دانشکده‌ی مهندسی کامپیوتر
شماره دانشجویی: ۹۹۱۰۵۵۶۱
شایان صالحی

سیستم‌های عامل

نیم‌سال اول ۰۳-۰۲

تمرین سوم

پاسخ مسئله‌ی ۱.

الف

در اینجا می‌دانیم که برنامه a می‌تواند y و z را لاک کند همچنین برنامه b توانایی لاک کردن x و z و در نهایت برنامه c توانایی لاک کردن x و y را دارد.

برای آنکه بتوانیم حالتی را معرفی کنیم که تردها دچار Deadlock شوند. می‌توانیم ترتیبی به این صورت داشته باشیم:

$$a \to lock(y), \quad b \to lock(z), \quad c \to lock(x)$$

به این ترتیب تمامی قسمت‌ها لاک شده و برنامه توانایی پیش‌روی را نخواهد داشت.

ب

با تغییر ترتیب mutex-unlock ها در اینجا نمی‌توان جلوی رخ دادن Deadlock را گرفت زیرا که در قسمت الف، برنامه در خط اول لاک می‌شود و ترتیب اجرای دستورات mutex-unlock اهمیتی نخواهند داشت و برنامه هیچ‌گاه به آنلاک نمی‌رسد.

ج

از آنجایی که در زمان‌بندی first come first served یک پردازه تا آخر انجام می‌شود و سپس به پردازه بعدی می‌رویم. هر پردازه در زمان اجرای خود دو واحد مموری را لاک کرده و سپس آنلاک می‌کند که به این ترتیب هیچ‌گاه به مشکل Deadlock نخواهیم خورد.

د

برای آنکه بتوانیم توالی ارائه دهیم که بتوان این مشکل را برطرف کرد می‌توان برنامه را از حالتی که هرکدام یک واحد جداگانه را لاک کند دربیاورویم.

برای اینکار کافیست در پردازه a ترتیب لاک کردن y و z را عوض کنیم. به این ترتیب هر دو پردازه a و b در ابتدا تلاش می‌کند که z را لاک کنند و یکی موفق می‌شوند. حال دو حالت داریم:

- اگر پردازه c هر دو x و y را لاک کرده باشد. آنگاه پس از اجرای آن هر دو آنلاک شده و پردازه c کار خود را انجام داده و دو پردازه دیگر نیز می‌توانند بدون مشکل کار خود را به اتمام برسانند.
- x و y لاک نشده باشند. به این ترتیب پردازه a کار خود را اتمام کرده و y و z را آزاد می‌کنند. سپس b می‌تواند به انتهای خود برسد و درنهایت پس از اتمام کار این دو پردازه c بدون هیچ اشکالی می‌تواند ران شود.

به این ترتیب Deadlock در این حالت برطرف شده است.

پاسخ مسئله‌ی ۲.

برای آنکه بتوانیم مطمئن شویم گراف داده شده از توالی پردازه‌ها به درستی رعایت می‌شوند می‌توانیم از دو سمافور S_1 و S_2 استفاده کنیم. به این صورت که از سمافور اولی برای ترتیب توالی پردازه T_1 با T_2 و T_3 استفاده می‌کنیم و از سمافور دومی برای پردازه‌های منتی به T_4 استفاده خواهیم کرد.

به این ترتیب داریم:

```
void T_1 () {
    T_1 Running;
    Signal(S_1);
    Signal(S_1);
}

void T_2 () {
    Wait(S_1);
    T_2 Running;
    Signal(S_2);
}

void T_3 () {
    Wait(S_1);
    T_3 Running;
    Signal(S_2);
}

void T_3 () {
    Wait(S_2);
    T_4 Running;
}
```

در اینجا برای مقدار اولیه S_1 و S_2 داریم:

$$Value(S_1) = 0, \qquad Value(S_2) = -1$$

دلیل اینکار برای این است که پرادازه T_4 باید منتظر بماند که هردو پردازه T_3 و T_2 کار خود را به اتمام برسانند و تابع `Signal(S_2)` را صدا بزنند.

به صورت کلی دو پردازه دوم و سوم در `Wait()` S_1 خواهند ماند تا زمانی که پردازه اول کار خود را به اتمام برسانند و دوباره `Signal(S_1)` را صدا بزند تا مقدار آن برابر دو شود و پردازه‌های بعدی به اجرا در بیایند.

پاسخ مسئله‌ی ۳.

الگوریتم Banker یکی از روش‌های پیشگیری از Deadlock در سیستم‌های عامل است. این الگوریتم از طریق مدیریت منابع سیستم به گونه‌ای عمل می‌کند که اطمینان حاصل شود سیستم همیشه در حالت Safe State باقی می‌ماند. حالت امن به وضعیتی از سیستم گفته می‌شود که در آن، برای هر تقاضای منابع توسط فرایندها، این تضمین وجود دارد که منابع مورد نیاز بتوانند در زمانی معین و بدون ایجاد Deadlock تأمین شوند.

در الگوریتم Banker ، هر فرایندی که به سیستم اضافه می‌شود، باید حداکثر تعداد منابع مورد نیاز خود را از پیش اعلام کند. الگوریتم سپس بررسی می‌کند که آیا پذیرش تقاضای فعلی فرایند و اختصاص منابع به آن، سیستم را به حالت غیر امن خواهد کشاند یا خیر. اگر پذیرش تقاضا منجر به حالت غیر امن شود، تقاضا معلق می‌ماند تا زمانی که تأمین منابع بدون ریسک بن‌بست امکان‌پذیر باشد.

به طور خلاصه، الگوریتم Banker بررسی می‌کند که آیا اختصاص منابع درخواستی به فرایند خاصی باعث ایجاد یک زنجیره بن‌بست می‌شود یا خیر. این کار با مقایسه درخواست‌های فرایندها با منابع موجود و تخمین اینکه آیا فرایندها می‌توانند به صورت ترتیبی تکمیل و منابع آزاد شوند یا خیر، انجام می‌گیرد. در صورتی که تمام فرایندها بتوانند بدون ایجاد بن‌بست کامل شوند، سیستم در حالت امن قرار دارد و درخواست منابع تأیید می‌شود. در غیر این صورت، درخواست‌ها تا زمان تغییر شرایط منابع به تعویق می‌افتند.

این رویکرد تضمین می‌کند که سیستم همواره در حالتی قرار دارد که بتواند به درخواست‌های منابع پاسخ دهد و در عین حال از بن‌بست جلوگیری کند، به طوری که هیچ یک از فرایندها به طور دائمی در انتظار منابع قرار نگیرند و سیستم بتواند به صورت موثر به فعالیت خود ادامه دهد.

پاسخ مسئله‌ی ۴.

الف

تفاوت اصلی بین زمان‌بندی Mesa و Hoare در نحوه بیدار کردن و ادامه دادن به کار فرآیندهایی که در مانیتور منتظر هستند، قرار دارد. در زمان‌بندی نوع Hoare ، وقتی یک فرآیند از یک متغیر شرطی برای اعلام بیداری استفاده می‌کند، کنترل فوراً به فرآیند منتظر منتقل می‌شود. این یعنی فرآیندی که اعلام بیداری کرده، باید صبر کنه تا فرآیند منتظر کارش تموم بشه و دوباره کنترل رو به اون برگردونه. این روش کمی پیچیده‌تره ولی باعث می‌شه که اطمینان داشته باشیم فرآیند منتظر به محض بیدار شدن می‌تونه کارش رو ادامه بده.

در مقابل، زمان‌بندی Mesa یه کم متفاوت عمل می‌کنه. تو این روش، وقتی یک فرآیند اعلام بیداری می‌کنه، فرآیند منتظر فقط به صف آماده‌ها منتقل می‌شه ولی کنترل فوراً به اون منتقل نمی‌شه. این یعنی فرآیند منتظر باید دوباره منتظر بمونه تا نوبتش برسه. این روش ساده‌تره ولی ممکنه باعث شه که فرآیندهای منتظر زمان بیشتری رو در انتظار بگذرونن. پس در حالی که Mesa به لحاظ برنامه‌نویسی راحت‌تره، ممکنه از نظر کارایی کمی ضعیف‌تر از Hoare باشه.

ب

برای این قسمت خواهیم داشت:

```
public class Semaphore {
    public int value;
    public Lock lock;
    public CondVar condition;

    public Semaphore (int initalValue) {
        /* Create and return a semaphore with initial value: initialValue */
        value = initalValue;
        condition = New CondVar();
        lock = New Lock()
    }

    public P() {
        /* Call P() on the semaphore */
        lock.Aqurie();
        while (value <= 0) {
            condition.wait();
        }
        value -= 1;
        lock.Release();
    }

    public V() {
        /* Call V() on the semaphore */
        lock.Aquire();
        value += 1;
        condition.Signal();
        lock.Release();
    }
}
```

که به این ترتیب می‌توان یک Semaphore با استفاده از Lock داشته باشیم.

ج

حال اگر بخواهیم Lock را براساس Semaphore داشته باشیم خواهیم داشت:

```
public class Lock {
    public Semaphore s;

    public Lock() {
        /* Create new Lock */
        s = new Semaphore(1);
    }

    public void Acquire() {
        /* Acquire Lock */
        s.P();
    }

    public void Release() {
        /* Release Lock */
        s.V();
    }
}
```

که با ازای هر Lock یک سمافور با مقدار اولیه $S = ۱$ خواهیم داشت.

د

تفاوت این دو در این خواهد بود که `CondVar.Signal()` تنها در زمانی ریسورس را آزاد می‌کند که پردازه دیگری در حالت `Wait()` باشد و در حالتی که پردازه‌ای در این حالت نداشته باشیم کاری انجام نمی‌دهد. اما در طرف مقابل با هربار صدا زدن تابع `V()` در سمافور یک مقدار به مقدار اولیه آن اضافه شده، حتی در حالتی که هیچ پردازه‌ای در حالت `Wait()` نداشته باشیم. که این در ادامه برنامه ممکن است برنامه را دچار ایراد کند.

پاسخ مسئله‌ی ۵.

پاسخ مسئله‌ی ۶.

پاسخ مسئله‌ی ۷.

پاسخ مسئله‌ی ۸.