

۱. در مسئله ی *critical section* ، چهار شرط لازم وجود دارد که برای یک راه حل صحیح باید برقرار باشد. هر یک از چهار شرط را برای زیر چک کنید و اگر برقرار است توضیح دهید چگونه برقرار است در غیر این صورت یک مثال نقض برای موقعیتی که شرط برقرار نمی شود بیاورید.

P0 :

```
{
    While(True)
    {
        flag[0]=True;
        while(flag[1]); /*do nothing*/
        /*critical section*/
        flag[0]=False;
        /*remainder_section*/
    }
}
```

P1 :

```
{
    While(True)
    {
        flag[1]=True;
        while(flag[0]); /*do nothing*/
        /*critical section*/
        flag[1]=False;
        /*remainder_section*/
    }
}
```

- I. **Mutual Exclusion** : این شرط برقرار است ، به دلیل وجود چک کردن *flag* پروسس مقابل قبل از ورود به *critical section* ، هیچ دو پروسسی نمیتوانند همزمان *critical section* بشوند.
- II. **Progress** : این شرط برقرار است ، زیرا هر پروسس بلافاصله بعد از خروج از *critical section* ، با False کردن *Flag* خود در حقیقت به پروسس های دیگر اطلاع میدهد که میتوانند وارد *critical section* شوند.
- III. **Bounded Waiting (Dead Lock)** : این حالت ممکن است رخ دهد اگر فرض کنیم ابتدا P0 اجرا شود و بلافاصله بعد از اینکه این پروسس ، خط اولش (*flag[0]=True*) را اجرا کرد ، scheduler پروسس P1 را وارد cpu کند و پروسس P1 هم خط اولش (*flag[1]=True*) را اجرا کند ، از این جا به بعد دیگر به بن بست خواهیم رسید چون هر دو پروسس در حلقه ی *while* گیر خواهند کرد. **Bounded Waiting (Starvation)** : در اجرای این دو پروسس ، دچار گرسنگی نخواهیم شد.

۲. می‌خواهیم یک قفل *mutex* را توسط دستورات اتمیک سخت افزاری پیاده سازی کنیم. این قفل به

صورت زیر تعریف میشود:

```
Struct{
    int available;    "مقدار صفر برای available بیانگر آزاد بودن قفل(در دسترس بودن)
}lock;              "و مقدار یک بیانگر غیر قابل دسترس بودن بودن است."
```

مشخص کنید چگونه دو تابع زیر به کمک دستور اتمیک *compare\_and\_swap* قابل پیاده سازی است (مقادیر اولیه ی لازم را به درستی مشخص کنید)

```
void acquire(lock *mutex) , void release(lock *mutex)
int compare_and_swap(int *value, int expected, int new_value)
{
    int temp = *value;
    if(*value==expected)
        *value=new_value;
    return temp;
}
```

```
void acquire(lock *mutex)
{
    while(compare_and_swap(&(mutex → available), 0, 1)); /*do nothing*/
}
```

```
void release(lock *mutex)
{
    compare_and_swap(&(mutex → available), 1, 0);
}
```

۳. تابع زیر با هدف پیاده سازی *mutex* دارای چه مشکلی است ؟ راه حلی با توضیح یا کد کامل برای این

مشکل پیشنهاد دهید.

```
acquire( ) {  
    while(!available);  
    available=False;  
}
```

مشکل این پیاده سازی این است که این دو خط باید به صورت اتمیک اجرا شوند تا دو پروسس همزمان نتوانند وارد *critical section* بشوند که البته اینجا هیچ تضمینی برای این امر وجود ندارد .

راه حل :

استفاده از CAS به این صورت :

```
void acquire()  
{  
    while(compare_and_swap(&available, 0, 1)); /*do nothing*/  
}
```

۴. فرض کنید در یک وب سرور امکان پاسخ گویی به بیش از  $m$  کاربر به صورت همزمان وجود ندارد . بنابراین اگر  $m$  کاربر در حال سرویس گرفتن باشند و کاربر ( یا کاربران ) دیگری نیز درخواست دهند ، به درخواست او ( یا آنها ) پاسخ داده نمیشود تا اینکه حداقل سرویس دهی به یکی از کاربران تمام شود. توضیح دهید در این مسئله از کدام یک از روش های همگام سازی ( *semaphore* یا *mutex* ) بهتر است استفاده شود و به چه صورت؟

بهتر است از *semaphore* استفاده کنیم (البته که دلیلش کاملاً بدیهی است!) چون سرور به بیشتر از یک کاربر میتواند در هر لحظه پاسخ دهد، در مدل سازی این مسئله با مسئله ی *critical section* ، درواقع انگار به طور همزمان  $m$  پروسس میتوانند وارد *critical section* بشوند در نتیجه استفاده از *semaphore* که این امکان را فراهم میکند، منطقی تر است .  
کد سمت سرور به این صورت پیاده میشود :

```
Semaphore client = m;  
do {  
    /* request has received */  
    wait(client);  
    /* service */  
    signal(client);  
} while(True);
```

۵. می‌خواهیم گذر ماشین‌ها از روی یک پل یک طرفه را مدیریت کنیم به صورتی که تا وقتی ماشین‌هایی در حال عبور از پل از یک سمت هستند به ماشین‌های دیگر که از سمت دیگر قصد ورود دارند اجازه ی ورود داده نشود . محدودیتی روی تعداد ماشین‌هایی که روی پل هستند وجود ندارد. اما می‌خواهیم مسئله ی گرسنگی را تا حدی حل کنیم بدین منظور اگر ۵ ماشین از سمت شمال از پل رد شدند و تعدادی ماشین (یک یا بیشتر ) ماشین در سمت جنوب قصد ورود به پل را داشتند ، از ادامه عبور ماشین‌ها از سمت شمال جلوگیری کرده تا ماشین‌هایی از سمت جنوب از پل رد شوند. همین قانون برای سمت جنوب هم به صورت برعکس برقرار است.

با تابع *north()* ماشینی که در شمال پل است ، قصد ورود به پل را کرده و از پل رد میشود . همچنین با تابع *south()* ماشین‌های سمت جنوب از پل رد میشوند. سمافور یا میوتکس‌های مورد نیاز برای حل مسئله را تعریف کرده و دو تابع نامبرده را با استفاده از آنها پیاده سازی کنید

```
/* North is used to ensure mutual exclusion when NorthCounter is updated i.e. when any north_entered_car enters to the bridge */
/* South is used to ensure mutual exclusion when SouthCounter is updated i.e. when any south_entered_car enters to the bridge */
/* bridge is used to ensure mutual exclusion when North or South wants to executes */
semaphore South , North , bridge = 1;
```

```
/* Counters are helpfull variables used for preventing starvation */
int NorthCounter = 0;
int SouthCounter = 0;
```

```
North :
{
    wait(North);
    NorthCounter++;
    if (NorthCounter==1)
        wait(bridge);
    signal(North);

    /* Crossing the Bridge */

    if (NorthCounter>=5 && SouthCounter)
    {
        signal(bridge);
        NorthCounter = 0;
    }
}
```


```
South :
{
    wait(South);
    SouthCounter++;
    if (SouthCounter==1)
        wait(bridge);
    signal(South);

    /* Crossing the Bridge */

    if (SouthCounter>=5 && NorthCounter)
    {
        signal(bridge);
        SouthCounter = 0;
    }
}
```

۶. در مورد روش های *synchronization* در یکی از زبان های زیر (در صورتی که مشخص شده است) تحقیق کنید و در هر یک از این زبان ها حداقل دو روش را با توضیح نوع کاربرد و نحوه استفاده شرح دهید.

الف) افرادی که ششمین رقم شماره دانشجویی آنها زوج است : پایتون

ب) افرادی که ششمین رقم شماره دانشجویی آنها فرد است : جاوا 

## Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

## Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
  1. Synchronized method.
  2. Synchronized block.
  3. static synchronization.
2. Cooperation (Inter-thread communication in java)

### Method 1 : Java synchronized method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Syntax :

```
Class [ClassName]
{
    synchronized void [FunctionName](args)
    {
        /* critical section */
    }
}
```

## Method 2 : Synchronized block in java

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

### Syntax :

```
synchronized (object reference expression)
{
    // code block
}
```

## Method 3 : Static synchronization

If you make any static method as synchronized, the lock will be on the class not on object.

```
Class [ClassName]
{
    synchronized static void [FunctionName](args)
    {
        /* critical section */
    }
}
```

[Source](#)