Operating Systems

Isfahan University of Technology
Electrical and Computer Engineering Department

Zeinab Zali

Synchronization

هماهنگ سازی

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to **shared data** may result in data **inconsistency**
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

-
توی سیستم پروسس ها ممکنه که همزمان اجرا بشن و هر پروسسی ممکنه که در هر لحظه ای بخاطر وقفه ای که اتفاق افتاده متوقف بشه بنابراین اجراش یه جورایی ناتموم می مونه
حالا اگر پروسس های همزمانی داشته باشیم که یک دیتای مشترک رو دارند استفاده می کنند ممکنه دسترسی نادرست به اون دیتای مشترک باعث data inconsistency ینی اون مقداری که ما انتظار داریم توی اون شیردیتا باشه توی اون شیر دیتا نخواهد بود بنابراین اون چیزی که میخوایم حل بکنیم و روش کار بکنیم قسمت ابی رنگ است:
ینی می خوایم مکانیزیم هایی داشته باشیم که مطمئن بشیم پروسس هایی که با هم دیگه یک تعاملی دارند یا یک شیر دیتایی دارند ترتیب اجرایی خواهند داشت که باعث data inconsistency نمیشه یا یک ترتیب درست است که باعث میشه دیتامون همانطور که می خوایم consistency باشه و consistency همان طور که گفتیم منظورمون این است که اون چیزی که انتظار داشتیم اتفاق بیوفته ینی ما برنامه نویس که اون کد رو نوشتم چه انتظاری داشتم از اون کدی که نوشتم:
دیتاها مقادیرشون بعد از اجرای اون کد به همون صورتی باشه که ما انتظار داشتیم نه اینکه با یک ترتیب اشتباهی اجرا شده باشه که باعث شده باشه مقادیر دیتا اونطوری که میخوایم نباشه

Illustration of the problem: Calculating summation of numbers with some threads

- We want to calculate the summation of some numbers with more than one thread to speed up the operation
- Consider a global variable sum
- We use data parallelism, divide numbers in to some sets, create a thread for each set to add numbers to sum
- Execute the code in next page and see the result

-
چندتا مثال برای روشن شدن مسئله:
1- قبلا اینو گفتیم
2- بحث جمع کردن یک تعداد عدد است ینی یک تعداد زیادی عدد داریم و می خوایم جمع این ها رو توی یک متغییری محاسبه بکنیم چون تعداد اعداد زیاده میایم توی چندتا ترد این جمع رو انجام میدیم ینی تا n رو می خوایم جمع بکنیم و اینو مثلا تقسیم می کنیم به 4 تا ترد و هر قسمت رو از اون اعداد رو جدا جدا جمع می زنیم توی ترد و روشمون هم به این صورت است که یک global variable به نام sum که قبل از اجرای این تردها مقدارش صفره و بعد اون عددهایی که داخل اون تردها است به این مقدار Sum توی هر تردی به طور جداگانه اضافه میشه و انتظار داریم وقتی که این 4 تا ترد تموم شد مقدار sum برابر با جمع کلش بشه : کدش صفحه بعدی...

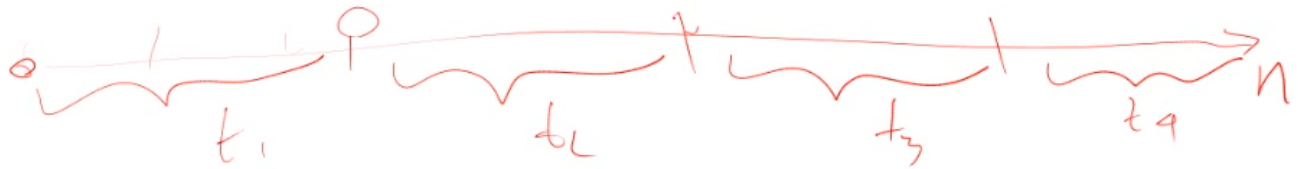


Illustration of the problem: Calculating summation of numbers with some threads

```
/*************************************************************
Concurrency problem
*************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>


struct thread_input{
int b;
int e;
};

int sum = 0;

void *summation_thread(void *input)
{
   struct thread_input *arg;
   arg = (struct thread_input*)input;
   for (int i=arg->b; i<=arg->e; i++){
  sum += i;
}
   pthread_exit(NULL);
}
```

```
int main(int argc, char *argv[])

{

   int n = 10000000;

   pthread_t threads[NUM_THREADS];

   int rc, t;

   struct thread_input beg_end[NUM_THREADS];

   int d = n / NUM_THREADS ;

   for(t=0;t<NUM_THREADS;t++){

        beg_end[t].b = t * d + 1;

         beg_end[t].e = beg_end[t].b + d - 1;

         rc = pthread_create(&threads[t], NULL, summation_thread, (void
         *)&beg_end[t]);

         if (rc){

                  printf("ERROR; return code from pthread_create() is %d\
         n", rc);

                  exit(-1);

          }

   }

   for (t=0;t<NUM_THREADS;t++){

          pthread_join(threads[t], NULL);

   }

   printf("sum= %d\n", sum);

   /* Last thing that main() should do */

   pthread_exit(NULL);

}
```

Illustration of the problem: updating a linked list

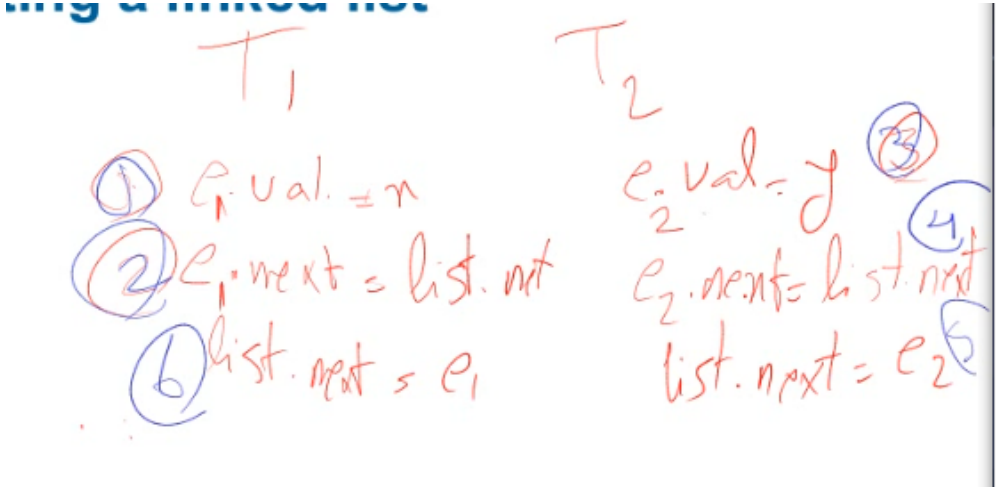
Create new list element e

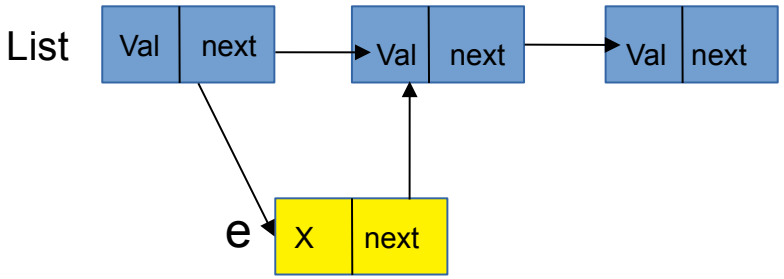
Set e.value = X

Read list and list.next

Set e.next=list.next

Set list.next=e





What happens if two threads try to add an element concurrently to the same list??

مثال بعدی:
در مورد لینکدلیست است
فرض میکنیم یک لینکدلیستی داریم و می خوایم اینو اپدیتش بکنیم ینی میخوایم یک المنت جدیدی به این لینکدلیست اضافه بکنیم
ما اگر یک فانکشنی نوشته باشیم برای اینکه این المان جدید رو به این لینکدلیست اضافه بکنیم و این فانکشن رو ممکنه ما جاهای مختلفی توی پروگرمی که نوشتیم استفاده بکنیم توی تردهای مختلفی و باعث بشه یه جایی توی دوتا ترد همزمان این فانکشن فراخوانی بشه در این حالت اتفاقی که می افته این است که :
مثلا ترد T1 داره x رو اضافه میکنه و ترد T2 هم داره y اضافه میکنه
اگر فرض کنیم که T2 , T1 دارن همزمان اجرا میشن
شماره هایی که کنار دستورات هست ینی به این ترتیب دارن اجرا میشن
توی این لیست اول y اضافه شد
توی شماره 6 افتاقی که می افته اینه که اول list.next به e1 وصل میشه و بعد از اون list.next که به e2 وصل بوده حذف میشه و در نهایت اون لیستی که می بینیم لیستیه که x بهش اضافه شده و y توش وجود نداره ینی توی این لیست حساب نمیشه
انتظار ما این بود که وقتی که این دوتا ترد اجرا شد قسمت سبزرنگ ساخته میشد

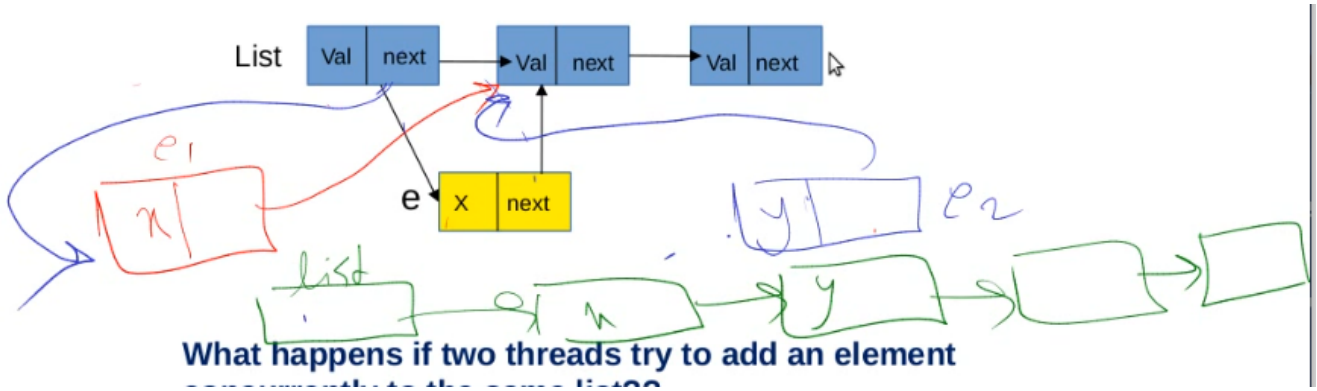


Illustration of the problem: producer-consumer problem

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers.
- We can do so by having an integer `counter` that keeps track of the number of full buffers.
- Initially, `counter` is set to 0.
- **counter** is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

-
مسئله بعدی مسئله producer-consumer است:
فرض کنید بخاطر اینکه بافرمون محدود است میخوایم از یک کانتری استفاده بکینم که هم چک بکنه اگر بافر پر بود تولید کننده دیگه نمیتونه مقدار جدیدی توش قرار بده و هم اینکه اگر بافر خالی بود مصرف کننده باید صبر بکنه که تولیدکننده یک مقداری توی بافر قرار بده
کدش صفحه بعدی...

Producer

```
while (true) {
    /* produce an item in next produced */

    while (counter == BUFFER_SIZE) ;
        /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Consumer

```
while (true) {
    while (counter == 0)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
        counter--;
    /* consume the item in next consumed */
}
```

Race Condition

- `counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter`	{register1 = 5}
S1: producer execute `register1 = register1 + 1`	{register1 = 6}
S2: consumer execute `register2 = counter`	{register2 = 5}
S3: consumer execute `register2 = register2 - 1`	{register2 = 4}
S4: producer execute `counter = register1`	{counter = 6 }
S5: consumer execute `counter = register2`	{counter = 4}

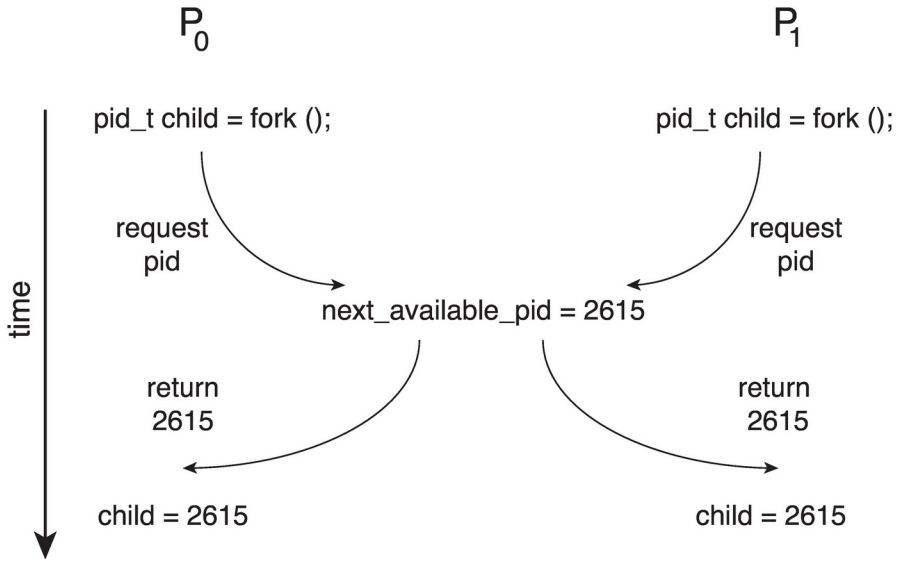
حالتی که این دوتا ینی تولید کننده و مصرف کننده دارن توی دوتا ترد جداگانه اجرا میشن و همزمان این دوتا ترد به خط اپدیت کردن کانتر می رسن ینی به خط counter++ , counter-- اتفاقی که می افته:
از توی اسلاید بخون
اگر قبلش counter=5 بوده بعد از اجرای این دوتا هم باز باید counter=5 میشد ولی این مقدار اینجا یا 4 میشه یا 6

نکته: اینجا دوتا پروسسور داریم پس اینطوری نشون داده میشه که وقتی که تک پروسسور داریم مشکلی پیش نمیاد در صورتی که حتی اگر تک پروسسور هم داشته باشیم هم باز ممکنه مشکل پیش بیاد چرا؟ چون مسئله به این برمیگرده که ما اون خط کدی که توی برنامه نوشتیم ینی counter++ یا counter-- است وقتی که می خواد باینریش تولید میشه و کامپایل میشه اسمبلی میشه و بعد اسمبلی میاد باینری میشه و توی اسمبلی یک عملیات جمع به یک همچین چیزی تبدیل میشه که توی اسلاید روبرو نوشته ینی + ینی اون مقداری که توی حافظه است باید بره توی رجیستر cpu و بعد رجیستر cpu هست که میتونه زیاد بشه و بعد رجیستر cpu ذخیره بشه توی اون قسمت حافظه

حالا فرض میکنیم یک cpu داریم و زمان بندی این دوتا پروسس توی این cpu به این صورت است ینی * : ینی به نوعی این ها زمان بندی میشن که به این ترتیب این خط ها بینابین هم اجرا میشن پس با وجود یک cpu هم اجرای همزمان این دوتا ینی تایم شیرینگشون باعث ایجاد مشکل شد

Race Condition

- Processes P_0 and P_1 are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable `next_available_pid` the same pid could be assigned to two different processes!

-
یک نمونه دیگه ای که توی کرنل اتفاق می افته تا مسئله Race Condition رو بتونیم ببینیم:
Race Condition موقعی است که برای تغییر یک ریسورس یا اپدیت یک ریسورس مشترک بین دوتا پروسس یا دوتا تردی که همزمان دارن اونو اپدیت می کنن یک رقابتی پیش بیاد و اگر ما رقابتشون رو در نظر نگیریم و نخوایم ترتیب اجرا بهشون بدیم باعث میشه که data consistency پیش بیاد و اون چیزی که مد نظرمون هست دیتا اپدیت نشه
اینجا مثال کرنل رو می بینیم : جایی که کرنل میخواد فورک بکنه:
اگر توی اون سرویس روتین فورک همزمانی پیش بیاد ینی توی دوتا پروسس همزمان فورک فراخوانی بشه و باعث بشه توی پروسس اول که فورک ایجاد شده و تا اون خطش که یک pid جدید بخواد بهش داده بشه انجام بشه مثلا pid قبلی که داشتیم مثلا 2614 بوده و p0 که فورک میشه کرنل بهش 2615 رو میده
اینجا برای p0 , p1 اگر همزمان pid جدید بخواد ریکوست بشه همزمان اون available_pid جدید یک مقدار مشترک خواهد داشت برای این دوتا پروسس و برای هر دوی این ها اون مقداری که برگردونده میشه برای اون پروسس جدید یک مقدار واحد است و اینجا به مشکل می خوریم چرا؟ چون کد فورک که داشت اجرا میشد همزمانی توی اجرای این کد فورک اتفاق افتاد

پس همونطوری که توی برنامه های سطح یوزر ممکنه همزمانی پیش بیاد و مشکل ایجاد بشه توی برنامه های سطح کرنل هم اجرای همزمان دوتا دستور کرنل می تونه باعث ایجاد مشکل بشه

Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

-
مسئله Critical Section:
فرض کنید n تا پروسس توی سیستم داریم از p0 تا pn-1 و هر کدوم این ها هم یک قسمت Critical دارن و منظورم از Critical چیه؟ اون جایی هست که یک common variables داره اپدیت میشه بین این پروسس ها حالا میتونه یک متغییر باشه یا یک جدول مشترک باشه یا یک فایل باشه و... به هرحال چندتا از این پروسس ها بخوان همزمان یک فایل مشخص یا یک جدول مشخص یا یک متغییر مشخص رو تغییر بدن اون قسمتی از کد که این تغییرات توش اعمال میشه بهش میگیم critical section
همیشه مسئله همزمانی برای ما پیش نمیاد بستگی به نحوی اجرای این پروسس ها داره
اگر p0 , p1 , p2 داشته باشیم و هر سه تاشون هم یک critical section داشته باشند که دارن یک متغییر واحدی رو توش تغییر میدن در این صورت اگر همشون همزمان وارد این critical section نشن ما مشکلی نداریم
بنابراین مسئله critical section تعریف شده که ما با حلش مطمئن بشیم که این همزمانی پیش میاد ینی در یک زمان واحد حداکثر فقط یکی از این پروسس ها توی critical section شون قرار میگیرن پس ما می خوایم مسئله رو به نوعی اجرا بکنیم که به هر ترتیبی این n تا پروسس ها اجرا بشن توی cpu یا cpu ها مطمئن بشیم که موقعی اجرا توی یک لحظه واحد فقط یکیشون توی قسمت critical section است و بقیه دارن کدهایی رو اجرا می کنن که خارج از critical section شون هستن
به اون نقطه ای که وارد critical section میشیم entry section و به اون نقطه ای که داریم خارج میشیم می گیم exit section
و بقیه کد رو ینی کدهایی که داخل critical section نیست رو میگیم remainder section

Critical Section

- General structure of process P_i

```
while (true) {
        entry section
            critical section
        exit section
            remainder section
}
```

مثل این:

این while(true) هم به این معنیه که ممکنه این تیکه کد تکرار بشه اجراش

Critical-Section Problem (Cont.)

Requirements for solution to critical-section problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

اگر بخوایم یک راه حلی برای مسئله critical section بدیم باید چه ویژگی هایی داشته باشه؟
1- Mutual Exclusion یا انحصار نامتقابل: فقط یکی از این پروسس ها بین n تا پروسس می تونه توی critical section اش باشه و بقیه مطمئنیم که خارج از critical section شون هستن
2- Progress: اگر هیچ پروسسی الان توی critical section اش نیست و یک پروسسی مایله که وارد critical section اش بشه روشی که ما ارائه می کنیم باید اجازه بده که اون پروسس وارد critical sectionاش بشه و ما یه جوری روش رو ارائه نداده باشیم که با این که بقیه پروسس ها نمی خوان وارد critical section شون بشن و الان اون منبع ازاد هست پروسس جدیدمون نتونه وارد critical section اش بشه
3- Bounded Waiting: اینطور نباشه که روش ما جوری نوشته شده باشه که پروسس ها برای وارد شن به critical section شون مقدار زمان نامحدودی رو بخوان صبر کنن برای ورود به critical section پس ما انتظار داریم که هر پروسسی اگر مایل بود وارد critical section بشه یک زمان محدودی رو منتظر بمونه و بعد دیگه یه جوری بهش نوبت و اجازه داده بشه که وارد critical section اش بشه پس باید یک باندی روی اون زمانی که پروسس منتظر می مونه که وارد critical section اش بشه وجود داشته باشه
و این دوتا مورد رو توی این قسمت باید در نظر بگیریم:
1- همه پروسس ها به هر حال اجرا میشن ینی مثلا اگر n تا پروسس داریم به هر حال این ها زمان بندی می شن و یه موقعی هر کدومشون اجرا میشن پس باید یه همچین فرضی رو داشته باشیم وقتی می خوایم Bounded Waiting رو چک بکنیم
2- هیچ فرضی برای اینکه بدونیم به چه نسبت و با چه سرعتی این ها دارن بین هم اجرا میشن یا زمان بندیشون به چه نسبته نخواهیم داشت توی همه حالات باید این Bounded Waiting رعایت بشه مثلا p1 یک پروسه ی با اولویت بالا باشه بنابراین تعداد دفعاتی که اجرا میشه و میخواد وارد critical section اش بشه زیاده و اسکجولر سیستم هم خیلی اینو اسکجولش میکنه توی یک زمان مشخصی ولی مثلا p5 اولویت پایینی داره و گهگاهی اجرا بشه ولی بلاخره اجرا میشه
این شرط سوم به دوتا شرط شکسته میشه: 1- عدم گرسنگی 2- عدم بن بست ینی اگر ما بن بست و گرسنگی نداشته باشیم شرط Bounded Waiting رعایت شده و اگر هر کدوم از این ها پیش بیاد این شرط نقض شده
پس باید 4 تا مورد رو برای مسئله چک بکنیم

Definitions

- **Critical resource** (منبع بحرانی) : a shared resource between more than one threads (processes) that can not be used or updated concurrently by them.
- **Critical section** (ناحیه بحرانی): a section of code for updating a critical resource. Ex: write to printer buffer, updating a table in database, writing to a file
- **Race condition** (شرایط رقابتی): a situation where several processes access and manipulate the same data (a critical resource) concurrently.
- **Synchronization** (هماهنگ‌سازی): Orderly execution of cooperating processes that share a critical resource
- **Deadlock** (بن‌بست): two or more processes are blocked with each other to enter the critical section and progress execution.
- **Starvation** (گرسنگی): a process wait indefinitely for entering the critical section

Critical resource: منظور همون ریسورس شیری است که چندتا پروسس یا ترد دارن همزمان ازش استفاده می کنن یا اپدیتش می کنن
Critical section: اون قسمت کدی است که اون Critical resource رو توی این داریم تغییر میدیم یا داریم ازش استفاده میکنیم
Race condition: اگر اجرای پروسس ها یا تردها به نحوی صورت بگیره که حداقل دوتاشون بتونن همزمان به اون قسمت Critical section برسن اون وقت شرایط رقابتی پیش میاد
Synchronization: راه حلمون برای حل مسئله Critical section است ینی ما میخوایم برای اینکه مطمئن بشیم این ها همزمان وارد Critical section شون نمی شن از همگام سازی استفاده بکنیم ینی به نوعی ترتیب بدیم به اجرای اون پروسس هایی که دارن یک منبع مشترک رو استفاده می کنن و با ترتیب اجرا وارد Critical sectionشون بشن که مطمئن بشیم هیچ وقت همزمان وارد نمیشن
Deadlock: شرایطی پیش بیاد که دو یا تعداد بیشتری پروسس به جایی برسن که میخوان وارد Critical sectionشون بشن ولی ما راه حلمون یه جوری بوده که هیچکدومشون نمی تونن وارد Critical sectionشون بشن و تا ابد توی نقطه entry می مونن در این حالت می گیم بن بست پیش اومده - ممکنه اگر n تا پروسس داریم برای دوتا از پروسس ها این مشکل پیش بیاد در این حالت بازم ما راه حلمون منجر به بن بست شده پس اگر دوتا یا تعداد بیشتری از پروسس ها به نقطه entry برسن و هیچکدوم نتونن وارد Critical sectionشون بشن این حالت بن بست است
Starvation: ینی حالتی پیش بیاد که پروسس p1 , p2 ما مدام از Critical section بتونن استفاده بکنن ولی p3 هیچ وقت نتونه استفاده بکنه اینجا بن بست پیش نیومده ینی جوری نیست که این ها نتونن برن جلو ولی گرسنگی پیش اومده ینی p3 نتونسته ریسورس بگیره ولی p1 , p2 تونستن جلو برن

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** قبضه شدنی یا غیرلنحصاری – allows preemption of process when running in kernel mode
- **Non-preemptive** قبضه نشدنی یا انحصاری– runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode
- preemptive kernels are difficult to design especially in SMP (Symmetric Multi-Processing), but they are more responsive and suitable for real-time

-
یکی از راه حل هایی که عنوان میشه برای حل مسئله Critical-Section راه حلی است برای سیستم عامل فرض کنیم داریم انتخاب می کنیم این راه حل رو ینی برای کرنل سیستم عامل ینی حالتی که ما وقفه ها رو بخوایم غیرفعالش بکنیم
سیستم عامل کلا Interrupt-based است بقیش صفحه بعد...
این صفحه...
کرنل رو دو دسته کرده:
1- یک حالت این که کرنل Preemptive باشه ینی ما اجازه داریم وقتی که یک کدی داره توی کرنل اجرا میشه cpu رو ازش بگیریم و یک کد دیگه ای رو اجرا بکنیم
2- وقتی که داریم وقفه ها رو غیرفعال میکنیم ینی این که داریم کرنل رو Non-preemptive میکنیم پس غیرفعال کردن وقفه ها ینی ما داریم کرنل رو Non-preemptive می کنیم ینی وقتی که یک کدی داره توی کرنل اجرا میشه ما هیچ وقت ازش cpu رو نمی گیریم مگر اینکه خودش بلاک بشه یا خودش cpu رو پس بده ینی اجراش تموم شده
اگر اینطوری باشه شرایط رقابت هچ وقت توی کرنل پیش نمیاد و مسئله ما حل شده
در حالت کلی Preemptive کرنل ها مناسب تر هستند برای وقتی که سیستممون قراره که پاسخگوییش بالا باشه یا ریل تایم باشه اما مشکل اون مدیریت اون Critical-Section رو اینجا خواهیم داشت مخصوصا وقتی که سیستممون SMP است در واقع خیلی سخت میشه مدیریت Critical-Section

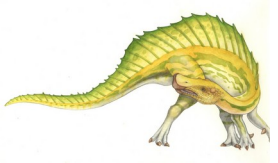
Interrupt-based Solution

- Entry section: disable interrupts
- Exit section: enable interrupts
- Will this solve the problem?
  - What if the critical section is code that runs for an hour?
  - Can some processes starve – never enter their critical section.
  - What if there are two CPUs?

-
ینی اگر ما مطمئن باشیم که وقتی که داره یک تیکه کد اجرا میشه وقفه پیش نمیاد راهی برای گرفتن cpu از این تیکه کد وجود نداره مثلا p1 داره اجرا میشه توی کرنل ما وقفه ای هم نداریم خب این باید اونقدر اجرا بشه که دیگه اجراش تموم بشه ینی دیگه خودش ادامه ای نداشته باشه و رسیده باشه به تهش در این صورت هیچ وقت همزمانی بین دوتا پروسسی یا دوتردی که توی سطح کرنل دارن اجرا میشن پیش نمیاد پس چون همزمانی پیش نمیاد پس مشکلی هم پیش نمیاد پس یکی از راه حل هایی خیلی ساده اینه که ما وقفه هایی سیستم رو وقتی که یک پروسس میخواد وارد critical sectionاش بشه غیر فعال بکنیم یا به طور کلی بگیم وقتی که یک فانکشنی یا یک سیستم کالی توی کرنل داره اجرا میشه همون ابتدا ما کلا وقفه ها رو غیر فعال میکنیم که مطمئن بشیم که فانکشن دیگه همزمان نمی تونه اجرا بشه
مشکل اولیه که این داره اینه که مثلا پروسس p1 برای زمان طولانی بخواد اجرا بشه که این خیلی بده چون کل cpu رو دادیم به این پروسس و هیچ کد دیگه ای توی کرنل نمیتونه اجرا بشه وقتی که این داره اجرا و این باعث میشه که بقیه پروسس ها به گرسنگی بخورن
مسئله بعدی هم اینه که ما دوتا cpu داریم اگر دوتا cpu داشته باشیم این غیرفعال کردن وقفه ینی وقفه های هر دوتا cpu رو غیرفعال بکنیم و اگر مثلا اگر p1 داره توی cpu یک اجرا میشه هیچ کد دیگه ای نباید روی Cpu های دیگه هم بتونن اجرا بشن پس وقفه های اونا هم باید غیرفعال بکنه و برای همچین کاری ارتباطی بین این cpu ها نیازه که خود این زمان بر است و یه مقدار این کار، کار سختی است که ما بخوایم همزمان بخوایم وقفه های همه cpu های سیستم رو غیر فعال بکنیم

Software Solution 1

- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share one variable:
  - **`int turn;`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- initially, the value of **`turn`** is set to *i*



-
راه حلی که تا اینجا گفتیم یک راه حل سخت افزاری بود البته مسئله رو حل میکرد و البته می تونست تا حدی گرسنگی ایجاد بکنه و پیاده سازیش هم سخت بود و پاسخگویی سیستم رو هم میاورد پایین
ایا می تونیم ترفند هایی توی نرم افزار داشته باشیم که اونجا جلوگیری بکنیم از پیش اومدن critical section و یه جورایی توی کدی که داریم می نویسیم ینی همون جایی که کدهای برنامه رو می نویسیم که ممکنه توی شرایط رقابتی قرار بگیرند اونجاها یه جوری کدشو بنویسیم که بهشون ترتیب اجرا بدیم ینی برای ورودشون به critical section یه جوری ترتیب اعمال بکنیم که مطمئن بشیم همزمان وارد اون critical section نمیشن و دوتا شرط دیگه ای که برای مسئله critical section رو هم گفتیم رو دارن
یک متغییر turn تعریف میکنیم توی کدمون و با استفاده از ست کردن مقدار turn سعی میکنیم جوری این turn رو مقدار دهی بکنیم که مطمئن بشیم فقط یکی از پروسس ها وارد critical section میشه

Algorithm for Process P_i

```
while (true){

    while (turn = = j);

    /* critical section */

    turn = j;

    /* remainder section */

}
```

این روش نایس برخورد میکنه ینی همیشه نوبت رو میده به پروسس مقابل مثلا اگر ما پروسس pi , pj رو داشته باشیم پروسس pi همیشه قبل از ورود به critical section چک میکنه که نوبت j هست یا نه و اگر نوبت j بود وارد critical section نمیشه و توی حلقه چک کردن j می مونه اینقدر تا اون یکی پروسس از critical section در بیاد و بعد این یکی بتونه وارد بشه
همین کد رو برای پروسس j داریم و برای turn==i داره صبر میکنه
بعد critical section هم توی نقطه exit یک turn=j گذاشتیم ینی اگر پروسسی وارد critical section اش شد و کدهاشو اجرا کرد توی این قسمت و چون دیگه کاری نداره توی critical section نوبت رو میده به طرق مقابل و اگر طرف مقابلی توی این خط while اش مونده باشه می تونه رد بشه و وارد critical sectionاش بشه
همین کدو که این طرف برای i داریم اون طرف هم برای j هم داریم
هیچ وقت هر دو نمی تونن همزمان وارد critical section بشن چون این turn یا برابره با i است یا j منظورم توی حلقس
چک کردن شرط Progress: فرض کنید pi اومده باشه و رفته باشه توی critical section اش و pj هم اصلا توی این حلقه اولی که while(true) نیست یا قبلشه یا بعدشه در کل pj اصلا نمیخواد الان وارد critical section اش بشه در این حالت pi وارد critical section اش میشه و بعد از اینکه خارج شد turn=j میکنه ینی نوبت رو می ده به j و pj نمی خواد وارد critical section اش بشه پس کاری باهاش نداریم و pi میاد remainder.. اش رو اجرا میکنه و برمیگرده و دوباره ممکنه برای دفعه بعدی بخواد وارد critical section اش بشه اینجا چک میکنه و می بینه turn==j هست یا نه که خودش قبلا مساوی j کرده ولی اون یکی پروسس نخواسته اصلا استفاده بکنه پس این پروسس pi اینجا توی این حلقه می مونه ینی turn==j و هر چی هم صبر بکنه دیگه هیچ وقت از این خط نمی تونه رد بشه چون pj دوست نداشته که وارد critical section بشه و بعد نوبت رو بده به pi پس روش ما توی این حالت مانع ورود pi به critical section میشه
پس کلا روشمون روش درستی نیست
چک کردن شرط Bounded Waiting: بن بست هیچ وقت پیش نمیاد - گرسنگی نداریم چون نوبت داره جابه جا میشه همش پس گرسنگی هیچ وقت نداریم

Correctness of the Software Solution

- Mutual exclusion is preserved

  P_i enters critical section only if:

  `turn = i`

  and `turn` cannot be both 0 and 1 at the same time
- What about the Progress requirement?
- What about the Bounded-waiting requirement?

-
نکته:
خود روش داره از یک متغییر شیری باز استفاده میکنه مثلا turn اینجا شیر بود بین دوتا پروسس و تغییر مقدار turn دوباره اینجا باید حواسمون بهش باشه که مسئله همزمانی رو نداشته باشه

Peterson's Solution

- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
  - **`boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section.
  - **`flag[i] = true`** implies that process $\mathbf{P_i}$ is ready!

-
راه حل Peterson:

Algorithm for Process P_i and P_j

P_i

```
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
     critical section
   flag[i] = false;
     remainder section
} while (true);
```

P_j

```
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
     critical section
   flag[j] = false;
     remainder section
} while (true);
```

روش پترسون:
این میاد مشکل روش قبلی رو حل میکنه ینی Progress نداره
اینجا بیایم چک بکنیم که طرف مقابل دوست داره که وارد critical section بشه یا نه و اگه دوست نداشت بیایم نوبتی که به اون دادیم رو پس بگیریم برای این کار پترسون میاد از یک فلگ استفاده میکنه
هر پروسسی یک فلگ برای خودش داره و اگه فلگ خودش رو true کرد ینی دوست داره که وارد critical section اش بشه و اگه False بود ینی دوست نداره
پس اینجا مثل همون روش قبلی است فقط یک فلگ داریم اینجا
وقتی که یک پروسسی critical section اش رو تموم کرد میاد فلگش رو False می کنه که یک پروسس دیگه نمونه پشت critical section خودش
شرط اول: برقرار است چون ما اینجا داریم turn رو چک میکنیم و turn مقدارش یا i است یا j پس فقط یکی از این پروسس ها وارد critical section اش میشه
شرط دوم: با اضافه کردن این فلگ progress حل شده پس برقرار است
شرط سوم: بن بست نداریم بخاطر این متغییر turn حالتی پیش نمیاد که دوتایشون پشت این while داخلی بمونن چون turn بلاخره یا i است یا j پس بن بست نداریم
گرسنگی :
برای چک کردن گرسنگی یکی از کارهایی که باید بکنیم اینه که سرعت اجرای این دوتا پروسس رو به نسبت هم تغییر بدیم ینی اسکجولر هی به یکیشون نوبت بده و به یکی دیگه نده و تک پروسسور هم در نظر بگیریم و کار کنیم چون اگر دو تا پروسسوره کار کنیم معمولا گرسنگی پیش نمیاد چون به محض اینکه پروسس اول از critical section خارج شد پروسس دوم می تونه کدش رو اجرا بکنه و وارد بشه پس وقتی که می خوایم شرایط گرسنگی رو پدید بیاریم با یه دونه پروسسور کار بکنیم و تایم شیرینگ بین این دوتا داشته باشیم و به یکی هی بیشتر زمان بدیم توی این مثال گرسنگی هیچ وقت نداریم اگه چک بکنیم چون اگر این پروسس یک وارد critical section اش بشه و بعد خارج بشه و فلگش رو False بکنه و j هم بخواد وارد بشه ینی فلگ j اینجا true میشه و وقتی که پروسس i به حلقه می رسه و چون نوبت j میشه cpu به اون داده میشه پس در هیچ حالتی اینجا ما گرسنگی نداریم مثلا زمانی گرسنگی داریم که j بخواد وارد بشه ولی هیچ وقت بهش cpu داده نشه

Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     P_i enters CS only if:

     either **`flag[j] = false`** or **`turn = i`**
  2. Progress requirement is satisfied

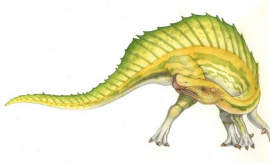
     after critical section flag[i] set to be false
  3. Bounded-waiting requirement is met

     it is achieved through setting turn correctly
- Peterson's solution is not guaranteed to work on modern computer architectures
  - Because of reordering read and write operations that have no dependencies

-
توی کامپیوترهای مدرن امروزی بازم یک مسئله ای داریم که این روش نرم افزاری پترسون هم ممکنه دچار مشکل بشه اون مسئله reordering read and write operations است مثالش صفحه 24..

Peterson's Solution and Modern Architecture

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures.
  - To improve performance, processors and/or compilers may reorder operations that have no dependencies
- Understanding why it will not work is useful for better understanding race conditions.
- For single-threaded this is ok as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!



-
پس مسئله reordering این است که توی سیستم های مدرن برای اینکه پرفرمنس بالا بره گاهی اوقات پروسسور یا کامپایلر میاد دو تا خطی که اجراشون توی حافظه ربطی بهم نداره رو جابه جا اجرا میکنه
این reordering توی یک پروسسور سینگل ترد مشکلی ایجاد نمیکنه ولی اگه مالتی ترد باشه می تونه مشکل ایجاد بکنه که مسئله های ما توی critical section هم مسائل مالتی ترد است

Modern Architecture Example

- Two threads share the data:

```
boolean flag = false;
int x = 0;
```

- Thread 1 performs

```
while (!flag)
;
print x
```

- Thread 2 performs

```
x = 100;
flag = true
```

- What is the expected output?

100

توی کامپیوترهای مدرن یک اتفاقی که می افته اینه که اون خطوطی که داره اجرا میشه توی cpu و اگر مربوط به دسترسی به حافظه باشه اون خطوط اگر اون حافظه که داره بررسی میشه مثلا اینجا داره مقدار فلگ رو اپدیت میکنه و بعد مقدار x را و x و فلگ دوتا مقدار متفاوت هستند ینی توی مموری بهم ربطی ندارند در این حالت cpu ممکنه به دلایلی این دوتا خط رو جابه جا بکنه و اجرا بکنه اگر این اتفاق بیوفته توی برنامه های مالتی ترد از جمله جاهایی که شیرد مموری داریم ممکنه که دچار مشکل بشیم حتی اگر راه حل نرم افزاری مثل پترسون رو به کار برده باشیم
بقیش صفحه قبلی..

مثال:
ترد یک داره یک فلگی رو چک میکنه و وقتی که فلگ true شد می تونه به خط print x برسه و بعد x رو 100 می کنه و بعد فلگ رو true میکنه
اگر reordering پیش بیاد مقدار x چند میشه؟ ینی جای x=100 با جای flag = true می تونه جابه جا بشه در این حالت اگر ترد دو در حال اجرا بوده و فلگش رو true بکنه و بعد cpu قبل از اینکه x بیاد و 100 بشه به ترد یک داده بشه در این حالت حلقه رو رد میکنه و مقدار x رو برابر با صفر چاپ میکنه و بعد cpu داده میشه به ترد دوم و مقدار x رو 100 می کنه در حالی که ما انتظار داشتیم جواب نهایی خروجی بشه 100 نه صفر پس reordering اینجا باعث شد که توی اون مقدار ها یک مشکلی به وجود بیاد وقتی که مالتی ترد است

Modern Architecture Example (Cont.)

- However, since the variables `flag` and `x` are independent of each other, the instructions:

```
flag = true;
x = 100;
```

for Thread 2 may be reordered

- If this occurs, the output may be 0!

Reordering problem for peterson

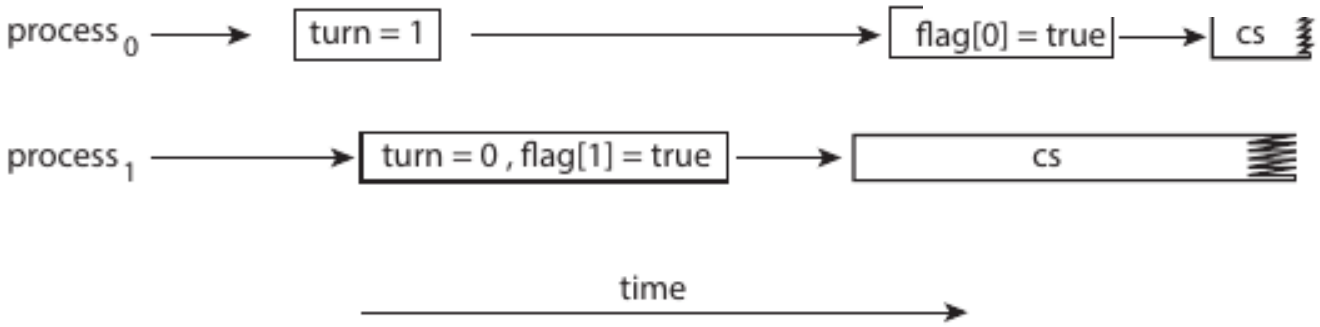
```
boolean flag = false;
int x = 0;
```

Thread 1

```
while (!flag)
   ;
print x;
```

Thread 2

```
x = 100;
flag = true;
```



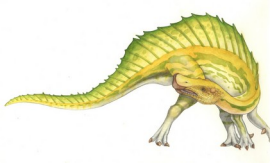
This allows both processes to be in their critical section at the same time!
To ensure that Peterson's solution will work correctly on modern computer architecture we must use **Memory Barrier**.

-
مثلا reordering توی روش پترسون:

Memory Barrier

How a computer architecture determines what memory guarantees it will provide to an application program is known as its **memory model**

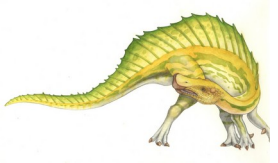
- Memory models may be either:
  - **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
  - **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors.
- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors.



-
یکی از روش های حل مسئله Reordering امکانی که توی سیستم ها قرار دادن که بهش Memory Barrier گفته میشه
سیستم ها از لحاظ مدل مموری دو نوع می تونن باشن:
Strongly ordered: ینی سیستم هایی که سیمتریک مالتی پروسسور هستند و تغییراتی که هر پروسسور روی حافظه انجام میده اگه سریعا توسط پروسسورهای دیگه قابل دیدن باشه بهش میگیم Strongly ordered ینی خود سیستم داره سیاستی رو اعمال میکنه که هر تغییری توی مموری با استفاده از یک cpu رخ داد cpu های دیگه هم فوری این تغییر رو ببینن
Weakly ordered: وقتی هست که تغییرات مموری از یک پروسسور ممکنه بلافاصله توسط پروسسورهای دیگه درک نشه و اگر یک سیستمی Weakly ordered باشه ممکنه در بعضی مواقع مشکلاتی ایجاد بشه و راه حلی که براش ارائه می کنن Memory Barrier هست
Memory Barrier در واقع مجبور میکنه سیستم رو که تغییراتی که توی مموری هرجا انجام میشه به بقیه پروسسورها حتما اعلام بشه و منتشر بشه

Memory Barrier Instructions

- When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed.
- Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.



-
بنابراین اگر ما یک دستور Memory Barrier داشته باشیم و یک دستور Memory Barrier اجرا بشه سیستم می تونه مطمئن باشه که load and stores ها به صورت کامل قبل از اینکه بریم سراغ قسمت دیگه ای از مموری انجام میشه
پس اگر ما Memory Barrier داشته باشیم می تونیم مطمئن باشیم که هر loads and stores که داریم کامل انجام میشه و بعد loads and stores بعدی انجام میشه
پس حتی اگر دستورها reordered بشن Memory Barrier تضمین میکنه که عملیات store کامل انجام میشه توی مموری و توسط بقیه پروسسورها هم بلافاصله بعد از اینکه store انجام شد قابل دیدن است قبل از اینکه loads and stores آپرشن های دیگه انجام بشه

Memory Barrier Example

- Returning to the example of slides 6.24 - 6.25
- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
 memory_barrier();
print x
```

- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true
```

- For Thread 1 we are guaranteed that that the value of `flag` is loaded before the value of `x`.
- For Thread 2 we ensure that the assignment to `x` occurs before the assignment `flag`.

-
مثال:
اینجا قبل از دسترسی به X اگر ما دستور Memory Barrier بذاریم اینطوری مطمئن میشیم که تا این خط دستور Memory Barrierهر کدی که داریم loads and stores کامل شده و بعد می ره سراغ بعدی پس اینجا دیگه اون reordered اعمال نمیشه به طوری که مشکلی ایجاد بکنه یا توی قسمت پایین اول مقدار x به طور کامل ذخیره میشه و بعد دستور بعد loads and stores انجام میشه بخاطر این Memory Barrier که اینجا گذاشتیم

Check Critical section requirements

```
P1:
while ( true )  {
        flag [1] = true ;
        turn = 1 ;
        while ( flag [2 ] and turn=2);
        <Critical - Section >
       flag [1] = false ;
       < remainder >
}
```

```
P2:
 while ( true ) {
       flag [2 ] = true ;
       turn = 2 ;
        while ( flag [1] and turn=1);
              < Critical - Section >
        flag [2 ] = false ;
       < remainder >
  }
```

تمرین:
هر سه تا شرط رو برای مثال چک بکنیم

این شرط اول رو نقض میکنه درست میگم؟؟؟؟

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- We will look at three forms of hardware support:

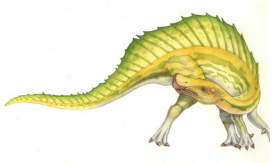
1. Hardware instructions
2. Atomic variables

-
روش های که در اختیار دولوپرها قرار گرفته برای حل مسئله critical section که همه اون ها مبتنی بر روش های سخت افزاری هست
قبلا یک روش گفتیم: گفتیم که توی سیستم های تک پروسسوره می تونیم وقفه ها رو غیرفعال بکنیم توی اون قسمت critical section یا وقتی که یک کد کرنل داره اجرا میشه وقفه ها غیرفعال بشه و به این صورت مطمئن میشیم اون قسمت کد کامل اجرا خواهد شد و بینابین دستوراتش کد دیگه ای نمی تونه اجرا بشه توی سیستم های تک پروسسوره اعمال چنین روشی راحت بود ولی اگر سیستم مالتی پروسسور بود این کار موثر نبود و overhead داشتیم برای غیرفعال کردن وقفه های cpu های دیگه پس این روش توی هر نوع سیستمی قابل استفاده نیست
پس روش های دیگه ای ارائه شد:

1. Hardware instructions
2. Atomic variables

Hardware Instructions

- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or to *swap* the contents of two words atomically (uninterruptedly.)
  - **Test-and-Set** instruction: Either **test** memory word and **set** value
  - **Compare-and-Swap** instruction: Or **swap contents** of two memory words



-
دستورات سخت افزاری که در واقع سیستم ها در اختیار ما قرار میدن که از طریقش بتونیم قسمت بحرانی رو مدیریت بکنیم:
Test-and-Set:
Compare-and-Swap:
مطمئن هستیم که وقتی بدنه ی هر کدوم از این دستورات ینی Test and set و compare and swap داره اجرا میشه اون دستور دیگه یه جوری مثل یک بلاکه و uninterruptedly ینی این test and set دیگه مثل یک فانکشنی نیست که مثلا ممکنه که وسط اجرای یک فانکشن وقفه بخوره و پروسس دیگه ای اجرا بشه مطمئنیم که کل این بدنه Test-and-Set بدون وقفه اجرا میشه و هیچ دوتا Test-and-Set روی یک متغییری نمی تونن همزمان اجرا بشن و برای Compare-and-Swap هم به همین صورت

The test_and_set Instruction

- Definition

```
boolean test_and_set (boolean *target)
   {
          boolean rv = *target;
          *target = true;
          return rv:
   }
```

- Properties
  - Executed atomically
  - Returns the original value of passed parameter
  - Set the new value of passed parameter to `true`

-
مثال:
یک مقدار بولین میگیره و بعد سعی میکنه مقدار این بولین رو true کنه
اگر مقدارش قبلا توسط یک پروسس دیگه ای true شده بود ینی قبلا این test_and_set یک جایی فراخوانی شده بود برای همین متغییر و این متغییر رو trrue کرده بوده می خوایم بفهمیم این اتفاق افتاده است یا نه؟ برای همین مقدار قبلی این متغییر رو نگهداری میکنیم توی یک temp و اون رو ریترن می کنیم توسط این فانکشن test_and_set برای همین همیشه این مقدار target رو true می کنیم
این ریترن کردن به چه دردی می خوره؟ چون ما بعد از فراخوانی این تابع test_and_set می تونیم چک بکنیم این مقداری که ریترن شده True است یا نه و اگر مقدارش true باشه می فهمیم قبلا کسی اینو true کرده بوده و این ما نبودیم که اینو true کردیم پس بخش بحرانی رو نمی گیریم پس می تونیم اینطوری در نظر بگیریم که این مثل یک قفلی است که دست ما است و قبلا اون متغییره قفل شده توسط کس دیگه ای پس ما دیگه نمی تونیم اینجا کاری بکنیم
ولی اگر این مقدار false بود پس مقدار rv میشه false و ما مقدار target رو true میکنیم و مقداری که این فانکشن برمی گردونه false است و می فهمیم این ماییم که این متغییر رو true کردیم
ویژگی هاش:
این قطعه کد به صورت اتمیک اجرا میشه --> اتمیک ینی همون بدون وقفه
original value اون پارامتری که بهش دادیم رو بهمون برمی گردونه برای اینکه بفهمیم این جدید True شده یا قبلا کسی اینو true کرده
و همیشه هم ما میخوایم این متغییر رو true بکنیم با این فانکشن

Solution Using test_and_set()

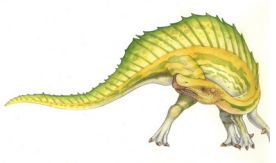
- Shared boolean variable **`lock`**, initialized to **`false`**
- Solution:

```
do {
        while (test_and_set(&lock))
        ; /* do nothing */

            /* critical section */

    lock = false;
            /* remainder section */
 } while (true);
```

While(lock==true);
lock=true;

- Does it solve the critical-section problem?

This solution does not satisfy bounded waiting: starvation



-
فرض کنید یک متغییر lock می خوایم داشته باشیم که این شیر است بین پروسس هایی که میخوان وارد یک بخش بحرانی بشن
اگر کسی این lock رو خودش true کرد می تونه وارد بخش بحرانی بشه و تا زمانی که true است کس دیگه ای نمی تونه وارد بخش بحرانیش بشه بنابراین مقدار اولیه این متغییر باید false باشه
پس هر کسی می خواد وارد بخش بحرانیش بشه این lock رو true میکنه و کسی هم که میخواد وارد بشه باید چک بکنه که قبلا این True نشده باشه --> کد ابیه میشه
و وقتی که وارد بخش بحرانی شد بعد بعدش lock رو false بکنه و اینجا که False اش کرد اگر کسی دیگه ای پشت این حلقه while موند بود می تونه وارد بخش بحرانی بشه
پس روش ما به طور کلی با استفاده از یک متغییر lock به این صورت است
البته این روش ساده یک مشکلی داره: همزمان اگر توی دوتا پروسس این اجرا بشه دوتا پروسس می تونن همزمان بعد از اینکه lock ما false بود توی حلقه lock رو چک بکنن و هر دوشون می بینن lock ما False است پس از حلقه رد می شن و lock رو true می کنن و فکر می کنن خودشون true کردن و وارد بخش بحرانیشون می شن هر دو --> پس این روش دقیق و درستی نیست
ادامش صفحه بعدی...

حالا میخوایم این روش رو درست بکنیم: با استفاده از test_and_set این کارو میکنیم --> اینجا به جای اینکه این شرط رو روی lock با استفاده از حلقه چک بکنیم از یک test_and_set استفاده میکنیم به این صورت که: تا زمانی که lock <-- test_and_set ما true هست صبر کن این ینی اگر کسی قبلا lock رو true کرده ما باید منتظر بمونیم و نمی تونیم وارد بخش بحرانی بشیم --> اینجا دیگه مطمئن هستیم این دستور test_and_set روی lock فقط توسط یک پروسس در یک لحظه اجرا میشه ینی حتی اگر سیستمون چند پروسسور باشه این test_and_set روی این lock نمی تونه روی دوتا پروسسور همزمان اجرا بشه پس این دیگه مثل اون حلقه while نیست و مشکل قبلی رفع میشه
بعد به محض اینکه یک پروسسی بخش بحرانیش تموم شد lock اش رو false میکنه و به محض اینکه false شد یکی از این پروسس هایی که داره این تابع test_and_set رو فراخوانی میکنه موفق میشه که lock رو خودش true بکنه و مقدار قبلیش هم که False بوده که ریترن میشه پس از این حلقه خارج میشه
ایا این کامل کامل داره درست کار میکنه یا نه ؟
از لحاظ شرط اول اره --> چون بلاخره مقداری که حلقه برمیگردونه یا false است یا true و تا وقتی که true است کس دیگه ای نمی تونه وارد بخش بحرانی بشه
شرط دوم هم برقراره
شرط سوم: در مورد مسئله بن بست ینی دو تا پروسس پشت حلقه گیر بکنند و نتونن رد بشن هیچ وقت پیش نمیاد این حالت بخاطر اینکه فقط یک پروسس است که میتونه این lock رو true بکنه و این همیشه یک مقداری رو برمیگردونه و هیچ وقت شرایطی پیش نمیاد که دوتا پروسس اینجا بمونن و هیچ وقت رد نشن
ادامشش...

مسئله گرسنگی: ینی شرایطی پیش میاد که اگر ما دوتا پروسس داریم یکی از پروسس ها مدام ازاین بخش بحرانی استفاده بکنه و یکی دیگه نتونه وارد بشه --> توی سیستم تک پروسسوره اینو بررسی میکنیم--> توی یک سیستم تک پروسسوره فرض می کنیم یک پروسسی وارد بخش بحرانیش شده وپروسس دوم توی این حلقه اولش مونده و چون تک پروسسوره هستیم و p1 داخل بخش بحرانیش است هیچی از p2 اجرا نمیشه ینی حتی نمی تونه این مقدار test_and_set رو چک بکنه --> بخش بحرانی p1 تموم می شه و کامل اجرا میشه می رسه به این خطی که باید lock رو false اش بکنه ولی هنوز lock رو false اش نکرده بعد cpu رو میدیم به p2 و p2 الان چک میکنه این lock ما True است یا False است و می بینه این true است و اجرا نمیشه و هرچقدر هم این test_and_set اجرا بکنه توی این حلقه ینی هی cpu داریم بهش میدیم که این test_and_set رو چک بکنه بازم نمیتونه وارد بشه و یک عالمه بار اینو اجرا میکنه و وقتی که تایم اسلایس این پروسس تموم شد باید cpu رو پس بده و cpu پس داده میشه به p1 و p1 میاد lock رو false میکنه و ممکنه که دوباره بیاد سر خط while و بخواد دوباره وارد بخش بحرانیش بشه و همچنان cpu دستش است و چون مقدار قبلی lock ما False است می تونه وارد بشه و بره توی بخش بحرانی و دوباره بدیم cpu به p2 و دوباره مثل قبل است این پس گرسنگی داریم اینجا
علت این بخاطر نوع پیاده سازی است که اینجا داریم ینی ما cpu را اینجا مشغول کردیم در حالتی که توی حالت wait هستیم و هیچ نتیجه خوبی نداره این کار

The compare_and_swap Instruction

- Definition

```
int compare_and_swap(int *value, int expected, int new_value)
 {
   int temp = *value;
   if (*value == expected)
       *value = new_value;
    return temp;
 }
```

- Properties
  - Executed atomically
  - Returns the original value of passed parameter **`value`**
  - Set the variable **`value`** the value of the passed parameter **`new_value`** but only if **`*value == expected`** is true. That is, the swap takes place only under this condition.

Intel x86 instruction
cmpxchg <destination operand>, <source operand>

-
compare_and_swap هم خیلی شبیه test and set است با این تفاوت که کاری که داره انجام میده اینه که مقدار یک متغییری رو میخواد عوض بکنه ینی value با مقداری که ما داریم مشخص میکنیم ینی new_value ولی فقط در صورتی این new_value رو ست میکنه که مقدار value قبلا با این مقدار expected برابر باشه ینی در صورتی مقدار value رو تغییر میده که به مقدار new_value که مقدار قبلی value برابر باشه با expected
ازادی عمل با این بیشتر است
شرایط مثل قبلی است ینی اتومیک است و مقدار original value متغییر value هم ریترن میشه و فقط در صورتی مقدار Value اپدیت میشه که مقداری که قبلاش بهش دادیم توی این value قرار گرفته باشه
نمونه این دستور هم توی intel x86 است

Solution using compare_and_swap

- Shared integer `lock` initialized to 0;
- Solution:

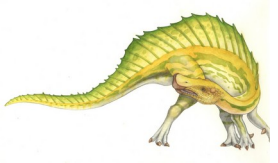
```
while (true){
        while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

        /* critical section */

        lock = 0;

    /* remainder section */
}
```

- Does it solve the critical-section problem?

-
مثال:
مقدار قبلی lock رو صفر می ذاریم که معادل اینه که اونجا lock رو false می گذاشتیم و میگیم این lock رو یکش بکن در صورتی که قبلا صفر بوده و مقداری هم که برمی گردونه این تابع مقدار قبلی lock است که اگر صفر بود از حلقه رد میشه و تا وقتی که صفر نیست توی حلقه می مونه
منطق این مثل روش test and set است پس این روش هم همون مشکل گرسنگی رو داره

Bounded-waiting with compare-and-swap

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      key = compare_and_swap(&lock,0,1);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = 0;
   else
      waiting[j] = false;
   /* remainder section */
}
```



برای حل مسئله که باعث گرسنگی نشه می تونیم یه کاری رو انجام بدیم:
هر جا که گرسنگی بود برای اینکه گرسنگی نباشه و زمانی که یک ریسورس محدودی داریم باید هی به نوبت این ریسورس رو به بقیه بدیم پس اینجا باید یک جوری نوبت اعمال کنیم
روش:
اگر n تا پروسس داشته باشیم هر دفعه نوبت به یکیشون برسه مثلا اگر n تا داریم از 0 تا n نوبت بدیم و اگر صفر اجرا شد دیگه دفعه بعدی نتونه اجرا بشه اگر مثلا 1 یا بقیشون می خواستن وارد بشن و این کار رو با یک waiting[i] مشخص میکنیم مثل همون روش پترسون است
میایم waiting[i] رو true میکنیم و اگر این ها هر کدوم مایل بودن و نوبتشون رسید اجازه میدیم وارد بشن و این lock رو با همون compare-and-swap داره استفاده میکنه تا مسئله اش از طریق سخت افزار حل شده باشه
و نوبت هم به صورت چرخشی میده

Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and booleans.
- For example:
  - Let **`sequence`** be an atomic variable
  - Let **`increment()`** be operation on the atomic variable **`sequence`**
  - The Command:

```
increment(&sequence);
```

    ensures **`sequence`** is incremented without interruption:

-
روش دیگه سخت افزاریه که وجود داره Atomic Variables است
Atomic Variables در واقع امکان محدودتری برای ما است فقط مواقعی که ما جایی نیازمون با increment کردن یک متغییر یا تغییر مقدار یک متغییر حل میشه می تونیم از Atomic Variables ستفاده بکنیم ینی همون متغییری که شیر است بین پروسس های مختلف رو اونو اتومیک تعریفش میکنیم و توی دستور increment(&sequence) مطمئن میشیم که اگر جایی اون متغییر رو تغییر دادیم مثلا تویی این دستور که روی متغییر sequence داره اپدیت صورت میگیره فقط همینی که داره اینو تغییرش میده ینی اگر ترد دیگه ای هم همزمان می تونست این sequence رو تغییر بده چون این sequence یک متغییر اتومیک است اخرش منجر به این میشه که به ترتیب این ها این کار رو انجام بدن اجرای این sequence به طور کامل بدون وقفه است و لود و استورش ممطمئنیم که کامل انجام میشه پشت سر هم و مسئله ای پیش نمیاد
نکته: این Atomic Variables موقعی که می خوایم یک مقداری رو اپدیت بکنیم قابل استفاده است و می تونیم مطمئن باشیم که اپدیت این ها همزمان انجام نمیشه
ولی همچنان یکسری مسائلی رو برای ما حل نمی کنه مثلا توی اون مسئله باند بافر می خواستیم بافر محدود رو توی مسئله تولیدکننده و مصرف کننده حل بکنیم از یک کانتر استفاده میکردیم اینجا می تونیم الان کانتر رو اتومیک تعریف بکنیم پس مطمئن میشیم اون خط هایی که توش کانتر رو زیاد می کردیم یا کم می کردیم اتومیک اجرا میشن ولی باز مشکلی که پیش میاد این است اونجایی که داریم کانتر رو چک میکنیم است ینی دوتا پروسس همزمان می تونن مقدار کانتر رو چک بکنن چون اپدیتش نمی کنن پس ممکنه دوتاشون همزمان وارد بخش بحرانی بشن و بازم مسئله رو نتونستیم اونجا حل بکنیم پس همچنان نیاز داریم بازم یکسری چیزهایی دیگه هم اضافه بکنیم به این روش ها که انواع مسئله های بخش بحرانی رو بتونیم توی شرایط مختلف حل بکنیم

Atomic Variables

- The `increment()` function can be implemented as follows:

```
void increment(atomic_int *v)
{
    int temp;
    do {
        temp = *v;
    }
    while (temp != (compare_and_swap(v,temp,temp+1));
}
```

-
مثال:
تابع increment رو با استفاده از compare_and_swap پیاده کرده ینی atomic variables ها هم می تونن از طریق توابع compare_and_swap یا test and set پیاده بشن ینی ما وقتی که اعلام میکنیم که یک متغییر اتومیک است موقعی که میخوایم اپدیتش بکنیم باید از دستور compare_and_swap سخت افزاری استفاده بکنیم ینی همچنان اون بیسیک ما همون دستورها هستن حتی توی Atomic Variables هم بیسیک ما دستورهای سخت افزاری هستن که اینجا حتی اگر increment هم نبود هر مقدار جدیدی که میخواستیم بذاریم با استفاده از compare_and_swap می تونستیم مقدار جدید بگذاریم روی یک متغییر ولی با استفاده از compare_and_swap هم مطمئن هستیم که همزمان کسی این متغییر v رو تغییر نمیده

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be **atomic**
  - Usually implemented via hardware atomic instructions such as compare-and-swap.

```
do {
    acquire lock
        critical section
    release lock
      remainder section
 } while (true);
```

-
به دو روش خیلی معروفه که دولوپر ها در سطح یوزر هم می تونن ازشون استفاده بکنن تا مسئله بخش بحرانی کدشون رو حل بکنن
روش اول یا Mutex خیلی شبیه همون lock که قبلا گفتیم می خوایم با صفر یا یک کردن بخش بحرانی رو مدیریت بکنیم و فقط یک نفر یکش بکنه
ولی اینجا دوتا تابع داریم:
acquire() and release() و مطمئنیم که این دوتا اتومیک هستن
acquire(): اگر خواستیم که متغییر رو تغییر بدیم باید حتما lock اش رو در اختیار بگیریم که با این تابع این کار رو می کنیم
اگر پروسس دیگه ای این lock رو نگرفته بود می تونیم از acquire() رد بشیم ولی در غیر اینصورت acquire() خودش بلاک میشه و اونقدر می مونه تا یه کسی که قبلا lock رو گرفته اونو release() اش بکنه و پروسس مدنظر بتونه از این قسمت رد بشه
همین دوتا تابع هم باید توسط compare_and_swap پیاده سازی بشن یا دستورهای سخت افزاری
توی این شکل: قبل از اینکه وارد بخش بحرانی بشیم باید lock رو acquire بکنیم و وقتی خواستیم خارج بشیم باید release بکنیم
نکته: اگر دوتا ترد داشتیم که میخواستن یکیشون S1 رو تغییر بده و اون یکی هم S2 رو این دوتا می تونن این کارو بکنن چون روی یک متغییر نیست نکته ای که هست T1 باید acquire اش رو روی S1 بذاره و T2 باید روی S2 بذاره

acquire() and release()

```
acquire() {
    while (!available) ; /* busy wait */
    available = false;
 }
release() {
    available = true;
}
```

-
```
  do {
   acquire lock
      critical section
   release lock
     remainder section
} while (true);
```

کار کلی که انجام میشه توی این دوتا تابع:
acquire می خواد صبر بکنه روی یک مقدار متغییری که اون متغییر مثل lockامون می مونه و میگه اگر اون متغییر در دسترسه و کسی نگرفتش می تونی ازش رد بشی و اگر در دسترس نیست باید صبر بکنی و بعد خودت اونو فالسش بکنی همین کد رو می تونیم با compare-and-swap هم بنویسیم
و توی release هم صرفا مقدار متغییر رو برمی گردونه این دقیقا مثل اون خطی است که lock که به compare-and-swap دادیم رو فالس می کردیم

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait()` operation**

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

یتی به پروسسی اجازه داده نمیشه وارد بخش بحرانی بشه هی باید این شرط حلقه s<=0 رو چک بکنه و برای چک کردن این شرط cpu خرج میشه

- Definition of the **`signal()` operation**

```
signal(S) {
    S++;
}
```

روش دیگه ای که استفاده میشه استفاده از سمافور است
سمافور: می تونه Mutex رو هم بپوشونه ینی سمافور امکانات Mutex رو هم داره ولی یکسری امکانات اضافی تری هم داره
یک جاهایی است که از ریسورسمون بیشتر از یک المان داریم مثلا حافظه کامپیوتر ما اجازه نداریم یک خونه مشخص ازاین حافظه رو دسترسی همزمان به دوتا پروسس بدیم اما وقتی که p1 داره به یک قسمت حافظه دسترسی پیدا میکنه p2 می تونه به یک قسمت دیگه ای دسترسی داشته باشه و این کار هیچ مانعی نداره و ما به تعداد خونه های حافظه می تونه دسترسی های همزمان به این خونه های حافظه وجود داشته باشه چون این ریسورس حافظه ما مقدارش یک نیست مثلا برابر سایز این حافظه است
پس اینجا یک ریسورسی داریم که صرفا اینطوری نیست که در یک لحظه فقط یک پروسس بتونه استفاده بکنه بیشتر از یک پروسس هم می تونه استفاده بکنه که به سایز این ریسورس برمیگرده که چه تعدادی می تونن ازش استفاده بکنن
توی همچین حالتی می تونیم از سمافورها استفاده بکنیم در واقع سمافور مقدار می تونه یک مقدار اینتجر باشه که می تونه بیشتر از یک باشه و ااگر مقدار سمافور یک بود ینی Mutex داریم ینی فقط یک پروسس می تونه در همون لحظه ازش استفاده بکنه ولی اگر دو بود ینی اجازه داریم دوتا پروسس همزمان ازش استفاده بکنن ولی پروسس سوم نمیتونه ازش استفاده بکنه تا اینکه یکی از این p2 or p1 اون ریسورس رو ازاد بکنن تا p3 بتونه از اون ریسورس استفاده بکنه
اینجا هم دو تا تابع داریم:
wait شبیه همون acquire است ینی صبر بکنه تا یکی از این پروسس ها از این ریسورس ازاد بشه و سیگنال شبیه release است ینی این p1 وقتی که می خواست این ریسورس رو پس بده به سیستم release ش کرد و وقتی که این p3 می خواست اون ریسورس رو بگیره wait گذاشت روش حالا اگر ازاد نبود باعث میشه که بلاک بشه روی اون خط و اگر ازاد بود می تونه بگیرتش و رد بشه
بدنه کد داخل اسلاید است --> منطق کار شبیه کدی است که داخل اسلاید است ولی باز همین هم با استفاده از اون compare-and-swap پیاده سازی میکنیم
توی wait همش باید چک بکنه که ببینه s کوچکتر و مساوی صفر هست یا نه اگر کوچکتر یا مساوی بود ینی هیچ ریسورسی از این نوع s نمونده پس باید همین جا wait بکنه ینی پشت این while می مونه از اون طرف اگریک پروسسی قبلا این ریسورس رو گرفته بود ینی s رو گرفته بود و الان دیگه کاری باهاش نداره یک سیگنال فراخوانی می کنه و فراخوانی سیگنال باعث میشه که مقدار این سمافور یکی زیاد بشه و یکی که زیاد بشه سریعا توی wait مون s از صفر به یک تبدیل میشه و پروسسی که wait رو فراخوانی کرده بود می تونه از اون خط رد بشه و مقدار s رو کم بکنه

Semaphore (Cont.)

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can implement a counting semaphore ***S*** as a binary semaphore
- With semaphores we can solve various synchronization problems

-
به طور کلی این سمافور می تونه مقادیر مختلفی داشته اباشه توی حالتی که مقدارش بیشتر از یک باشه بهش میگیم Counting semaphore و محدودیت نداریم که مقدار اولیه اش چند باشه ولی می تونیم اونو با یک ستش بکنیم و این باعث میشه که شبیه mutex عمل بکنه
توی مسائلی که می خوایم با سمافور حل بکنیم وقتی از سمافور داریم استفاده میکنیم حتما باید سمافور رو مقدار دهی اولیه بکنیم چون مهمه که مقدار اولیه اش چنده و وابسته به مقدار اولیه اش است ولی mutex نه mutex همیشه مقدا اولیه اش ازاده و صفره در واقع یا میتونه صفر باشه یا یک که مقدار اولیه اش باید ازاد باشه که صفره ولی سمافور نه ینی کد نویس باید مشخص بکنه که مقدار اولیه سمافور چنده و وابسته به شرایط و ریسورسی است که می خوایم استفاده بکنیم ----->>
<<<<<<<<<<<<<نکته مهمی است >>>>>>>>>>>>>

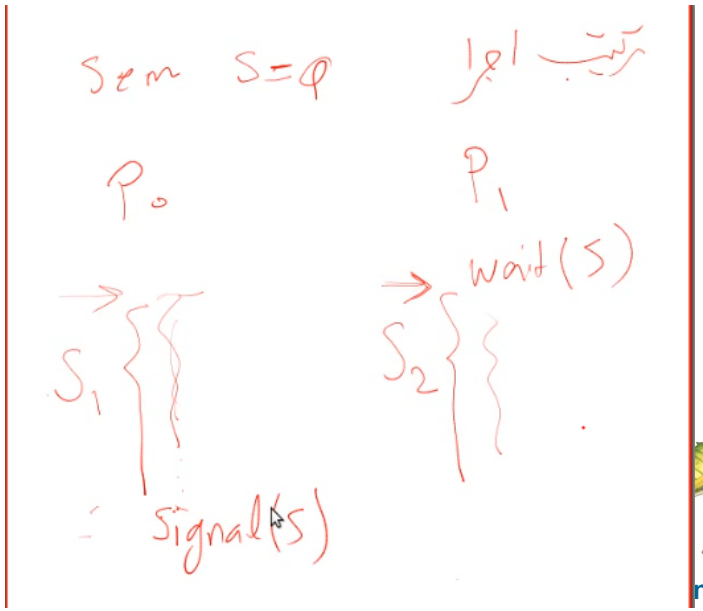
Semaphore Usage Example

- Solution to the CS Problem
  - Create a semaphore "**mutex**" initialized to 1

```
wait(mutex);
    CS
signal(mutex);
```

- Consider P_1 and P_2 that with two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```



-
با استفاده از سمافور ما می تونیم مسائل متفاوتی رو حل بکنیم
از سمافور به جای mutex استفاده بکنیم: +

مثال: فرض کنید می خوایم این مسله رو حل بکنیم که به دوتا کد ترتیب اجرا بدیم ینی مطمئن باشیم که S1 پروسس p0 قبل از اجرای S2 اجرا میشه ینی اگر اسکجولینگ cpu هم به صورتی بود که اول p1 اسکجول شد و رسید به این نقطه اول --> اگر S1 اجرا نشده بود S2 اجرا نشه پس اول باید یه جوری مطمئن بشیم S1 کامل اجرا بشه و بعد S2 اجرا بشه
اینجا بحث بخش بحرانی هم نیست بحث ترتیب اجرا است
باید ترفند هایی به کار ببریم توی کدمون که اگر P1 اول اسکجول شد باعث موندن در ابتدا اون بخش بشه تا P0 اسکجول بشه و S1 اجرا بکنه و بعد بیاد سر S2
راه حل:
با استفاده از سمافور اگر بخوایم اینو حل بکنیم باید یه جورایی از وارد شدن این P1 در اون بخش جلوگیری بکنیم در صورتی که این P0 هنوز وارد نشده باشه
یک سمافوری تعریف کنیم اینجا به اسم S که مقدار اولیه اش رو صفر بدیم (مقدار سمافور رو میتونیم هر مقدار غیر منفی بدیم)
کاری که باید بکنیم اینه که باید قبل از اینکه P1 بتونه وارد اون قسمت بشه یک wait بذاره روی S ولی برای P0 نه ینی اون می تونه بلافاصله اجرا بشه پس نیاز نیست اونجا wait بذاریم ولی باید وقتی که کارش تموم شد باید به P1 خبر بده پس انتهاش باید S رو سیگنال بکنه
استفاده از این سمافور ها به این صورت خیلی دقت می خواد چون اگر درست استفاده نکنیم ممکنه که دچار بن بست بشیم

Example: Mistakes using semaphores

- Let ***S*** and ***Q*** be two semaphores initialized to 1, What happens executing P0 and P1 ?

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

-
مثال:
یک نمونه دیگه از حالت هایی که ممکنه منجر به بن بست بشه با استفاده از سمافورها این قطعه کد است:
اینجا دوتا سمافور تعریف شده که مقدار اولیه هر دوتا یک است ولی P0 اول روی S صبر کرده در حالی که P1 اول روی Q صبر کرده که اینجا مشکلی نیست و هردوتا می تونن عبور بکنن ولی بعدش P0 روی Q صبر کرده و P1 روی S و اتفاقی که اینجا می افته این است که هر دوتاشون اینجا می مونن و کی از اینجا رد میشن در صورتی که برای پروسس مقابلی که wait کرده روی S یعنی P1 یکبار سیگنالش بکنه یعنی پروسس P0 از wait(Q) رد باید بشه که بتونه S رو سگینال بکنه که P1 بتونه رد بشه ایا این اتفاق می تونه بیوفته اصلا؟ خیر چون خودش هم روی Q الان منتظره --> پس هر دوتاشون اینجا بلاک میشن و باعث میشه هیچ وقت نتونن از اون قسمت خارج بشن

Example: Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

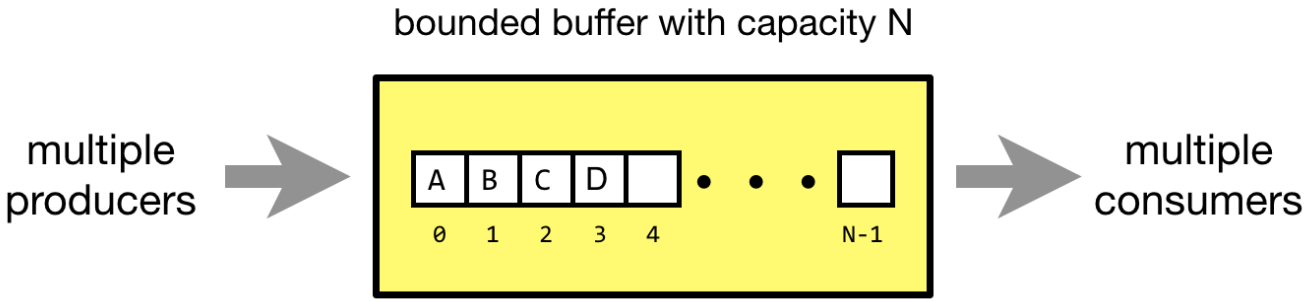
-
مثال:
یک بافری داشتیم که اندازه اش n بود
ومسئله این است که تولیدکننده و مصرف کننده به طور صحیحی از این استفاده بکنند ینی تولیدکننده نتونه بیشتر از n تا توی این بافر قرار بده و مصرف کننده هم اگر این بافر خالی بود چیزی نمیتونه ازش برداره و باید منتظر بمونه تا داخلش پر بشه
راه حل:
سه تا سمافور تعریف میکنیم
یک سمافور empty تعریف میکنیم --> چندتا از خونه های بافر خالی است پس مقدار اولیه اش رو n می ذاریم
یک سمافور full تعریف میکنیم که مقدار اولیه اش رو صفر می ذاریم
یک سمافور mutex تعریف میکنیم که مقدار اولیه اش یک است

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n



+ n بافر، هر کدام می توانند یک آیتم را در خود جای دهند
+ Semaphore mutex به مقدار 1 مقداردهی اولیه شد
+ Semaphore full به مقدار 0 مقداردهی اولیه شد
+ سمافور خالی به مقدار n مقدار دهی اولیه شده است

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true) {
        ...
        /* produce an item in next_produced */

        ...
        /* add next produced to the buffer */
        ...

}
```

Bounded Buffer Problem (Cont.)

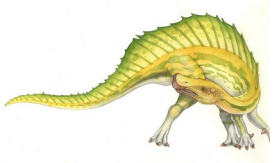
- The structure of the producer process

```
while (true) {
      ...
       /* produce an item in next_produced */
      ...
   wait(empty);
   wait(mutex);
      ...
      /* add next produced to the buffer */
      ...
   signal(mutex);
   signal(full);
}
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

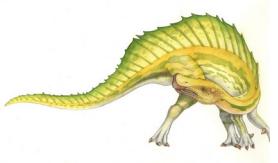
```
while (true) {

    ...
  /* remove an item from buffer to next_consumed */
    ...

    /* consume the item in next consumed */
    ...
}
```



Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {
    wait(full);
    wait(mutex);
        ...
     /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
       /* consume the item in next consumed */
        ...
   }
```



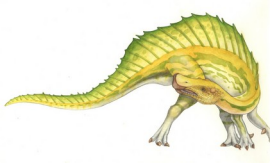
Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
- Could now have **busy waiting** in critical section implementation
  - But implementation code is short
  - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

-
busy waiting ینی جایی که ما cpu رو مشغول یک کار بیهوده ای کردیم ینی یک پروسسه ای که صرفا داره wait می کنه یه جایی و با wait کردنش داره cpu خرج میکنه

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - Value (of type integer)
  - Pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue



-
توی بحث پروسس ها و مدیریت و زمان بندیشون یک صف داشتیم برای پروسسس ها و وقتی یک پروسس می رفت توی حالت I/O یا می خواست منتظر یک وقفه ای باشه می رفت توی صف انتظارها و دیگه ready نبود و اسکجولر هم از صف ready ها یک پروسسی رو انتخاب میکرد و بهش cpu می داد
ازاین نکته می خوایم استفاده بکنیم برای پیاده سازی سمافور به صورتی که busy waiting رخ نده
اینجا برای هر سمافوری یک صفی تعریف میکنیم
صفحه بعدی...
اگر این سمافور هیچ ریسورسیش ازاد نبود و یک پروسس درخواست ریسوس داشت باید اول بلاکش بکنیم و بلاک کردن یک پروسس به این معناست که دیگه بهش cpu داده نمیشه و اسکجولر دیگه هیچ وقت انتخابش نمیکنه
وخودمون هم برای اینکه بخوایم از بلاک درش بیاریم می گذاریم توی اون صفی که توی سمافورمون داریم ینی struct process *list و در مقابل بلاک یک تابع wakeup داریم که این wakeup باعث میشه که اون پروسس دوباره جز ready ها حساب بشه و cpu بهش تعلق بگیره
پس وقتی که یک پروسسی به یک نقطه ای رسید و یک ریسورسی می خواست که ازاد نبود اینجا اینو وارد یک صفی می کنیم که همون صف سمافور است و از طرفی هم اینجا براش بلاک رو فراخوانی میکنیم و دیگه هیچ وقت cpu اینو انتخابش نمیکنه مگر اینکه یک جا دیگه این wakeup بشه

Implementation with no Busy waiting (Cont.)

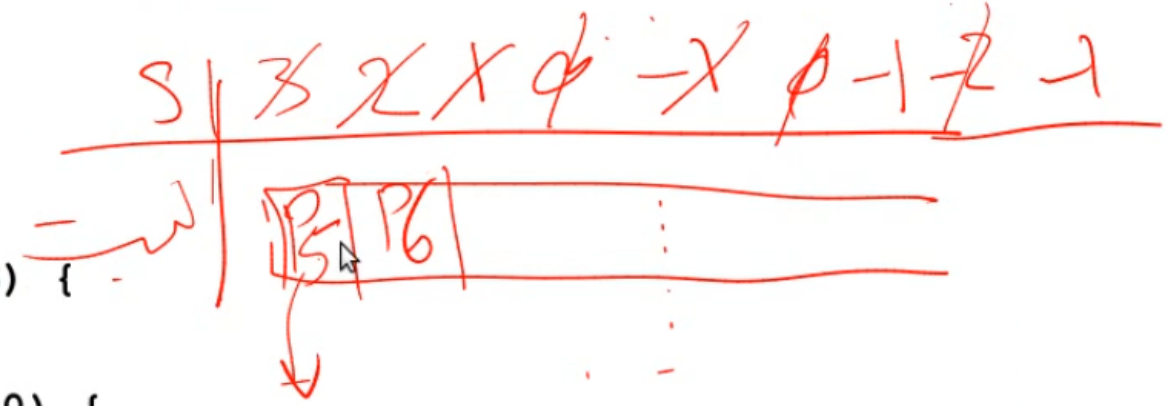
- Waiting queue

```
typedef struct {
  int value;
  struct process *list;
} semaphore;
```

یک استراکچری اینجا تعریف میکنیم برای خود سمافور که این یک مقداری داره که همون مقدار سمافورمون است ینی value و یک لیست هم داره که این همون لیست waiting ها است

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```



P1 wait signal
P2 wait
P3 wait signal
P4 wait
S = 3
P5 wait
P6 wait

کد wait و سیگنال رو چجوری با حرفایی که قبلا زدیم الان بنویسیم؟
wait رو وقتی که می خوایم بنویسیم بخاطر اینکه اونایی که ریسورس براشون ازاد نیست رو باز می خوایم ببریمشون توی اون صفه یه جا باید یک روش تشخیصی هم داشته باشیم که کیا توی صف هستن پس value سمافور رو به هر حال همیشه کم میکنیم

سیگنال:
سیگنال در مقابل داره خط اول مقدار value یکی زیاد میکنه و بعد از اینکه یکی زیاد کرد چک میکنه اگر این value<=0 است پس حتما قبلا یک پروسسی بوده که مقدار این S رو منفی کرده بوده و باعث شده بود که بره توی لیست انتظارها پس باید الان اون پروسس رو از توی لیست درش بیاریم و wakeupاش بکنیم
درمقابل اون طرف توی wait اگر ببینه مقدار value<0 شد ینی قبل از اینکه این پروسس فعلی بیاد و مقدار S رو کم بکنه صفر شده بوده و ریسورسی ما نداشتیم پس ما اینجا اول باید اونو اضافه بکنیم به لیست waiting اون سمافورمون و بعد بلاکش

توی این مثال دیگه ما Busy waiting نداریم