

CS240 Operating Systems, Communications and Concurrency

Limitations of Approaches we have used to solve the Mutual Exclusion problem

Software Solutions for two or for N processes (Bakery Algorithm)

Each process executes an **agreed sequence of entry code** before using the critical section and an **agreed sequence of exit code** when completed.

One of the problems with this scheme is that the entry and exit code and associated shared data structures are a bit **unwieldy** and awkward when used by a programmer to protect any arbitrary critical section.

CS240 Operating Systems, Communications and Concurrency

/* Shared structures */

boolean[] choosing; /* An array of n elements initially false */

int[] number; /* An array of n elements initially 0 */

/* Thread code for access to critical section */

/* This thread is thread i */

/* Choose a ticket */

choosing[i] = true;

number[i] = Max(number[0], number[1], ..., number[n-1]) + 1;

choosing[i] = false;

for (j = 0; j < n; j++) { /* Check the ticket of each other thread against this thread's ticket */

while choosing[j] { /* Wait while a thread is choosing, in case its ticket is smaller */ }

while ((number[j] != 0) && ((number[j], j) < (number[i], i))) { /* Wait if a smaller ticket exists */ }

Critical Section

/* Not seeking to enter critical section anymore */

number[i] = 0;

Remainder Section

Unwieldy code
prone to coding
error

CS240 Operating Systems, Communications and Concurrency

Limitations of Approaches we have used to solve the Mutual Exclusion problem


Software algorithms and Multiprocessor operation

Due to the **operation of caches** in multiprocessor systems, software solutions based on the state of shared data structures between threads may not work if processors have different views of these data structures (not thread safe) when stored in local caches.

CS240 Operating Systems, Communications and Concurrency

Limitations of Approaches we have used to solve the Mutual Exclusion problem

Spinlock solutions based on the use of **indivisible instructions** such as **TestandSet** or **Swap/Xchg**



An **indivisible instruction** like TestandSet or Swap uses a non cached lock variable which is **read-and-set** in a single operation.

CS240 Operating Systems, Communications and Concurrency

Limitations of spinlock solutions based on the use of indivisible instructions

These provide a simple solution to mutual exclusion but do not make any guarantees about **progress** or **bounded waiting**.

As the competing threads spin the lock, it is arbitrary (based on scheduling) as to which thread may acquire it.

Hardware TestandSet operations are also **less efficient for multiple processor systems** to execute than instructions which can use the local caches and it is desirable to minimise the need for their execution.

CS240 Operating Systems, Communications and Concurrency

Busy Waiting

Another problem with software algorithms and spinlocks

<http://www.cyberiapc.com/os/ashes/busywait-swf.htm>

CS240 Operating Systems, Communications and Concurrency

Busy Waiting

All of these software solutions involve **Busy Waiting** - an execution loop in the code where the thread continually tests the locking condition during its scheduled timeslice until the condition allows it to continue into the critical section.

When a thread is scheduled, it could spend all its time repeatedly re-evaluating a condition which can never change until another thread is scheduled and changes that condition.

This is OK if the waiting period is short, but not for long waits, we need to suspend the process.

CS240 Operating Systems, Communications and Concurrency

Busy Waiting implementations have a number of disadvantages:-

System performance degrades due to a **wastage of processor time** reevaluating conditions repeatedly.

Timed waiting - It is not possible to implement a Bounded Waiting primitive for a lock where a thread might want to **wait for a certain period of time only**.

Priority inversion can occur where a lower priority task in the critical section may hold a lock which causes a higher priority task to wait.

CS240 Operating Systems, Communications and Concurrency

Semaphores

In order to deal with the **unwieldy** aspects of using synchronisation code (seen with the software solutions)

and to deal with the **fairness** of access to the critical section (seen with the indivisible hardware instruction approach)

and the problem of **busy waiting** (seen in both approaches), we introduce a more convenient and efficient synchronisation construct known as a semaphore.

CS240 Operating Systems, Communications and Concurrency

Semaphores

A semaphore is a general synchronisation tool.

In its **classical definition**, a semaphore consists of a **shared integer variable** accessible only through two defined operations, **P & V**, (sometimes known as Wait and Signal, or Acquire and Release).

The introduction of Semaphores by Dijkstra in 1965, is why the operations are classically known as P & V (P from the Dutch *proberen* “to test” and V from *verhogen* “to increment”).

CS240 Operating Systems, Communications and Concurrency

Semaphores

The **P operation** on Semaphore S is defined as:-

```
P(S) {  
    while (S <= 0) { // Do Nothing}  
    S = S - 1;  
}
```

The **V operation** on Semaphore S is defined as:-

```
V(S) {  
    S = S+1;  
}
```

Both P & V operations must be indivisible.

CS240 Operating Systems, Communications and Concurrency

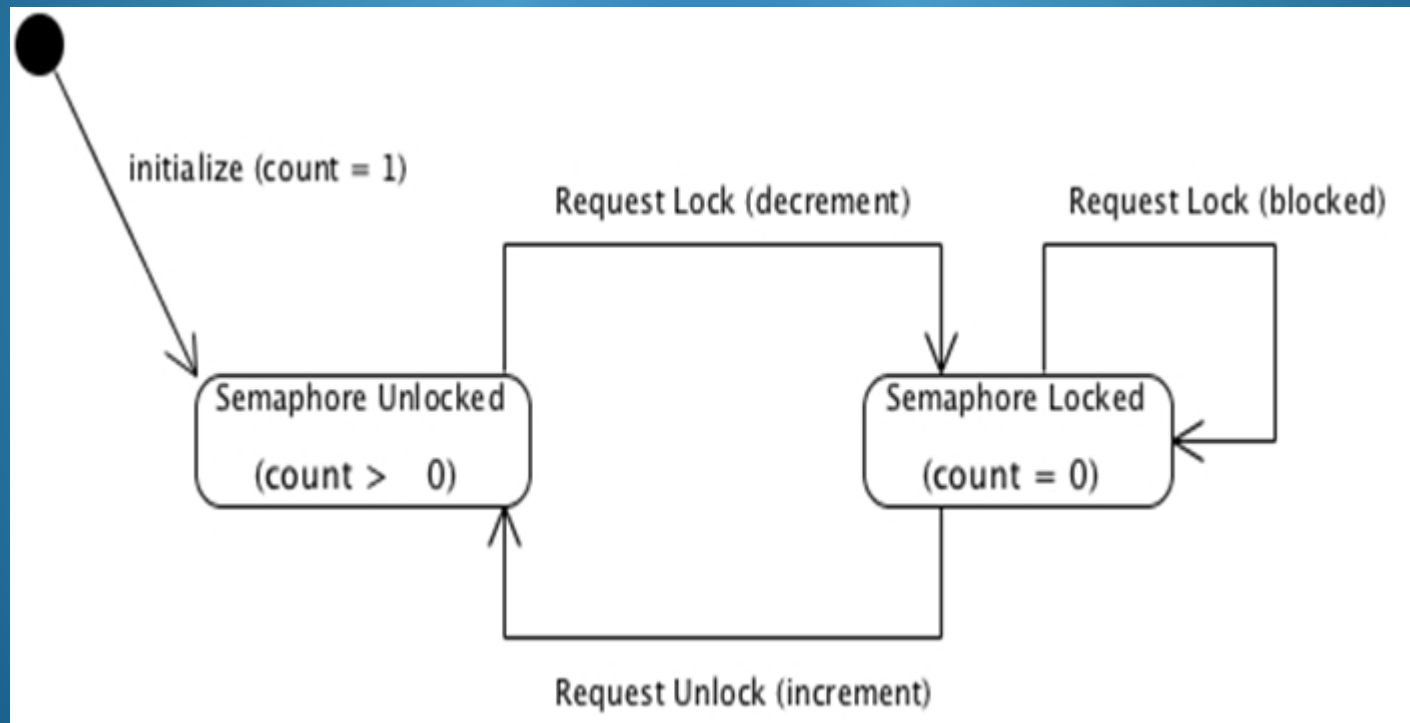
Two types

When the semaphore value is used to control mutual exclusion, then a **binary semaphore** is used which can only have the value 0 or 1. It is used as a mutex locking mechanism.

A **counting semaphore** (or **rate limiting semaphore**) can also be implemented by setting the count value to a maximum number (say n) of threads allowed into a region of code at one time.

In this case, n threads will be able to decrement the semaphore but the $n+1$ thread will block until one of the earlier ones releases it. The semaphore is used as a signalling mechanism between threads.

CS240 Operating Systems, Communications and Concurrency



CS240 Operating Systems, Communications and Concurrency

Suggestions for making P and V indivisible

To make these operations indivisible in single processor systems we could:-

Treat the P & V methods as critical sections and **use our N process software solutions** at entry and exit to the P & V operations.

Implement P and V as operating system calls and **turn off interrupts** during their execution to prevent the executing thread from being swapped.

CS240 Operating Systems, Communications and Concurrency

Suggestions for making P and V indivisible

Spinlocks - We could use indivisible hardware **TestandSet instructions** and a lock variable which must be exclusively acquired first by any process wishing to execute P or V.

As the **P and V code is relatively short**, there is less likely to be much contention to execute them at the same time in different threads and the effect of busy waiting on the spinlock associated with the semaphore code would therefore be minimal.

CS240 Operating Systems, Communications and Concurrency

Usage of Semaphores

So if we can create P and V operations that are indivisible, we can use the much simpler coding below for synchronisation:-

Mutual Exclusion

// mutex is shared between threads who wish to use the

// protected critical section

Semaphore mutex = 1; // Initially unlocked

P(mutex);

Critical Section

V(mutex);

CS240 Operating Systems, Communications and Concurrency

Usage of Semaphores

General Synchronisation

In the sequence below we want to stop statement S2 in thread 2 from executing until statement S1 in thread 1 has completed. By creating a semaphore S and initialising it to 0, thread 2 will be held up until thread 1 increments S.

Semaphore S = 0; // initially locked

<u>thread 1</u>	<u>thread 2</u>
S1;	P(S);
V(S);	S2;

CS240 Operating Systems, Communications and Concurrency

USING Operating System Mutex within C program, POSIX API

```
include <pthread.h>
```

```
//create an unlocked mutex structure with default attributes using initializer macro  
// Note: could also use pthread_mutex_init(&mutex1, NULL) in main instead
```

```
pthread_mutex_t mutex1= PTHREAD_MUTEX_INITIALIZER;
```

```
int counter = 0;
```

```
int main () {  
    pthread_mutex_lock(&mutex1);  
        counter++;  
    pthread_mutex_unlock(&mutex1);  
}
```

CS240 Operating Systems, Communications and Concurrency

Practical Implementation of Semaphores

Note that implementing the classical definition of a semaphore standardises the implementation of mutual exclusion but the P and V operations **still include a busy waiting loop** and do not address **fairness** of access to the critical section.

A thread will loop indefinitely in the P operation below checking the value of the semaphore if it is not > 0 .

```
P(S) {  
    while (S <= 0) { // Do Nothing}  
    S = S - 1;  
}
```

CS240 Operating Systems, Communications and Concurrency

To overcome the problem of busy waiting and fair access we can provide a better implementation for the P and V operations as follows:-

```
P(S) {  
    value = value - 1;  
    if( value < 0) {  
        Add this thread to waiting queue  
        Block;  
    }  
}
```


CS240 Operating Systems, Communications and Concurrency

To overcome the problem of busy waiting and fair access we can provide a better implementation for the P and V operations as follows:-

```
V(S) {  
    value = value + 1;  
    if (value <= 0) {  
        Remove a thread T from waiting queue  
        wakeup(T);  
    }  
}
```

Both P & V operations must be indivisible.

CS240 Operating Systems, Communications and Concurrency

An animation of the general idea can be found here

[http://williamstallings.com/OS-
Animation/Queensland/SEMA.SWF](http://williamstallings.com/OS-Animation/Queensland/SEMA.SWF)

CS240 Operating Systems, Communications and Concurrency

Semaphores can be used to solve a variety of **classical coordination problems**. We will examine some classes of problems common in many computing applications:-

Producer Consumer Problem

A situation where two or more threads exchange data with each other through a shared buffer.

Dining Philosophers Problem

A situation where limited resources must be allocated among competing threads in a way that avoids deadlock.

Readers Writers Problem

A situation where some threads alter the contents of a database and other thread simply query the contents.