



Operating Systems

Synchronization Tools-Part3

Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Spring 2023

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
- Then **release()** the lock

```
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?

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.
- 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**
- Can implement a counting semaphore S as a binary semaphore.
- **With semaphores we can solve various synchronization problems.**



Semaphore Usage Example

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

```
wait(mutex) ;
```

CS

```
signal(mutex) ;
```

Requirement	Yes/No
Mutual Exclusion	
Progress	
Bounded waiting	

Semaphore Usage Example (Cont.)

- 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 :

S_1 ;

signal (synch) ;

P2 :

wait (synch) ;

S_2 ;



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.



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.
- 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;
```


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);  
    }  
}
```

Problems with Semaphores

- Incorrect use of semaphore operations:
 - `signal(mutex) ... wait(mutex)`
 - `wait(mutex) ... wait(mutex)`
 - Omitting of `wait(mutex)` and/or `signal(mutex)`
- 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/slide-dir/index.html>



In-class Quiz

(همگام‌سازی ۱، نوشتن نام خانوادگی و شماره دانشجویی در بالای هر برگه پاسخ فراموش نشود)

در سیستم‌های عامل قدیمی مشابه Unix، قابلیت‌های اولیه (primitives) مورد نیاز برای همگام‌سازی پردازش (process synchroniztation) فراهم نشده بود. با این وجود این سیستم‌ها قابلیت pipe را فراهم می‌کردند. یک pipe کانالی برای انتقال جریانی از داده‌ها (data stream) بر اساس یک بافر با اندازه ثابت است (اغلب 4KB). خواندن (read) از یک pipe خالی باعث می‌شود که پردازش فراخواننده تا زمانی که داده‌ای در pip نوشته شود، بلوکه شود (block). نوشتن (write) در یک pipe پر (full) باعث می‌شود که پردازش فراخواننده تا زمانی که داده‌ای خوانده شود، بلوکه گردد. با توجه به این توصیف، به دو سوال زیر پاسخ دهید:

الف) توضیح دهید که سمافور را چگونه با pipe پیاده‌سازی کنیم. (راهنمایی: از منطق بلوکه شدن pipe طبق توضیحات بالا استفاده کنید). (۵ نمره)

ب) سمافور پیاده‌سازی شده با استفاده از pipe ممکن است در وضعیتی دچار بن‌بست (deadlock) شود که سمافور معمولی این مشکل را ندارد. این وضعیت را شرح دهید. (۵ نمره)

