

## آزمایشگاه سیستم عامل

پروژه چهار

اعضای گروه:

الهه خداوردی - 810100132

فرشته باقری - 810100089

عاطفه میرزاخانی - 810100220

Repository: <https://github.com/elahekhodaverdi/Operating-System-Lab-Projects>

Latest Commit: 14c3a401662dd8905bbea98af968686ae40f2f8f

## همگام سازی 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؟

زیرا ممکن است در هنگام استفاده از این قفل ما برای مثال یک lock را در اختیار داشته باشیم که وقفه رخ داده نیز دقیقا به آن نیاز داشته باشد. برای مثال فرض کنید ما قطعه زیر را داریم:

```
acquire(tickslock);
```

و قفل مربوطه را بدست می آوریم حال فرض کنید وقفه ای رخ میدهد که در interrupt handler آن نیز داریم:

```
acquire(tickslock);
```

حال چون از قبل این قفل را قسمت قبل بدست آوردیم و آن پردازش متوقف شده interrupt handler تا ابد منتظر می ماند تا قفل آزاد شود تا آن را بدست آورد و بنابراین با deadlock مواجه می شویم.

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

همانطور که در سوال قبل توضیح داده شده، استفاده از spinlock به صورتی که وقفه ها فعال باشند منجر به deadlock می شود. برای حل مشکل توابع popcli و pushcli را داریم.

این توابع به این صورت عمل می کنند که هرگاه که یک قفل فعال می شود پیش از فراخوانی تابع xchg تابع pushcli را فراخوانی می کنیم و بعد از رهاسازی یک قفل نیز تابع popcli را فراخوانی می کنیم. با فراخوانی این توابع در واقع با فعال شدن هر قفل یکی به تعداد ncli اضافه می کنیم و مطمئن می شویم که وقفه ها غیرفعال باشند و هنگام آزادسازی قفل ها نیز یکی از ncli کم می کنیم تا هنگامی که مقدار آن به 0 برسد، در این زمان مطمئن هستیم که هیچ spinlock فعال نیست و وقفه ها را دوباره فعال می کنیم.

تفاوت آن‌ها با تابع cli و sli این است که این توابع کنترل‌کننده تعداد قفل‌ها هستند و در واقع توابعی که وقفه‌ها را فعال و یا غیر فعال می‌کنند در اصل sli و cli هستند.

### (3) چرا قفل مذکور در سیستم‌های تک هسته‌ای مناسب نیست؟ روی کد توضیح دهید.

```
void
acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to avoid deadlock.
    if(holding(lk))
        panic("acquire");
    while(xchg(&lk->locked, 1) != 0)
        ;
    __sync_synchronize();
    lk->cpu = mycpu();
    getcallerpcs(&lk, lk->pcs);
}
```

همانطور که در قطعه کد مربوط به تابع acquire می‌بینیم از آنجا که هنگامی که یک پردازش منتظر قفل می‌ماند هنوز در حالت running قرار می‌گیرد و متوقف نمی‌شود تا زمانی که قفل بدست آید. اگر پردازش دیگری داشته باشیم که از قبل قفل را بدست آورده باشد و پردازش جدیدی به همان قفل نیاز داشته باشد، پردازش جدید تا زمانی که پردازش دیگری قفل را رها کند منتظر می‌ماند؛ این موضوع در سیستم‌های چند هسته در مشکل busy waiting را بوجود می‌آورد ولی از آنجا که در سیستم‌های تک هسته فقط یک پردازش در هر زمان در حال اجراست اگر یکی از پردازش‌های قبلی قفل را بدست آورده و رها نکرده باشد پردازش جدید نمی‌تواند قفل را بدست بیاورد و لذا در سیستم‌های تک هسته استفاده از این قفل در بدترین حالت منجر به deadlock می‌شود.

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

amoswap.w rd, rs2, (rs1)

این تابع یک دستور اتمی مربوط به حافظه در معماری RISC-V است که مراحل زیر را به صورت اتمی انجام می‌دهد:

مقدار موجود در آدرس ذخیره شده در رجیستر rs1 را در rd ذخیره می‌کند.

کلمه ذخیره شده و کلمه ذخیره شده در رجیستر rs2 را جا به جا می‌کند. نتیجه را در خانه حافظه ذخیره می‌کند. در مدت زمانی که این دستور اجرا می‌شود هیچ thread دیگری اجازه دسترسی و تغییر در آن مکان حافظه را ندارد.

از آنجا که این دستور به صورت اتمی اجرا می‌شود به دلایل زیادی از آن استفاده می‌شود:

- باعث می‌شود race condition جلوگیری شود و باعث بهبود در کنترل همگامی می‌شود و از inconsistencies در دیتا جلوگیری می‌کند.
- با استفاده از آن می‌توان مکانیسم‌های مختلف lock کردن از جمله mutex، spinlock را پیاده‌سازی کرد.
- از آنجا که مستقیماً در سطح سخت‌افزار پیاده‌سازی می‌شوند از مکانیسم‌های نرم‌افزاری معادل بسیار بهینه‌تر و سریع‌تر هستند.
- از آنجا که یکی از دستورات RISC-V است می‌تواند در سیستم‌های مختلفی به سادگی استفاده شود.

## (5) مختصری راجع به تعامل میان پردازنده‌ها توسط دو تابع مذکور توضیح دهید.

آدرس قفل به تابع `acquiresleep` به عنوان ورودی پاس داده می‌شود و سپس در بدنه تابع پردازنده تا زمانی که شرایط برای در دست گرفتن قفل به آن داده نشده است `sleep` میکند.

```
22 void
23 acquiresleep(struct sleeplock *lk)
24 {
25     acquire(&lk->lk);
26     while (lk->locked) {
27         sleep(lk, &lk->lk);
28     }
29     lk->locked = 1;
30     lk->pid = myproc()->pid;
31     release(&lk->lk);
32 }
```

به تابع `releasesleep` نیز به عنوان ورودی آدرس قفل پاس داده می‌شود، پردازنده‌ای که قفل را نگه داشته بود، تابع `wakeup` را فراخوانی می‌کند که در آن `wakeup1` فراخوانی می‌شود و در این تابع، پردازنده‌هایی که روی آن قفل خاص در وضعیت `sleep` هستند بیدار می‌شوند و در وضعیت `RUNNABLE` قرار می‌گیرند.

```
34 void
35 releasesleep(struct sleeplock *lk)
36 {
37     acquire(&lk->lk);
38     lk->locked = 0;
39     lk->pid = 0;
40     wakeup(lk);
41     release(&lk->lk);
42 }
```

```

573 // PAGEBREAK!
574 // Wake up all processes sleeping on chan.
575 // The ptable lock must be held.
576 static void
577 wakeup1(void *chan)
578 {
579     struct proc *p;
580
581     for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
582         if (p->state == SLEEPING && p->chan == chan)
583             p->state = RUNNABLE;
584 }
585
586 // Wake up all processes sleeping on chan.
587 void wakeup(void *chan)
588 {
589     acquire(&ptable.lock);
590     wakeup1(chan);
591     release(&ptable.lock);
592 }
593

```

## 6) حالات مختلف پردازها در xv6 را توضیح دهید. تابع sched چه وظیفه ای دارد؟

در فایل proc.h حالات مختلف پردازها را به صورت زیر داریم:

```

58 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
59

```

UNUSED: در این حالت استفاده‌ای از پردازنده نمی‌شود.

EMBRYO: زمانی که پردازهای از وضعیت UNUSED خارج می‌شود به وضعیت EMBRYO می‌رود.

SLEEPING: زمانی که پردازها به منابعی نیاز دارد که آماده نیست، پردازها در این حالت قرار می‌گیرد و دیگر در cpu نیست. این آماده نبودن به دلیل در اختیار داشتن منابع توسط پردازها یا عملیات I/O یا ... است.

RUNNABLE: حالتی است که پردازها آماده اجرا است و آماده است که cpu به آن اختصاص داده شود.

RUNNING: وقتی پردازهای در این حالت قرار دارد یعنی در حال اجرا است و cpu در حال حاضر به آن اختصاص داده شده است.

ZOMBIE: زمانی که کار پردازهای تمام می‌شود و پایان می‌یابد، به این حالت در می‌آید و تا زمانی که والدش wait را فراخوانی نکرده است در این حالت می‌ماند زیرا با وجود پایان این پردازها، اطلاعات آن هنوز در ptable وجود دارد.

```

478 // Enter scheduler. Must hold only ptable.lock
479 // and have changed proc->state. Saves and restores
480 // intena because intena is a property of this
481 // kernel thread, not this CPU. It should
482 // be proc->intena and proc->ncli, but that would
483 // break in the few places where a lock is held but
484 // there's no process.
485 void sched(void)
486 {
487     int intena;
488     struct proc *p = myproc();
489
490     if (!holding(&ptable.lock))
491         panic("sched ptable.lock");
492     if (mycpu()->ncli != 1)
493         panic("sched locks");
494     if (p->state == RUNNING)
495         panic("sched running");
496     if (readeflags() & FL_IF)
497         panic("sched interruptible");
498     intena = mycpu()->intena;
499     swtch(&p->context, mycpu()->scheduler);
500     mycpu()->intena = intena;
501 }

```

تابع sched() برای switch context کردن به context زمانبند است. پردازش برای رها کردن CPU به این تابع می‌آید (که از قبل باید state اش از RUNNING عوض شده باشد و قفل ptable را داشته باشد). در تابع فلگ enable interrupt ذخیره شده و پس از بازگشت برگردانده میشود. این تابع با استفاده از swtch، کانتکست را تغییر میدهد و ادامه تابع scheduler اجرا میشود که به context پردازش RUNNABLE دیگری تعویض میکند.

## 7) تغییری در توابع دسته دوم داده تا تنها پردازش صاحب قفل، قادر به آزادسازی آن باشد. قفل معادل در هسته لینوکس را به طور مختصر معرفی نمایید

برای اینکه تنها پردازش صاحب قفل قادر به آزادسازی آن باشد، تابع releasesleep را مطابق زیر تغییر دادیم تا اگر شماره پردازش فعلی با شماره پردازش صاحب قفل یکی بود، بتواند آن را آزاد کند.

```

33 void releasesleep(struct sleeplock *lk)
34 {
35     acquire(&lk->lk);
36     if (lk->pid == myproc()->pid)
37     {
38         lk->locked = 0;
39         lk->pid = 0;
40         wakeup(lk);
41     }
42     release(&lk->lk);
43 }

```

قفل معادل در هسته لینوکس mutex است که کد آن در لینک زیر موجود است.

<https://github.com/torvalds/linux/blob/master/include/linux/mutex.h>

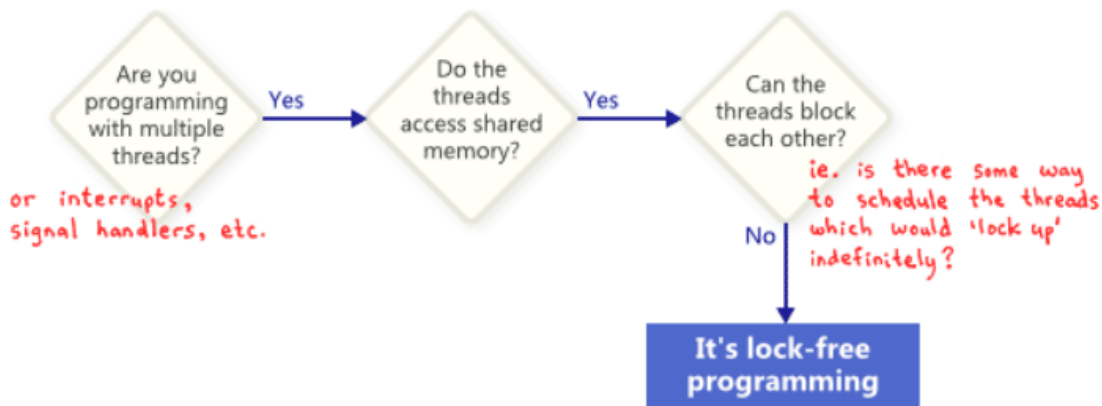
```
struct mutex {
    atomic_long_t      owner;
    raw_spinlock_t      wait_lock;
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
    struct optimistic_spin_queue osq; /* Spinner MCS lock */
#endif
    struct list_head     wait_list;
#ifdef CONFIG_DEBUG_MUTEXES
    void                *magic;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map    dep_map;
#endif
};
```

در استراکت mutex همانطور که قابل مشاهده است یک فیلد به نام owner در نظر گرفته شده است، این فیلد در حین آزادسازی قفل چک می‌شود تا تنها صاحب قفل مجاز به این کار باشد.

## 8) روشی دیگر برای نوشتن برنامه‌ها استفاده از الگوریتم‌های free-lock است. مختصری راجع به آن‌ها توضیح داده و از مزایا و معایب آنها نسبت به برنامه‌نویسی با lock بگویید.

برنامه‌نویسی بدون قفل یک روش است که به روزرسانی همزمان داده‌ساختارهای مشترک را بدون نیاز به انجام همگام‌سازی پر هزینه بین thread‌ها امکان‌پذیر می‌کند. این روش اطمینان می‌دهد که هیچ thread ای برای مدت زمان نامحدودی مسدود نشود، و پیشرفت برخی از thread‌ها را زمانی که thread‌های متعددی وجود دارد، تضمین می‌کند.

الگوریتم‌های بدون قفل داده‌ساختارها و توابعی را که با دقت طراحی شده‌اند، فراهم می‌کنند تا به thread‌های متعدد اجازه دهند تا به صورت مستقل از یکدیگر کار کنند. این به این معنی است که شما سعی نمی‌کنید قبل از اجرای critical section خود یک قفل را acquire کنید. به جای آن، شما به صورت مستقل یک کپی از بخشی از داده‌ساختار را به‌روز می‌کنید و سپس آن را به صورت اتمیک به ساختار مشترک با CAS (مقایسه و تعویض) اعمال می‌کنید.



الگوریتم های lock-free می‌توانند مزایای متنوعی نسبت به داده‌ساختارهای با قفل ارائه دهند مانند scalability و performance در سناریو های با رقابت بالا که بسیاری از thread ها برای گرفتن یک resource یکسان رقابت می‌کنند. این الگوریتم ها می‌توانند سربار context-switch، خطر deadlock livelock یا priority inversion را کاهش دهند. همچنین می‌توانند fault tolerant و پاسخگویی را با جلوگیری از thread failure یا از پیش برداشتن بدون اختلال در سازگاری یا پیشرفت داده افزایش دهند. این الگوریتم ها با چالش‌ها و محدودیت‌هایی همراه هستند. آنها نیاز به درک عمیق‌تری از مفاهیم concurrency و همگام‌سازی دارند، مانند اتمی بودن، ترتیب حافظه، یا شرایط پیشرفت. طراحی و آزمایش برنامه های بدون قفل پیچیده‌تر از داده‌ساختارهای با قفل است، و ممکن است اشکالات یا خطاهای ریزی را معرفی کنند که سخت است آنها را تشخیص داد یا تکرار کرد. علاوه بر این، آنها ممکن است برای همه انواع برنامه ها مناسب نباشند نباشند، و ممکن است با برخی از زبان‌ها، کتابخانه‌ها، یا پلتفرم‌ها خوب کار نکنند. علاوه بر این، داده‌ساختارهای بدون قفل ممکن است مشکلات سازگاری با سایر مکانیزم‌های همزمانی داشته باشند، مانند معاملات یا سیگنال‌ها. اگرچه آنها ممکن است در برخی موارد عملکرد بهتری نسبت به داده‌ساختارهای با قفل داشته باشند، اما همچنین برخی از معایبی نیز دارند، مانند مصرف بیشتر حافظه، کارایی کمتر cache، یا عدالت کمتر.

## پیاده‌سازی متغیرهای مختص هر هسته پردازنده

### الف) cache coherence protocols

این پروتکل‌ها اطمینان می‌دهند که تمام core ها نمای یکپارچه‌ای از داده‌ها در حافظه را ببینند. زمانی که یک core مکانی در حافظه را تغییر می‌دهد، این پروتکل‌ها مکانیسم‌هایی را برای به‌روزرسانی یا ابطال ورودی‌های

متناظر در سایر کش‌ها فراهم می‌کنند، سازگاری داده‌ها در تمام کش‌ها را حفظ می‌کنند. به این ترتیب، تمام هسته‌ها می‌توانند به درستی با داده‌های مشترک کار کنند.

## ب) ticket lock

Ticket lock یک روش همگام‌سازی است که برای کنترل دسترسی به منابع مشترک در برنامه‌های multi-thread استفاده می‌شود. در این روش، هر thread ای که می‌خواهد منبع را قفل کند، یک "بلیت" دریافت می‌کند. رشته‌ها سپس بر اساس ترتیب بلیت‌هایشان به منبع دسترسی پیدا می‌کنند. با این حال، قفل بلیت ممکن است با مشکلات cache coherence روبرو شود. این مشکل زمانی پیش می‌آید که یک thread مقدار قفل را تغییر می‌دهد و این تغییر باید در کش‌های دیگر نیز منعکس شود. این می‌تواند باعث ایجاد ترافیک سنگین بر روی باس شود و عملکرد سیستم را کاهش دهد. برای مقابله با این مشکل، می‌توان از پروتکل‌های هماهنگی کش استفاده کرد. این پروتکل‌ها اطمینان می‌دهند که تمام هسته‌ها داده‌های به‌روز را می‌بینند و هنگامی که یک هسته مقداری را تغییر می‌دهد، مکانیسم‌هایی را فراهم می‌کنند تا ورودی‌های متناظر در سایر کش‌ها را به‌روز یا نامعتبر کنند. این روش باعث می‌شود تمام هسته‌ها بتوانند به درستی با داده‌های مشترک کار کنند.

## ج) per-core variables

در لینوکس با استفاده از ماکروی زیر می‌توان متغیرهای مختص هر هسته را تعریف کرد:

```
DEFINE_PER_CPU(int, per_cpu_n);
```

با استفاده از این ماکرو، یک متغیر با تایپ `int` در `data..percpue` ساخته می‌شود که زمانی که `initialize` می‌شود تابع `setup_per_cpu_areas` فراخوانی می‌شود و بخش `data..percpue` به تعداد هسته‌ها لود می‌شود و متغیرهای مختص هر هسته ایجاد می‌شوند.

## اضافه کردن system call

برای اینکار به استراکت `cpu` یک متغیر `syscallcount` اضافه می‌کنیم که تعداد سیستم کال‌های هر `core` را ذخیره می‌کند. برای صحت‌سنجی یک `global variable` به نام `shared_syscallcount` هم تعریف می‌شود. برای اینکه تعداد سیستم کال‌های فراخوانی شده در یک بار اجرای برنامه را پیدا کنیم ابتدای تابع `exec` مقادیر `syscallcount` را صفر می‌کنیم.

در ادامه یک برنامه تست نوشته شده است که پردازش‌ها روی یک فایل مشترک می‌نویسند.

توابع مورد نیاز:



```
int
sys_getsyscallcount(void)
{
    int i, total = 0;
    for (i = 0; i < ncpu; i++) {
        int count = syscallcount(i);
        if(count >= 0) {
            cprintf("System call count for %d-th core: %d\n", i, count);
            total += count;
        }
    }
    cprintf("total syscall count: %d\n", total);
    cprintf("shared syscall count: %d\n", shared_syscallcount);
    return total;
}
```

```

int
syscallcount(int cpu)
{
    if(cpu < 0 || cpu >= ncpu)
        return -1;
    return cpus[cpu].syscallcount;
}

```

```

int reset_syscallcount(int cpu){
    if(cpu < 0 || cpu >= ncpu)
        return -1;
    cpus[cpu].syscallcount = 0;

    return 0;
}

```

```

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

            char* write_data = "OS Lab4 - G29";
            int max_length = 13;

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

            release_user();
            exit();

        }
    }

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

    getsyscallcount();
    exit();
}

```

```
QEMU
Machine View
SeaBIOS (version 1.13.0-1ubuntu1.1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CB00+1FECCB00 CA00

Booting from Hard Disk...
cpu1: starting 1
cpu2: starting 2
cpu3: starting 3
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
tesinit: starting sh
t$ syscall
System call count for 0-th core: 2
System call count for 1-th core: 4
System call count for 2-th core: 9
System call count for 3-th core: 7
total syscall count: 22
shared syscall count: 22
$ _
```

## پیاده‌سازی سازوکار همگام سازی با قابلیت اولویت دادن

این ساز و کار را در قالب یک قفل به نام priority lock پیاده‌سازی کردیم. کد مربوط به این قفل را در قالب دو فایل جدید به نام prioritylock.c و prioritylock.h به سیستم عامل xv6 اضافه کردیم. ابتدا یک استراکت تعریف کردیم که به صورت زیر است:

```
struct prioritylock {
    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
};
```

توابع مربوط به این قفل که مربوط به init کردن قفل و acquire و release کردن این قفل هستند نیز در فایل c. به صورت زیر نوشته شده اند.

### Init:

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

### Acquire:

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

### Release:

```
void
releasepriority(struct prioritylock *lk)
{
    if (myproc()->pid != lk->pid)
        panic("release");

    acquire(&lk->lk);
    lk->locked = 0;
```

```
lk->pid = 0;
wakeup2(lk);
release(&lk->lk);
}
```

## Holding:

```
Int holdingpriority(struct prioritylock *lk)
{
    int r;
    acquire(&lk->lk);
    r = lk->locked && (lk->pid == myproc()->pid);
    release(&lk->lk);
    return r;
}
```

همانطور که در کد مربوط به release میبینیم برای بیدار کردن پردازش هایی که در انتظار رها شدن قفل هستند از تابع جدید wakeup2 استفاده کردیم که براساس اولویت مخصوص به این قفل، در صورت وجود پردازش در صف این قفل، پردازش با بالاترین اولویت را بیدار میکند و در حالت RUNNABLE قرار می دهد. تعریف این تابع نیز به شرح زیر است:

```
void wakeup2(void *chan)
{
    acquire(&ptable.lock);
    struct proc *p;
    struct proc *p_f = 0;

    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if (p->state == SLEEPING && p->chan == chan)
        {
            if (p_f)
            {
                if (p_f->pid > p->pid)
                    p_f = p;
            }
            else
            {
                p_f = p;
            }
        }
}
```

```

    }
    if (p_f)
        p_f->state = RUNNABLE;
    release(&ptable.lock);
}

```

حال برای تست این قفل بایستی یک متغیر به صورت گلوبال تعریف کرده که شامل یک قفل باشد و در سیستم کال مربوطه آن را acquire و release کنیم. این متغیر را در proc.c اضافه کردیم:

```

struct
{
    int number;
    struct prioritylock lock;
} buffer_test;

```

حال یک تابع برای init کردن این استراکت تعریف کردیم:

```

void buf_test_init(void)
{
    buffer_test.number = 0;
    initprioritylock(&buffer_test.lock, "test_buffer");
}

```

این تابع را بایستی در فایل main.c در تابع main فراخوانی کنیم تا با راه اندازی سیستم عامل این استراکت نیز مقداردهی شود.

حال به سراغ سیستم کال مربوطه می‌رویم:

```

void prioritylock_test()
{
    cprintf("Process with pid %d entering critical section\n", myproc()->pid);
    acquirepriority(&buffer_test.lock);

    cprintf("Process with pid %d accessed the lock\n", myproc()->pid);

    volatile long long a = 3;
    volatile long long b = 4;
    volatile long long temp = 0;
    for (long long l = 0; l < 10000; l++)
        for (long long s = 0; s < 100; s++)
            for (long long k = 0; k < 200; k++)

```

```

    {
        temp += a * b;
    }
    print_priority_queue(&buffer_test.lock);

    buffer_test.number += 1;

    releasepriority(&buffer_test.lock);
    cprintf("\nProcess with pid %d leaving critical section\n\n", myproc()->pid);
}

```

تابعی که صف مربوط به قفل را چاپ می‌کند نیز :

```

void print_priority_queue(void *chan)
{
    acquire(&ptable.lock);

    struct proc *p;
    int m = 0;
    struct proc * p_f = 0;
    cprintf("\nPriority Queue:\n");
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if (p->state == SLEEPING && p->chan == chan)
        {
            if(p_f && p_f->pid < p->pid){
                p_f = p;
            }
            else {
                p_f = p;
            }
            cprintf("pid: %d\n",p->pid);
            m++;
        }
    }
    if(m == 0)
        cprintf("Queue is empty.\n");
    if(p_f)
        cprintf("Process with highest priority has pid: %d\n",p_f->pid);
    release(&ptable.lock);
}

```

```
}
```

حال یک برنامه سطح کاربر به xv6 اضافه کردیم که در آن چند پردازش ساخته و برای هر پردازش سیستم کال مربوطه را فراخوانی می‌کنیم تا همه پردازش‌ها به دنبال بدست آوردن قفل باشند اسم این برنامه سطح کاربر را نیز test\_prioritylock گذاشتیم.

```
#define PROCS_NUM 3

int main()
{
    for (int i = 0; i < PROCS_NUM; i++)
    {
        int pid = fork();
        if (pid == 0)
        {
            prioritylock_test();
            exit();
        }
    }
    for (int i = 0; i < PROCS_NUM; i++)
        wait();
    exit();
}
```

نتیجه اجرای برنامه سطح کاربر:



```

$ test_prioritylock
Process with pid 4 entering critical section
Process with pid 4 accessed the lock
Process with pid 5 entering critical section
Process with pid 6 entering critical section

Priority Queue
pid: 5
pid: 6
Process with highest priority has pid: 6

Process with pid 4 leaving critical section

Process with pid 6 accessed the lock

Priority Queue
pid: 5
Process with highest priority has pid: 5

Process with pid 6 leaving critical section

Process with pid 5 accessed the lock

Priority Queue
Queue is empty

Process with pid 5 leaving critical section
$

```

آیا این پیاده‌سازی ممکن است که دچار گرسنگی شود؟ راه‌حلی 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 دچار گرسنگی می‌شود زیرا ممکن است در هر لحظه پردازش‌های جدیدی وارد شوند که به دنبال قفل هستند و از آنجا که هر پردازش جدید یک pid بزرگتر از قبلی‌ها می‌گیرد پس اولویت بالاتری نسبت به آن‌ها که از قبل در صف بودند می‌گیرد و اینگونه ممکن است پردازش‌هایی که ابتدا در صف رفته‌اند دچار گرسنگی شوند. برای حل این مشکل می‌توانیم از مکانیزم aging استفاده کنیم. در این مکانیزم مدت زمان مشخصی را معین کرده و هر پردازش‌های پس از گذشت این مدت زمان منتظر ماندن هر بار اولویتش افزایش می‌یابد و به موقعیت جلوتری در صف اولویت منتقل می‌شود. اگر اولویت دو پردازش نیز با هم یکسان بود می‌توانیم waiting time را

معیار قرار داده و مدت زمان انتظار هر یک که بیشتر بود اولویت آن را برتری دهیم و اگر آن هم یکسان بود می‌توانیم براساس pid تصمیم بگیریم و هر یک pid کوچک تری داشت آن را اولویت دهیم زیرا پردازش با pid کوچک تر در سیستم ما زودتر ساخته شده است.

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

قفل بلیت نیز یک مکانیزم همگام‌سازی با الگوریتم قفل است که از ورود غیرمجاز چند پردازش و یا thread به critical section جلوگیری می‌کند. این قفل با استفاده از spinlock پیاده‌سازی می‌شود و مبتنی بر ساختار صف FIFO یا First In First Out است.

این مکانیزم و صف مورد نظر همانند سیستم نوبت‌دهی در مکان‌هایی مانند نانویی است که مشتریان پس از ورود یک نوبت براساس وضعیت آن لحظه دریافت می‌کنند و براساس همان نوبت به ترتیب خدمات خود را دریافت می‌کنند. این شرایط در سیستم عامل با استفاده از قفل بلیت به این صورت است که هر Thread یا پردازش پس از درخواست برای کسب قفل مربوطه یک نوبت دریافت می‌کند و سپس براساس همان نوبت پردازش‌ها به قفل دسترسی پیدا خواهند کرد.

هنگامی که یک نخ یا یک پردازش قصد ورود به ناحیه بحرانی را دارد به صورت اتمی یک بلیت دریافت می‌کند و بلیت مجاز بعدی مربوط به قفل نیز یکی افزایش پیدا می‌کند و همچنین برای ورود هر پردازش به ناحیه بحرانی باید بلیت آن با مقدار معتبر بلیت برای ورود به قفل مقایسه می‌کند و اگر برابر بود قفل را بدست می‌آورد (همانند سیستم نوبت‌دهی بانک).

از آن طرف قفل اولویت نیز مکانیزم همگام‌سازی با الگوریتم قفل است که باعث می‌شود پردازش‌ها یا thread‌هایی که قصد ورود به ناحیه بحرانی دارند به ترتیب اولویت وارد ناحیه بحرانی شوند. ایده آن‌ها مانند هم است و همچنین هر دو عملیات خود را به صورت اتمی انجام می‌دهند ولی به دلیل اینکه در قفل بلیت نسبت به ترتیب ورود پردازش‌ها حساسیت وجود دارد دسترسی به قفل عادلانه تر است و گرسنگی رخ نمی‌دهد در حالی که این موضوع برای قفل اولویت برقرار نیست.

در شرایط خاص قفل اولویت و بلیت می‌توانند یکسان باشند.