

CS240 Operating Systems, Communications and Concurrency

Algorithmic Solutions to Mutual Exclusion

Conditions necessary for a correct solution to the Mutual Exclusion Problem

(a) Mutual Exclusion

No two threads can be able to execute simultaneously in the critical code section. (i.e. the section which manipulates the nonshareable resource.)

(b) Progress

A thread operating outside the critical section cannot prevent another thread from entering the critical section.

(c) Bounded Waiting

Once a thread has indicated its desire to enter a critical section, it is guaranteed that it may do so in a finite time.

CS240 Operating Systems, Communications and Concurrency

Algorithmic Solutions to Mutual Exclusion

Objective

Design a set of instructions to be executed by any thread entering or leaving a critical section which will solve the mutual exclusion problem in a fair manner.

First we solve it for two threads, then for n threads.

CS240 Operating Systems, Communications and Concurrency

Two Thread Software Solutions

Both threads execute the same entry and exit code, we will refer to the current thread executing the code as **i** and the other thread as **j**.

Where **i** appears in the code, the thread is referring to its own Id, where **j** appears the thread is referring to the Id of the other thread.

This code is only to be executed by **two** processes simultaneously.

CS240 Operating Systems, Communications and Concurrency


Starting Point – Method 2 Solution to Mutual Exclusion (but Lockstep Execution)

```
int turn; /* a shared variable which can store  
           the value 0 or 1. */  
while (turn != i)  
    { /* While it's not my turn, do nothing */ }
```

Critical Section

```
turn = j;
```

Process leaving the critical section is deciding who goes next without accounting for which processes actually want to use it.



Remainder Section

CS240 Operating Systems, Communications and Concurrency

Recall - Method 3 (two bowls) was also Solution to Mutual
Exclusion

(but is susceptible to deadlock)

```
/* Shared boolean array */  
boolean[] flag = {false, false};
```


CS240 Operating Systems, Communications and Concurrency

Recall - Method 3 Solution to Mutual Exclusion (but susceptible to deadlock)

```
flag[i] = true; /* set my flag, I want in */  
while flag[j]    ← Deadlock results if both set flags simultaneously  
{ /* while the other thread's flag is set,  
    do nothing */}
```

Critical Section

```
flag[i] = false;
```

Remainder Section

CS240 Operating Systems, Communications and Concurrency

Solution for two processes is combination of these two approaches

(Deadlock free and lockstep free)

```
flag[i] = true; /* This thread wants to enter */
turn = j; /* Let other thread go first */
while (flag[j] && (turn == j))
    { /* Yield to thread j, Do nothing */ }
```

Critical Section

flag [i] = false;

Remainder Section

Turn can only have one value

This condition cannot be true for both processes
so no deadlock

CS240 Operating Systems, Communications and Concurrency

Two Process Solution

Ensures Mutual Exclusion Because

If both processes attempt entry together, both their flags are set but the value of variable turn decides which gets entry. It can only have one value.

Progress

Only processes trying to gain access to critical section are involved in decision as to which one of them goes next.

Bounded Waiting

A thread cannot access the critical section a second time if the other thread is waiting because it yields first to the other thread by setting turn.

CS240 Operating Systems, Communications and Concurrency

Next we look at a general software solution for N processes

This algorithm is known as the **bakery algorithm**, and is a commonly used queue servicing approach. A thread chooses a ticket, from an ordered set of numbers, which determines its position in the queue for the critical section.

- Tickets are unique, **there is a service order and so waiting is bounded**

CS240 Operating Systems, Communications and Concurrency

Bakery Algorithm - Choosing a Ticket

Look at the tickets of all threads so far and pick one bigger

Note this operation could be done in parallel by other threads and they could possibly generate the same ticket number.

In cases of where the ticket number is the same, we will order the threads by their thread ID, with lower numbers going first.

CS240 Operating Systems, Communications and Concurrency

Bakery Algorithm - Waiting for Critical Section

For each of the threads {

 If a thread is choosing a ticket then wait

 (as it may have been calculated before ours and maybe
 smaller or equal)

 If the thread holds a non zero ticket and it is smaller than
 ours then wait (until the thread is finished with its ticket)

}

Enter the Critical Section

CS240 Operating Systems, Communications and Concurrency

Solutions to the Mutual Exclusion Problem

```
/* Shared structures */
```

```
boolean[] choosing;
```

```
/* An array of n elements initially false */
```

```
int[] number;
```

```
/* An array of n elements initially 0 */
```

CS240 Operating Systems, Communications and Concurrency


Solutions to the Mutual Exclusion Problem

/* Thread i - Choose a ticket */

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


CS240 Operating Systems, Communications and Concurrency

```
/* Wait until we have the lowest ticket */  
for (j = 0; j < n; j++) {  
    while choosing[j]  
        {}  
  
    while ((number[j] != 0) &&  
           ((number[j], j) < (number[i], i)))  
        {  
            /* Wait if a smaller ticket exists */  
        }  
}
```



This thread's ticket

Enter Critical Section

CS240 Operating Systems, Communications and Concurrency

Exit Code

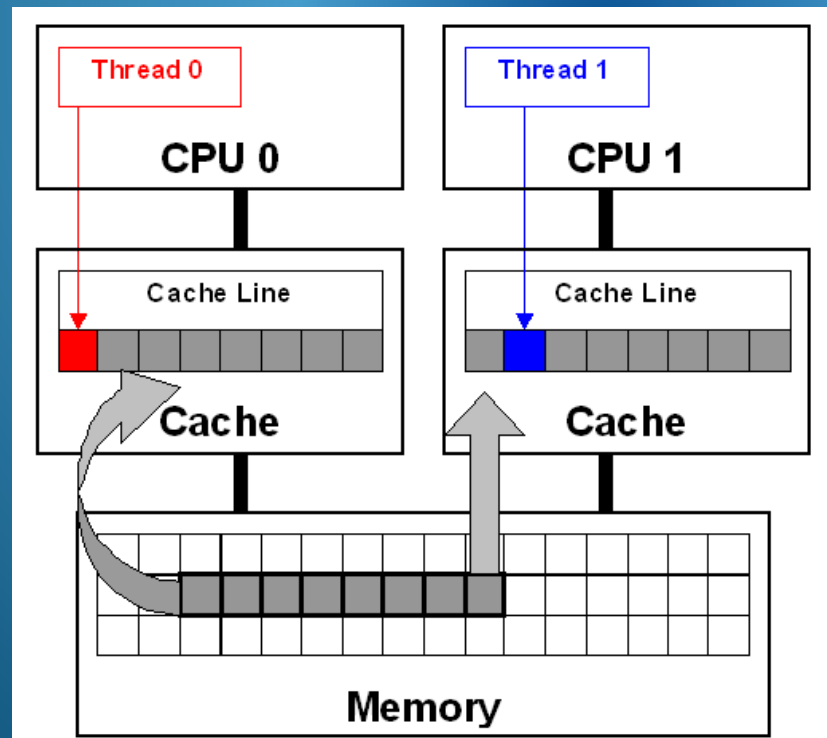
```
/* Not seeking to enter critical section  
   anymore */  
number[i] = 0;
```

Remainder Section

CS240 Operating Systems, Communications and Concurrency

Solving Mutual Exclusion on Multiprocessor Systems and Systems with Caches

Memory is cached within processors for performance reasons. This means there may be several copies of data.



CS240 Operating Systems, Communications and Concurrency

One of the assumptions of using software algorithmic solutions to the mutual exclusion problem, is that when the data structures used for controlling access to a critical section are shared among the threads, **all threads must have a consistent view of those data structures** in order for the algorithm to work.

They must all be working with the same data values.

CS240 Operating Systems, Communications and Concurrency

Considering caches at many levels

Compilers or virtual machine run times may try to **optimise code execution or data access by using a processor register for storing a variable or reorganising the code** in some way.

In multiprocessor systems, the shared data structures may be held in the cache or registers of each processor, so that when one thread alters its cache copy, other threads that read close to that time may read obsolete values from their respective copies. Alterations **may not be visible to other processors immediately**.

This means the algorithms can fail to preserve mutual exclusion.

CS240 Operating Systems, Communications and Concurrency

Solving Mutual Exclusion on Multiprocessor Systems

Disabling Interrupts or Disabling Pre-emption

On a single processor system, if you disable interrupts, the current thread cannot be interrupted and swapped, so it can perform a series of test and set operations on data structures without the consequences of potential interleaved execution by another. So the entry code for testing and setting variables controlling entry to a critical section could be executed easily with interrupts turned off to make it indivisible.

But, disabling pre-emption is risky.

It is inefficient and impractical to disable interrupts simultaneously for all processors **in a multiprocessor system.**

CS240 Operating Systems, Communications and Concurrency

Solving Mutual Exclusion on Multiprocessor Systems

Atomic test-and-set

A safer approach to solving the indivisibility problem is to use **special processor instructions** which can **read and write to memory indivisibly** of the activities of other processors and which assert lock mechanisms on areas of memory to avoid cache coherence problems.

e.g. A **test-and-set** instruction is an instruction used to both write to a memory location and return its old value in a single atomic (i.e., non-interruptible) operation. The memory location is locked from access by other processors during a test-and-set operation.

CS240 Operating Systems, Communications and Concurrency

Indivisible Hardware Instructions

If a processor supports the *test-and-set* hardware instruction then we can implement n-process mutual exclusion as follows:-

A lock variable is shared by all threads, threads should not cache its value at the software level, specify to be volatile
`volatile boolean lock = false; //lock available`

Each thread then tries to set lock to true before entering the critical section. The test-and-set operation returns the current value of lock and also sets it to a new value in an indivisible operation.

If the value returned is true, then the lock was already set and the thread has to wait until the value returned by test-and-set is false.

CS240 Operating Systems, Communications and Concurrency

Indivisible Hardware Instructions

// Entry code to critical section

```
while ( test-and-set(lock, true) )  
    { /* loop until the get-and-set function  
       returns false, i.e. we got the lock */ }
```

Spinlock
loop



critical section

```
test-and-set(lock, false);
```

remainder section

NB: test-and-set method must be implemented as a single indivisible processor instruction.

CS240 Operating Systems, Communications and Concurrency

Indivisible Hardware Instructions

Another type of indivisible read/write instruction found on some processors is the **xchg** or **swap** instruction which allows the contents of two memory locations to be exchanged indivisibly.

If the processor supports the **swap** hardware instruction then we can implement n-process mutual exclusion as follows:-

CS240 Operating Systems, Communications and Concurrency

```
// lock is shared by all threads  
Boolean lock = false; // lock available
```

```
// Each thread has its own copy of key  
Boolean key = true;
```

```
// Entry code to critical section
```

```
do {  
    swap(lock, key);  
} while (key == true);
```

Spinlock
loop



```
critical section
```

```
swap(lock, key);
```

```
remainder section
```

CS240 Operating Systems, Communications and Concurrency

Busy Waiting

All of these software solutions involve **Busy Waiting**.

When a thread is scheduled, it could spend its time repeatedly re-evaluating the lock condition which can never change until another thread is scheduled and changes that condition. This is OK if the waiting period is short, but for long waits, we need to suspend the process.

The spinlock solutions don't guarantee **Bounded Waiting** either. It is pure luck which thread manages to acquire the lock.