پاسخ تکلیف دوم سیستم عامل

پاییز ۹۸

دانشگاه صنعتی اصفهان

جواب سوال ۱:

شروط لازم برای صحیح بودن یک راه حل عبارتند از:

۱) انحصار متقابل
۲) پیشرفت
۳) زمان انتظار محدود که خود به دو دسته تقسیم می‌شود: ۱. عدم وجود گرسنگی ۲. عدم وجود بن بست[۱]

شرط اول: برقرار است زیرا شرط هر نخ برای ورود به ناحیه‌ی بحرانی، falseبودن فلگ(پرچم) نخ دیگر است. فلگ نخ دیگر تنها در شرایطی false است که یا قبل از اعلام تمایل به ورود به CS باشد (قبل از flag[i]=true که قبل از حلقه while است و یا بعد از CS باشد، یعنی در شرایطی که درون CS نباشد. بنابراین هردو نمیتوانند درون CS باشند.

شرط دوم: برقرار است. اگر یکی از نخ ها (برای مثال نخ ۰) قصد نداشته باشد که وارد ناحیه‌ی بحرانی شود پس پرچم خود را رها می‌کند(flag[0] = 0) و نخ دیگر(نخ ۱) اگر بخواهد وارد شود ابتدا پرچم نخ دیگر(flag[0]) را چک می‌کند و چون پرچم رها شده است پس وارد ناحیه‌ی بحرانی می‌شود.

گرسنگی: رخ نمی دهد، بلکه بن بست می شود. فرض کنید که نخ ۰ هم اکنون در ناحیه‌ی بحرانی قرار دارد و نخ ۱ منتظر است که وارد ناحیه‌ی بحرانی شود (در حال چک کردن شرط while) حال اگر یک پروسسور داشته باشیم و نخ ۰ از ناحیه بحرانی خارج شود و قبل از اینکه نوبت اجرا به نخ ۱ برسد، نخ ۰ دوباره اجرا شده و flag خود را ۱ کند، از آنجا که نخ ۱ قبل از انتظار پرچم خود را گرفته است(flag[1] = 1)، پس نخ ۰ هم در حلقه while با شرط یک بودن flag[1] گیر می افتد و حال اگر نوبت اجرا به نخ ۱ برسد او نیز همچنان در شرط while خود است (flag[0]=0) پس هیچ یک وارد CS نمیشوند.

در واقع اگر نخ ۰ بعد از اتمام ناحیه‌ی بحرانی بخواهد سریعا دوباره وارد ناحیه‌ی بحرانی شود، باید منتظر باشد که نخ ۱ پرچم خود را رها کند. بنابراین امکان ندارد که گرسنگی رخ دهد.

بن بست: امکان دارد اتفاق بیافتد (یک حالت رخداد بن بست در بخش قبل آمد) اما با ترتیب اجرای زیر یا اجرای دقیقا همزمان دو نخ در دو پروسسور نیز به بن بست می خوریم:

---

[۱] اگر تعریف گرسنگی را "منتظر ماندن یک نخ به اندازه‌ای نامعین" بدانیم آن گاه می‌توان گفت که بن بست نیز نوعی گرسنگی است اما در این جا تعریف گرسنگی را "منتظر ماندن یک نخ به اندازه‌ای نامعین در حالی که دیگر نخ ها وارد ناحیه‌ی بحرانی می‌شوند" در نظر می‌گیریم و از این رو بین آنها تفاوت قائل می‌شویم.

```
P0: while(true){
P0: flag[0] = true
P1: while(true{
P1 flag[1] = true
P0: while(flag[1]);
P1 while(flag[0]);
```

جواب سوال ۲:

```
Void acquire(lock* mutex){
     While( compare_and_swap( &(mutex->available), 0, 1) );
}
Void release(lock* mutex){
     compare_and_swap( &(mutex->available), 1, 0);
}
```

جواب سوال ۳:

فرض می‌کنیم که در اینجا ۱ بودن مقدار available به معنای موجود بودن آن باشد.

۱. اگر این تابع اتمیک نباشد، ممکن است دو نخ با صدا زدن این تابع، همزمان وارد ناحیه‌ی بحرانی شوند، به ترتیب دستورات زیر دقت کنید:

```
Assume before starting execution: available = 1;
P0: while(!available);
P1: while( !available);
P0: available = false;
P1 available = false;
P0: exit from acquire and continue!
P1: exit from acquire and continue!
```

۲. اگر این تابع اتمیک باشد، باز هم مشکل busy waiting ، بازدهی پردازنده را پایین می‌آورد.

راه‌حل: ساختمان داده mutex را به صورت زیر تعریف می کنیم:

```
Struct mutex{
    available;
    waiting_q;
}
```

تابعهای اتمیک زیر را پیاده سازی می کنیم:

```
  Acquire(mutex m){
     If(m.available=0)
          Add this process to m.waiting_q;
          Block();  //this function force calling thread to
                    // sleep and add it to blocked list.

     }
     m.available=0;
  }
  Release(mutex m){
     If(len(m.waiting_q)>0){
          P = pop a process from m.waiting_q
          Wakeup p
      }
      m.available=1
  }
```

جواب سوال ۴:

چون بیش از یک منبع بحرانی داریم بهتر است که از یک semaphore استفاده شود و مقدار اولیه‌ی آن برابر با m باشد؛ به ازای هر کاربری که آماده ی پردازش می‌شود یک بار wait() را صدا بزنیم و پس از اتمام پردازش نیز تابع signal() را صدا بزنیم. بنابراین اگر مقدار سمافور به صفر برسد یعنی m کاربر در حال سرویس گرفتن هستند و به کاربر m+1ام و بقیه اجازه سرویس گرفتن داده نمیشود تا اینکه سرویس دهی به حداقل یکی از کاربران قبلی به پایان برسد.

نکته: پیاده سازی آن با mutex نیز انجام پذیر است اما حجم محاسبات را افزایش می‌دهد و برنامه‌نویسی آن را سخت‌تر می‌کند..

جواب سوال ۵:

در فایل ضمیمه قرار دارد.