علی معز غلامی ، سینا کاشی پزها ، آرش شفائی اردکانی شماره های دانشجویی به ترتیب : (378 ، 355 ، 307) 810191

آشنایی با پیاده سازی سمافور در لینوکس

1- ساختار داده اصلی سمافور با نام semaphore در فایل semaphore.h تعریف شده است. به طور کامل توضیح دهید که هر یک از سه عضو این ساختار چه کاربردی دارند ؟

ساختار سمافور شامل سه عضو count ، wait_list و lock است. در زیر توضیح این سه عضو خواهد آمد

1- count

همان طور که می دانید سمافور یک مقدار دارد (در پیاده سازی ساختارداده سمافور در لینوکس نام این مقدار count است) که برابر با تعداد پردازه هاییست که در آن لحظه می توانند وارد ناحیه ی بحرانی شوند (برای mutex این مقدار 1 است). وقتی برای سمافور wait می شود باید count یک عدد کم شود و با هر سیگنال یکی به count اضافه شود. مخلص کلام این که تعداد پردازه هایی که در آن لحظه می توانند وارد ناحیه ی بحرانی شوند در count نگه داری می شود.

توجّه: count تعریف شده از نوع عدد صحیح بدون علامت است، که این به این معناست که تعداد پردازه های موجود در صف نامعلوم است و با هر سیگنال کردن از سر صف انتظار یک پردازه بیدار می شود.

2- wait_list

بدیهیست که باید لیستی از پردازه های در حال انتظار برای ورود به ناحیه بحرانی وجود داشته باشد. این صف تحت عنوان wait_list به عنوان یک جزء از ساختار داده سمافور است.

3- lock

برای مدیریت نواحی بحرانی up و down از آن استفاده می شود.

2- توابع و ماکروهایی که برای مقدار دهی اولیه به سمافور تعریف شده اند را نام برده و به طور مختصر شرح دهید.

- تابع __SEMAPHORE_INITIALIZER

در این تابع ابتدا یک spinlock که در ابتدا قفل نیست را initialize می کند. سپس از آرگومان n استفاده می کند تا count را مقدار دهی کند. در خط بعد ماکرو LIST_HEAD_INIT یک head را مقداردهی می کند و wait_list را به آن نسبت می دهد.
این تابع فقط در init_MUTEX و init_MUTEX_LOCKED استفاده شده است.

- sema_init

این تابع به صورت زیر تعریف شده است:

```
1-  static inline void sema_init ( struct semaphore * sem , int val )
2-  {
3-          static struct lock_class_key __key;
4-          *sem = ( struct semaphore ) __SEMAPHORE_INITIALIZER ( *sem , val );
5-          lockdep_init_map( & sem -> lock.dep_map , “semaphore->lock” , & __key , 0 );
6-  }
```

و برای مقدار دهی اولیه ی سمافور استفاده می شود. در خط 3 __key از جنس lock_class_key تعریف می شود. __key به عنوان آرگومان به تابع lockdep_init_map پاس داده می شود. در این تابع مقدار lock->key با __key مقدار دهی می شود. در خط 4 مقدار sem توسط تابع __SEMAPHORE_INITIALIZER مقدار دهی اولیه می شود. در خط بعد تابع lockdep_init_map برای مقدار دهی متغیر اولیه استفاده می کند. برای مثال همان طور که در بالا گفته شد از __key استفاده می کند تا lock->key را مقدار دهی کند.

3- به پیاده سازی توابع ()up و ()down در فایل semaphore.c مراجعه کنید. توضیح دهید که برای حفاظت از ناحیه ی بحرانی در این دو تابع چگونه عمل شده است.

همانطور که می دانید صحت عمل سمافور منوط به حفاظت توابع up و down از حضورشان به طور همزمان در نواحی بحرانی است. این مساله به این معناست که باید مساله ناحیه بحرانی را برای این دو تابع نیز حل کرد. برای این کار از روش ابتدائی ترbusy waiting استفاده می شود. عیب اصلی این روش در حالت کلّی این است که برای یک ناحیه بحرانی با اندازه نامشخص باعث افت کارایی سیستم می شود، ولی در این مورد خاص اندازه نواحی بحرانی (up و down) مشخص و کوچک است که باعث میشود استفاده از busy waiting باعث افت محسوس کارآیی نباشد. (گرچه به علت کوچک بودن این نواحی احتمال رخ دادن busy waiting کم است و در صورت بروز هم بیش از یک بار برای هر up یا down اتفاق نمی افتد.) این کار با قفل و باز کردن خصیصه lock که به ترتیب پذیرش وقفه ها را قطع و وصل میکنند انجام میشود.

توجه : توابع up و down_tryloc می توانند در زمان رسیدگی به وقفه انجام شوند در زمان گرفتن up و down وقفه-ها غیر فعال می شوند.

نکته ی مهم در باره ی سوال ۴و۵:

در پیاده سازی صف سمافور به جای داشتن صفی از اشاره گرها به پردازه ها صفی از یک ساختار داده به عنوان semaphore_waiter نگهداری می شود. یک عضو آن بدیهتا اشاره گر به پردازه است. عضو دیگر آن waiter.up است که پردازه ها از طریق آن می فهمند که نوبت به آن ها رسیده یا نه. یک عضو دیگر (که بعد از حدود یک ساعت مطالعه کد فهمیدم نقش آن چیست :O) هم هست که جنس آن لینک لیست (سر لینک لیست) است. این لیست در هیچ کجای کد نه مقداری به آن وارد می شود و نه از آن مقداری خارج می شود. این لیست صرفا attribute ای است که بتوان با آن لیستی از ساختار داده ی semaphore_waiter درست کرد. یعنی این attribute صرفا برای type-checking و نکات کار با list پیاده سازی شده در linux است!!!

4- به طور کامل توضیح دهید که تابع ()up در ناحیه ی بحرانی خود چه می کند؟

تابع up به طور کلی این کار را می کند (وقتی که در ناحیه ی بحرانی است.) :

اگر لیست انتظار سمافور خالی باشد (که احتمال این پیشامد زیاد در نظر گرفته شده) به سادگی مقدار count سمافور را یک واحد اضافه می کند و از تابع خارج می شود.

اما اگر صف انتظار خالی نباشد تابع up__ فراخوانی می شود. در این تابع پردازه ی اول صف سمافور برداشته و از صف پاک می شود. سپس flag روشن بودن waiter.up آن پردازه true می شود تا وقتی (when نه while) که دوباره بیدار شد و نوبت به او رسید از down خود خارج شود.

در ادامه با فراخوانی یک تابع از زمانبند لینوکس پردازه ی سر صف بیدار می شود تا از حالت block به ready بیاید و down خود را ادامه دهد تا از آن خارج شده و سمافور را در اختیار بگیرد.

5- به طور کامل توضیح دهید که تابع ()down در ناحیه ی بحرانی خود چه می کند؟

تابع down ورژن های مختلفی دارد. وقتی وارد تابع می شویم اگر count سمافور مثبت باشد (که این پیشآمد محتمل در نظر گرفته شده) به سادگی از count یکی کم کرده و سمافور را در اختیار می گیریم. اگر ورژن down_trylock باشد و count مثبت نباشد با نتیجه ی خطا از تابع خارج می شویم. در ورژن های دیگر وارد down_common می شویم که بسته به نوع ورژن پارامتر state آن فراخوانی را تعیین می کنیم.

پارامتر های ممکن برای state در زیر آمده اند:

- task_uninterruptible

دراین حالت که توسط تابع down عادی فراخوانی می شود پردازه خوابیده است تا وقتی که بیدار شود. دراین حالت به هیچ سیگنالی از جمله سیگنال های تقاضای بسته شدن برنامه رسیدگی نمی شود. این برنامه نویسی برنامه ی کاربر آن را آسان تر می کند اما دلیل استفاده ی کم از down هم همین است مثلا ما یک برنامه را می بندیم اما چون منتظر یک سمافور است بسته نمی شود!

- task_interruptible

در این حالت که توسط تابع down_interruptible فراخوانی می شود پردازه خوابیده است تا وقتی که بیدار شود یا یک سیگنال دریافت کند. همانطور که گفته شد این برنامه نویسی برای برنامه ی کاربر را سخت تر می کند چون برای down باید بفهمیم که به خاطر سیگنال از خواب بیدار شده ایم نه ورود به ناحیه ی بحرانی.

- task_killable

در این حالت که توسط تابع down_killable فراخوانی شده است مانند task_uninterruptible عمل می شود با این تفاوت که سیگنال های مربوط به تقاضای بسته شدن رسیدگی می شود. یعنی هم برنامه نویسی آسان است و هم وقتی یک برنامه را می بندیم منتظر سمافور نمی ماند.

در تابع down_common در ابتدا ، پردازه ی تقاضا کننده به انتهای لیست انتظار می رود و نیز مقدار waiter.up خود را false می کند (تا اگر بعدا کسی آن را true کرد بفهمد.)

حال وارد یک حلقه ی نامتناهی می شویم و در هر بار اجرای حلقه روند زیر را طی می کنیم:

اگر در حالت فعلی و با توجه به پارامتر state یک سیگنال دریافت شود خود را از صف برداشته و با پیغام خطای مربوط به دریافت سیگنال خارج می شویم.

در ادامه اگر زمان انتظار تمام شده باشد (در همه ی ورژن های حاضری (با پارامتر های آماده) down زمان انتظار بی نهایت است.) خود را از صف برداشته و خارج می شویم.

حال وضعیت پارامتر state به scheduler اطلاع داده می شود. در اینجا lock ای که برای حفاظت از نواحی بحرانی up و down بود را آزاد می کنیم و با توجه به state ای که داشتیم به خواب می رویم تا timeout تمام شود که timeout بی نهایت است. یعنی رسما block می شویم. پس حلقه ی بی نهایتی که گفته شد در اینجا متوقف می شود. حال وقتی از خواب در آمدیم ابتدا دوباره lock سمافر را می گیریم. اگر مقدار waiter.up ما true شده بود یعنی کسی ما را بیدار کرده و سمافر را در اختیار داریم. پس با موفقیت از تابع خارج می شویم. اما اگر waiter.up ما false باشد یک بار دیگر حلقه را تکرار می کنیم و به همین ترتیب.

توجه شود که در صورتی که در دو شرط ابتدای حلقه چه با شرط timeout و چه با شرط گرفتن سیگنال از تابع خارج شویم lock خود سمافر را آزاد نمی کنیم. یعنی در این حالت ها هیچ پردازه ی دیگر مربوط به این سمافر نمی تواند up و down خود را تمام کند.

جواب سوال پایانی

اگر مکانیزم همگام سازی به کل در محیط کاربر انجام شود ، مثل داشتن یک pc مجازی برای هر ریسمان و... می توان کل سیستم همگام سازی از جمله سیستم سمافور را در لایه ی کاربر انجام داد. در واقع کتابخانه multithreading مربوطه ریسمان ها را مدیریت می کند (برای مثال ریسمان های قبل انتقال گنو (GNU portable threads) کل مکانیزم همگام سازی را در لایه ی کاربری مدیریت کرده اند.)

اما اگر بخواهیم بخشی از همین سیستم را به لایه ی کاربر منتقل کنیم ، همه چیز را نمی توان به سطح کاربر برد. مثلا spin lock درون سمافور که وقفه های پردازنده را غیر فعال می کند به هیچ وجه نباید به لایه ی کاربر منتقل شود. اما می توان مدیریت صف انتظار و سیاست های وراثت اولویت و اولویت بندی را به لایه ی کاربر منتقل کرد) مثلا هنگام up کردن یک سمافور می توان پردازه ی بعدی را پیشنهاد داد .(اما از طرف دیگر وظایف و مسئولیت های مدیریت نیز به لایه کاربر منتقل شده.)