به نام خدا

**گزارش کار**

**پروژه سوم آزمایشگاه**

**اعضای گروه:**‌

۱- محمدرضا یزدانی فر ۸۱۰۱۹۴۵۵۱

۲- ملیکه احقاقی ۸۱۰۱۹۴۲۵۴

۳- 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](https://elixir.bootlin.com/linux/v3.1/ident/mutex) {

*/\* 1: unlocked, 0: locked, negative: locked, possible waiters \*/*

[atomic\_t](https://elixir.bootlin.com/linux/v3.1/ident/atomic_t) count;

[spinlock\_t](https://elixir.bootlin.com/linux/v3.1/ident/spinlock_t) [wait\_lock](https://elixir.bootlin.com/linux/v3.1/ident/wait_lock);

struct [list\_head](https://elixir.bootlin.com/linux/v3.1/ident/list_head) wait\_list;

#if [defined](https://elixir.bootlin.com/linux/v3.1/ident/defined)(CONFIG\_DEBUG\_MUTEXES) || [defined](https://elixir.bootlin.com/linux/v3.1/ident/defined)([CONFIG\_SMP](https://elixir.bootlin.com/linux/v3.1/ident/CONFIG_SMP))

struct [task\_struct](https://elixir.bootlin.com/linux/v3.1/ident/task_struct) \*[owner](https://elixir.bootlin.com/linux/v3.1/ident/owner);

#endif

#ifdef CONFIG\_DEBUG\_MUTEXES

const char \*name;

void \*[magic](https://elixir.bootlin.com/linux/v3.1/ident/magic);

#endif

#ifdef [CONFIG\_DEBUG\_LOCK\_ALLOC](https://elixir.bootlin.com/linux/v3.1/ident/CONFIG_DEBUG_LOCK_ALLOC)

struct [lockdep\_map](https://elixir.bootlin.com/linux/v3.1/ident/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 کهread-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 تواندinclusive-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 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.

\*\* https://lwn.net/Articles/167034/

\*\*http://tuxthink.blogspot.com/2011/10/atomic-variables.html

\*\*https://www.unix.com/man-page/all/3C/mutex\_init/

**بخش دوم**

ساختمان داده mysync را به صورت زیر تعریف میکنیم.



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)

1. 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.
2. 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.
3. 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:

