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

Latest Commit Hash: <hash>

# همگام‌سازی در xv6

## علت غیرفعال شدن وقفه در حین اجرای ناحیه بحرانی، توضیح توابع pushcli و popcli و تفاوت آنها با cli و sti

تابع cli برای غیرفعال کردن وقفه‌ها و تابع sti برای فعال کردن وقفه‌ها استفاده می‌شود. توابع pushcli و popcli به ترتیب، به نوعی یک wrapper برای توابع cli و sti هستند با این تفاوت که می‌توانیم فرض کنیم یک stack مدیریت فعال کردن و یا غیرفعال کردن وقفه‌ها را به عهده می‌گیرد. تا زمانی که این استک خالی است، وقفه‌ها فعال‌اند و به محض اینکه با استفاده از تابع pushcli مقداری در استک push شود، وقفه‌ها غیرفعال می‌شوند. در واقع پیاده‌سازی توابع pushcli و popcli به این شکل است که تابع pushcli به ازای هربار فراخوانی، تابع cli را صدا می‌زند و وقفه‌ها را غیرفعال می‌کند اما تابع popcli تنها زمانی با استفاده از تابع sti وقفه‌ها را فعال می‌کند که استک کاملا خالی باشد. البته لازم به ذکر است که در واقعیت استکی وجود ندارد و فقط تعداد فراخوانی‌های هر یک از توابع در متغیری به نام ncli در هر پردازنده ذخیره می‌شود (به ازای فراخوانی تابع pushcli، مقدار این متغیر یک واحد افزایش پیدا می‌کند و به ازای فراخوانی تابع popcli، مقدار آن یک واحد کاهش می‌یابد) و زمانی که این متغیر برابر با 0 شود، وقفه‌ها فعال می‌شوند و هر موقع مقدار این متغیر بیشتر از 0 شود، وقفه‌ها غیرفعال می‌شوند. پس در واقع 2 بار فراخوانی تابع pushcli، نیازمند 2 بار فراخوانی تابع popcli برای فعال‌سازی مجدد وقفه‌ها است. کاربرد این توابع این است که اگر برای مثال به طور همزمان از دو قفل استفاده می‌کردیم، آزاد کردن یکی از قفل‌ها سبب فعال شدن وقفه‌ها نشود و این مورد فقط زمانی انجام شود که هر دو قفل آزاد شده باشند.

## چرا Spinlock در سیستم‌های تک‌هسته‌ای مناسب نیست؟

در ابتدا نیاز به توضیح مختصری راجع به نحوه عملکرد تابع acquire می‌باشد؛ کد این تابع به شرح زیر است (کامنت‌ها در اینجا حذف شده‌اند):

void  
acquire(struct spinlock \*lk)  
{  
  pushcli();  
  if(holding(lk))  
    panic("acquire");  
  
  while(xchg(&lk->locked, 1) != 0)  
    ;  
  
  \_\_sync\_synchronize();  
  
  lk->cpu = mycpu();  
  getcallerpcs(&lk, lk->pcs);  
}

در این تکه کد، ابتدا تمام وقفه‌ها توسط تابع pushcli غیرفعال می‌شوند. سپس در یک حلقه while دستور اتمی xchg اجرا می‌شود؛ این دستور به صورت اتمی، محتوای یک خانه از حافظه را با یک رجیستر جابجا می‌کند. تابع acquire بصورت مداوم مقدار lk->locked را توسط دستور xchg یک می‌کند؛ در صورتی که قفل از قبل توسط پردازه‌ای دیگر نگه داشته شده باشد، مقدار آن از قبل یک بوده و با اجرای این دستور تفاوتی ایجاد نمی‌شود و تابع xchg مقدار یک را برمی‌گرداند. اما اگر مقدار lk->locked صفر باشد، این دستور مقدارش را یک کرده (که به معنای درگیر بودن قفل است) و تابع xchg مقدار صفر را بازمی‌گرداند؛ در این صورت تابع acquire از حلقه خارج شده و پردازه ادامه روند اجرایش را از سر می‌گیرد.

واضح است که این روش باعث انتظار مشغول[[1]](#footnote-1) می‌شود. انتظار مشغول در سیستم‌های چند پردازنده‌ای باعث هدر رفتن زمان پردازنده و در نتیجه افت بهینگی سیستم شود؛ اما در سیستم‌های تک پردازنده‌ای این موضوع در بدترین حالت می‌تواند منجر به deadlock شود. حالتی را فرض کنید که در آن یک پردازه قفلی را در اختیار می‌گیرد؛ سپس پردازه‌ای دیگر سعی می‌کند قفل را به روش مذکور بدست بیاورد؛ در این صورت پردازه دوم هیچگاه از حلقه خارج نشده و پردازه‌‌های دیگر زمان‌بندی نمی‌شوند.

## مختصری راجع به تعامل میان پردازه‌ها توسط توابع Sleeplock و چرا استفاده از Spinlock در مثال producer-consumer ممکن نیست؟

## توضیح حالات مختلف پردازه‌ها در xv6 و وظیفه تابع sched()

## تغییری در توابع Sleeplock بدهید تا تنها پردازه صاحب قفل، قادر به آزادسازی آن باشد و قفل معادل در هسته لینوکس را به طور مختصر معرفی کنید

در استراکت sleeplock متغیر pid قرار داده شده که منظور اولیه از ایجاد آن برای دیباگ کردن بوده. از همین متغیر برای چک کردن صاحب قفل در releasesleep استفاده می‌کنیم. تکه کد زیر، تابع جدید است که قابلیت خواسته شده در آن ایجاد شده:

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

در هسته لینوکس از قفل mutex استفاده می‌شود. تعریف این قفل بصورت زیر می‌باشد (در فایل [mutex.h](https://github.com/torvalds/linux/blob/master/include/linux/mutex.h) موجود است):

    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  
    };

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

## افزونه‌های همگام‌سازی تراکنشی (TSX) را شرح دهید و نقش حذف قفل را در آن بیان کنید

# مانع

## پیاده‌سازی ماکروی barrier() در لینوکس برای معماری x86

## آیا یک دستور مانع حافظه باید مانع بهینه‌سازی هم باشد؟ نام ماکرو پیاده‌سازی سه نوع مانع حافظه در لینوکس برای معماری x86 را به همراه دستورهای ماشین پیاده‌سازی آنها ذکر کنید

بله یک مانع حافظه باید یک مانع بهینه‌سازی نیز باشد زیرا پردازنده فقط برنامه کامپایل شده را می‌بیند و ترتیت خطوط کدها پیش از بهینه‌سازی را نمی‌داند. برای مثال شبه کد زیر را در نظر بگیرید:

Statement1;

Statement2;

MemBarrier;

Statement3;

اگر فرض کنیم Statement2 و Statement3 به همدیگر وابسته نباشند، کامپایلر در هنگام بهینه‌سازی ممکن است ترتیب این 2 دستور را عوض کند. اما پردازنده به دلیل وجود مانع، تغییری در ترتیب دستورات نمی‌دهد و دستورات به ترتیب زیر اجرا می‌شوند:

Statement1;

Statement3;

Statement2;

در نتیجه ترتیب دستورات برخلاف خواسته توسعه‌دهنده، تغییر کرده است.

در لینوکس به طور کلی 3 دستور برای ایجاد مانع حافظه وجود دارد که در بخش زیر توضیح داده شده‌اند:

1. دستور rmb: این دستور از انتقال هرگونه دستور read access به سمت دیگر مانع، جلوگیری می‌کند. دستورالعمل ماشین آن در x86، lfence است که حرف l در آن، نشان‌دهنده کلمه load است.
2. دستور wmb: این دستور از انتقال هرگونه دستور write access به سمت دیگر مانع جلوگیری می‌کند. دستورالعمل ماشین آن در x86، sfence است که حرف s در آن، نشان‌دهنده کلمه store است.
3. دستور mb: این دستور از انتقال هرگونه دستور memory access به سمت دیگر مانع جلوگیری می‌کند (در واقع ترکیبی از 2 دستور قبلی است). دستورالعمل ماشین آن در x86، mfence است که حرف m در آن، نشان‌دهنده کلمه memory است.

برای یافتن دستورات مانع حافظه در لینوکس و دستورالعمل ماشین معادل آن‌ها در معماری x86، از فایل [linux/arch/x86/include/asm/barrier.h](https://github.com/torvalds/linux/blob/master/arch/x86/include/asm/barrier.h) استفاده شده است که کد آن در بخش زیر قابل مشاهده است:

#ifdef CONFIG\_X86\_32  
#define mb() asm volatile(ALTERNATIVE("lock; addl $0,-4(%%esp)", "mfence", \  
 X86\_FEATURE\_XMM2) ::: "memory", "cc")  
#define rmb() asm volatile(ALTERNATIVE("lock; addl $0,-4(%%esp)", "lfence", \  
 X86\_FEATURE\_XMM2) ::: "memory", "cc")  
#define wmb() asm volatile(ALTERNATIVE("lock; addl $0,-4(%%esp)", "sfence", \  
 X86\_FEATURE\_XMM2) ::: "memory", "cc")  
#else  
#define \_\_mb() asm volatile("mfence":::"memory")  
#define \_\_rmb() asm volatile("lfence":::"memory")  
#define \_\_wmb() asm volatile("sfence" ::: "memory")  
#endif

در لینوکس توابع دیگری نیز برای ایجاد مانع وجود دارند. برای مثال تابع barrier یک مانع نرم‌افزاری (بهینه‌سازی) به وجود می‌آورد. توابع smp\_rmb، smp\_wmb و smp\_mb نیز در سیستم‌های چندهسته‌ای به ترتیب معادل دستورات 1 تا 3 هست و در سیستم‌های تک هسته‌ای معادل دستور barrier هستند.

## کابرد مانع در پردازش موازی

# پیاده‌سازی Semaphore

# شبیه‌سازی مسئله فلاسفه خورنده

1. Busy waiting [↑](#footnote-ref-1)