به نام خدا

پروژه اصول سیستم های عامل

پویا هزاوه و امیر ظفری

قسمت اول: در سیستم عامل *Dead Lock* و *Race Condition* چه مفهومی دارد؟

Dead Lock

بن بست یا Dead Lock زمانی رخ می‌دهد که در سیستم عامل چندین فرآیند در حال ۰ هستند و به منابعی باید دسترسی پیدا کنند تا ادامه آن ها قابل اجرا باشد اما سیستم در وضعیتی قرار دارد که این فرآیند ها هرگز به منابع مورد نظرشان نخواهند رسید؛ این وضعیت هر چهار شرط زیر را دارد:

شرط ۱- انحصار متقابل: در هر لحظه تنها یک فرآیند بتواند به منبع یا منابع مورد نظر دسترسی داشته باشد

شرط ۲- نگهداشت و انتظار: هنگام درخواست منبع جدید فرآیند قبلی تخصیص یافته را آزاد نمی‌کند

شرط ۳- قبضه نکردن: منابع به زور قابل پس گرفتن نیستند

شرط ۴- انتظار مدور: زنجیر بسته ای از فرآیند ها وجود دارد، به طوری که هر یک حداقل یک منبع مورد نیاز فرآیند بعدی زنجیره را نگه دارد

Race Condition

مسابقه شرطی زمانی اتفاق می‌افتد که چندین فرآیند به یک یا چند منبع مشترک دسترسی داشته و اگر ترتیب متفاوت اجرای فرآیند ها نتایج مختلفی به دنبال داشته باشند، در این صورت نتایج غیر قابل پیش بینی و ناپایدار خواهند بود

قسمت دوم: *برنامه ای تهیه کنید که در آن تعدادی فرایند طوری کار می‌کنند که در آن شرایط Race Condition و Dead Lock اتفاق بیفتند*

کد زیر را بررسی کنید

برای سادگی و راحتی در تست برنامه آن را با زبان سی++ شبیه سازی کرده ام

سمافور ها و منابع با کلاس ها و فرآیند ها به شکل تابع شبیه‌سازی شده اند دستور زیر

```
this_thread::sleep_for(std::chrono::seconds());
```

برای یک ثانیه فرآیند (تابعی که به عنوان یک فرآیند شبیه‌سازی شده) را متوقف می‌کند تا مطمئن شویم فرآیند-دیگر فرصت داشته باشد تا تابع-توقف منبع دیگر را اجرا کند. و بن‌بست ایجاد شود

پس به احتمال نزدیک به صددرصد این برنامه در بن‌بست گیر خواهد کرد

(.این کد را در فایل فشرده قرار داده ام)

Cpp > OS-project.cpp > main()

```cpp
#include <iostream>
#include <chrono>
#include <thread>
using namespace std;

class Semaphore
{
public:
    int index;
    Semaphore()
    {
        index = 1;
    }
    Semaphore(int number_of_resources)
    {
        index = number_of_resources;
    }

    void signal()
    {
        index++;
    }

    void wait()
    {
        while (index <= 0);
        index--;
    }
};

```

```cpp
class ExampleResource
{
public:
    int mem;
    ExampleResource()
    {
        mem = 0;
    }
    ExampleResource(int initial)
    {
        mem = initial;
    }
};

Semaphore sem_1;
Semaphore sem_2;
ExampleResource resource_1;
ExampleResource resource_2;

void process_1()
{
    sem_1.wait();
    this_thread::sleep_for(std::chrono::seconds(1));
    sem_2.wait();

    cout<<"C-S of process 1\n";
    resource_1.mem += resource_2.mem;

    sem_1.signal();
    sem_2.signal();
}

void process_2()
{
    sem_2.wait();
    this_thread::sleep_for(std::chrono::seconds(1));
    sem_1.wait();

    cout<<"C-S of process 2\n";
    resource_2.mem += resource_1.mem;

    sem_1.signal();
    sem_2.signal();
}

int main()
{
    resource_1 = ExampleResource(10);
    resource_2 = ExampleResource(3);
    thread t1(process_1);
    thread t2(process_2);

    // تابع جوین باعث میشه این تابع اصلی تمام نشه و صبر کنه تا فرآیند های
    // تی۱ و تی۲ هم تمام شوند سپس برنامه جافطه مینابع را چاپ کند و تمام شود
    t1.join();
    t2.join();

    cout << "memory of the first resource: " << resource_1.mem << endl;
    cout << "memory of the second resource: " << resource_2.mem << endl;
}

```

Race Condition:

وجود ریس-کاندیشن در این برنامه که کاملا مشهود است، چندین فرآیند به منابع مشترکی می‌توانند دسترسی داشته باشند.
همچنین ترتیب اجرای متفاوت دستورات فرآیند ها نتایج متفاوتی خواهد داشت برای مثال اگر ابتدا فرآیند اول وارد کریتیکال سکشن شود مقدار داخل حافظه منبع-اول برابر ۱۳ و حافظه منبع-دوم برابر ۱۶ خواهد بود اما اگر برعکس اتفاق بیوفتد یعنی ابتدا فرآیند دوم بتواند وارد کریتیکال سکشن شود آنگاه حافظه منبع-اول برابر ۲۳ و حافظه منبع-دوم برابر ۱۳ خواهد بود پس در ترتیب اجرا های متفاوت مقادیر مختلفی خواهیم گرفت

Dead Lock:

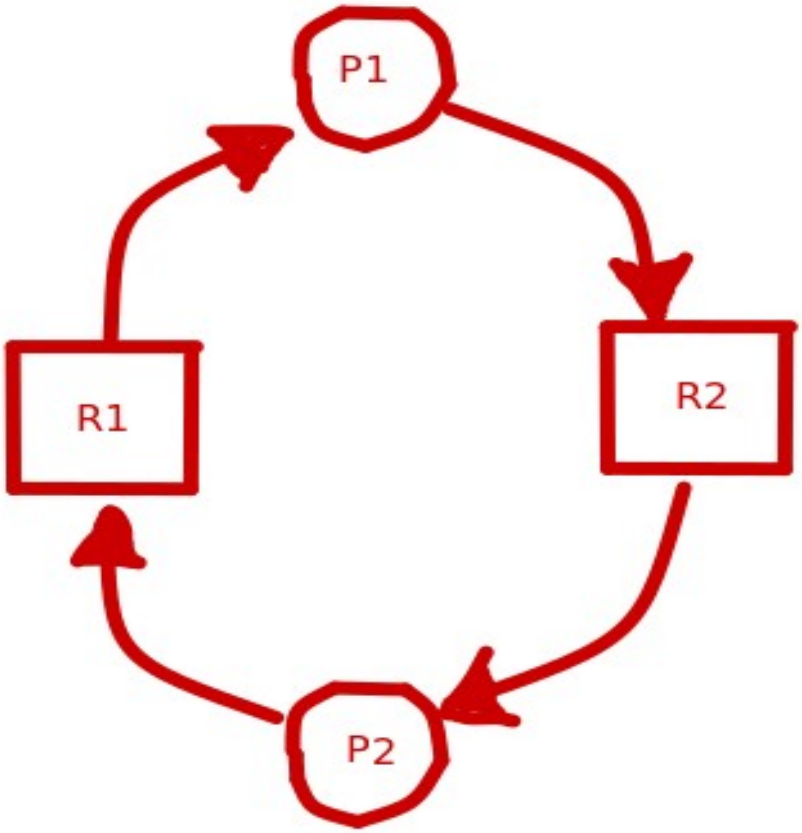
در برنامه ای که نوشته ایم اگر اول تابع-توقف سمافور-اول در فرآیند-اول اجرا شود سپس تابع-توقف سمافور-دوم که در فرآیند-دوم قرار دارد اجرا شود هر چهار شرط بن بست همزمان ایجاد می‌شود:

شرط ۱- انحصار متقابل: این شرط برقرار است زیرا ما به وسیله سمافور ها انحصار متقابل ایجاد کرده ایم

شرط ۲- نگهداشت و انتظار: این شرط برقرار است زیرا همانطور که می‌بینید در هر دو فرآیند دو تابع توقف سمافور پشت هم قرار دارند یعنی قبل از اینکه سمافور منبعی که قبلا اشغال نموده اند را سیگنال کنند منبع دیگری را می‌خواهند اشغال کنند

شرط ۳- قبضه نکردن: این شرط برقرار است همانطور که می‌بینید دستور یا تابعی برای قبضه کردن در هیچ جای این برنامه کد نویسی نشده است چه برسد به آنکه اجرا شود

شرط ۴- انتظار مدور: این شرط برقرار است زیرا در آن حالت خاصی از ترتیب اجرا که قبلا گفتیم بن بست رخ خواهد داد، منبع اول دست فرآیند اول است و منبع دوم دست فرآیند دوم است و هر دو منبع منتظرند تا منبع دیگری که دستشان نیست را دریافت کنند در نتیجه گراف تخصیص آن به شکل زیر خواهد بود که انتظار مدور در آن مشهود می‌باشد



اگر ما دستور

```
this_thread::sleep_for(std::chrono::seconds());
```

در توابع فرآیندها را به شکل زیر قرار می‌دادیم:

```
50  void process_1()
51  {
52      this_thread::sleep_for(std::chrono::seconds(1));
53      sem_1.wait();
54      sem_2.wait();
55
56      cout<<"C-S of process 1\n";
57      resource_1.mem += resource_2.mem;
58
59      sem_1.signal();
60      sem_2.signal();
61  }
62  void process_2()
63  {
64      sem_2.wait();
65      sem_1.wait();
66
67      cout<<"C-S of process 2\n";
68      resource_2.mem += resource_1.mem;
69
70      sem_1.signal();
71      sem_2.signal();
72  }
73
```

در این صورت از آنجا که کریتیکال-سکشن فرآیند-دوم زمان بسیار ناچیزی نسبت به یک-ثانیه نیاز دارد پس به احتمال نزدیک به صددرصد ابتدا کریتیکال-سکشن فرآیند دوم انجام شده سپس فرآیند-اول وارد کریتیکال-سکشن خود می‌شود در این صورت بن‌بست رخ نخواهد داد

نکته: ما با اسلیپ کردن فرآیند ها صرفا احتمال رخداد بن بست را در کد قبلی افزایش و در این کد، برعکس، کاهش داده ایم. در هر صورت این برنامه همان طور که صورت این سؤال از ما می‌خواهد احتمال رخداد بن‌بست وجود دارد. در نظر داشته باشید که حتی اگر دستور اسلیپ

فرآیند ها را به این شکل قرار دهیم باز هم احتمال رخداد بن بست وجود دارد برای مثال ممکن است سیستم مدتی هنگ کند یا این پردازش ها به علت پردازش های دیگری طول بکشد و به تعویق بیفتد یا در کریتیکال-سکشن ها دستورات وقت گیر دیگری هم اضافه نماییم در این صورت احتمال اینکه دستورات فرآیند ها به همان ترتیبی که بن‌بست ایجاد می‌شود اجرا شوند دیگر ناچیز نخواهد بود. (قبل تر درمورد این ترتیب خاص توضیح داده ایم که باعث ایجاد شرط چهارم بن بست می‌شود)

خلاصه: پس صرفا با گذاشتن دستور اسلیپ برای فرآیند ما صرفا احتمال رخداد بن بست را تغییر دادیم که برنامه را تست کنیم و نشان دهیم که در این برنامه چه حالاتی ممکن است رخ دهد

نتیجه: در برنامه فعلی ریس-کاندیشن و همچنین احتمال رخداد دد-لاک وجود دارد

توضیحات تکمیلی:

از کتابخانه آی-او-استریم برای نمایش خروجی در کنسول با استفاده از دستور سی-اوت استفاده شده
از کتابخانه ترد برای شبیه سازی فرآیند ها استفاده شده است
از کتابخانه کرونو برای شبیه سازی اسلیپ شدن فرآیند استفاده شده است

کد هر دو نمونه" احتمال بن بست نزدیک به صد" و" احتمال بن بست نزدیک به صفر" را به شکل فایل در فایل فشرده قرار داده ام تا بتوان آن ها را بررسی کرد

قسمت سوم: *سپس یک الگوریتم پیاده سازی کنید که مشکل Dead Lock را در این سیستم حل کند.*

روش های مختلفی برای رفع مشکل بن‌بست وجود دارد

ما از روش جلوگیری از بن‌بست با نقض شرط چهارم بن‌بست یعنی "انتظار مدور" عمل کردیم؛ به این صورت که دیگر هر فرآیند به طور مستقیم به سمافور دسترسی ندارد و از طریق یک رابط کاربری منبع مورد نظرش را درخواست میکند و رابط کاربری به روشی که شرط انتظار مدور را نقض میکند منابع را به فرآیند ها تخصیص میدهد

این روش اینطور کار میکند که هر فرآیند در ابتدا آرایه ای از منابعی که نیاز خواهد داشت را به شکل آرایه ای به رابط کاربری خود اعلام میکند

Cpp > Question3-DeadLock-Fixed.cpp > ProcessInterface

```cpp
#include <iostream>
#include <chrono>
#include <thread>
using namespace std;

class Semaphore
{
public:
    int index;
    Semaphore()
    {
        index = 1;
    }
    Semaphore(int number_of_resources)
    {
        index = number_of_resources;
    }

    void signal()
    {
        index++;
    }

    void wait()
    {
        while (index <= 0)
            ;
        index--;
    }
};
// ExampleResource manabaye farzi hast ke hafezei ham darad.
class ExampleResource
{
public:
    int mem;
    ExampleResource()
    {
        mem = 0;
    }
    ExampleResource(int initial)
    {
        mem = initial;
    }
};

```

رابط کاربری (که میتوانید کلاس مربوط به آن را در این صفحه مشاهده کنید) هنگامی که یک فرآیند از طریق تابع ریکوئست–ریسورس درخواست منبعی میدهد ابتدا منابعی که ایندکس کوچکتر داشته و همچنین فرآیند ممکن است فرآینده آن ها را درخواست کند، به فرآیند اختصاص داده میشوند. همچنین وقتی درخواست آزادسازی منابع از طرف فرآیندی به رابط ارسال میشود، تمام منابع با ایندکس بزرگتر از بزرگترین ایندکس درخواستی آزاد میشود

```
const int NUMBER_OF_RESOURCE_TYPES = 2;
ExampleResource resource[NUMBER_OF_RESOURCE_TYPES];
Semaphore semaphore[NUMBER_OF_RESOURCE_TYPES];

class ProcessInterface
{
private:
    // cell i-th contain the maximum need of i-th resource of the process
    bool is_needed[NUMBER_OF_RESOURCE_TYPES];
    // cell i-th contains 'true' if(i-th resource requested by the process) else: 'false'
    bool is_requested[NUMBER_OF_RESOURCE_TYPES];
    // cell i-th contains 'true' if(i-th resource is currently captured by the process) else:'false'
    bool is_captured[NUMBER_OF_RESOURCE_TYPES];
    int find_the_highest_requested_id()
    {
        int result = -1;
        for (int i = 0; i < NUMBER_OF_RESOURCE_TYPES; i++)
        {
            if (is_requested[i])
            {
                result = i;
            }
        }
        return result;
    }

public:
    // the interface recieves and sets the max needs list at the initial define.
    ProcessInterface(int max_need[])
    {
        for (int i = 0; i < NUMBER_OF_RESOURCE_TYPES; i++)
        {
            this->is_needed[i] = max_need[i];
            this->is_requested[i] = false;
            this->is_captured[i] = false;
        }
    }
    void request_resource(int id_of_the_resource)
    {
        for (int i = find_the_highest_requested_id() + 1; i <= id_of_the_resource; i++)
        {
            if (is_needed[i] == true)
            {
                semaphore[i].wait();
                is_captured[i] = true;
            }
        }
        is_requested[id_of_the_resource] = true;
    }
    void release_resource(int id_of_the_resource)
    {
        is_requested[id_of_the_resource] = false;
        for (int i = find_the_highest_requested_id() + 1; i <= id_of_the_resource; i++)
        {
            if (is_captured[i] == true)
            {
                semaphore[i].signal();
                is_captured[i] = false;
            }
        }
    }
};

```

```cpp
void process_1()
{
    int needed_resources[] = {true, true};
    ProcessInterface interface = ProcessInterface(needed_resources);

    interface.request_resource(0);
    this_thread::sleep_for(chrono::seconds(1));
    interface.request_resource(1);

    cout << "C-S of process 1\n";
    resource[0].mem += resource[1].mem;

    interface.release_resource(0);
    interface.release_resource(1);
}

void process_2()
{
    int needed_resources[] = {true, true};
    ProcessInterface interface = ProcessInterface(needed_resources);

    interface.request_resource(1);
    this_thread::sleep_for(chrono::seconds(1));
    interface.request_resource(0);

    cout << "C-S of process 2\n";
    resource[1].mem += resource[0].mem;

    interface.release_resource(0);
    interface.release_resource(1);
}

int main()
{
    // bayad tedade i-th resource ra roye i-th semaphor set konim
    semaphore[0] = Semaphore(1);
    semaphore[1] = Semaphore(1);
    resource[0] = ExampleResource(10);
    resource[1] = ExampleResource(3);
    thread t1(process_1);
    thread t2(process_2);

    // تابع جوین باعث میشه فرآیند این تابع اصلی صبر کنه
    // تا فرآیند های تی۱ و تی۲ هم تمام شوند
    t1.join();
    t2.join();

    // سپس برنامه جافطه منابع را چاپ میکند و تمام میشود
    cout << "memory of the first resource: " << resource[0].mem << endl;
    cout << "memory of the second resource: " << resource[1].mem << endl;

    cout << endl;
    return 0;
}
```

قسمت چهارم: *راه حل های دیگری که می‌توانند این مشکل را رفع کنند پیشنهاد کنید و بگویید کدام راه حل بهتر است.*

سه راه برای مقابله با بن بست وجود دارد و ما از هر کدام یک مثال زده ایم:

راه ۱- پیشگیری از بن‌بست: برای راه پیشگری از بن بست باید یکی از چهار شرط ایجاد بن بست را نقض کنیم؛ ما هم از یکی از روش های پیشگیری از بن بست یعنی با نقض شرط چهارم بن‌بست (انتظار مدور)، مشکل بن‌بست را حل نمودیم

این روش باعث میشود به فرآیند ها منابعی که لازم ندارند اختصاص یابد و در نتیجه فرآیند های دیگر نتوانند به منابع دسترسی پیدا کنند و کند شوند

همچنین در این روش باید حداکثر منابعی که هر فرآیند نیاز دارد از قبل مشخص باشد

راه ۲- اجتناب از بن‌بست: الگوریتم معروفی که با این روش با بن‌بست مقابله میکند الگوریتم بانکداران است

تعدادی از معایب:

وقوع بن‌بست را نمی‌تواند به طور حتم پیش‌بینی کند بلکه فقط میتواند بروز آن را حدس زده یا گاهی اطمینان دهد

* باید حداکثر منابعی که هر فرآیند نیاز دارد از قبل مشخص باشد
* امکان گرسنگی فرآیند ها وجود دارد

راه ۳- برخورد با بن‌بست: دراین روش هیچ تلاشی برای پیشگیری یا اجتناب از بن‌بست قبل از وقوع آن صورت نمی‌گیرد بلکه سیستم صبر می‌کند تا زمانی که بن‌بست رخ دهد تا در صورت رخ دادن بن‌بست با آن برخورد کند

به نظر ما در این روش در صورت رخداد بن‌بست سیستم عامل دستورات متعددی را اجرا خواهد کرد تا فرآیند هارا متوقف، منابع را قبضه و فرآیند ها را به حالتی قبل از رخداد بن‌بست منتقل کند در نتیجه اگر منابع و فرآیند ها طوری باشند که بن‌بست ها به وفور اتفاق بیفتند این راهبرد باهث کاهش بازدهی کلی سیستم میشود

نتیجه: هر استراتژی مزایا و معایب خودشو داره و بهترین راه اینه که از ترکیبی از آن ها استفاده کرد