

آزمایشگاه سیستم عامل

تمرین کامپیوتری ۴

اعضای گروه:

علی حمزه پور - ۸۱۰۱۰۰۱۲۹

نرگس سادات سیدحائری - ۸۱۰۱۰۰۱۶۵

مینا شیرازی - ۸۱۰۱۰۰۲۵۰

همگام‌سازی در xv6

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

برای جلوگیری از مشکل deadlock باید قبل از استفاده از این قفل وقفه‌ها را غیر فعال کرد. همچنین غیرفعال کردن وقفه‌ها این اطمینان را به ما می‌دهند که دستورات مربوط به قفل به صورت اتمیک اجرا می‌شوند و چیزی بینشان قرار نمی‌گیرد و ترتیبشان عوض نمی‌شود.

اگر در وقفه نیاز به قفلی داشته باشیم که در اختیار پردازش‌های که متوقف شده باشد، آن‌گاه ددلاک رخ می‌دهد زیرا آن وقفه منتظر می‌ماند تا قفل آزاد شود اما پردازش دیگری نیز منتظر وقفه است تا وقفه تمام شود. با غیرفعال کردن وقفه‌ها این مشکل دیگر پیش نمی‌آید.

سوال دو: توابع pushcli و popcli به چه منظور استفاده شده و چه تفاوتی با cli و sti دارند؟

توابع pushcli و popcli معمولاً در زمینه مدیریت اینتراپت‌ها و اطمینان از انحصار متقابل در یک محیط چندنخی یا چندپردازنده‌ای استفاده می‌شوند. این توابع معمولاً با غیرفعال‌سازی و فعال‌سازی اینتراپت‌ها ارتباط دارند:

۱. cli و sti :

- دستور clear interrupt flag برای غیرفعال‌سازی اینتراپت‌ها استفاده می‌شود.

- دستور set interrupt flag برای فعال‌سازی اینتراپت‌ها استفاده می‌شود.

این دستورها توسط پردازنده ارائه شده‌اند تا مکانیزم اینتراپت را کنترل کنند. غیرفعال‌سازی اینتراپت‌ها ("cli") از پاسخگویی پردازنده به اینتراپت‌های خارجی جلوگیری می‌کند، تا بخش‌های حیاتی از کد بتوانند به صورت اتمیک و بدون وقفه اجرا شوند. فعال‌سازی اینتراپت‌ها ("sti") امکان پاسخگویی پردازنده به اینتراپت‌های خارجی را دوباره فراهم می‌کند.

۲. Pushcli و Popcli :

Pushcli یک تابع سفارشی است که معمولاً در هسته یا سیستم‌عامل پیاده‌سازی می‌شود. برای غیرفعال‌سازی اینتراپت‌ها و ذخیره وضعیت جاری اینتراپت (اگر اینتراپت‌ها قبلاً فعال یا غیرفعال بوده‌اند) در یک پشته استفاده می‌شود.

"Popcli" معادل "pushcli" است. برای بازیابی وضعیت اینتراپت از طریق پاپ کردن وضعیت ذخیره‌شده از روی پشته و در صورت لزوم، فعال‌سازی مجدد اینتراپت‌ها استفاده می‌شود.

سوال سه: چرا قفل مذکور در سیستم های تک هسته ای مناسب نیست؟ روی کد توضیح دهید.

در سیستم های تک هسته ای، استفاده از قفل چرخشی یا "spin lock" به عنوان یک مکانیزم همگام سازی مناسب نیست. دلایل اصلی عبارتند از:

۱. **هدررفت زمان در انتظار فعال (Busy-Waiting):** در یک سیستم تک هسته ای، وقتی یک نخ در یک حلقه چرخشی منتظر آزاد شدن یک قفل باشد، به طور مداوم CPU را اشغال می کند. این موضوع موجب هدر رفتن زمان پردازشی می شود و منابع CPU را به طور ناکارآمد مصرف می کند.

۲. **ضایعات منابع:** قفل های چرخشی ممکن است منابع را در محیط تک هسته ای هدر دهند. زمانی که یک نخ به طور مداوم در حلقه چرخشی است، حتی زمانی که پیشرفتی ندارد، مصرف برق و منابع CPU را افزایش می دهد.

۳. **افزایش تاخیر:** استفاده از قفل های چرخشی ممکن است تاخیر را افزایش دهد، به خصوص در سیستم های تک هسته ای. زمانی که نخ ها باید منتظر آزاد شدن یک قفل باشند، زمان انتظار می تواند به کارایی و واکنش پذیری سیستم آسیب بزند.

کد acquire در xv6:

```
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);
}
```

همانطور که می‌بینید پردازش بدون اینکه به sleep برود در یک حلقه منتظر آزاد شدن قفل می‌ماند و این باعث اشغال شدن هسته می‌شود.

سوال چهار : در مجموعه دستورات RISC-V، دستوری با نام amoswap وجود دارد. دلیل تعریف و نحوه کار آن را توضیح دهید.

دستور amoswap یک دستور اتمیک در مجموعه دستورات RISC-V است و برای عملیات‌های حافظه اتمیک استفاده می‌شود که در برنامه‌نویسی چندنخی برای همگام‌سازی دسترسی به منابع مشترک بسیار حیاتی است.

```
amoswap.w rd, rs2, (rs1)
```

این دستور به صورت اتمیک یک مقدار داده ۳۲ بیتی با علامت را از آدرس موجود در "rs1" می‌خواند، مقدار را در رجیستر "rd" قرار می‌دهد، مقدار خوانده شده را با مقدار اولیه ۳۲ بیتی با علامت موجود در "rs2" جابه‌جا (swap) می‌کند، سپس نتیجه را دوباره در آدرس موجود در "rs1" ذخیره می‌کند.

مراحل این دستور به شرح زیر است:

۱. **بارگذاری (load):** ابتدا دستور یک مقدار داده ۳۲ بیتی با علامت را از آدرس مشخص شده توسط "rs1" می‌خواند.

۲. **جابه‌جایی (swap):** سپس این مقدار خوانده شده را با مقدار موجود در "rs2" جابه‌جا می‌کند.

۳. **ذخیره (store):** در نهایت، مقدار جابه‌جا شده (که ابتدا در "rs2" بوده است) را دوباره در آدرس حافظه مشخص شده توسط "rs1" ذخیره می‌کند.

این عملیات به صورت اتمیک انجام می‌شود، به این معنا که در یک مرحله اجرا می‌شود. این باعث می‌شود که اگر چند نخ همزمان دستورات amoswap را در هسته‌های مختلف اجرا کنند، هر عملیات amoswap یا کامل اجرا می‌شود یا هیچ‌کدام، اجرا نمی‌شوند. اما هرگز در یک وضعیت نیمه کامل قرار نمی‌گیرد.

دستور amoswap معمولاً در پیاده‌سازی ابزارهای همگام‌سازی مانند قفل‌های چرخشی (spinlock) استفاده می‌شود. به عنوان مثال، می‌تواند برای بررسی اتمیک و به‌دست آوردن یک قفل در صورت آزاد بودن آن استفاده شود.

سوال پنج: مختصری راجع به تعامل میان پردازنده‌ها توسط دو تابع `acquiresleep` و `releasesleep` توضیح دهید.

تابع `acquiresleep` به شکل زیر تعریف شده است:

```
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);
}
```

این تابع یک `sleep lock` دریافت می‌کند. اگر قفل از قبل گرفته شده باشد، پردازنده فراخوانی‌کننده به حالت خواب می‌رود و واحد پردازش را به سایر پردازنده‌ها می‌سپارد. پردازنده تا زمانی که قفل در دسترس قرار گیرد، در حالت خواب باقی می‌ماند. این تابع با تابع `acquire spinlock` (که تا زمانی که قفل در دسترس نشود، در حالت `busy waiting` منتظر می‌ماند) متفاوت است. در واقع در این تعامل بین پردازنده‌ها، پردازنده فراخوانی‌کننده می‌گوید "من را هنگامی که قفل در دسترس بود بیدار کنید."

تابع `releasesleep` به شکل زیر است:

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

```
lk->pid = 0;
wakeup(lk);
release(&lk->lk);
}
```

این تابع یک sleep lock را که قبلاً توسط پردازش فراخواننده گرفته شده بوده است، آزاد می‌کند. اگر همچنان پردازش‌هایی وجود دارد که در حالت خواب هستند (چون قبلاً سعی کرده‌اند با استفاده از acquire_sleep قفل را بگیرند اما قفل از پیش گرفته شده بوده است)، یکی از آن‌ها بیدار می‌شود. این صورت دیگری از تعامل میان پردازش‌ها می‌باشد، پردازش در اصل می‌گوید: "حالا که قفل در دسترس است، یکی از فرآیندهای خوابیده می‌تواند بیدار شود و آن را گرفته و استفاده کند."

سوال شش: حالات مختلف پردازش‌ها را در xv6 توضیح دهید. تابع sched چه وظیفه‌ای دارد؟

پردازش‌ها در سیستم عامل xv6 در یکی از حالات زیر هستند:

1. **UNUSED**: اگر در یک خانه از جدول پردازش‌ها (ptable)، واقعا یک پردازش وجود نداشته باشد. (برای مثال در آن خانه پردازش ساخته نشده یا پردازشی مربوط به آن خانه کارش تمام شده و به اصطلاح terminate شده)، حالت متغیر پردازشی مربوط به آن خانه UNUSED می‌شود. این به این معناست که اگر بخواهیم پردازشی جدید بسازیم می‌توانیم از این خانه در جدول پردازش‌ها استفاده کنیم.
2. **EMBRYO**: زمانی که یک پردازش در مرحله‌ی ساخته شدن هست اما هنوز کاملاً آماده‌ی اجرا شدن نیست، در این حالت قرار می‌گیرد.
3. **SLEEPING**: زمانی که یک پردازش منتظر اتفاقی برای رخ دادن است در این حالت قرار می‌گیرد. برای مثال اگر منتظر پاسخ I/O باشد یا منتظر سیگنال یک تایمر باشد در حالت SLEEPING قرار می‌گیرد و در این حالت پردازش زمانبندی نمی‌شود و تا زمانی که آن رخدادی که منتظرش است رخ ندهد، اجرا نمی‌شود.
4. **RUNNABLE**: زمانی که پردازش آماده‌ی اجرا شدن است و منتظر پردازنده است که آن را برای اجرا زمانبندی کند، در این حالت قرار می‌گیرد.
5. **RUNNING**: زمانی که پردازش در یک پردازنده در حال اجرا است، در این حالت قرار می‌گیرد. در هر لحظه در هر پردازنده در سیستم عامل xv6 یک پردازش این حالت را داراست.
6. **ZOMBIE**: زمانی که کار یک پردازش تمام می‌شود و به اصطلاح terminate می‌شود، به حالت ZOMBIE می‌رود و در این حالت منتظر پردازشی پدرش می‌ماند تا تمام شدن آن را با سیستم کال wait متوجه شود و بعد از آن کاملاً از سیستم عامل پاک شود و به حالت UNUSED برود.

هر پردازش زمانی که قرار است از حالت RUNNING خارج شود (که به ۳ دلیل ممکن است رخ دهد: تمام شدن پردازش، تمام شدن تایمر یا یک اینتراپت دیگر)، تابع sched را صدا می‌زند. تابع sched context-switch را انجام می‌دهد تا به همان پردازشی که عملیات scheduling را انجام می‌داد برگردیم. به همین دلیل بعد از context-switch در تابع sched، ادامه‌ی تابع scheduler، و از آنجایی که به یک پردازش سوئیچ کرده بودیم، اجرا می‌شود.

سوال هفت: تغییری در توابع دسته دوم (sleeplock) داده تا تنها پردازشی صاحب قفل، قادر به آزادسازی آن باشد. قفل معادل در هسته لینوکس را به طور مختصر معرفی نمایید.

اگر به تعریف sleeplock مراجعه کنیم، می‌بینیم که شناسه‌ی پردازشی صاحب قفل را نگه می‌دارد:

```
// Long-term locks for processes
struct sleeplock {
    uint locked;    // Is the lock held?
    struct spinlock lk; // spinlock protecting this sleep lock

    // For debugging:
    char *name;     // Name of lock.
    int pid;        // Process holding lock
};
```

پس کافیست در تابع releasesleep ابتدا بررسی کنیم که آیا پردازشی فعلی همان پردازشی صاحب قفل هست یا خیر:

```
void
releasesleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    if (lk->pid != myproc()->pid) {
        release(&lk->lk);
        panic("Process is not the owner of the lock!");
    }
    lk->locked = 0;
    lk->pid = 0;
```

```
wakeup(lk);
release(&lk->lk);
}
```

قفل معادل با sleeplock در سیستم عامل لینوکس، mutex است. مانند sleeplock، در صورتی که پردازهای بخواهد قفلی که در اختیار پردازشی دیگری است را بگیرد، پردازش وارد حالت sleep می‌شود و پردازنده پردازشی دیگری را برای اجرا زمانبندی می‌کند تا مشکل busy waiting رخ ندهد.

ساختار mutex و نحوه‌ی استفاده از این قفل به شکل زیر است:

```
#include <linux/mutex.h>

struct mutex {
    raw_spinlock_t wait_lock;
    struct list_head wait_list;
    struct task_struct *owner;
#ifdef CONFIG_DEBUG_MUTEXES
    unsigned long owner_cpu;
    void *owner_stack;
#endif
};
```

```
#include <linux/mutex.h>

struct mutex my_mutex;

/* ... initialization ... */

/* Lock the mutex */
mutex_lock(&my_mutex);

/* Critical section */
```



```
/* Unlock the mutex */  
mutex_unlock(&my_mutex);
```

سوال هشت: روشی دیگر برای نوشتن برنامه‌ها استفاده از الگوریتم‌های lock-free است. مختصری راجع به آن‌ها توضیح داده و مزایا و معایب آن‌ها نسبت به برنامه‌نویسی با lock را بگویید.

در برنامه‌نویسی lock-free همانطور که از نام آن پیداست، از قفل‌ها در همگام‌سازی استفاده نمی‌کنیم و به جای آن از دستورات در سطح سخت افزار (مانند دستور compare_and_swap و یا استفاده از memory_barriers) و یا متغیرهای atomic برای همگام‌سازی و جلوگیری از race condition استفاده می‌کنیم.

از مزایای این روش می‌توان به نداشتن هزینه عملیات‌های مربوط به قفل مثل گرفتن قفل و یا آزادسازی آن اشاره کرد. همچنین به دلیل عدم نیاز به قفل‌ها، در توسعه نرم‌افزار امکان مقیاس پذیری بهتری (scalability) وجود دارد. در این روش مشکل deadlock نیز رخ نمی‌دهد.

از طرفی توسعه نرم‌افزار و تست آن با روش lock-free سخت‌تر و زمان‌برتر می‌شود. همچنین چون در این روش از دستورهای سطح پایین‌تر و متغیرهای atomic استفاده می‌کنیم، برنامه‌نویس نیاز به درک عمیق‌تر و دانش بیشتری از موضوع دارد تا بتواند برنامه‌ای بدون مشکل، توسعه دهد.

در کل در حالت‌هایی که بیشتر ممکن است پردازش قبل از critical section صبر کند و به اصطلاح contention بیشتری رخ دهد، استفاده از قفل‌ها توصیه می‌شود و در غیر این صورت استفاده از دستورات سخت‌افزاری و عدم استفاده از قفل، بهینه‌تر است زیرا دیگر هزینه قفل را متحمل نمی‌شویم.

پیاده‌سازی متغیرهای مختص هر پردازنده

الف) روشی جهت حل این مشکل (نامعتبرسازی متغیر در حافظه پنهان هسته‌های دیگر) در سطح سخت‌افزار وجود دارد. مختصراً آن را توضیح دهید.

یکی از راه‌های حل این مشکل در سطح سخت‌افزار روشی به نام snooping است. در این روش هسته‌ها دائماً bus سیستم را برای عملیات‌های خواندن و نوشتن روی خانه‌هایی از حافظه که در حافظه پنهان (کش) هسته هم موجود است بررسی می‌کند. در صورتی که هسته متوجه شد که مقدار خانه‌ای از حافظه که آن را در کش دارد عوض شده است، مقدار آن را در کش نامعتبر می‌کند و یا آن را به روزرسانی می‌کند.

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

در صورتی که از قفل بلیت استفاده کنیم، مقدار متغیر ticket_counter برای هر پردازنده باید با هسته‌های دیگر یکسان باشد در غیر این صورت ممکن است مشکلاتی نظیر ددلاک، بهم خوردن اولویت و یا افزایش contention رخ دهد. از طرفی فرآیند نامعتبر کردن مقدار موجود در کش نیست همانطور که گفته شد سربار بالایی دارد و باعث می‌شود کارایی سیستم پایین بیاید، زیرا متغیر ticket_counter دائما در حال به‌روز شدن است و عملا کش برای آن بی‌فایده می‌شود و هسته‌ها باید دائما آن را از حافظه‌ی اصلی به‌روز کنند.

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

با استفاده از ماکروی DECLARE_PER_CPU می‌توان متغیری مختص هسته تعریف کرد:

```
DECLARE_PER_CPU(int, a);
```

اولین ورودی این ماکرو تایپ متغیر تعریفی و دومین ورودی اسم آن متغیر است. در مثال بالا متغیر a از نوع int مختص به هسته تعریف شده است. این ماکرو معمولا در فایل header استفاده می‌شود.

ماکروی DEFINE_PER_CPU نیز برای اختصاص حافظه برای متغیر مختص به هسته استفاده می‌شود. این ماکرو معمولا در فایل سورس برنامه (فایل c) استفاده می‌شود.

```
DEFINE_PER_CPU(int, a);
```

با استفاده از دستور get_cpu_var می‌توان به متغیر مختص به هسته دسترسی داشت و با استفاده از دستور put_cpu_var می‌توان آن‌ها را آزاد کرد. برای مثال در کد زیر یک متغیر مختص به هسته گرفته شده، مقدار آن تغییر داده شده و بعد آزاد شده است.

```
#include <linux/percpu.h>

DEFINE_PER_CPU(int, my_per_cpu_variable);

void func(void){
    int *per_cpu_ptr = &get_cpu_var(my_per_cpu_variable);
```

```
// Access and modify per-CPU variable
(*per_cpu_ptr)++;

put_cpu_var(my_per_cpu_variable);
}
```

با استفاده از این روش، یک فراخوانی سیستمی تعریف نمایید که تعداد فراخوانی های سیستمی اجرا شده در یک بار کاری را روی یک سیستم چهار هسته ای برمیگرداند.

در این بخش ما برای اینکه متغیر مختص به هسته تعریف کنیم متغیر `executed_syscalls` را به استراکت `cpu` اضافه کردیم. فایل `syscalls.c` دارای یک فانکشن به نام `syscall` می باشد، هر موقع این فانکشن صدا زده شود این متغیر یکی زیاد می شود. از آنجایی که گفته شده جهت اطمینان از صحت عملکرد باید یک نسخه مشترک میان همه هسته ها تعریف شده و با مقدار برگشتی مقایسه گردد. برای این قسمت در فایل `mp.c` یک متغیر به نام `executed_syscalls` تعریف می کنیم. می دانیم که فایل `mp.c` دیتاهای مربوط به مولتی پراسسر را در خود نگه می دارد. در مرحله ی بعد در `proc.h` آن را اکسترن کردیم تا تمام فایل های دیگر بتوانند به آن دسترسی پیدا کنند. در `syscalls.c` بار `executed_syscalls` کلی را یکی زیاد می کنیم. از `cli()` و `sti()` برای `enable/disable` کردن اینترپت ها استفاده می کنیم و برای اینکه `race condition` اتفاق نیافتد تابع `sync_synchronization` را صدا میزنیم. در تابع `exec` نیز هر بار مقادیر این متغیرها را صفر می کنیم تا تعداد سیستم کال ها در همان یک بار اجرای برنامه ی سطح کاربر حساب شود.

```
void
syscall(void)
{
    int num;
    struct proc *curproc = myproc();
    cli();
    mycpu()->executed_syscalls++;
    sti();
    executed_syscalls++;
    __sync_synchronize();
}
```

```

num = curproc->tf->eax;
if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
} else {
    cprintf("%d %s: unknown sys call %d\n",
        curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
}
}

```

در سیستم‌کال هم مقادیر `executed_syscalls` را در سی‌پی‌یوهای مختلف جمع می‌زنیم و آن را خروجی می‌دهیم. همچنین مقدار `executed_syscalls` را هم چاپ می‌کنیم تا مطمئن شویم این دو مقدار با هم برابر است:

```

int
sys_get_num_syscalls(void){
    int sum = 0;
    for (int i = 0; i < ncpu ;i++){
        sum += cpus[i].executed_syscalls;
        //cpus[i].executed_syscalls = 0;
    }
    cprintf("shared: %d per: %d\n", executed_syscalls, sum);
    // executed_syscalls = 0;
    //__sync_synchronize();
    return sum;
}

```

در برنامه‌ی سطح کاربر تعدادی پردازش می‌سازیم که هر کدام در یک فایل می‌نویسند و در پایان سیستم‌کال مدنظر را صدا می‌زنیم:

```

#include "types.h"
#include "user.h"

```

```

#include "stat.h"
#include "fcntl.h"

#define NUM_FORKS ୨

int main(int argc, char* argv[]){
    int fd=open("shared_file.txt",O_CREATE|O_WRONLY);
    for (int i = ୦; i < NUM_FORKS; i++){
        int pid = fork();
        if (pid == ୦){
            acquire_user_lock();

            char* write_data = "salam";
            int max_length = ୦;

            write(fd,write_data,max_length);
            write(fd,"\n",1);

            release_user_lock();
            exit();

        }
        //while(1);
    }

    while (wait() != -1);
    close(fd);

    get_num_syscalls();
    exit();
}

```

خروجی برنامه‌ی سطح کاربر به شکل زیر خواهد بود:

```
pid_t lock = 0;
$ get_num_syscalls
shared: 25 per: 25
$
```

پیاده‌سازی قابلیت همگام‌سازی با قابلیت اولویت دادن

ابتدا یک قفل جدید به نام prioritylock مانند پیاده‌سازی sleeplock درست می‌کنیم:

```
#define MAX_PRIORITY_QUEUE_SIZE NPROC

// Long-term locks for processes
struct prioritylock {
    uint locked;    // Is the lock held?
    struct spinlock lk; // spinlock protecting this sleep lock
    int queue[MAX_PRIORITY_QUEUE_SIZE];
    int queue_front;
    int queue_size;

    // For debugging:
    char *name;    // Name of lock.
    int pid;       // Process holding lock
};
```

این ساختار دقیقاً همان متغیرهای sleeplock را دارد و علاوه بر آن صفی که در آن پردازنده‌ها قرار می‌گیرند را هم دارد. این صف حالت حلقوی دارد و برای آن اندیس شروع و اندازه‌ی آن را نگه می‌داریم.

تابع init را برای این ساختار تعریف می‌کنیم:

```
void
initprioritylock(struct prioritylock *lk, char *name){
    initlock(&lk->lk, "priority lock");
    lk->name = name;
    lk->locked = 0;
    lk->pid = 0;
    lk->queue_front = 0;
    lk->queue_size = -1;
}
```

سپس توابع اضافه کردن و برداشتن از صف اولویت را پیاده‌سازی می‌کنیم تا بتوانیم acquire و release را با استفاده از آن‌ها پیاده‌سازی کنیم. صف ما همیشه برحسب مقدار pid مرتب است و هنگام اضافه کردن یک پردازش، pid آن را در جای مناسب insert می‌کنیم. هنگام برداشتن هم صرفاً عنصر اول (که بزرگترین pid را دارد) برمی‌داریم.

```
void
add_process_to_priority_lock(struct prioritylock *lk, int pid){
    lk->queue_size++;
    int last_idx = (lk->queue_size + lk->queue_front) % MAX_PRIORITY_QUEUE_SIZE;
    lk->queue[last_idx] = pid;
    int cur_idx = last_idx;
    for (int i = 0; i < lk->queue_size; i++){
        int prev_idx = (cur_idx - 1) % MAX_PRIORITY_QUEUE_SIZE;
        if (lk->queue[prev_idx] > lk->queue[cur_idx]){
            break;
        }
        int temp = lk->queue[cur_idx];
        lk->queue[cur_idx] = lk->queue[prev_idx];
        lk->queue[prev_idx] = temp;
        cur_idx = prev_idx;
    }
}
```

```

}

void
pop_priority_queue(struct prioritylock *lk){
    lk->queue_front = (lk->queue_front + 1) % MAX_PRIORITY_QUEUE_SIZE;
    lk->queue_size--;
}

```

حال توابع acquire و release را پیاده‌سازی می‌کنیم. این توابع مانند توابع متناظر در sleeplock هستند اما شرطی در acquire اضافه می‌کنیم که تا وقتی که pid پردازش برابر با pid عنصر اول صف نشده، قفل را به آن اختصاص نمی‌دهیم:

```

int
acquirepriority(struct prioritylock *lk){
    acquire(&lk->lk);
    if (lk->pid == myproc()->pid){
        release(&lk->lk);
        return -1;
    }
    add_process_to_priority_lock(lk, myproc()->pid);
    while (lk->locked || lk->queue[lk->queue_front] != myproc()->pid) {
        sleep(lk, &lk->lk);
    }
    lk->locked = 1;
    lk->pid = myproc()->pid;
    pop_priority_queue(lk);
    release(&lk->lk);
    return 0;
}

int
releasepriority(struct prioritylock *lk){

```



```

acquire(&lk->lk);
if (lk->pid != myproc()->pid) {
    release(&lk->lk);
    return -۱;
}
lk->locked = ۰;
lk->pid = ۰;
wakeup(lk);
release(&lk->lk);
return ۰;
}

```

تابع `print_queue` را برای نمایش صف به صورت زیر پیاده‌سازی می‌کنیم:

```

void
print_priority_queue(struct prioritylock *lk){
    acquire(&lk->lk);
    cprintf("Queue: [");
    for (int i = ۰; i < lk->queue_size + ۱; i++){
        int idx = (i + lk->queue_front) % MAX_PRIORITY_QUEUE_SIZE;
        cprintf("%d, ", lk->queue[idx]);
    }
    cprintf("]\n");
    release(&lk->lk);
}

```

حال در `kernel` متغیری به نام `userlock` تعریف می‌کنیم تا کاربر بتواند آن را با فراخوانی سیستمی تغییر دهد:

```

struct {
    struct prioritylock pl;
} userlock;

```

سپس سه فراخوانی سیستمی برای `acquire` و `release` و `print_queue` پیاده‌سازی می‌کنیم:

```

int
acquire_user_lock(void){
    return acquirepriority(&userlock.pl);
}

int
release_user_lock(void){
    return releasepriority(&userlock.pl);
}

void print_queue(void){
    print_priority_queue(&userlock.pl);
}

```

فراخوانی‌های acquire و release در صورتی که ناموفق باشند ۱- خروجی می‌دهند و در صورت موفقیت ۰ را خروجی می‌دهند.

همچنین مشکلاتی مانند زمانی که یک پردازش‌ای که قفل را در اختیار دارد و می‌خواهد exit کند و یا پردازش‌ای بخواهد دوباره همان قفل را acquire کند و یا بخواهد قفلی که در اختیار ندارد را release کند را برطرف کرده‌ایم.

برنامه‌ی سطح کاربری نوشتیم که تعدادی پردازش ایجاد می‌کند سپس هر پردازش در ناحیه‌ی بحرانی‌اش در یک فایل مشترک می‌نویسد:

```

#define NUM_FORKS ۱۰

void intToStr(int num, char* buffer) {
    int i = ۰;
    do {
        buffer[i++] = num % ۱۰ + '۰';
        num /= ۱۰;
    } while (num > ۰);
    int start = ۰;
    int end = i - ۱;
}

```

```

while (start < end) {
    char temp = buffer[start];
    buffer[start] = buffer[end];
    buffer[end] = temp;

    start++;
    end--;
}
buffer[i] = '\0';
}

int main(int argc, char* argv[]){
    int fd=open("lock-test.txt",O_CREATE|O_WRONLY);
    for (int i = 0; i < NUM_FORKS; i++){
        int pid = fork();
        if (pid == 0){

            acquire_user_lock();
            print_queue();
            printf(1, "pid lock: %d\n", getpid());

            char* write_data = "pid: ";
            int max_length = strlen(write_data);
            write(fd,write_data,max_length);
            char pid_str[12];
            intToStr(getpid(), pid_str);
            write(fd, pid_str, strlen(pid_str));
            write(fd,"\n",1);
            sleep(10);
        }
    }
}

```

```

        release_user_lock();
        exit();
    }
    //while(1);
}

while (wait() != -1);
close(fd);

exit();
}

```

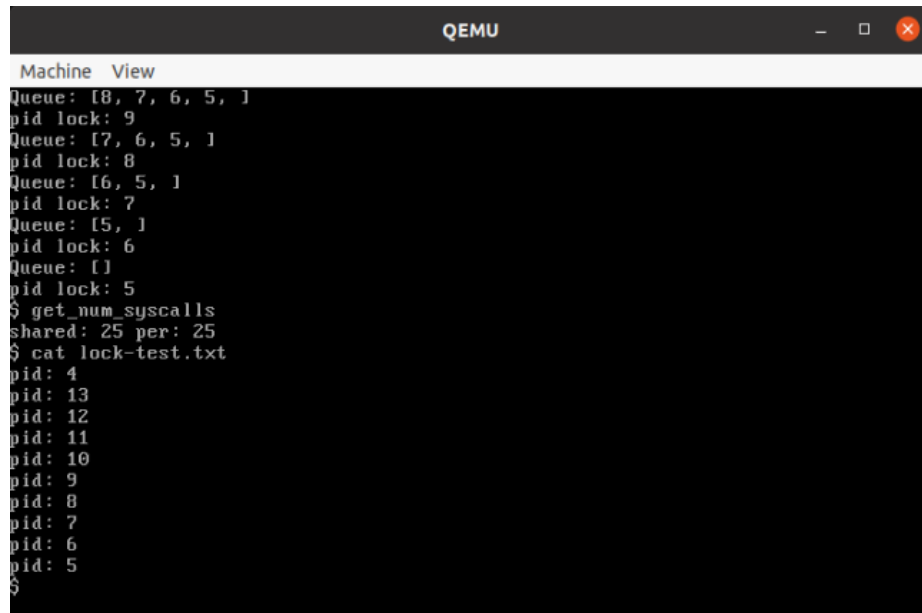
حالا در اجرای این برنامه می بینیم که بجز پردازشی اول باقی پردازها به ترتیب بزرگی pid قفل را دریافت می کنند. (پردازشی اول چون هنوز صف خالیست قفل را می گیرد و بعد در زمانی که ناحیه بحرانی آن پردازش اجرا می شود باقی پردازها وارد صف می شوند و از آن به بعد به ترتیب بزرگی pid قفل گرفته می شود).

همچنین در فایل مشترکی که پردازها می نویسند هم می بینیم race condition رخ نداده و پردازها به نوبت در فایل نوشته اند:

```

2. Ali Hamzhepour
3. Mina Shirazi
$ lock_test
Queue: []
pid lock: 4
Queue: [12, 11, 10, 9, 8, 7, 6, 5, 1]
pid lock: 13
Queue: [11, 10, 9, 8, 7, 6, 5, 1]
pid lock: 12
Queue: [10, 9, 8, 7, 6, 5, 1]
pid lock: 11
Queue: [9, 8, 7, 6, 5, 1]
pid lock: 10
Queue: [8, 7, 6, 5, 1]
pid lock: 9
Queue: [7, 6, 5, 1]
pid lock: 8
Queue: [6, 5, 1]
pid lock: 7
Queue: [5, 1]
pid lock: 6
Queue: []
pid lock: 5
$

```



```
QEMU
Machine View
Queue: [8, 7, 6, 5, 1]
pid lock: 9
Queue: [7, 6, 5, 1]
pid lock: 8
Queue: [6, 5, 1]
pid lock: 7
Queue: [5, 1]
pid lock: 6
Queue: [1]
pid lock: 5
$ get_num_syscalls
shared: 25 per: 25
$ cat lock-test.txt
pid: 4
pid: 13
pid: 12
pid: 11
pid: 10
pid: 9
pid: 8
pid: 7
pid: 6
pid: 5
$
```

آیا این پیاده‌سازی ممکن است که دچار گرسنگی شود؟ راه حلی برای برطرف کردن این مشکل ارائه دهید. روش ارائه‌شده توسط شما باید بتواند شرایطی را که قفلها دارای اولویت یکسان میباشند را نیز پوشش دهد.

بله زیرا ممکن است pid یک پردازش پایین باشد و دائما یک پردازش با pid بزرگتر وارد صف شود و قفل به پردازش اولیه هیچ‌وقت نرسد.

می‌توان خاصیتی مانند aging در این قفل پیاده‌سازی کرد که اگر یک پردازش از مدت زمان مشخصی بیشتر در صف منتظر ماند، اولویت آن بالاتر می‌رود و می‌تواند به ابتدای صف برود.

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

در روش قفل بلیت هر پردازش یک بلیت دارد که شامل یک شماره است و قفل نیز یک شمارنده دارد که نشان می‌دهد در لحظه‌ی فعلی کدام بلیت می‌تواند قفل را بگیرد.

در این روش صرفا نیاز است یک عدد به عنوان شمارنده‌ی بلیت در قفل نگه‌داری کنیم و نیاز به نگه‌داشتن یک آرایه نیست. همچنین در این روش دیگر مشکل گرسنگی رخ نمی‌دهد.