«به نام خدا»

پروژه چهارم آزمایشگاه سیستم عامل

اعضای گروه:

نرگس غلامی

سارا رضایی منش

(1

اگر شما یک lock را acquire کنید، این lock متعلق به پراسس حاضر است. بقیه پراسس ها اگر سعی کنند همان lock را دریافت بکنند بلاک میشوند. lockها برای یک مدتی متعلق به پراسس ها میباشد ولی تنها این موضوع کافی نیست تا از دسترسی همزمان بقیه به صورت موثر جلوگیری بکند.(مخصوصا در مورد threadهای موازی و interrupt handlers) مثلا فرض کنید یک interrupt حین پراسس رخ میدهد که این منجر به این میشود که interrupt handlers مربوط به آن سیگنال وسط ناحیه بحرانی شما فعال شود. بعد حالا شما فرض کنید که این interrupt handlers بخواهد به همان lock پراسس شما دسترسی پیدا کند و قابلیت انجام این کار را هم دارد که این موضوع اتفاق خوبی نیست و همچنین به وسیله خود main thread قابل پیشگیری نیست. پس برای این که main thread موثر واقع شود، interrupt handlers موقتا غیرفعال می گردد و بعد از unlock کردن دوباره enable میشوند.

pushcli وقفه ها را غیرفعال میکند و همچنین به ازای هر CPU متغیری به نام ncli را یک واحد افزایش می دهد.

popcli شمارنده را کاهش می دهد و اگر شمارنده برابر با صفر باشد وقفه ها را فعال می کند.

این توابع برای اطمینان از این نوشته شده اند که وقفه ها فقط زمانی غیرفعال می شوند که همه قفل ها آزاد شده باشند.

از آن طرف توابع CLI و STI را داریم که CLI اختصار Clear Interrupt است و STI که اختصار Set Interrupt است. این توابع با فلگی تحت عنوان IF یا interrupt flag کار دارند. این فلگ یک بیت است که مشخص میکند که آیا CPU سریع به maskable hardware interrupts جواب بدهد یا ندهد. فرق این توابع این است که فقط روی single processor پاسخگو هستند و در سیستم های multiprocessor روش های دیگر را باید به کار گرفت.

(2

روش spinlock باعث می‌شد lock waiters حلقه را تکرار کنند و باعث صرف زمان CPU شوند. از آن طرف sleeplock باعث می‌شود waiters پراسس را sleep کنند که تا زمانی که وقت wakeup نرسیده از زمان CPU استفاده نکنند.

با این حال، هنگامی که یک دارنده sleeplock آن را رها می کند، پراسسی که spinlock of sleeplock را اول به دست آورده است، wakeup میکند. همچنین بقیه پراسس ها همان مکانیزم sleeplock را به کار می برند که ترتیب waiter ها را تضمین میکند.

در spin-lock شرط اول و دوم راه حل critical section برقرار است ولی Bounded waiting برقرار نیست. فرض بکنید این حالت پیش بیاید که هنوز بازه زمانی پراسس قبل تمام نشده و دوباره همان قبلی بیاید و lock را اشغال بکند و بعد دوباره ممکن است این اتفاق تکرار شود. در حقیقت هیچ کرانی در این که پراسس رقیب چندبار وارد این حلقه می شود وجود ندارد. پس این راه حل مناسبی برای مسئله producer/consumer نمیباشد. به طور مثال producer ممکن است در حلقه while بماند و اجازه اجرای critical section را به consumer ندهد در نتیجه ما حدی برای تعداد بار انتظار پراسس رقیب نداریم که این یک مشکل اساسی است.

(3

حالات مختلف پردازه در xv6:

Runnable, Unused, Embryo, Sleeping, Running, Zombie

وظیفه تابع shed:

هنگامی که یک پردازه می خواهد پردازنده را آزاد کند، تابع shed را صدا می زند. این تابع، عملیات تعویض متن را فعال می کند و وقتی که در زمان دیگری مجدد تعویض متن به پردازه ی فراخواننده ی sched انجام می شود، تابع ادامه اجرای خود را از تابع sched فرا می‌گیرد. در واقع می توان گفت فراخوانی تابع sched به صورت موقت باعث توقف اجرای یک پردازه می شود. پردازه های در موقعیت های زیر پردازنده را با فراخوانی sched آزاد می کنند:

۱) هنگامی که یک timer interrupt رخ می دهد که به این معناست که کوانتوم زمانی پردازه به پایان رسیده است و باید نوبت را به پردازه دیگری بدهد. در این زمان توسط تابع trap تابع yield را فراخوانی می کند و تابع yield هم تابع shced را فراخوانی می کند.

۲) زمانی که یک پردازه کار خود را با فراخوانی تابع exit پایان می دهد که باعث فراخوانی sched می شود.

۳) وقتی که باید به علت رخداد یک event به حالت Sleeping برود، sched را فراخوانی می کند و پردازنده را آزاد می کند.

هنگامی که یک پردازه می خواهد پردازنده را آزاد کند باید ptable.lock را بدست بیاورد و هر lock دیگری که دارد را آزاد کند. سپس وضعیت خود را از حالت running خارج کند و سپس تابع sched را صدا بزند که این فرایند از طریق توابع sleep، yield و exit انجام می شود. Sched این شرایط را مجدد چک می کند و در صورتی که هر یک از عملیات ها انجام نشده بود، ارور مربوطه را ارسال می کند. سپس تابع swtch را فراخوانی می کند تا متن فعلی در proc->context ذخیره شود و به متن scheduler در cpu->scheduler تعویض متن می کند.

(4

```
void
releasesleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  if(myproc()->pid == lk->pid) {
    lk->locked = 0;
    lk->pid = 0;
    wakeup(lk);
  }
  release(&lk->lk);
}
```

استراکت sleeplock در خود مقدار عددی به نام pid دارد که شناسه پردازه نگهدارنده را در خود ذخیره می کند. هنگام صدا زدن تابع aquiresleep این شناسه در lk->pid ذخیره می شود.

در تابع releasesleep، افزودن شرط نشان داده شده در تصویر بالا موجب می شود که تنها در صورتی که پردازه فعلی که در cpu در حال اجرا شدن است. و releasesleep را فراخوانی کرده است، شناسه یکتا یکسانی با نگهدارنده sleeplock دارد، عملیات آزادسازی قفل انجام شود.

قفل های mutex در لینوکس کارکرد مشابهی دارند. به این صورت که تسکی که می خواهد وارد critical section بشود باید ابتدا mutext_lock و هنگام خروج تابع mutex_unlock را فراخوانی کند. در صورتی که mutex lock قابل دسترسی نبود و توسط تسک دیگری گرفته شده بود، تسک فعلی به وضعیت sleep وارد می شود تا mutex lock توسط نگهدارنده آن آزاد شود. لینوکس قفل های semaphore را که بر اساس توضیحات قبل با وارد کردن تسک ها به وضعیت sleep کار می کنند را نیز دارد.

(5

حافظه تراکنشی مدلی جایگزین برای lock ها برای کنترل دسترسی به حافظه به صورت همروند در برنامه نویسی موازی می باشد.
در مدل حافظه تراکنشی به جای مشخص کردن بخش هایی از کد به صورت critical، بخش هایی از کد به صورت transaction مشخص می شوند. هر transaction مجموعه ای از عملیات هاست که می‌توانند اجرا شوند و در متغیر ها تا زمانی که conflict رخ ندهد تغییر ایجاد کنند.
حافظه تراکنشی تضمین می کند که اجرای موازی یک برنامه چند ریسه ای معادل اجرای آن در حالتی خواهد بود که هیچ کدام از بخش های transaction پشت سر هم انجام شوند و نه همزمان.
این کار به صورت انجام می شود که بر خلاف lock ها، حافظه تراکنشی به بخش های transaction اجازه می دهد که همزمان اجرا شوند اما متغیر های مربوط به این بخش ها را رصد می کند. در صورتی که دو بخش تراکنش همزمان تصمیم به دسترسی به این متغیر ها داشته باشند، یکی از آنها abort می شود و به ابتدای تراکنشی باز می‌گردد که در حال انجام آن بوده است(rollback) و به صورت خودکار آن را از سر می‌گیرد و به این ترتیب دیگر برای کنترل ورود و خروج ریسه ها از critical section احتیاجی به lock ها نداریم.

توضیح قسمت کد پروژه چهارم:

در ابتدا برای پیاده سازی سمافور سه فراخوانی سیستمی sem_init و sem_acquire و sem_release پیاده سازی شد.

توابع اصلی را در پایین مشاهده می کنید:

```
int sem_init(int i , int v)
{
  sems[i].value = v;
  sems[i].last = 0;
  return 0;
}

int sem_acquire(int i)
{
  if(sems[i].value <= 0)
  {
    struct proc* p = myproc();
    sems[i].list[sems[i].last ++] = p;
    sem_sleep(p);
  }
  else
    sems[i].value--;

  cprintf("philosopher %d acquired %d with value %d\n", myproc()->pid-3 , i, sems[i].value );

  return 0;
}

int sem_release(int i)
{
  if(sems[i].last)
  {
    struct proc* p = sems[i].list[--sems[i].last];
    sem_wakeup(p);
  }
  else
    sems[i].value++;

  cprintf("philosopher %d released %d with value %d\n",  myproc()->pid-3 ,i, sems[i].value );
  return 0;
}
```

تابع **sem_init** به value سمافور iام مقدار v را اختصاص می‌دهد و همچنین مقدار last را که به ایندکس آخرین خانه خالی صف انتظار سمافور اشاره می‌کند به صفر مقدار دهی می‌کنیم.

تابع **sem_acquire** اول چک میکند که آیا این سمافور ظرفیت دارد یا خیر. اگر داشته باشد که ظرفیتش را یک واحد کم می‌کند و به کار خود ادامه می‌دهد ولی اگر نداشته باشد پراسس به آرایه پراسس های در انتظار روی آن سمافور اضافه می‌شود و به خواب می‌رود.

تابع **sem_release** از آنجایی که مد نظر دارد یکی از ظرفیت ها را خالی بکند، اول نگاه میکند که ببیند آیا پراسسی هست که ما بخواهیم آن را wakeup بکنیم. اگر وجود داشت آن را wakeup می‌کند و در پایان ظرفیت را یک واحد بیشتر می‌کند.

به علت پرهیز از شلوغی گزارش از توضیح نحوه نوشتن این فراخوانی‌های سیستمی در برنامه خودداری می‌شود.

برنامه تست فلاسفه خورنده:

```
int main()
{
    sem_init(0, 1);
    sem_init(1, 1);
    sem_init(2, 1);
    sem_init(3, 1);
    sem_init(4, 1);
    sem_init(5, 4);

    for(int i = 0 ; i < 5 ; i++)
    {
        int id = fork();
        if(!id)
            phil(i+1);
    }
    exit();
}
```

منطق این برنامه به این صورت جلو می‌رود که ابتدا 5 سمافور مخصوص قاشق‌ها و سپس یک سمافور 4 تایی مخصوص حداکثر تعداد افرادی که می‌توانند بدون deadlock غذایشان را بخورند تعریف می‌کنیم. سپس برای این که فیلسوف‌ها را دور میز بچینیم از یک فور کمک می‌گیریم و 5 بار fork را انجام می‌دهیم و هر کدام در برنامه‌ای که در قسمت بعد شرح می‌شود شروع به خوردن یا فکر کردن می‌کنند.

برنامه فلاسفه خورنده:

```
void phil(int phil_id)
{
    double x = 0;
    for(int j = 0 ; j < 10 ; j++)
    {
        sem_acquire(5);
        sem_acquire(phil_id % 5);
        sem_acquire(phil_id - 1);

        //eat
        for(int i = 0 ; i < 2000000000 ; i+= 1)
            x += 80.86;

        sem_release(phil_id % 5);
        sem_release(phil_id - 1);
        sem_release(5);

        //think
        for(int i = 0 ; i < 200000 ; i++)
            x += 80.86;
    }
    exit();
}
```

برنامه بالا برنامه ایست که در پردازنده هایی که توسط fork ساخته می شوند اجرا می شود. در این برنامه به جای هر chopstick یک سمافور با ظرفیت یک قرار داده شده است. در بخش خوردن که همان critical section است برای جلوگیری از deadlock(هر فیلسوف chopstick سمت چپ یا راست خود را بردارد) یک سمافور با ظرفیت چهار قرار دادیم که اجازه ندهد همزمان هر پنج فیلسوف عمل خوردن را انجام دهند که با توجه به اصل لانه کبوتری همیشه حداقل یک فیلسوف به دو chopstick دسترسی داشته باشد.(سمافور ششم) هر فیلسوف باید بعد از گرفتن سمافور ششم، chopstick های سمت راست و چپ خود را برای شروع غذا خوردن بردارد. از آنجایی که ظرفیت این سمافور ها یک است، دو فیلسوف نمی توانند همزمان یک chopstick را بردارند. فرایند خوردن یک حلقه زمانبر است تا فیلسوف ها را به اندازه کافی در بخش خوردن نگه دارد که بقیه فلاسفه هم به این بخش برسند.

پس از اتمام این حلقه، هر فیلسوف باید قفل مربوط به chopstick هایی که برداشته و قفل سمافور ششم را آزاد کند. پس از انجام این بخش، فلاسفه وارد مرحله فکر کردن می شوند که خود یک حلقه زمانبر است.

```c
void sem_wakeup(struct proc *p1)
{
  acquire(&ptable.lock);
  p1->state = RUNNABLE;
  release(&ptable.lock);
}

int sem_init(int i , int v)
{
  sems[i].value = v;
  sems[i].last = 0;
  return 0;
}
```

دو تابع بالا تابع های wakeup و sleepای که می‌باشد که اختصاصی تابع sem ها هستند.

```
philosopher 4 released 3 with value 1
philosopher 5 acquired 4 with value 0
philosopher 4 released 5 with value 1
philosopher 5 released 0 with value 1
philosopher 5 released 4 with value 1
philosopher 2 released 1 with value 0
philosopher 5 released 5 with value 2
philosopher 2 released 5 with value 3
philosopher 5 acquired 5 with value 2
philosopher 2 acquired 5 with value 1
philosopher 5 acquired 0 with value 0
philosopher 2 acquired 2 with value 0
philosopher 5 acquired 4 with value 0
philosopher 1 acquired 1 with value 0
philosopher 5 released 0 with value 1
philosopher 1 acquired 0 with value 0
philosopher 5 released 4 with value 1
philosopher 1 released 1 with value 0
philosopher 5 released 5 with value 2
philosopher 1 released 0 with value 1
philosopher 1 released 5 with value 3
philosopher 2 acquired 1 with value 0
philosopher 1 acquired 5 with value 2
philosopher 2 released 2 with value 1
```

بخشی از لاگ های برنامه به صورت بالا می باشد.