



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:

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

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 پردازش تا آخر انجام می‌شود و سپس به پردازش بعدی می‌رویم. هر پردازش 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 دربیاریم.

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

- اگر پردازش c هر دو x و y را لاک کرده باشد. آنگاه پس از اجرای آن هر دو آنلاک شده و پردازش c کار خود را انجام داده و دو پردازش دیگر نیز می‌توانند بدون مشکل کار خود را به اتمام برسانند.

- x و y لاک نشده باشند. به این ترتیب پردازش a کار خود را اتمام کرده و y و z را آزاد می‌کنند. سپس b می‌تواند به انتهای خود برسد و در نهایت پس از اتمام کار این دو پردازش 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 پردازدها 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 پردازده T_1 با T_2 و T_3 استفاده می‌کنیم و از سمافور دومی برای پردازده‌های متنی 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_4 () {
    Wait(S_2);
    T_4 Running;
}
```

در اینجا برای مقدار اولیه S_1 و S_2 داریم:

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

دلیل اینکار برای این است که پردازده T_4 باید منتظر بماند که هر دو پردازده T_2 و T_3 کار خود را به اتمام برسانند و تابع $Signal(S_2)$ را صدا بزنند.

به صورت کلی دو پردازده دوم و سوم در $Wait()$ S_1 خواهند ماند تا زمانی که پردازده اول کار خود را به اتمام برسانند و دوباره $Signal(S_1)$ را صدا بزنند تا مقدار آن برابر دو شود و پردازده‌های 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 کند 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 به 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 initialValue) {
        /* Create and return a semaphore with initial value: initialValue */
        value = initialValue;
        condition = New CondVar();
        lock = New Lock()
    }

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

    public V() {
        /* Call V() on the semaphore */
        lock.Acquire();
        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 $S = 1$ خواهیم داشت.

د

تفاوت این دو در این خواهد بود که `CondVar.Signal()` تنها در زمانی ریسورس را آزاد می‌کند که پردازش دیگری در حالت `Wait()` باشد و در حالتی که پردازش 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 پردازشی در حالت `Wait()` نداشته باشیم. که این در ادامه برنامه ممکن است برنامه را دچار ایراد کند.

پاسخ مسئله‌ی ۵.

الف

برای آنکه رشته مورد نظر چاپ شود نیاز داریم که مقادیر اولیه سمافورها را به این ترتیب مقدار دهی کنیم:

$$S_1 = 0, \quad S_2 = 0, \quad S_3 = 1$$

به این ترتیب ابتدا پردازش سوم اجرا خواهد شد و $\text{Signal}(S_2)$ را صدا زده و پردازش دوم اجرا شده و در نهایت پردازش اول اجرا خواهد شد و رشته

$$\text{output} = 123123123\dots$$

به عنوان خروجی چاپ خواهد شد.

ب

خیر در این حالت امکان ندارد چنین رشته‌ای چاپ شود، زیرا که مقدار S_1 یک محدودیت برای ما خواهد شد. برای چاپ رشته مورد نظر ابتدا نیاز است که پردازش سوم دو بار اجرا شود، می‌دانیم پس از دو بار اجرا شدن این پردازش خواهیم داشت:

$$S_1 = 1, \quad S_2 = 4, \quad S_3 = 4$$

بنابراین در این صورت در ادامه چاپ دو عدد ۱ فقط می‌توان یک عدد ۳ را به عنوان خروجی داشته باشیم که ساخت رشته مدنظر را غیرممکن می‌سازد.

پاسخ مسئله‌ی ۶.

الف

بله سیستم مورد نظر در حالت safe قرار داشته زیرا که اگر تمام منابع allocate شده را در نظر بگیریم برای منابع A ، B و C خواهیم داشت:

$$Capacity(A) = ۲, \quad Capacity(B) = ۱, \quad Capacity(C) = ۱$$

برای اجرا پردازها نیز می‌توانیم روندی همانند روند زیر را داشته باشیم:

$$P_۳ \rightarrow P_۲ \rightarrow P_۴ \rightarrow P_۱$$

که روند اجرایی آنها با جزئیات به صورت زیر خواهد بود:

$Exec(P_۳)$

$$Capacity(A) = ۱, \quad Capacity(B) = ۱, \quad Capacity(C) = ۱$$

$FreeResource(P_۳)$

$$Capacity(A) = ۷, \quad Capacity(B) = ۵, \quad Capacity(C) = ۴$$

$Exec(P_۲)$

$$Capacity(A) = ۳, \quad Capacity(B) = ۴, \quad Capacity(C) = ۴$$

$FreeResource(P_۲)$

$$Capacity(A) = ۸, \quad Capacity(B) = ۷, \quad Capacity(C) = ۷$$

$Exec(P_۴)$

$$Capacity(A) = ۶, \quad Capacity(B) = ۰, \quad Capacity(C) = ۷$$

$FreeResource(P_۴)$

$$Capacity(A) = ۱۰, \quad Capacity(B) = ۸, \quad Capacity(C) = ۹$$

$Exec(P_۱)$

$$Capacity(A) = ۸, \quad Capacity(B) = ۰, \quad Capacity(C) = ۸$$

$FreeResource(P_۱)$

$$Capacity(A) = ۱۲, \quad Capacity(B) = ۹, \quad Capacity(C) = ۱۲$$

که بدون هیچ مشکلی تمام پردازها اجرا خواهند شد.

ب

خیر، با اختصاص دو واحد از منبع A به پرداز $P_۱$ به حالت Deadlock خواهیم رسید. زیرا که در آن لحظه $Capacity(A) = ۰$ شده و هیچ پردازهای را نمی‌توانیم اجرایش را تمام کنیم که منابع را آزاد کنیم، برای همین به بن‌بست خواهیم خورد.

ج

در اینجا با توجه به دیاگرام اجرای پردازها در قسمت الف، از آنجایی که پرداز $P_۱$ آخرین پردازهای است که اجرا می‌شود، به ظرفیت منابع در خط‌های پردازشی بالای آن نگاه می‌کنیم. از آنجایی که دو بار $Capacity(B)$ به صفر می‌رسد. نمی‌توان از این منابع به این پرداز اختصاص داد در عوض از هر دو منابع A و C در همه حالات یک واحد اضافی داریم که آن را می‌توانیم به پرداز $P_۱$ اختصاص دهیم. (در واقع مینیموم ظرفیت منابع در پردازهای قبلی حداکثر ظرفیتی است که در لحظه شروع می‌توان به پرداز اول اختصاص داد.)

پاسخ مسئله‌ی ۷.

الف

نادرست، لزومی ندارد که به حالت Deadlock بر نخوریم زیرا این حالت بر سر منابع رخ داده همچنین علت‌های دیگری همانند IO نیز می‌تواند باعث ایجاد بن‌بست در سیستم ما شود و اختصاص یک پردازنده به پردازنده تضمینی برای عدم رخ دادن این مورد به ما نمی‌دهد.

ب

نادرست، ممکن است پردازنده‌های حذف شده تاثیر در ایجاد Deadlock نداشته باشند و می‌بایست پردازنده‌هایی که در حالت چرخه وجود داشته را شناسایی کرده و سپس حذف کردن آنها یکی از راه‌های ممکن برای حل این مشکل خواهد بود.

ج

درست، در حین اجرای برنامه به Deadlock بر نخواهیم خورد به خاطر اینکه هر پردازنده تمام ریسورس‌های لازم خود را در ابتدا برداشته است و نیاز ندارد که در حین اجرا به ریسورس دیگر درخواست دهد. البته این در حالتی امکان پذیر است که در همان ابتدا نیز بتوانیم این منابع را بین همه پردازنده‌ها اختصاص دهیم و ممکن است در ابتدا این کار در سیستم ممکن نباشد.

د

نادرست، لزوماً این کار باعث از بین بردن Deadlock نمی‌شود. اولویت بندی اختصاص منابع به پردازنده‌ها می‌تواند باعث جلوگیری رخ دادن یک بن‌بست شود اما ممکن است در حالتی قرار گرفته باشیم که با هر اولویت بندی ممکن هم نتوانیم سیستم را از Deadlock خارج کنیم. بنابراین این کار همیشه جوابگو نخواهد بود.