به نام خدا

گزارش پروژه چهارم آزمایشگاه سیستم عامل

پریا خوش‌تاب ۸۱۰۱۹۸۳۸۷ - علی اخگری ۸۱۰۱۹۸۳۴۱ - پرنیان فاضل ۸۱۰۱۹۸۵۱۶

همگام سازی در xv6

۱- علت غیرفعال کردن وقفه چیست؟ توابع ()pushcli و ()popcli به چه منظور استفاده شده و چه تفاوتی با cli و sti دارند؟

علت غیرفعال کردن وقفه ها این است که از deadlock جلوگیری شود، زیرا وقفه ها به صورت asynchronous عمل می کنند؛ یعنی هر زمان که یک وقفه تولید شود، کد های در حال اجرا متوقف می شوند و trap handler اجرا می شود و تا زمانی که کارش تمام نشود، کار دیگری صورت نمی گیرد. حال موقعیتی را فرض کنید که در یک ناحیه بحرانی هستیم و یک وقفه رخ دهد و سپس trap handler اجرا شود. همچنین در اجرای این وقفه نیاز داشته باشیم که به همان قفل دوباره دسترسی داشته باشیم، به این صورت برنامه تلاش می کند که به آن قفل دسترسی پیدا کند، اما چون آن قفل در اختیار برنامه دیگری می باشد، با deadlock مواجه می شویم.

توابع pushcli و popcli به ترتیب منظور غیرفعال کردن و دوباره فعال کردن این وقفه ها به کار می روند.(در توابع acquire و release فراخوانی شده اند.)

در این توابع، توابع cli و sti که در xv6 برای غیرفعال کردن و فعال کردن وقفه ها استفاده می شوند، فراخوانی شده اند. اما علت اینکه به صورت مستقیم از cli و sti استفاده نمی شود این است که این دو تابع قابلیت شمارش ندارند. به طور مثال اگر دو قفل مجزا، هر کدام درخواست غیرفعال کردن وقفه ها را بدهند، باید دو بار دستور آزادی وقفه ها داده(sti) صدا زده شود، اما با استفاده از توابع pushcli و popcli این مشکل حل می شود.

۲ـ مختصری راجع به تعامل میان پردازه ها توسط دو تابع مذکور توضیح دهید. چرا در مثال تولیدکننده/مصرف کننده استفاده از قفل های چرخشی ممکن نیست.

تابع aquiresleep یک آرگومان دارد که از جنس استراکت sleeplock است.در اینجا آدرس قفلی که داده شده به sleep میرود و تابع sleep روی آن صدا زده میشود. یک پردازنده روی آدرس قفل که به عنوان ورودی داده شده تا زمانی که قفل در حالت lock قرارداد لوپ میزند. در تابع sleep به طور اوتومات قفل آزاد میشود. در realeasesleep که در استراکت sleeplock قفل شده بود و به خواب رفته بود بیدار میشود و فیلدهای locked و pid آن برابر صفر قرار داده میشود و تابع wakeup رو آن ها صدا زده میشود. در این تابع همه پردازه‌هایی که در chan به حالت sleep رفته اند بیدار می‌شوند.

در مثال تولیدکننده/مصرف کننده که یک bounded buffer وجود دارد، مسائلی که وجود دارد یکی این است که تولیدکننده و مصرف‌کننده هر دو نباید به طور همزمان به بافر دسترسی داشته باشند یک مورد دیگر اینکه اگر بافر پر باشد نباید تولیدکننده چیزی رو آن بنویسد و مورد آخر اینکه مصرف کننده نباید وقتی که بافر خالی است چیزی ار بافر بخواند. حال اگر بخواهیم از قفل‌های چرخشی استفاده کنیم، ممکن است مصرف کننده هنگامی از بافر بخواند که بافر خالی باشد و یا تولید کننده هنگامی روی بافر بنویسد که بافر خالی نشده باشد و پر باشد. همچنین ممکن است مشکل گرسنگی ایجاد شود.

۳ـ حالات مختلف پردازه ها در xv6 را توضیح دهید. تابع ()sched چه وظیفه ای دارد؟

حالات مختلف پردازه در xv6:

- UNUSED: حالتی که پردازه قابل استفاده نمی باشد.
- EMBRYO: حالتی که پردازه به تازگی ایجاد شده است. در واقع زمانی که در تابع allocproc، اولین پردازه UNUSED پیدا شود، حالت این پردازه EMBRYO می شود، تا به عنوان یک پردازه USED مارک شود و به این پردازه جدید یک pid اختصاص داده شود.

- SLEEPING: حالتی که cpu در اختیار پردازه قرار ندارد، برای مثال زمانی که یک پردازه برای انجام یک عمل I/O متوقف می شود.
- RUNNABLE: حالتی که پردازه آماده اجرا می باشد و cpu می تواند با استفاده از scheduler در اختیار پردازه قرار بگیرد.
- RUNNING: حالتی که پردازه در حال اجرا می باشد و cpu در اختیار پردازه قرار دارد.
- ZOMBIE: زمانی که کار پردازه فرزند تمام شده باشد، اما پردازه پدر هنوز wait را صدا نکرده باشد، ، پردازه فرزند فورا نمیمیرد، بلکه اطلاعاتش همچنان در ptable وجود داشته و به حالت ZOMBIE می رود. زمانی که پردازه ی پدر حالت zombie را برای فرزندان خود ببیند شروع به پاک کردن آنها از سیستم عامل میکند که به آن عمل process reaped می گویند. اگر به هر دلیلی پدر ها نتوانند حالت zombie فرزندان را ببینند آن پردازه ها تا بینهایت در حافظه حضور دارند و اصطلاحا resource leak رخ میدهد.

در تابع yield وقتی که حالت یک پردازه به RUNNABLE تغییر پیدا میکند، تابع sched صدا زده میشود. در تابع sched ابتدا شروطی مثل lock بودن، running بودن و وجود داشتن امکان وقفه چک می شود و در صورتی که هیچ کدام از این شروط برقرار نباشند با استفاده از تابع swtch عمل switch context انجام می شود و پس از ذخیره کردن context فعلی به scheduler سوییچ می کنیم، پس scheduler جایگزین پردازه فعلی می شود.

۴ـ تغییری در توابع دسته دوم داده تا تنها پردازه صاحب قفل، قادر به آزادسازی آن باشد. قفل معادل در هسته لینوکس را به طور مختصر معرفی نمایید.

به این صورت می توانیم این کار را انجام دهیم که به struct sleeplock یک متغیر owner اضافه کنیم که مشخص شود که کدام پردازه قفل را ایجاد کرده است که فقط همان پردازه بتواند قفل را آزاد کند.

در لینوکس این قفل معادل mutex است، که در struct mutex یک متغیر به نام owner داریم که این کار را انجام می دهد. در mutex، اگر پردازه ای بخواهد به قفل دسترسی پیدا کند و قفل در دسترس نباشد، خود به خود حالت پردازه به SLEEPING تبدیل می شود و هر زمان mutex آزاد شد، دوباره به حالت آماده باش در می آید. در نتیجه مشکل spinlock حل می شود.

```
63  struct mutex {
64          atomic_long_t           owner;
65          raw_spinlock_t          wait_lock;
66  #ifdef CONFIG_MUTEX_SPIN_ON_OWNER
67          struct optimistic_spin_queue osq; /* Spinner MCS lock */
68  #endif
69          struct list_head        wait_list;
70  #ifdef CONFIG_DEBUG_MUTEXES
71          void                    *magic;
72  #endif
73  #ifdef CONFIG_DEBUG_LOCK_ALLOC
74          struct lockdep_map      dep_map;
75  #endif
76  };
77
```

۵- یکی از روش‌های افزایش کارایی در بارهای کاری چندریسه‌ای استفاده از حافظه تراکنشی بوده که در کتاب نیز به آن اشاره شده است. به عنوان مثال این فناوری در پردازنده‌های جدیدتر اینتل تحت عنوان افزونه‌های همگام‌سازی تراکنشی (TSX) پشتیبانی می‌شود. آن را مختصراً شرح داده و نقش حذف قفل را در آن بیان کنید؟

تراکنش حافظه دنباله ای از عملیات‌های read-write حافظه است که اتمی هستند. اگر تمام عملیات در یک تراکنش کامل شود، تراکنش حافظه کامل شده است. در غیر این صورت، عملیات باید متوقف شود و به عقب برگشت. مزیت استفاده از حذف قفل، این است که سیستم حافظه تراکنشی مسئول تضمین atomic بودن آن است. علاوه بر این، از آنجایی که هیچ قفلی در کار نیست، deadlock ممکن نیست. علاوه بر این، یک سیستم حافظه تراکنشی می تواند تشخیص دهد که کدام دستورات در بلوک های atomic را می‌توان همزمان اجرا کرد، مانند خواندن همزمان از یک متغیر مشترک. حافظه تراکنشی را می توان در نرم افزار یا سخت افزار پیاده سازی کرد. حافظه تراکنشی نرم افزار (STM)، همانطور که از نام آن پیداست، حافظه تراکنشی را منحصراً در نرم افزار پیاده سازی می کند. حافظه تراکنشی سخت‌افزار (HTM) از سلسله‌مراتب کش سخت‌افزار و پروتکل‌های انسجام حافظه پنهان برای مدیریت و حل تداخل‌های مربوط به داده‌های اشتراک‌گذاری شده در حافظه پنهان پردازنده‌های جداگانه استفاده می‌کند.

شبیه سازی مسئله فلاسفه خورنده

ابتدا سیستم کال های گفته شده در صورت پروژه را مشابه آزمایش های قبلی اضافه میکنیم، سپس مسئله فلاسفه خورنده را با پیاده سازی متغیر شرط با استفاده از سمافور و سپس استفاده از مانیتور، شبیه سازی میکنیم.

فراخوانی‌های سیستمی مربوط به سمافور را به صورت زیر در سیستم عامل تعریف کردیم:

```
int sem_init(int i, int v, int init);
int sem_acquire(int i);
int sem_release(int i);
void initialization();
```

- sem_init: در این تابع مقدار های اولیه برای سمافور iام را برای متغیرهای استراکت سمافور قرار می‌دهیم. به این صورت که مقدار max_proc و init که مقدار اولیه میباشد را برای سمافور قرار میدهیم و lock آن را با استفاده از تابع initlock مقداردهی می‌کنیم.
- sem_acquire: در این تابع، در صورتی که سمافور جا داشته باشد، تعداد پردازه ها در سمافور را یک واحد افزایش می دهیم و در غیر این صورت پردازه را به صف پردازه های منتظر در استراکت سمافور اضافه میکنیم و آن را sleep میکنیم و مقدار proc_no در سمافور را یک واحد اضافه می‌کنیم.
- sem_release: در این تابع یک پردازه را از لیست پردازه‌های در انتظار در سمافور pop میکنیم و آن را wakeup می‌کنیم.

- initialization: در این تابع به استفاده از یک حلقه همه‌ی سمافورها را با sem_init مقداردهی اولیه میکنیم، استیت همه فیلسوفان را برابر THINKING قرار داده، مقدار count در conditionها را برابر صفر قرار داده و مقدار sem_no هم مقداردهی اولیه بر اساس شماره سمافور میدهیم. همچنین سمافور mutex و next را با استفاده از sem_init مقداردهی اولیه می‌کنیم.

حال برای پیاده سازی مسئله فلاسفه خورنده داریم:

برای این منظور یک ساختار جدید یک Semaphore و Condition تعریف میکنیم و سپس توابع مانیتور را دقیقا مشابه کتاب مرجع پیاده سازی میکنیم. این راه‌حل این محدودیت را ایجاد می‌کند که یک فیلسوف فقط در صورتی می‌تواند چنگال‌های خود را بردارد که هر دوی آنها در دسترس باشند. برای کدگذاری این راه حل، باید بین سه حالتی که ممکن است فیلسوفی در آنها پیدا کنیم، تمایز قائل شویم. برای این منظور، ساختار داده زیر را معرفی می کنیم:

```
enum {THINKING, HUNGRY, EATING} state[5];
```

فیلسوف iام می تواند متغیر state[i] = EATING را فقط در صورتی ست کند که استیت دو همسایه او EATING نباشد.

همچنین متغیر های شرط را به صورت زیر تعریف میکنیم:

```
condition self[5];
```

این به فیلسوف iام اجازه می‌دهد تا زمانی که گرسنه است، خود را به تأخیر بیاندازد، اما نمی‌تواند چنگال های مورد نیاز خود را به دست آورد. توزیع چنگال ها توسط مانیتور DiningPhilosophers کنترل می شود که تعریف آن در شکل زیر نشان داده شده است:

```
monitor DiningPhilosophers
{
   enum {THINKING, HUNGRY, EATING} state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait();
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
       (state[i] == HUNGRY) &&
       (state[(i + 1) % 5] != EATING)) {
         state[i] = EATING;
         self[i].signal();
      }
   }

   initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
    }
}
```

هر فیلسوف، قبل از خوردن، باید pickup را فراخوانی کند. این عمل ممکن است منجر به تعلیق روند فیلسوف شود. پس از اتمام موفقیت آمیز عمل، فیلسوف ممکن است غذا بخورد. به دنبال

این، فیلسوف عملیات putdown را فراخوانی می کند. بنابراین، فیلسوف iام باید عملیات pickup و putdown را به ترتیب زیر فراخوانی کند:

```
DiningPhilosophers.pickup(i);
                ...
                eat
                ...
DiningPhilosophers.putdown(i);
```

مانیتور را دقیقا مشابه اسلاید های درس با استفاده از سمافورها به صورت زیر پیاده سازی میکنیم:

```
wait(mutex);
    ...
  body of F
    ...
if (next_count > 0)
  signal(next);
else
  signal(mutex);
```

x.wait()

```
x_count++;
if (next_count > 0)
  signal(next);
else
  signal(mutex);
wait(x_sem);
x_count--;
```

x.signal()

```
if (x_count > 0) {
  next_count++;
  signal(x_sem);
  wait(next);
  next_count--;
}
```

توجه شود که با پیاده سازی مانیتور مشکل بن بست نداریم.