییر کند که lock را در اختیار داشته باشیم. (دقت شود که چون از این متغیر ها در چند فایل استفاده می کنیم به صورت extern در defs.h تعريف شده)

```
// trap.c
void idtinit(void);
extern uint ticks;
void tvinit(void);
extern struct spinlock tickslock;
extern int total_syscall;
extern struct spinlock nsyscall_lock;
```



```
uint ticks,
int total_syscall;
struct spinlock nsyscall_lock;
```

حال کافی است زمانی که یک فراخوانی سیستمی اتفاق می افتد و به OS، ترپ می شود (با استفاده از تابع trap در trap.c)، قبل از شروع اجرای این سیستم کال، مقادیر متغیر هایی که بالا تعريف کردیم را آپدیت کنیم. این بخش به این صورت کار می کند که ابتدا هزینه سیستم کال یک در نظر گرفته می شود. سپس با توجه به نوع سیستم کال که توسط tf->eax مشخص می شود (آیدی سیستم کال در این رجیستر ذخیره می شود)، می بینیم که اگر سیستم کال open یا write بود هزینه سیستم کال را به مقدار مورد نظر افزایش می دهیم. همچنین ابتدا وقفه ها را غیر فعال کرده، مقدار سیستم کال های فراخوانی شده توسط cpu ای که در حال اجرا کردن است را آپدیت می کنیم و دوباره وقفه ها را فعال می کنیم. در آخر هم به وسیله همان nsyiscal_lock ابتدا lock را میگیریم سپس مقدار total_syscall را آپدیت می کنیم سپس دوباره lock را آزاد می کنیم.

```

//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
    if(tf->trapno == T_SYSCALL){
        if(myproc()->killed)
            exit();
        myproc()->tf = tf;
        int cost_syscall = 1;
        // if (tf->eax == SYS_write)
        //     cprintf("amooooooooo\n");
        if (tf->eax == SYS_write){
            cli();
            mycpu()->syscallnum += 2;
            sti();
            cost_syscall = 2;
        }
        else if (tf->eax == SYS_open){
            cli();
            mycpu()->syscallnum += 3;
            sti();
            cost_syscall = 3;
        }
        else{
            cli();
            mycpu()->syscallnum++;
            sti();
        }
        acquire(&nsyscall_lock);
        total_syscall += cost_syscall;
        release(&nsyscall_lock);
        syscall();
    }
}

```

حال نوبت به اضافه کردن فراخوانی سیستمی مورد نیاز می‌رسد. مثل روال سابق جاهایی که برای سیستم کال اضافه کردن تغییر می‌دادیم را تغییر می‌دهیم و خلاصه یک سیستم کال به اسم sys_nscalls اضافه می‌کنیم که تابع get_syscallsnum واقع در proc.c را فراخوانی می‌کند.

کد این تابع به صورت زیر است که مقدار سیستم کال های فراخوانی شده توسط تمام cpu ها ، مجموعشان و مقدار total_syscallnum را پرینت می‌کند.

```

87
88     void get_syscalls_num(void){
89         int sum = 0;
90         for (int i = 0; i < NCPU; i++)
91             sum += cpus[i].syscallnum;
92         cprintf("%d, %d, %d, %d, sum = %d, ", cpus[0].syscallnum, cpus[1].syscallnum, cpus[2].syscallnum,
93                 cpus[3].syscallnum, sum);
94         acquire(&nsyscall_lock);
95         cprintf("%d\n", total_syscall);
96         release(&nsyscall_lock);
97     }

```

برنامه سطح کاربری که این سیستم کال را تست می‌کند هم به صورت زیر است که NPROCESS تا پردازه که در اینجا برابر ۵ است، ایجاد می‌کند و با آن‌ها در یک فایل می‌نویسد. تعداد کل سیستم کال‌های این پردازه برابر ۳۴ تا است که اگر تا قبل از exit صدا زده شده را حساب کنیم ۳۳ تا می‌شود و خروجی کد ما هم ۳۳ تا است.

```

inal xv6 in lab4 > C nsystest.c > main(int argc, char *argv[])
1  #include "types.h"
2  #include "user.h"
3  #include "fcntl.h"
4
5  #define NPROCESS 5
6
7  int main(int argc, char *argv[])
8  {
9      unlink("ns.txt");
10     int fd = open("ns.txt", O_CREATE | O_WRONLY);
11     for (int i = 0; i < NPROCESS; i++)
12     {
13         if (!fork())
14         {
15             write(fd, "nothing", 8);
16             exit();
17         }
18         else
19             continue;
20     }
21     for (int i = 0; i < NPROCESS; i++)
22         wait();
23     write(fd, "\n", 1);
24     close(fd);
25     nsyscalls();
26     exit();
27 }

```

هنگامی که خط write را کامنٹ کنیم، خروجی به صورت زیر می‌شود.

```
$ nsystest  
14, 56, 22, 1, sum = 93,93
```

حال اگر write را انجام دهیم باید به ازای هر پردازه ۲ فراخوانی سیستمی اضافه شود و چون ۵ پردازه فرزند ایجاد کرده‌ایم پس در کل باید ۱۰ فراخوانی سیستمی اضافه شود. همانطور که انتظار داشتیم خروجی به صورت زیر تغییر می‌کند.

```
$ nsystest  
10, 28, 57, 8, sum = 103,103
```

میوتکس پردازه مالک (Reentrant Mutex)

هدف ما پیاده‌سازی یک قفل جدید با نام reentrantlock می‌باشد. در فایل reentrantlock.h می‌باشد. در فایل reentrantlock.h یک struct جدید برای این قفل تعریف می‌کنیم. وظیفه اصلی قفل کردن بر عهده spinlock lock می‌باشد. متغیر owner نشان دهنده پردازه‌ای می‌باشد که قفل را گرفته است. همچنین برای مدیریت چندین بار متغیر recursion کردن تو در تو توسط یک پردازه، متغیر recursion را قرار می‌دهیم تا تعداد release و acquire لایه‌ها را بشمارد.

```
// Reentrant mutual exclusion lock  
struct reentrantlock {  
    struct spinlock lock;          // Underlying spinlock for atomicity  
    struct proc *owner;            // Current owner of the lock  
    int recursion;                // Recursion depth for reentrancy  
};
```

پیاده‌سازی توابع را در فایل reentrantlock.c انجام می‌دهیم. در ابتدای استفاده از قفل باید آن را به کمک تابع initreentrantlock مقدار دهی اولیه کنیم.

```
void  
initreentrantlock(struct reentrantlock *lk)  
{  
    initlock(&lk->lock, "reentrant lock");  
    lk->owner = 0;  
    lk->recursion = 0;  
}
```

از آنجایی که وظیفه اصلی قفل کردن بر عهده spinlock lock می‌باشد، پس آن را با نام reentrant recursion مقداردهی اولیه می‌کنیم. همچنین مقدار owner را به یک مقدار نامعتبر یعنی صفر و lock را به صفر که یعنی عمق بازگشتی صفر، مقدار دهی اولیه می‌کنیم. پس از صدا زدن این تابع بر روی قفل، قفل آماده استفاده می‌شود.

یک پردازه قبل از آن که وارد critical section شود باید با صدا زدن تابع acquire_reentrant را روی قفل، آن را بگیرد تا از race condition جلوگیری شود.

```
void
acquire_reentrant(struct reentrantlock *lk)
{
    pushcli(); // Disable interrupts to avoid deadlock.

    // Check for the new process to hold the lock
    if (lk->owner != myproc()) {
        // Acquire the underlying spinlock
        acquire(&lk->lock);
        if (lk->recursion != 0) // For debugging
            panic("reentrant lock: acquire");
        lk->owner = myproc();
    }

    lk->recursion++;
    popcli();
}
```

در این تابع، در ابتدا برای جلوگیری از deadlock و سایر مشکلات به کمک تابع pushcli وقفه‌ها غیرفعال می‌شوند. سپس بررسی می‌کند که پردازه فعلی آیا قفل را گرفته است یا خیر. اگر قفل را گرفته باشد، بدین معنی است که این یک acquire بازگشتی می‌باشد و نباید پردازه متوقف شود و فقط به عمق بازگشتی یعنی recursion اضافه می‌کنیم. اما اگر پردازه صاحب قفل نباشد، پس باید قفل را بگیرد که در واقع همان گرفتن spinlock lock به کمک تابع acquire می‌باشد. پس از آن یک قسمت چک کردن مقدار recursion برابر با صفر انجام می‌دهیم که صرفا جنبه دیبگ کردن دارد. زیرا هنگامی که یک پردازه جدید قفل را می‌گیرد، قبل از آنکه مقدار recursion را زیاد کنیم، مقدار آن باید برابر با صفر باشد. وقتی که پردازه جدید قفل را گرفت، آن پردازه را به عنوان صاحب آت قفل ذخیره می‌کنیم تا در

صورت acquire بازگشتی، دیگر در آن گیر نکند. در نهایت نیز چه پردازه جدید باشد چه پردازه صاحب به صورت بازگشتی این تابع را صدا زده باشد، مقدار recursion را یکی افزایش می‌دهیم و چون در ابتدا وقهه‌ها را متوقف کرده بودیم، دوباره با تابع popcli آن‌ها را فعال می‌کنیم.

هنگامی که کار پردازه در critical section به پایان رسید، باید با صدا زدن تابع releasereentrant روی قفل، آن را رها کند.

```
void
releasereentrant(struct reentrantlock *lk)
{
    pushcli();

    if (lk->owner != myproc())
        panic("reentrant lock: release");

    lk->recursion--;

    // Check the depth to release the spinlock
    if (lk->recursion == 0) {
        lk->owner = 0;
        release(&lk->lock);
    }

    popcli();
}
```

در این تابع نیز وقهه‌ها را در حین عملیات غیر فعال می‌کنیم. در ابتدا چک می‌کنیم که آیا پردازه‌ای که این تابع را فراخوانی کرده است، صاحب آن قفل می‌باشد یا خیر. در صورتی که پردازه‌ای غیر از صاحب قفل این تابع را صدا بزند با تابع panic ارور را نمایش می‌دهیم و به اجرای عملیات سیستم پایان می‌دهیم. اگر هم صاحب پردازه این تابع را صدا زده باشد، باید مقدار recursion را یکی کم کنیم. در نهایت باید چک کنیم که اگر پردازه از تمام فراخوانی‌های بازگشته بازگشته بود (یعنی به همان تعداد که به صورت بازگشتی تابع acquire را فراخوانی کرده بود، به همان تعداد هم تابع release را فراخوانی کرده است). قفل باید رها شود. پس اگر عمق برابر صفر بود، صاحب قفل را به یک مقدار نامعتبر تغییر داده و قفل اصلی یعنی spinlock lock را رها می‌کنیم.

حال که توابع آماده هستند، باید declaration آنها را در فایل def.h قرار دهیم تا در تمام قسمت‌های کرنل شناخته شده و قابل استفاده باشند.

```
struct reentrantlock;

// reentrantlock.c
void acquirereentrant(struct reentrantlock* );
void releasereentrant(struct reentrantlock* );
void initreentrantlock(struct reentrantlock* );
```

همچنین برای کامپایل شدن و استفاده از توابع فایل reentrantlock.c باید makefile را تغییر دهیم.
برای این کار OBJS را به متغیر reentrantlock.o اضافه می‌کنیم.

```
OBJS = \
...
reentrantlock.o\
```

از آنجایی که این قفل فقط در سطح کرنل قابل استفاده است، برای تست آن نیاز به تعریف یک system call جدید داریم. پس برای آن sys_reentrantlocktest را تعریف می‌کنیم که در آن ابتدا یک متغیر با تابع struct reentrantlock درست می‌کنیم که همان قفل بازگشتی می‌باشد و سپس آن را با تابع initreentrantlock مقدار دهی اولیه می‌کنیم. حال که قفل آماده استفاده است، تابع بازگشتی را صدا می‌زنیم.

```
void
sys_reentrantlocktest(void)
{
    struct reentrantlock lk;
    initreentrantlock(&lk);
    cprintf("Initiating lock: recursion = %d\n", lk.recursion);
    lock_and_call(3, &lk);
}
```

ما از یک تابع بازگشتی ساده با نام lock_and_call استفاده می‌کنیم که صرفاً به تعدادی که به عنوان ورودی به آن می‌دهیم، به صورت بازگشتی صدا زده می‌شود. البته قبل از صدا زدن بازگشتی قفل را گرفته و پس از آن قفل را رها می‌کنیم.

```
void
lock_and_call(int i, struct reentrantlock *lk)
{
    acquirereentrant(lk);
```

```

cprintf("Acquiring lock: recursion = %d\n", lk->recursion);

if (i > 0)
    lock_and_call(i - 1, lk);

releasereentrant(lk);
cprintf("Releasing lock: recursion = %d\n", lk->recursion);
}

```

همچنین برای استفاده از این فراخوانی سیستمی یک برنامه سطح کاربر می‌نویسیم که در آن صرفاً فراخوانی سیستمی reentrantlocktest صدا زده می‌شود.

```

int
main(int argc, char *argv[])
{
    reentrantlocktest();
    exit();
}

```

خروجی حاصل از اجرای برنامه سطح کاربر بالا به صورت زیر است:

```

$ reentranttest
Initiating lock: recursion = 0
Acquiring lock: recursion = 1
Acquiring lock: recursion = 2
Acquiring lock: recursion = 3
Acquiring lock: recursion = 4
Releasing lock: recursion = 3
Releasing lock: recursion = 2
Releasing lock: recursion = 1
Releasing lock: recursion = 0

```

همان طور که انتظار داشتیم هنگام مقداردهی اولیه، عمق برابر صفر بوده و با هر بار acquire کردن یکی به عمق اضافه می‌شود و با در بار release کردن یک از عمق کم می‌شود. در نهایت هم عمق برابر با صفر می‌شود یعنی قفل به صورت کامل رها شده است و پردازه‌های دیگر می‌توانند آن را بگیرند.