به نام خدا

گزارش کار
پروژه سوم آزمایشگاه

اعضای گروه:
۱ـ محمدرضا یزدانی فر ۸۱۰۱۹۴۵۵۱
۲ـ ملیکه احقاقی ۸۱۰۱۹۴۲۵۴
۳ـ سیدعلی طباطبایی ۸۱۰۱۹۴۴۶۲

بخش اول

۱.

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

};
```

- ✓ lock: این متغیر از نوع raw_spinlock_t است. این متغیر تضمینی جهت atomic بودن دستورهای semaphore هست. در واقع این متغیر هر بار می تواند در اختیار یک پردازه باشد و دیگر پردازه ها توان تغییر مقادیر semaphore و ورود به ناحیه ی بحرانی را نخواهند داشت.
- ✓ count: نشان دهنده ی تعداد پردازه هایی است که در هر لحظه می توانند وارد ناحیه ی انحصاری شوند. (به تعبیر دیگر متغیر lock را در اختیار داشته باشند.)
- ✓ wait_list: این شیء از نوع list_head است و درواقع یک لیست پیوندی از PCB های پردازه هایی است که خواستار ورود به ناحیه ی بحرانی هستند.

۲.

```
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);
}

void up(struct semaphore *sem) {
        unsigned long flags;
        raw_spin_lock_irqsave(&sem->lock, flags);
        if (likely(list_empty(&sem->wait_list)))
                sem->count++;
        else
                __up(sem);
        raw_spin_unlock_irqrestore(&sem->lock, flags);

}
```

Down

این تابع مسءولیت به دست آوردن سمافور را دارد و در صورتی که نتواند سمافور را به دست آورد (در صورتی که مقدار count صفر باشد) پردازه موقتا بلاک می شود(sleep) و در این حالت در wait_list قرار می گیرد. در غیر این صورت ناحیه ی بحرانی را در دست خواهد داشت و در نتیجه مقدار count را یکی کم می نماید.

Up

این تابع عملکردی خلاف down دارد. هنگامی که یک پردازه مایل به خروج از ناحیه ی بحرانی باشد در جهت آزاد کردن semaphore این تابع را فراخوانی می نماید. این تابع با فراخوانی list_empty() چک می کند که آیا پردازه ای داخل لیست انتظار مایل به دریافت سمافور هست یا خیر. در صورت خالی نبودن لیست آن پردازه را از حالت انتظار به حالت آماده می برد تا در صورت امکان cpu به آن اختصاص داده شود. در غیر این صورت مقدار count را یکی افزایش می دهد تا در صورت درخواست ورود به ناحیه ی بحرانی از سمت پردازه ای دیگر امکان ورود به آن داده شود.

.۳

```
struct mutex {
        /* 1: unlocked, 0: locked, negative: locked, possible waiters */
        atomic_t            count;
        spinlock_t                  wait_lock;
        struct list_head    wait_list;
#if defined(CONFIG_DEBUG_MUTEXES) || defined(CONFIG_SMP)
        struct task_struct  *owner;
#endif
#ifdef CONFIG_DEBUG_MUTEXES
        const char                  *name;
        void                        *magic;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
        struct lockdep_map          dep_map;
#endif
};
```

فیلد count حاوی حالت mutex است. مقدار یک نشان می دهد که در دسترس است، صفر یعنی قفل شده است و مقدار منفی بدین معنی است که قفل شده و فرآیندها ممکن است منتظر باشند.

نوع داده ی این عضو atomic_t هست که با کمک آن متغیرهایی atomic خواهیم داشت. کاربرد این گونه زمانی است که مایل نیستیم سربار spinlock ویا semaphore را برای یک سری دستور متحمل شویم. در یک سیستم multiprocessor فرض کنید که داده بین پردازنده های share باشد و یک برنامه روی چند پردازنده اجرا شود.به روز رسانی مقدار یک داده به عملکرد پردازنده ی دیگر وابسته خواهد بود. مثالی را در نظر بگیر که مقدار اولیه ی val برابر با ۳ باشد دو پردازنده ی A و B دستور increment را روی این داده انجام می دهند. انتظار می رود که بعد از پایان مقدار نهایی برابر ۵ شود در حالی که اگر اتفاقات زیر رخ دهد:

-Processor A reads the value 3 and lets go of the bus
-While processor A updates the *val* processor B also reads *val* as 3 and lets the go of the bus
-Now processor A updates the new value 4 in the *val*
-Then processor B also updates the value of *val* as 4.

در این حالت مقدار نهایی ۴ خواهد شد. این مساله ناشی از دسترسی هم زمان دو پردازنده به یک داده ی مشترک است در حالی که کار دیگری با آن تمام نشده است. راه حلی که برای این مشکل مطرح می شود استفاده از atomic variable هاست. این داده های به این صورت هستند که read-modify-write یک دستور محسوب می شوند و تا پایان آن ها interrupt قابل پذیرش نخواهد

بود. دسترسی به چنین داده هایی با دستورات استاندارد امکان پذیر نیست و از دستوراتی مانند آن چه در ادامه آمده است استفاده می شود:

*atomic_t *val*; Declaration
atomic_read(val); Returns the value of *val
atomic_set(val,i); Sets *val to i
atomic_add(i,val); adds i to *val
.
.
.

.۴

int mutex_init(mutex_t *mp, int type, void * arg);

mutex_init

این تابع به صورت داینامیک mutex را initialize می نماید.mutex ها یا intra-process هستند یا inter-process بسته به این که چه آرگمان هایی به صورت مستقیم یا غیر مستقیم پاس شود. یک mutex استاتیک نیازی به initialize به صورت مستقیم نخواهد داشت.(به صورت پیش فرض در این حالت تمام مقادیر صفر در نظر گرفته می شود و scope آن در بدنه ی پردازه ای خواهد بود که آن را فراخوانی می کند)

برای همگام سازی میان پردازه ای می بایست یک mutex در shared memory اختصاص داده شود. از آن جا که حافظه برای چنین mutex ای بایستی به صورت داینامیک اختصاص داده شود mutex بایستی به صورت مستقیم initialize شود.(به کمک تابع ()mutex_init).
این تابع mutex را که با reference ای به نام mp داده شده است با نوع داده ی اختصاص داده شده initialize می نماید. در یک initialization موفق state آن initialized و unlock می شود. تنها type ای به نام LOCK_PRIO_PROTECT از arg استفاده می نماید. Type بایستی از لیست زیر انتخاب شود:

USYNC_THREAD
USYNC_PROCESS

arg نیز می تواند inclusive-or از یک سری flag باشد.

mutex_lock(struct mutex *mp)

فراخوانی این تابع mutex object ای را که توسط mp به آن اشاره می شود را قفل می کند. اگر mutex از پیش قفل شده باشد thread ای که آن را فراخوانی می کند تا زمانی که mutex آزاد شود قفل می شود. بازگشت این تابع زمانی است که یک mutex که اشاره گر به آن در حالت locked خواهد بود به این صورت که thread ای که آن را فراخوانی می کند owner آن خواهد بود. اگر owner فعلی تلاش بر relock کردن mutex کند deadlock رخ خواهد داد.

mutex_unlock(struct mutex *mp)

این تابع توسط mutex owner فراخوانی می شود تا آن را آزاد کند. Mutex باید locked باشد و threadای که آن را فراخوانی میکند باید آخرین کسی باشد که mutex را قفل می نماید(owner). اگر thread هایی وجود داشته باشند که روی آن mutex قفل شده باشند وقتی که این تابع فراخوانی می شود mp آزاد می شود و scheduling policy تصمیم میگیرد که چه thread ای mutex را بگیرد. اگر thread ای که این تابع را فراخوانی می کند owner ای برای lock نباشد هیچ خطایی برگردانده نمی شود و رفتار برنامه تعریف نشده خواهد بود.

mutex_trylock(struct mutex *mp)

این تابع مشابه mutex_lock است با این تفاوت که اگر mutex object ای را که توسط mp به آن اشاره می شود قفل شود (توسط هر thread ای شامل thread فعلی). این تابع سریعا با یک error بازمی گردد.

۵. یک سری تفاوت های کلیدی بین mutex و semaphore وجود دارد که به آن ها اشاره می نماییم:

- سمافور یک مکانیسم signaling هست و از wait() و signal() استفاده می کند که نشان می دهد که یک منبع در اختیار یک پردازه قرار گرفته یا آزاد شده است. از طرفی mutex یک مکانیسمlocking هست که برای در دست داشتن یک منبع بایستی که mutex object قفل شود و برای آزاد کردن آن قفل mutex را باز می نماییم.
- سمافور معمولا یک متغیر از نوع int هست. در حالی که mutex از جنس object هست.
- سمافور اجازه می دهد که چند thread از برنامه به تعداد محدودی از منابع دسترسی پیدا کنند. در حالی که mutex اجازه می دهد چندین thread به یک single shared resource در یک لحظه دسترسی پیدا کنند.
- مقدار semaphore variable می تواند توسط هر پردازه ای که منبع را در دست دارد یا آزاد می کند با انجام wait() و signal() تغییر کند. از طرفی lock ای که روی mutex object به دست می آید تنها می تواند توسط پردازه ای که آن را به دست آورده است آزاد شود.
- سمافور های دو نوع هستند : counting و binary که نوع دوم تقریبا مشابه mutex عمل می کند.
- وقتی تمامی منابع در دست پردازه های دیگر باشد هیچ منبعی آزاد نخواهد بود و اگر پردازه ی جدیدی متقاضی منبع باشد wait() را صدا می زند و خودش را بلاک می کند تا وقتی که count بیشتر از صفر شود.اما اگر mutex object قفل باشد پردازه ای که متقاضی دریافت آن هست منتظر می ماند و توسط سیستم در صف قرار می گیرد تا وقتی که منبع آزاد شود و mutex object قفلش باز شود.

بخش دوم

ساختمان داده mysync را به صورت زیر تعریف میکنیم.

```
struct mysync {
  int event_id;
  wait_queue_head_t wait_queue;
  int go_aheads;
};
```

event_id شناسه هر event را مشخص میکند. wait_queue صفی است که پردازه های مربوط به آن event در آن در انتظار به سر میبرند. go_aheads به عنوان یک متغییر مشترک برای همگام سازی بکار میرود. به این ترتیب به هر پردازه از لحظه ای که بر روی event در حال صبر کردن است منتظر تغییر این مقدار میماندو و در صورت سیگنال شدن event یک واحد به این مقدار افزوده میشود.

متغیر های جهانی زیر را تعریف کرده ایم:

id_event_map : متغییری است از جنس idr که مسئولیت آن ایجاد تناظر بین event_id ها و event هاست.

mysync_lock : متغییری از جنس RW_LOCK است که رقابت بر سر نوشتن و خواندن از لیست event ها و تغییر مقدار event ها را کنترل میکند.

mysync_initialized : وضعیت initialize شدن ماژول را نشان میدهد.

- **ساخت رویداد** (sys_mysync_make_event):

به سادگی یک mysync را kmalloc میکنیم و بوسیله تابع idr_alloc آن event را به id_event_map اضافه میکنیم. تابع idr_alloc با دریافت حد بالا و پایین برای id و اشاره گر از نوع دلخواه اقدامات لازم در ساختمان داده را جهت تخصیص یک id جدید انجام میدهد. مقدار خروجی این تابع در صورتی که در بازه ذکر شده id باقی نمانده باشد ENOSPC ، در صورت پر بودن ساختمان داده ENOMEM و در صورت موفقیت برابر با id تخصیص داده شده است. دقت شود که برای استفاده از idr با گرفتن قفل ورود مشکل ورود به ناحیه بحرانی را حل کردیم.

- **تخریب یک رویداد** (sys_mysync_destroy_event):

با در اختیار گرفتن قفل و با استفاده از تابع idr_find اقدام به پیدا کردن event میکنیم. در صورت معتبر بودن شناسه رویداد همه پردازه هایی که برروی آن رویداد لنتظار میکشند را signal میکنیم سپس بار دیگر با در اختیار گرفتن قفل آن رویداد را از map بوسیله تابع idr_remove حذف میکنیم.

- **انتظار برروی یک رویداد**(sys_mysync_wait_event):
در ابتدا ورود به این تابع مقدار go_aheads را میخوانیم و تا وقتی مقدار go_aheads یتغییری نکرده انتظار میکشیم. تابع wait_event_interruptible پردازه جاری تا زمانی که شرط آرگومان دومش صحیح نشده باشد در حالت انتظار میبرد. این تابع همچنین آن پردازه را به صفی از جنس wait_queue که به آرگومان اولش داده میشود میبرد.
- **سیگنال کردن یک رویداد**(sys_mysync_sig_event): این تابع با افزودن مقدار go_aheads شرط آرگومان دوم تابع wait_event_interruptible درست میکند. همچنین تابع wake_up_interruptible پردازه های داخل صف را از حالت انتظار خارج میکند. در نهایت سایز wait_queue را باز میگردانیم.

معرفی ساختمان داده IDR :

بطور کلی یک map سه قابلیت افزودن، حذف و پیدا کردن را دارد. هر چند جداول در هم سازی نوعی از map ها هستند اما همه map ها بوسیله جداول درهم سازی پیاده سازی نمیشوند. از درخت های خود متوازن شونده نیز برای پیاده سازی map ها استفاده میشوند. هرچند جداول درهم سازی هزینه زمانی میانگین کمتری نسبت به درخت جستجوی دو دویی خود متوازن شونده دارند اما هزینه بدترین حالت در درخت های دودویی خود متوازن شونده کمتر است. درخت های دودویی خود متوازن شونده در بدترین حالت هم هزینه لگاریتمی دارند در حالیکه هزینه جداول درهم سازی در بدترین حالت خطی است. علاوه بر این درخت های دودویی این امکان را فراهم میکنند تا به شکل مرتب شده بر اساس شناسه ها بین عناصر حرکت کنیم. همچنین درخت های جستجوی دودویی نیازی به تابع درهم سازی ندارند.

بخش سوم

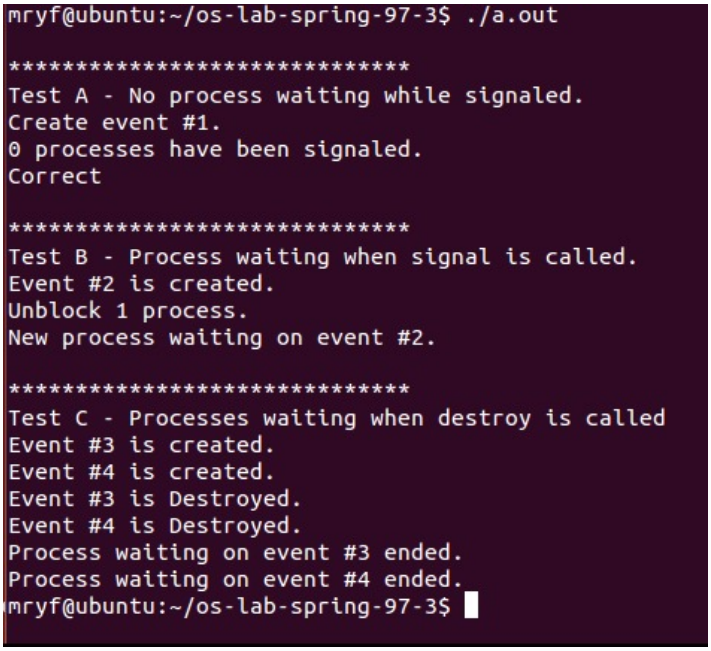
در این بخش سه سناریو برای تست کد سطح کرنل داده شده است. برای پیاده سازی این قسمت ما ۳ تابع برای تست ها در نظر میگیرم و به ترتیب آن ها را در main صدا میکنیم.(بین هر دو تست یک ثانیه وقفه میگزاریم تا تست ها تداخل نخورند)

A. برای تست اول ابتدا یک event ، با استفاده از فراخوانی سیستمی sys_mysync_make_event() میسازیم و پس از بررسی موفقیت آمیز بودن آن فراخوانی سیستمی sys_mysync_sig_event(eid) را صدا میزنیم و بررسی میکنیم که به صورت موفقیت آمیز صورت گرفته است یا خیر.

B. برای این تست ابتدا لازم است که یک event بسازیم سپس یک پردازه fork کنیم و در پردازه ی جدید sys_mysync_wait_event(eid) را صدا کنیم، پس از بررسی موفقیت آمیز بودن آن، کار پردازه ی جدید به پایان رسیده است. در پردازه ی والد یک ثانیه فرصت میدهیم تا مطمئن باشیم تا فرزند این پردازه کارش را انجام دهد پس از آن فراخوانی سیستمی سیگنال را صدا کرده و موفقیت بودن آن را بررسی میکنیم.

C. در تست آخر ابتدا دو event میسازیم پس از آن دو پردازه fork میکنیم و در پرازه های جدید فراخوانی سیستمی wait را صدا میکنیم. در پردازه ی والد به مانند تست قبلی یک ثانیه وقفه ایجاد میکنیم سپس هر دو event را با فراخوانی سیستمی destroy از بین میبریم.

نتایج به دست آمده از تست به این صورت است:


mryf@ubuntu:~/os-lab-spring-97-3$ ./a.out
****
Test A - No process waiting while signaled.
Create event #1.
0 processes have been signaled.
Correct
****
Test B - Process waiting when signal is called.
Event #2 is created.
Unblock 1 process.
New process waiting on event #2.
****
Test C - Processes waiting when destroy is called
Event #3 is created.
Event #4 is created.
Event #3 is Destroyed.
Event #4 is Destroyed.
Process waiting on event #3 ended.
Process waiting on event #4 ended.
mryf@ubuntu:~/os-lab-spring-97-3$