G52CON:
Concepts of Concurrency

Lecture 5: Algorithms for Mutual Exclusion I

Brian Logan
School of Computer Science
bsl@cs.nott.ac.uk

Outline of this lecture

- mutual exclusion protocols
- criteria for a solution
  - safety properties
  - liveness properties
- simple spin lock
- spin lock using turns
- spin lock using the Test-and-Set special instruction

Archetypical mutual exclusion

Any program consisting of *n* processes for which mutual exclusion is required between critical sections belonging to just one class can be written:

```
// Process 1        // Process 2   ...    // Process n
init_1;             init_2;               init_n;
while(true) {       while(true) {         while(true) {
  crit_1;             crit_2;               crit_n;
  rem_1;              rem_2;                rem_n;
}                   }                     }
```

where $init_i$ denotes any (non-critical) initialisation, $crit_i$ denotes a critical section, rem_i denotes the (non-critical) remainder of the program, and i is 1, 2, … n.

Archetypical mutual exclusion

We assume that `init`, `crit` and `rem` may be of any size:

- `crit` must execute in a finite time—process does not terminate in `crit`
- `init` and `rem` may be infinite—process *may* terminate in `init` or `rem`
- `crit` and `rem` may vary from one pass through the `while` loop to the next

With these assumptions it is possible to rewrite *any* process with critical sections into the archetypical form.

Ornamental Gardens problem

```
// West turnstile               // East turnstile

init1;                          init2;
while(true) {                   while(true) {
  // wait for turnstile           // wait for turnstile
  < count = count + 1; >          < count = count + 1; >
  // other stuff ...              // other stuff ...

}                               }
```

```
// Shared datastructures
count == 0
```

Ornamental Gardens problem

```
// West turnstile

init1;
while(true) {
  // wait for turnstile
  < INCR count; >
  // other stuff ...

}
```

```
// East turnstile

init2;
while(true) {
  // wait for turnstile
  < INCR count; >
  // other stuff ...

}
```

```
// Shared datastructures
count == 0
```

Limitations of special instructions

This solution will work if:

- we have a multiprogramming implementation of concurrency (or we can lock memory)
- we have an atomic *increment* instruction available on the target CPU
- we know how a given high-level program statement will be compiled

However, the range of things you can do with a single atomic action is limited — we can't write a critical section longer than one instruction.

Shared Queue problem

```
// Process 1

$init_1$
while(true) {
  tail = tail + 1;
  queue[tail] = data1;

  // other code ...
  $rem_1$
}
```

```
// Process 2

$init_2$
while (true) {
  tail = tail + 1;
  queue[tail] = data2;

  // other code ...
  $rem_2$
}
```

```
// Shared datastructures

Object queue[SIZE];
integer tail;
```

Example: coarse-grained atomic action

```
// Process 1

init1
while(true) {
 < tail = tail + 1;
   queue[tail] = data1; >

  // other code ...
}
```

```
// Process 2

init2
while (true) {
 < tail = tail + 1;
   queue[tail] = data2; >

  // other code ...
}
```

```
// Shared datastructures

Object queue[SIZE];
integer tail;
```

Defining a mutual exclusion protocol

To solve the mutual exclusion problem, we adopt a standard Computer Science approach:

- we design a *protocol* which can be used by concurrent processes to achieve mutual exclusion and avoid interference;
- our protocol will consist of a sequence of instructions which is executed before and possibly after the critical section;
- such protocols can be defined using standard sequential programming primitives, special instructions and what we know about when process switching can happen.

There are many ways to implement such a protocol.

General form of a solution

We assume that each of the *n* processes have the following form, $i = 1, \ldots, n$

```
// Process i
init_i;
while(true) {
    // entry protocol
    crit_i;
    // exit protocol
    rem_i;
}
```

Shared Queue problem

```
// Process 1

init1
while(true) {
  // entry protocol
  tail = tail + 1;
  queue[tail] = data1;
  // exit protocol

  // other code ...
}
```

```
// Process 2

init2
while (true) {
  // entry protocol
  tail = tail + 1;
  queue[tail] = data2;
  // exit protocol

  // other code ...
}
```

```
// Shared datastructures

Object queue[SIZE];
integer tail;
```

Correctness of concurrent programs

A concurrent program must satisfy two types of property:

- **Safety Properties:** requirements that something should never happen, e.g., failure of mutual exclusion or condition synchronisation, deadlock etc.
- **Liveness Properties:** requirements that something will eventually happen, e.g. entering a critical section.

Note that establishing liveness may require proving safety properties.

Criteria for a solution

The protocols should satisfy the following properties

- **Mutual Exclusion:** at most one process at a time is executing its critical section
- **Absence of Deadlock (Livelock):** if no process is in its critical section and two or more processes attempt to enter their critical sections, at least one will succeed
- **Absence of Unnecessary Delay:** if a process is trying to enter its critical section and other processes are executing their noncritical sections (or have terminated), the first process is not prevented from entering its critical section
- **Eventual Entry:** a process that is attempting to enter its critical section will eventually succeed

– Andrews (2000), p 95.

Deadlock vs livelock

A processes is deadlocked or livelocked when it is unable to make progress because it is waiting for a condition that will never become true

- a **deadlocked** process is blocked waiting on the condition, e.g, in `wait()` — process does not consume any CPU
- a **livelocked** process is alive and waiting on the condition, e.g, busy waiting — process does consume CPU

A simple spin lock

```
bool lock = false;          // shared lock variable

// Process i
init_i;
while(true) {
  while(lock) {};           // entry protocol
  lock = true;              // entry protocol
  crit_i;
  lock = false;             // exit protocol
  rem_i;
}
```

Properties of the simple spin lock

Does the simple spin lock satisfy the following properties:

- **Mutual Exclusion:** yes/no
- **Absence of Livelock:** yes/no
- **Absence of Unnecessary Delay:** yes/no
- **Eventual Entry:** yes/no

An example trace 1

```
// Process 1                   // Process 2

init1;                         init2;









}                              }
```

lock == *false*

An example trace 2

```
// Process 1                  // Process 2

init1;                        init2;
while(true) {








}                             }
```

`lock == false`

An example trace 3

```
// Process 1                    // Process 2

init1;                          init2;
while(true)                     while(true)






}                               }
```

`lock == false`

An example trace 4

```
// Process 1                // Process 2

init1;                      init2;
while(true) {               while(true)
    while(lock)



}                           }
```

`lock == false`

An example trace 5

```
// Process 1

init1;
while(true) {
    while(lock)



}
```

```
// Process 2

init2;
while(true) {
    while(lock)



}
```

`lock == false`

An example trace 6

```
// Process 1

init1;
while(true) {
    while(lock)
    lock = true;



}
```

```
// Process 2

init2;
while(true) {
    while(lock)




}
```

`lock == true`

An example trace 7

```
// Process 1

init1;
while(true) {
    while(lock)
    lock = true;
    crit1;



}
```

```
// Process 2

init2;
while(true) {
    while(lock)





}
```

`lock == true`

An example trace 8

```
// Process 1

init1;
while(true) {
    while(lock)
    lock = true;
    crit1;


}
```

```
// Process 2

init2;
while(true) {
    while(lock)
    lock = true;



}
```

`lock == true`

An example trace 9

```
// Process 1

init1;
while(true) {
    while(lock)
    lock = true;
    crit1;



}
```

```
// Process 2

init2;
while(true) {
    while(lock)
    lock = true;
    crit2;



}
```

`lock == true`

Mutual exclusion violation

```
// Process 1                 // Process 2

init1;                       init2;
while(true) {                while(true) {
    while(lock)                  while(lock)
    lock = true;                 lock = true;
    crit1;                       crit2;



}                            }

                  lock == true
```

Properties of the simple spin lock

The simple spin lock has the following properties:

- **Mutual Exclusion:** no
- **Absence of Livelock:** yes
- **Absence of Unnecessary Delay:** yes
- **Eventual Entry:** is guaranteed only if the scheduling policy is *strongly fair*.

A *strongly fair* scheduling policy guarantees that if a process requests to enter its critical section infinitely often, the process will *eventually* enter its critical section.

Properties of the simple spin lock

- **Mutual Exclusion:** doesn't hold because there are interleavings which allow both processes to pass their entry protocols
- **Absence of Livelock:** holds because if all processes are outside their critical sections, `lock` must be *false*, and hence (at least) one of the processes will be allowed to enter its critical section
- **Absence of Unnecessary Delay:** holds because if all the other processes are outside their critical sections and stay there, `lock` is *false* and stays *false*, and hence the process that is trying to enter can immediately do so
- **Eventual Entry:** holds because if a process tests `lock` infinitely often, it must eventually see the value *false*—`lock` must become *false* eventually as no process can spend infinitely long in its critical section, so must eventually execute its exit protocol, setting `lock` to *false*

Spin lock using turns

```
// Process 1

$init_1$;
while(true) {
    // entry protocol
    while(turn == 2) {};
    $crit_1$;
    // exit protocol
    turn = 2;
    $rem_1$;
}
```

```
// Process 2

$init_2$;
while(true) {
    // entry protocol
    while(turn == 1) {};
    $crit_2$;
    // exit protocol
    turn = 1;
    $rem_2$;
}
```

`turn == 1`

Properties of round robin

Does round robin satisfy the following properties:

- **Mutual Exclusion:** yes/no
- **Absence of Livelock:** yes/no
- **Absence of Unnecessary Delay:** yes/no
- **Eventual Entry:** yes/no

Properties of round robin

Round robin has the following properties:

- **Mutual Exclusion:** yes
- **Absence of Livelock:** no
- **Absence of Unnecessary Delay:** no
- **Eventual Entry:** no

Properties of round robin

- **Mutual Exclusion:** holds because `turn` can't be both 1 and 2, so at most one process can be in its critical section at any given time
- **Absence of Livelock:** doesn't hold — if there are three processes, one of which has terminated (e.g., in `rem`), then the other two processes may not be able to enter their critical sections
- **Absence of Unnecessary Delay:** fails for two reasons— (1) if any processes terminates outside its critical section, then a process that wants to enter may be unable to do so; (2) even if no process terminates, all processes are constrained to enter their critical sections in order and equally often
- **Eventual Entry:** doesn't hold because the processes can Livelock

Test-and-Set instruction

The Test-and-Set instruction effectively executes the function

```
bool TS(bool lock) {
    bool v = lock;
    lock = true;
    return v;
}
```

as an atomic action.

Spin lock using Test-and-Set

```
// Process i

$init_i$;
while(true) {
  while (TS(lock)) {};   // entry protocol
  $crit_i$;
  lock = false;          // exit protocol
  $rem_i$;
}
```

```
// shared lock variable
bool lock = false;
```

An example trace 1

```
// Process 1                    // Process 2

init1;                          init2;







}                               }
```

`lock == false`

An example trace 2

```
// Process 1                  // Process 2

init1;                        init2;
while(true)









}                             }
```

`lock == false`

An example trace 3

```
// Process 1                 // Process 2

init1;                       init2;
while(true)                  while(true)





}                            }
```

`lock == false`

An example trace 4

```
// Process 1                    // Process 2

init1;                          init2;
while(true) {                   while(true) {
    while(TS(lock))




}                               }
```

`lock == true`

An example trace 5

```
// Process 1

init1;
while(true) {
    while(TS(lock))




}
```

```
// Process 2

init2;
while(true) {
    while(TS(lock))




}
```

`lock == true`

An example trace 6

```
// Process 1                     // Process 2

init1;                           init2;
while(true) {                    while(true) {
   while(TS(lock)) {};              while(TS(lock)) {};




}                                }
```

`lock == true`

An example trace 7

```
// Process 1

init1;
while(true) {
    while(TS(lock)) {};
    crit1;



}
```

```
// Process 2

init2;
while(true) {
    while(TS(lock)) {};




}
```

`lock == true`

An example trace 7

```
// Process 1

init1;
while(true) {
    while(TS(lock)) {};
    crit1;
    lock = false;

}
```

```
// Process 2

init2;
while(true) {
    while(TS(lock)) {};




}
```

`lock == false`

An example trace 8

```
// Process 1

init1;
while(true) {
    while(TS(lock)) {};
    crit1;
    lock = false;
    rem1;
}
```

```
// Process 2

init2;
while(true) {
    while(TS(lock)) {};
    crit2;



}
```

`lock == true`

An example trace 9

```
// Process 1

init1;
while(true) {
    while(TS(lock)) {};
    crit1;
    lock = false;
    rem1;
}
```

```
// Process 2

init2;
while(true) {
    while(TS(lock)) {};
    crit2;


}
```

`lock == true`

An example trace 10

```
// Process 1                  // Process 2

init1;                        init2;
while(true) {                 while(true) {
    while(TS(lock)) {};           while(TS(lock)) {};
    crit1;                        crit2;
    lock = false;
    rem1;
}                             }
```

`lock == true`

Properties of the Test-and-Set solution

The solution based on Test-and-Set has the following properties:

- **Mutual Exclusion:** yes
- **Absence of Livelock:** yes
- **Absence of Unnecessary Delay:** yes
- **Eventual Entry:** is guaranteed only if the scheduling policy is *strongly fair*.

Solving the Shared Queue problem

```
// Process 1

init1
while(true) {

  tail = tail + 1;
  queue[tail] = data1;

  // other code ...
}
```

```
// Process 2

init2
while (true) {

  tail = tail + 1;
  queue[tail] = data2;

  // other code ...
}
```

```
// Shared datastructures

Object queue[SIZE];
integer tail;
```

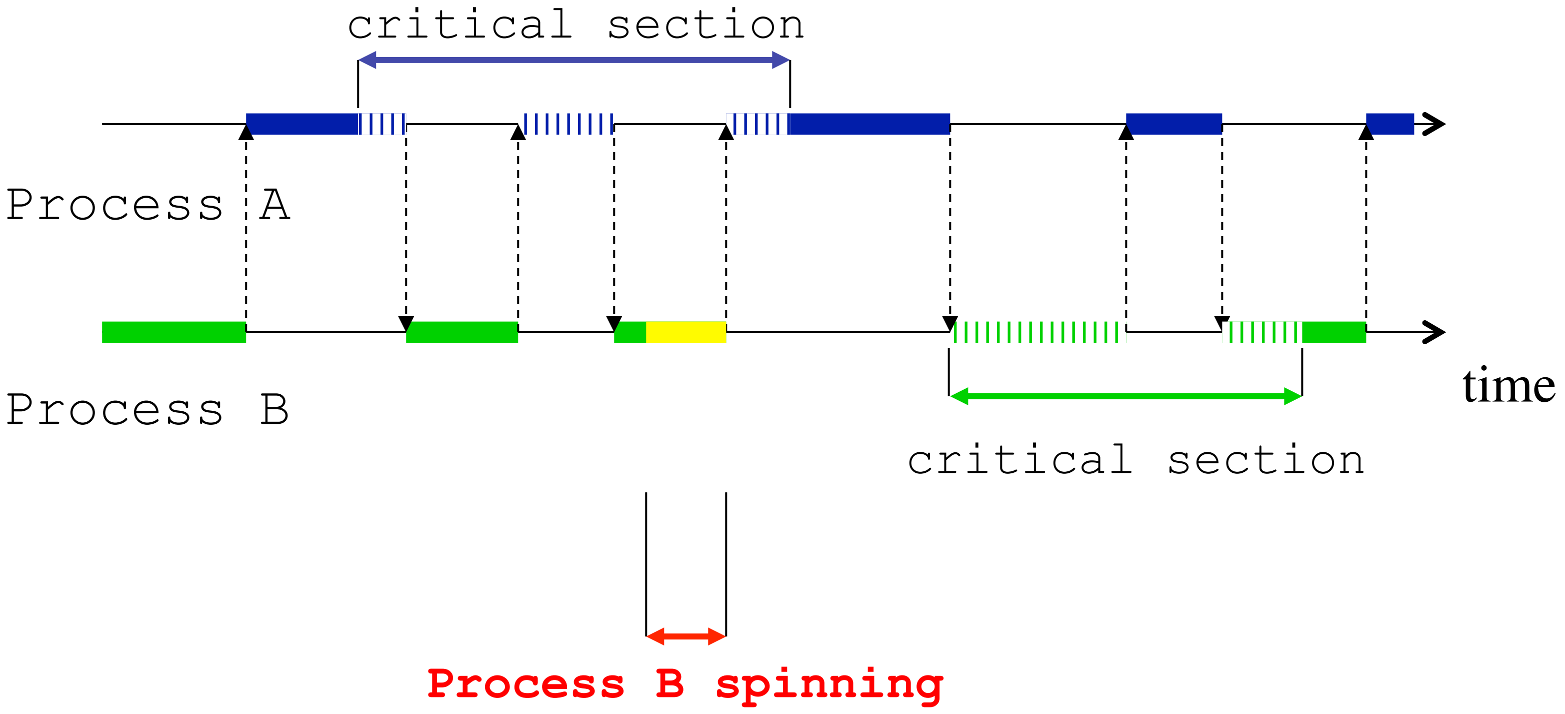
Solving the Shared Queue problem

```
// Process 1

init1
while(true) {
  while(TS(lock)) {};
  tail = tail + 1;
  queue[tail] = data1;
  lock = false;
  // other code ...
}
```

```
// Process 2

init2
while (true) {
  while(TS(lock)) {};
  tail = tail + 1;
  queue[tail] = data2;
  lock = false;
  // other code ...
}
```

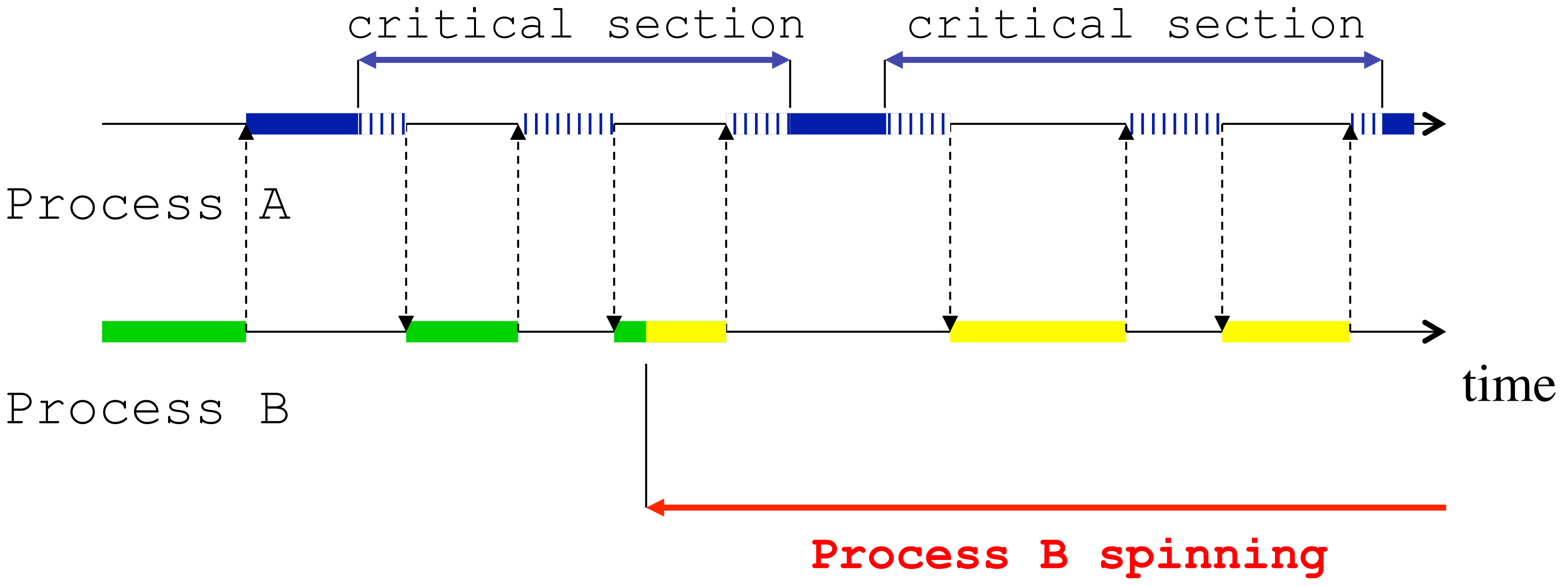
```
// Shared datastructures
Object queue[SIZE];
integer tail;
lock = false;
```

Overhead of spin locks



Process B spinning

Possible starvation with spin locks



Test-and-Set summary

- Test-and-Set must be atomic
- in a multiprocessing implementation Test-and-Set must effectively lock memory
- if both processes don't try to enter their critical section at the same time neither will have to wait (no *Unnecessary Delay*)
- if there is contention, so long as the critical sections are short the amount of time that each process should have to spend spinning (or *busy waiting*) will be small
- for Eventual Entry, the scheduling policy must be strongly fair
- since all processes execute the same protocol it works for any number of processes

The next lecture

Mutual Exclusion Algorithms II

Suggested reading:

- Ben-Ari(1982), chapter 3;
- Burns & Davies (1993), chapter 3, section 3.4.