



Operating Systems

Synchronization Tools-Part4

Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Fall 2021

Second solution using compare-and-swap

- The common data structures are:

```
boolean waiting[n];  
int lock;
```



if `Waiting[i] == true`
جای نیست

threads جو سرمه

- The elements in the ***waiting*** array are initialized to ***false***
- Variable ***lock*** is initialized to ***0***.



Second solution using compare-and-swap

```
while (true) {  
    thread حادثه میگیرد که waiting[i] = true;  
  
    key = 1;  
  
    while (waiting[i] && key == 1)  
        key = compare_and_swap(&lock, 0, 1);  
        اگر lock=0 بود، میتوانیم رو تکن و در حالت بینی خود را داشت  
    waiting[i] = false;  
  
    /* critical section */  
    ...  
}
```

کو در اینجا باید اولین بار (جذب)

- ① waiting[i] = true
- ② key = 1
- ③ while (*true*)
 > *پنهان* *در حالت*
- ④ compare_and_swap
 lock=0 *lock=1*
 key=0

- ⑤ false *نہ باید*
- ⑥ waiting[i] = false
- ⑦ *کنیت* *در حالت* *Critical*

Entry
Section

Second solution using compare-and-swap

```
while (true) {
```

اپنے پردازہ کی لائیں ...

$j = (\underline{i} + 1) \% n;$ مسلک سے! (هر کس بارہ دوبارہ ایسا کرنے کا کام کرنے کا!)

while ($(j \neq i) \&& !\text{waiting}[j]$)

$j = (j + 1) \% n;$ صرف سارے بعدی

چھ جعلیں! 80 وہ while جس کی خدمت! ایسا کام کرنے کا!

if ($j == i$)

چھ کسی خدام وروج نہ کر! ایسا کام کرنے کا! بیرون!

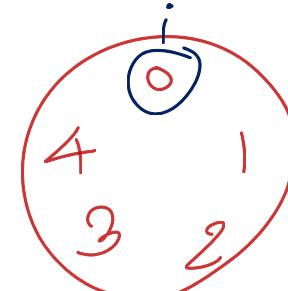
lock = 0; کسی ملک کا ملک کرنے کا! نعمتی چھ آجئنا ہے!

else

$\text{waiting}[j] = \text{false};$

/* remainder section */

}



اول صورت سرخ ۱ ← آتا مخواهد اجرا ہے؟
آخر حصے صورت سرخ ۴ ← آتا مخواهد اجرا ہے؟
آخر حصے صورت سرخ ۳ ← آتا مخواهد اجرا ہے؟

Requirement	Yes/No?
Mutual Exclusion	✓
Progress	✓
Bounded waiting	✓

چون حالات مغلقی در راه
از لمحہ نی کسی پردازہ درفلوت
ملا ہد، ۱-۱ پردازہ مل کوئی مدد نہیں
اپنے کارروائی کو white
 $j \neq i, j \neq i$ while
false (اگر ($\text{waiting}[j]$ اور $i == j$))
و بطری بعوئے وار
critical section

Synchronization Hardware Support

■ Hardware instructions

- `test_and_set()`
- `Compare_and_swap()`

■ Atomic variables

- We unfortunately do not have enough time to cover this
- Please read the related section in the reference book

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**
 - Boolean variable indicating if lock is available or not
 - In fact, the term **mutex** is short for mutual exclusion.



Mutex Locks

- Protect a critical section by

- First **acquire()** a lock
بندقی اوردن
- Then **release()** the lock
رها کردن

The diagram illustrates the implementation of a mutex lock. It shows two code snippets: `acquire()` and `release()`. The `acquire()` code is annotated with a red bracket labeled **atomic**, indicating that the entire block is atomic. The `release()` code is also annotated with a red bracket labeled **atomic**, indicating that the entire block is atomic.

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

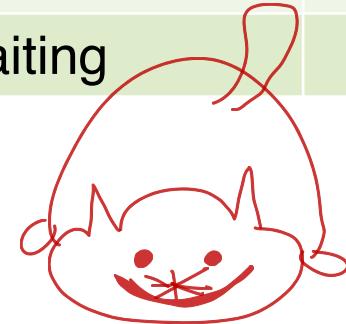
- Calls to **acquire()** and **release()** must be **atomic**

- Usually implemented via hardware atomic instructions such as **compare-and-swap**.

Solution to CS Problem Using Mutex Locks

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

Requirement	Yes/No
Mutual Exclusion	✓
Progress	✓
Bounded waiting	✗



- But this solution requires **busy waiting**.

- This lock therefore called a **spinlock**.
- Is this a disadvantage all the time?

این کار خوبی نیست اوند چنانچه هو بینم توی صفحه و کنترل پنل رو باز نمایم!

while ...
وی CPU رو گیرم
نه ممکن است
کسی دیگر
آن را بگیرد!

Busy Waiting: Advantage or Disadvantage?

- Advantage of Spinlocks: no context switch is required
 - When a process must wait on a lock
 - A context switch may take considerable time
- When we prefer spinlocks (on multi core systems)? *

- If a lock is to be held for a short duration
 - One thread can “spin” on one processing core while another thread performs its critical section on another core.



Busy Waiting: Advantage or Disadvantage?

On modern multicore computing systems, spinlocks are
widely used in many operating systems.



Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.

* ! Mutex نیز سینکرونایز کاربرد دارد *

- Semaphore S – **integer variable**.
- Can only be accessed via two indivisible (atomic) operations
 - **wait()** and **signal()**
 - ▶ Originally called **P()** and **V()**

Semaphore (Cont.)

- Definition of the wait() operation

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

- Definition of the signal() operation

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



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 → ...، *javac C++ > بیز*
- Can implement a counting semaphore S as a binary semaphore.
مکن بیز فعال و غیرفعال!
- With semaphores we can solve various synchronization problems.



Semaphore Usage Example

■ Solution to the CS Problem

- Create a semaphore “mutex” initialized to 1

: ترکیب این سازمان را
سوال
سازمان! این سازمان را
خواه! این سازمان را
خواه! این سازمان را
خواه! این سازمان را
خواه!
Semaphore $x = 0$ خواه! این سازمان را
خواه! این سازمان را
خواه! این سازمان را
خواه! این سازمان را
خواه!
entry section
CS
exit section
↑ signal (mutex) ;

Requirement	Yes/No
Mutual Exclusion	✓
Progress	✓
Bounded waiting	✗

Semaphore Usage Example (Cont.)

semaphore سیمافور

- 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

$P_1 :$

$S_1 ;$

signal(synch) ;

$P_2 :$

wait(synch) ;

سیمافور را
استفاده کردن
برای خود را

$S_2 ;$

Semaphore Implementation

- Must guarantee that no two processes can execute the **wait()** and **signal()** on the **same semaphore** at the same time.

! لیست است و این سریع می‌تواند هم‌زمان برای دو پردازنده اجرا شود.
برای اینکه یک سریع را بسازیم، **wait** و **signal** را در یک قسمت خاص قرار دهیم.
این قسمت را **semaphore** می‌نامیم!



! تا زمانی که دو پردازنده هم‌زمان برای این سریع اجرا نمی‌کنند،
هم‌زمانی در این سریع را می‌توانیم داشتیم.
→ {mutual exclusion, progress, bounded waiting}

- Thus, the implementation becomes **the critical section** problem where the **wait** and **signal** code are placed **in** the critical section.

Semaphore Implementation

- 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.**



Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue.

لهم حفظكم الله $\underline{\text{ما}} \leftarrow$ $\underline{\text{بـ}} \text{Semaphore}$ $\underline{\text{ما}} \text{ هو}$

Waiting queue \rightarrow Ready queue

- Each entry in a waiting queue has two data items:

- Value (of type integer)
- Pointer to next record in the list.

Semaphore Implementation with no Busy waiting

- 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

Implementation with no Busy waiting (cont.)

- Waiting queue

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

کل لست از بردازدها که سمت این
سیمافور است! که شرکت کردند!

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);  
    }  
}
```

نحوی نام : سیمپلیکس / wait و/or ایجاد میراث

value = 0
شرط ایجاد میراث
حالاتی هست که میراث ایجاد نمی شود

value = -1
شرط ایجاد نمی شود (بردازه را به لیست بردازهای که نیست مانع ایجاد نمی شوند)
شرط ایجاد نمی شود

ان مکانیزم باعث می شود CPU هدر نمود!

دعا برای پردازشی بکار رفته باشد
value = -2
شرط ایجاد نمی شود

value ++ \Rightarrow value = -1
شرط ایجاد نمی شود (هر کجا زوایا اجرا (هر کجا زوایا اجرا))

value ++ \Rightarrow value = 0
شرط ایجاد میراث

value ++ \Rightarrow value = 1
شرط ایجاد میراث

شرط ایجاد میراث



Problems with Semaphores

- Incorrect use of semaphore operations:

- **signal (mutex)** ... **wait (mutex)** → *mutual exclusion*
- **wait (mutex)** ... **wait (mutex)** → *progress*
- Omitting of **wait (mutex)** and/or **signal (mutex)**
(! خطای) ! پیشرفت ← ! اخراج

- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.

Summary of What Are Not Covered

- **Monitors**
- **Condition Variables**
- **We also skip chapter 7 slides**
 - <https://www.os-book.com/OS10/slides-dir/index.html>