دانشگاه صنعتی شریف

دانشکده مهندسی کامپیوتر

پروژه درس سیستم عامل

عنوان پروژه :

بررسی ابزارهای همگام سازی

در سیستم عامل لینوکس

استاد :جناب دکتر جلیلی

دانشجو:فاطمه مظلومیان

400211268

قبل از پرداختن به مسائل مربوط به همگام سازی، سعی خواهیم کرد بدانیم که به طور کلی یک همگام سازی اولیه چیست. در واقع، یک همگام سازی اولیه یک مکانیسم نرم افزاری است، که تضمین می کند که دو یا چند فرآیند یا رشته موازی به طور همزمان در یک بخش کد اجرا نمی شوند.

هسته لینوکس مجموعه‌ای از همگام‌سازی‌های ابتدایی مختلف را فراهم می‌کند مانند :

;seqlock/mutex /سمافورها / عملیات اتمی

Spinlocks در هسته لینوکس:

اسپینلاک یک مکانیسم همگام سازی سطح پایین است که به عبارت ساده، متغیری را نشان می دهد که می تواند در دو 1)اکتسابی 2)منتشر شد حالت باشد:

هر فرآیندی که می‌خواهد یک spinlock به دست آورد، باید مقداری بنویسد که حالت اکتسابی spinlock را برای این متغیر نشان دهد و وضعیت انتشار spinlock را روی متغیر بنویسد. اگر فرآیندی سعی کند کدی را اجرا کند که توسط spinlock محافظت می شود، قفل می شود در حالی که فرآیندی که این قفل را نگه می دارد آن را آزاد می کند. در این حالت تمام عملیات مربوط باید اتمی باشد تا از وضعیت شرایط race جلوگیری شود.در زیر spinlock با نوع spinlock_t در هسته لینوکس نشان داده می شود. اگر به کد هسته لینوکس نگاهی بیندازیم، خواهیم دید که این نوع به طور گسترده مورد استفاده قرار می گیرد. spinlock_t به صورت تعریف شده است:

```
typedef struct spinlock {
        union {
              struct raw_spinlock rlock;

#ifdef CONFIG_DEBUG_LOCK_ALLOC
# define LOCK_PADSIZE (offsetof(struct raw_spinlock, dep_map))
                struct {
                        u8 __padding[LOCK_PADSIZE];
                        struct lockdep_map dep_map;
                };
#endif
        };
} spinlock_t;
```

و در فایل هدر include/linux/spi lock_types.h قرار دارد. ممکن است ببینیم که اجرای آن به وضعیت گزینه پیکربندی هسته CONFIG_DEB G_LOCK_ALL C بستگی دارد.. بنابراین، اگر گزینه پیکربندی هسته CONFIG_DEB G_LOCK_ALL C غیرفعال باشد، spinlock_t دارای اتحاد با یک فیلد است که - raw_spinlock:

```
typedef struct spinlock {
        union {
              struct raw_spinlock rlock;
        };
} spinlock_t;
```

ساختار raw_spinlock تعریف شده در همان فایل هدر نشان دهنده اجرای spinlock معمولی است. حال به این میپردازیم که ساختار raw_spinlock چگونه تعریف شده است:

```
typedef struct raw_spinlock {
        arch_spinlock_t raw_lock;
#ifdef CONFIG_DEBUG_SPINLOCK
    unsigned int magic, owner_cpu;
    void *owner;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map dep_map;
#endif
} raw_spinlock_t;
```

که در آن arch_spinlock_t اجرای spinlock خاص معماری را نشان می دهد.

هسته لینوکس عملیات اصلی زیر را روی یک اسپینلاک ارائه می کند:

- spin_lock_initمقدار دهی اولیه اسپینلاک داده شده را ایجاد می کند

spin_lock - spinlockداده شده را بدست می آورد

- spin_lock_bhوقفه‌های نرم‌افزار را غیرفعال می‌کند و اسپین‌لاک داده‌شده را بدست می‌آورد

spin_lock_irqsa e وspin_lock_irq - وقفه‌ها را در پردازنده محلی غیرفعال کنید، حالت وقفه قبلی را در پرچم‌ها حفظ کنید و اسپین‌لاک داده شده را بدست آورید

spin_unlock - spinlockداده شده را آزاد می کند

spin_unlock_bh - spinlockداده شده را آزاد می کند و وقفه های نرم افزار را فعال می کند

- spin_is_lockedوضعیت spinlock داده شده را برمی گرداند

حال اجرای ماکرو spin_lock_init را بررسی کنیم. این ماکرو در فایل هدر include/linux/spi lock.h تعریف شده است و ماکرو spin_lock_init به نظر می رسد:

```
#define spin_lock_init(_lock)            \
do {                          \
    spinlock_check(_lock);                \
    raw_spin_lock_init(&(_lock)->rlock);    \
} while (0)
```

همانطور که می بینیم، ماکرو spin_lock_init یک spinlock می گیرد و دو عملیات را اجرا می کند: spinlock داده شده را بررسی میکند و raw_spin_lock_i آن را اجرا میکند. اجرای spinlock_check بسیار آسان است، این تابع فقط raw_spinlock_t اسپینلاک داده شده را برمی گرداند تا مطمئن شود که ما دقیقاً قفل خام اولیه را دریافت کرده ایم.

```
static __always_inline raw_spinlock_t *spinlock_check(spinlock_t
*lock)
{
    return &lock->rlock;
}
```

ماکرو raw_spin_lock_init:

```
# define raw_spin_lock_init(lock)        \
do {                                 \
    *(lock) = __RAW_SPIN_LOCK_UNLOCKED(lock);    \
} while (0)
```

مقدار __RAW_SPIN_ OCK_UNLOCK D را با اسپینلاک داده شده به raw_spinlock_t داده شده اختصاص می دهد. همانطور که ممکن است از نام ماکرو __RAW_SPIN_ OCK_UNLOCK D متوجه شویم، این ماکرو مقداردهی اولیه اسپینلاک داده شده را انجام می دهد و آن را در حالت آزاد قرار می دهد.

سمافور:

همگام سازی اولیه بعدی بعد از spinlock که در این قسمت خواهیم دید سمافور است. سمافور مکانیسم دیگری برای پشتیبانی از همگام سازی نخ یا فرآیند است. هسته لینوکس قبلاً یک مکانیسم همگام سازی را ارائه می دهدspinlocks - ،

حال چرا ما به مکانیزم دیگری نیاز داریم؟ برای پاسخ به این سوال باید جزئیات هر دوی این مکانیسم ها را بدانیم. ما قبلاً با spinlocks آشنا شدیم، بنابراین از این مکانیسم شروع کنیم. Spinlock یک قفل ایجاد می کند که برای محافظت از یک منبع مشترک در برابر تغییر توسط بیش از یک فرآیند به دست می آید. در نتیجه، سایر فرآیندهایی که سعی می کنند قفل فعلی را بدست آورند متوقف می شوند) معروف به "spin-in-place" یا انتظار مشغول .(سوئیچ زمینه مجاز نیست زیرا برای جلوگیری از بن بست، preemptionغیرفعال است. در نتیجه، spinlockفقط در صورتی باید استفاده شود که قفل فقط برای مدت زمان بسیار کوتاهی به دست آید، در غیر این صورت مقدار انتظار مشغول که توسط سایر فرآیندها انباشته شده است منجر به عملکرد بسیار ناکارآمد می شود. برای قفل هایی که باید برای مدت نسبتاً طولانی به دست آیند، به سمافور روی می آوریم. سمافورها راه حل خوبی برای قفل هایی است که ممکن است برای مدت طولانی به دست بیاید. به عبارت دیگر این مکانیسم برای قفل هایی که برای مدت کوتاهی به دست می آیند بهینه نیست

سمافور با ساختار زیر نشان داده میشود

```
struct semaphore {
    raw_spinlock_t        lock;
    unsigned int         count;
    struct list_head     wait_list;
};
```

در هسته لینوکس ساختار سمافور از سه قسمت تشکیل شده است

spinlock – lock:برای محافظت از داده ها

count : تعدا منابع در دسترس

wait_list : لیست فرآیند های منتظر به دست آوردن قفل

هسته لینوکس API زیر را برای دستکاری سمافورها فراهم می کند:

```
void down(struct semaphore *sem);
void up(struct semaphore *sem);
int  down_interruptible(struct semaphore *sem);
int  down_killable(struct semaphore *sem);
int  down_trylock(struct semaphore *sem);
int  down_timeout(struct semaphore *sem, long jiffies);
```

دو عملکرد اول: up , down برای بدست آوردن و رها کردن سمافور داده شده است. تابع down_interruptible سعی می کند یک سمافور بدست آورد. اگر این تلاش موفقیت آمیز بود، فیلد شمارش سمافور داده شده کاهش می یابد و قفل به دست می آید، به عبارت دیگر وظیفه به حالت مسدود شده یا به عبارت دیگر پرچم TASK_INTERR PTIBLE تنظیم می شود. این پرچم TASK_INTERR PTIBLE به این معنی است که فرآیند ممکن است توسط سیگنال به حالت خراب برگردد. تابع down_killable همانند تابع down_interruptible عمل می کند، اما پرچم TASK_KILLABL را برای فرآیند فعلی تنظیم می کند. این بدان معنی است که فرآیند انتظار ممکن است توسط سیگنال کشتن قطع شود. تابع down_trylock مشابه تابع spin_trylock است. اگر این عملیات ناموفق بود، این تابع سعی می کند قفل را بدست آورد و از آن خارج شود. در این صورت فرآیندی که می خواهد قفل را بدست آورد، منتظر نخواهد ماند. آخرین تابع down_timeout سعی می کند یک قفل بدست آورد. زمانی که مهلت داده شده منقضی شود در حالت انتظار قطع می شود. به‌علاوه، ممکن است متوجه شوید که مهلت زمانی در چند لحظه است. از تابع پایین شروع می کنیم. این تابع در فایل کد منبع kernel/locking/s maphore.c تعریف شده است.

```
void down(struct semaphore *sem)
{
        unsigned long flags;

        raw_spin_lock_irqsave(&sem->lock, flags);
        if (likely(sem->count > 0))
                sem->count--;
        else
                __down(sem);
        raw_spin_unlock_irqrestore(&sem->lock, flags);
}
EXPORT_SYMBOL(down)
```

ممکن است تعریف متغیر flags را در ابتدای تابع down ببینیم. این متغیر به ماکروهای raw_spin_lock_i qsave و raw_spin_lock_i qrestore ارسال می‌شود که در فایل هدر include/linux/spi lock.h تعریف شده‌اند و از یک شمارنده از سمافور داده شده در اینجا محافظت می‌کنند. در واقع هر دوی این ماکروها همان کار ماکروهای spin_lock و spin_unlock را انجام می‌دهند، اما علاوه بر این، ارزش فعلی پرچم‌های وقفه را ذخیره/بازیابی می‌کنند و وقفه‌ها را غیرفعال می‌کنند. همانطور که قبلاً ممکن است حدس بزنید، کار اصلی بین ماکروهای raw_spin_lock_i qsave و raw_spin_unloc _irqrestore در تابع down انجام می‌شود. مقدار شمارنده سمافور را با صفر مقایسه می کنیم و اگر بزرگتر از صفر باشد ممکن است این شمارنده را کاهش دهیم. این بدان معنی است که ما قبلاً قفل را بدست آورده ایم. به عبارت دیگر شمارنده صفر است. این بدان معنی است که تمام منابع موجود قبلاً تمام شده است و ما باید منتظر بمانیم تا

این قفل را بدست آوریم. همانطور که می بینیم، تابع __down در این حالت فراخوانی می شود. تابع __down در همان فایل کد منبع تعریف شده است و پیاده سازی آن به صورت زیر است

```
static noinline void __sched __down(struct semaphore *sem)
{
        __down_common(sem, TASK_UNINTERRUPTIBLE,
MAX_SCHEDULE_TIMEOUT);
}
```

تابع __down فقط تابع __down_comm n را با سه پارامتر فراخوانی می کند:

سمافور;

پرچم - برای کار؛

مهلت - حداکثر زمان انتظار برای سمافور.

قبل از اینکه اجرای تابع __down_comm n را در نظر بگیریم، توجه داشته باشید که اجرای توابع down_trylock، down_timeout و down_killable بر اساس __down_common نیز هستند:

```
static noinline int __sched __down_interruptible(struct semaphore
*sem)
{
        return __down_common(sem, TASK_INTERRUPTIBLE,
MAX_SCHEDULE_TIMEOUT);
}
```

The __down_killable

```
static noinline int __sched __down_killable(struct semaphore *sem)
{
        return __down_common(sem, TASK_KILLABLE,
MAX_SCHEDULE_TIMEOUT);
}
```

And the __down_timeout:

```
static noinline int __sched __down_timeout(struct semaphore *sem,
long timeout)
{
        return __down_common(sem, TASK_UNINTERRUPTIBLE, timeout);
}
```

Mutex (Mutual EXclusion)

ساختاری که اطلاعاتی در مورد وضعیت یک قفل و لیست خدمت دهندگان قفل را در خود دارد. بسته به مقدار فیلد شمارش، یک سمافور می‌تواند دسترسی به منبعی با بیش از یک مورد از این منبع را فراهم کند. مفهوم mutex بسیار شبیه به مفهوم سمافور است. اما تفاوت هایی دارد. تفاوت اصلی بین semaphore و mutex synchronization این است که mutex معنایی دقیق تری دارد. برخلاف سمافور، فقط یک فرآیند ممکن است در یک زمان mutex را نگه دارد و فقط صاحب یک mutex می‌تواند آن را آزاد یا باز کند. تفاوت اضافی در اجرای قفل API است.. اجرای API قفل mutex اجازه می دهد از سوئیچ های زمینه گران قیمت جلوگیری شود.

همگام سازی mutex با موارد زیر نشان داده می شود:

```
struct mutex {
        atomic_t                count;
        spinlock_t              wait_lock;
        struct list_head        wait_list;
#if defined(CONFIG_DEBUG_MUTEXES) ||
defined(CONFIG_MUTEX_SPIN_ON_OWNER)
        struct task_struct      *owner;
#endif
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
        struct optimistic_spin_queue osq;
#endif
#ifdef CONFIG_DEBUG_MUTEXES
        void                    *magic;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
        struct lockdep_map      dep_map;
#endif
};
```

این ساختار در فایل هدر include/linux/mu ex.h تعریف شده است و شامل مجموعه فیلدهای ساختار سمافور مشابه است. اولین فیلد ساختار mutex - count است. مقدار این فیلد نشان دهنده وضعیت یک mutex است. در حالتی که مقدار فیلد شمارش 1 باشد، یک mutex در حالت قفل باز است. وقتی مقدار فیلد شمارش صفر است، یک mutex در حالت قفل است. علاوه بر این، مقدار فیلد شمارش ممکن است منفی باشد. در این حالت یک mutex در حالت قفل است و دارای پیشخدمت های احتمالی است.

دو فیلد بعدی ساختار mutex - wait_lock و wait_list برای محافظت از یک صف انتظار و لیست منتظران هستند که این صف انتظار را برای یک قفل خاص نشان می دهد. همانطور که متوجه شدید، شباهت ساختارهای mutex و سمافور به پایان می رسد. فیلدهای باقی مانده از ساختار mutex، به گزینه های پیکربندی مختلف هسته لینوکس بستگی دارد.

اولین فیلد - مالک فرآیندی را نشان می دهد که یک قفل به دست آورده است. وجود این فیلد در ساختار mutex به گزینه های پیکربندی هسته CONFIG_DEB G_MUTEXES یا CONFIG_MUT X_SPIN_ON_ WNER بستگی دارد. دو فیلد آخر - magic و dep_map فقط در حالت اشکال زدایی استفاده می شوند.

اکنون، پس از در نظر گرفتن ساختار mutex، ممکن است نحوه عملکرد این همگام سازی اولیه در هسته لینوکس را در نظر بگیریم. فرآیندی که می خواهد یک قفل را بدست آورد، در صورت امکان باید مقدار count<-mutex را کاهش دهد. و اگر فرآیندی بخواهد یک قفل را آزاد کند، باید همان مقدار را افزایش دهد. در واقع، هنگامی که یک فرآیند سعی می کند یک mutex به دست آورد، سه مسیر ممکن وجود دارد:fastpath/midpath/slowpath

که ممکن است بسته به وضعیت فعلی mutexگرفته شود. همانطور که ممکن است از نام آن متوجه شوید، اولین مسیر یا fastpath سریعترین است. همه چیز در این مورد آسان است. هیچ کس یک mutex به دست نیاورد، بنابراین مقدار فیلد شمارش ساختار mutex ممکن است مستقیماً کاهش یابد. در مورد باز کردن قفل mutex، الگوریتم یکسان است. یک فرآیند فقط مقدار فیلد شمارش ساختار mutex را افزایش می دهد. البته، همه این عملیات باید اتمی باشد. چه اتفاقی می‌افتد اگر فرآیندی بخواهد یک mutex را که قبلاً توسط فرآیند دیگری بدست آمده است به دست آورد؟ در این صورت کنترل به مسیر دوم - midpath منتقل می شود. چرخش میانی سعی می‌کند با قفل MCS ، در حالی که صاحب قفل در حال اجرا است را بچرخاند. این مسیر تنها در صورتی اجرا می شود که هیچ فرآیند دیگری آماده اجرا نباشد که دارای اولویت بالاتر باشد. این مسیر خوش بینانه نامیده می شود. این کار اجازه می دهد تا از سوئیچ زمینه گران قیمت جلوگیری کنید. در آخرین مورد، زمانی که ممکن است مسیر سریع و میانی اجرا نشود، آخرین مسیر - مسیر کند اجرا خواهد شد. این مسیر مانند یک قفل سمافور عمل می کند. اگر نتوان قفل را توسط یک فرآیند بدست آورد، این فرآیند به صف انتظار اضافه می شود که به صورت زیر نشان داده می شود:

```
struct mutex_waiter {
        struct list_head        list;
        struct task_struct      *task;
#ifdef CONFIG_DEBUG_MUTEXES
        void                    *magic;
#endif
};
```

ممکن است متوجه شوید که ساختار mutex_waiter شبیه ساختار semaphore_waiter از فایل کد منبع kernel/locking/semaphore.c است:

```
struct semaphore_waiter {
        struct list_head list;
        struct task_struct *task;
        bool up;
};
```

سمافور خواننده نویسنده:

در واقع دو نوع عملیات ممکن است روی داده ها انجام شود. ممکن است داده ها را بخوانیم و تغییراتی در داده ها ایجاد کنیم. دو عملیات اساسی - خواندن و نوشتن. معمولا (اما نه همیشه)، عملیات خواندن بیشتر از عملیات نوشتن انجام می شود. در این حالت، منطقی است که داده ها را به گونه ای قفل کنیم که برخی از فرآیندها ممکن است داده های قفل شده را در یک زمان بخوانند، به شرطی که هیچ کس داده ها را تغییر ندهد. قفل خواننده/نویسنده به ما امکان می دهد این قفل را دریافت کنیم.

وقتی فرآیندی که می‌خواهد چیزی را در داده بنویسد، تمام فرآیندهای نویسنده و خواننده دیگر مسدود می‌شوند تا زمانی که فرآیندی که قفل شده است، آن را آزاد نکند. هنگامی که یک فرآیند داده ها را می خواند، سایر فرآیندهایی که می خواهند همان داده ها را نیز بخوانند، قفل نخواهند شد و می توانند این کار را انجام دهند. اجرای سمافور خواننده/نویسنده بر اساس اجرای سمافور معمولی است. ببینیم چگونه سمافور خواننده/نویسنده در هسته لینوکس نمایش داده می شود.

```
struct semaphore {
    raw_spinlock_t        lock;
    unsigned int        count;
    struct list_head    wait_list;
};
```

اگر به فایل هدر include/linux/rwsem.h نگاه کنید، تعریف ساختار rw_semaphore را خواهید دید که نشان دهنده سمافور خواننده/نویسنده در هسته لینوکس است. بیایید به تعریف این ساختار نگاه کنیم:

```
#ifdef CONFIG_RWSEM_GENERIC_SPINLOCK
#include <linux/rwsem-spinlock.h>
#else
struct rw_semaphore {
        long count;
        struct list_head wait_list;
        raw_spinlock_t wait_lock;
#ifdef CONFIG_RWSEM_SPIN_ON_OWNER
        struct optimistic_spin_queue osq;
        struct task_struct *owner;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
        struct lockdep_map      dep_map;
#endif
};
```

اگر نگاهی به تعریف ساختار rw_semaphore بیندازیم، متوجه خواهیم شد که سه فیلد اول مانند ساختار semaphore است. این شامل فیلد شمارش است که مقدار منابع موجود را نشان می‌دهد، فیلد فهرست انتظار که فهرست پیوندی دوگانه از فرآیندهایی را نشان می‌دهد که در انتظار به‌دست‌آوردن قفل قفل و wait_lock spinlock برای محافظت از این لیست هستند. علاوه بر فیلد شمارش، همه این فیلدها مشابه فیلدهای ساختار سمافور هستند. سه فیلد آخر به دو گزینه پیکربندی هسته لینوکس بستگی دارد: CONFIG_RWS M_SPIN_ON_ WNER و CONFIG_DEB G_LOCK_ALL C. ممکن است دو فیلد اول با اعلام ساختار mutex از قسمت قبلی برای ما آشنا باشند. اولین فیلد osq نشان دهنده چرخنده قفل MCS برای چرخش خوش بینانه و دومی نشان دهنده فرآیندی است که صاحب فعلی قفل است. آخرین فیلد ساختار rw_semaphore مربوط به -debugging- dep_map است.

:Seqlock

می دانیم که قفل نویسنده خواننده یک مکانیسم قفل ویژه است که امکان دسترسی همزمان برای عملیات فقط خواندن را فراهم می کند، اما یک قفل انحصاری برای نوشتن یا اصلاح داده ها مورد نیاز است. این روش ممکن است منجر به مشکلی شود که به آن گرسنگی نویسنده می گویند. به عبارت دیگر، یک فرآیند نویسنده نمی‌تواند قفلی را تا زمانی که

حداقل یک فرآیند خواننده که قفل را دریافت کرده است، به دست آورد. بنابراین، در شرایطی که مشاجره زیاد است، منجر به شرایطی می‌شود که فرآیند نویسنده‌ای که می‌خواهد قفلی به دست آورد، مدت زیادی منتظر آن باشد. همگام سازی seqlock اولیه می تواند به حل این مشکل کمک کند

نکته اصلی این همگام سازی اولیه، ارائه دسترسی سریع و بدون قفل به منابع مشترک است. از آنجایی که قلب اولیه همگام سازی قفل متوالی، همگام سازی spinlock اولیه است، قفل های متوالی در شرایطی کار می کنند که منابع محافظت شده کوچک و ساده هستند. علاوه بر این، دسترسی به نوشتن باید نادر باشد و همچنین باید سریع باشد. کار این همگام سازی اولیه بر اساس توالی شمارنده رویدادها است. در واقع یک قفل متوالی امکان دسترسی رایگان به یک منبع را برای خوانندگان فراهم می کند، اما هر خواننده باید وجود تضاد با نویسنده را بررسی کند. این همگام سازی اولیه شمارنده خاصی را معرفی می کند. الگوریتم اصلی کار قفل های متوالی ساده است: هر نویسنده ای که یک قفل متوالی به دست آورد این شمارنده را افزایش می دهد و علاوه بر آن یک قفل چرخشی نیز بدست می آورد. هنگامی که این رایتر به پایان می رسد، اسپینلاک بدست آمده را آزاد می کند تا به سایر رایترها دسترسی داشته باشد و شمارنده یک قفل متوالی را دوباره افزایش دهد. دسترسی فقط خواندنی بر اساس اصل زیر کار می کند، قبل از ورود به بخش بحرانی، مقدار یک قفل شمار متوالی را دریافت می کند و آن را با مقدار آن مقایسه می کند.

(همان شمارنده قفل متوالی در خروجی بخش بحرانی) اگر ارزش آنها برابر باشد، به این معنی است که برای این دوره نویسندگانی وجود نداشته است. اگر مقادیر آنها برابر نباشد، به این معنی است که نویسنده شمارنده را در طول بخش انتقادی افزایش داده است. این تضاد به این معنی است که خواندن داده های محافظت شده باید تکرار شود.

```
unsigned int seq_counter_value;

do {
    seq_counter_value = get_seq_counter_val(&the_lock);
    //
    // do as we want here
    //
} while (__retry__);
```