G52OSC
OPERATING SYSTEMS AND CONCURRENCY

Mutual Exclusion
and Critical Sections I

Dr Jason Atkin

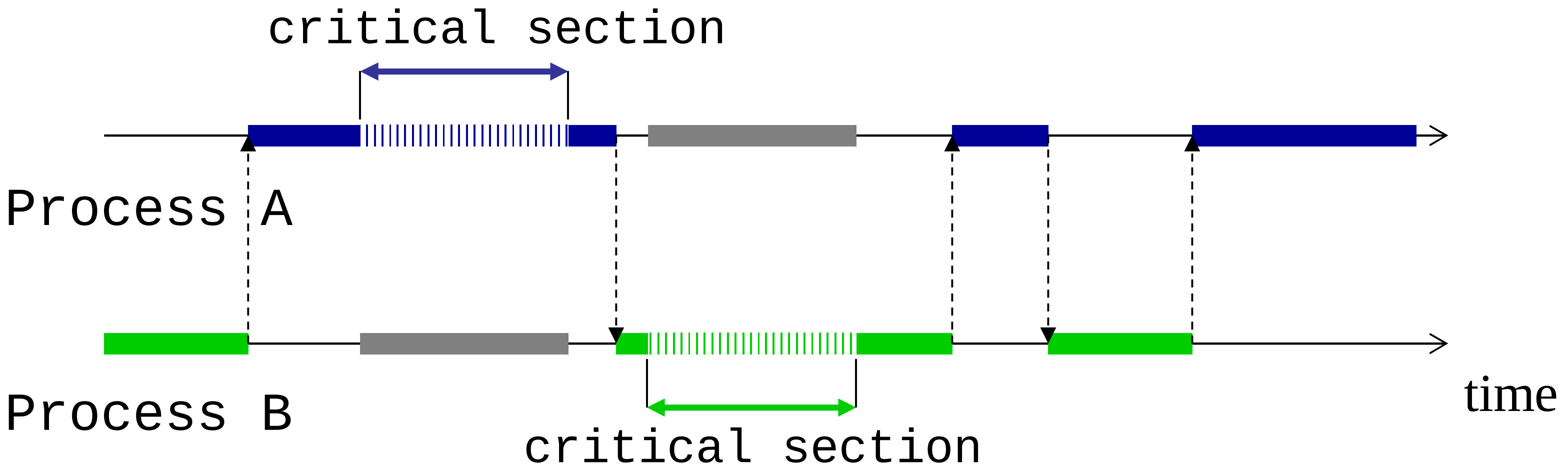
Previous lectures

- Creating processes and threads
  - Creating windows programs
  - Event loops and windows messages
  - Sharing memory between processes
- Process traces
  - Tracing the possible orders of execution
    - Do all possibilities work?
- Atomic Operations
- Spin-locks

Comment

- I've given you a toolbox of code and functions to play with
  - We will see a few more
- From here we move closer to the previous G52CON course
  - G52CON had no labs or coursework
    - Previous students asked for these to be added
  - I also moved us 'closer to the operating system'
- So I would like to repeat my appreciation to Brian Logan for providing his slides
  - Many of the slides in the following lectures are modified versions of his (often changed to my style)

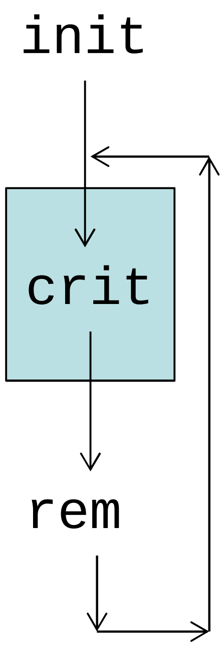
Reminder: critical sections



- Problem: Shared resources/data items can be altered simultanously by multiple threads/processes
- Key solution principle: We **identify *Critical Sections*** and prevent more than one thread being in these at once
- Note: You can have multiple types of these. E.g. to protect different data. We assume one for now

Reminder: Critical Section

- Simple mutual exclusion:
  – Mutually exclude each other – only one at a time
- We can think of the code having a ‘critical section’ that only one thread can run at once
- Consider a generic structure like this:
  1. Initial code (init) – before critical section
  2. Enter critical section – apply protocol here
  3. Critical section code (crit)
  4. Exit critical section – apply protocol here
  5. Remainder (rem) – after critical section



Typical 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
init1;
while(true)
{
    crit1;
    rem1;
}
```

```
// Process 2
init2;
while(true)
{
    crit2;
    rem2;
}
```

```
...
```

```
// Process n
initn;
while(true)
{
    critn;
    remn;
}
```

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 and do any operations:
  - `crit` must execute in a finite time
  - process must not terminate in `crit`
  - `init` and `rem` may be infinite length
  - process *may* terminate in `init` or `rem`
  - `crit` and `rem` may vary from one pass through the `while` loop to the next
  - i.e. not always the same
- With these assumptions it is possible to rewrite *any* process with critical sections into the typical form.

This lecture

- Improved (entry and exit) protocols for ensuring mutual exclusion
  - Spin locks (!)
  - Test and set
  - Dekker's algorithm
- Tomorrow:
  - Peterson's algorithm
  - Operating system support
    - Mutex and CriticalSection objects
  - Disadvantages of critical sections

Properties of a concurrent program

- 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
- Establishing liveness may require proving safety properties

Properties of a good protocol

Mutual Exclusion: at most one process at a time is executing its critical section

i.e. it works to protect the 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

i.e. it doesn't 'get stuck'

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

i.e. can enter critical section if nobody else is in it

Eventual Entry: a process that is attempting to enter its critical section will eventually succeed

i.e. nothing is 'waiting forever'

– Andrews (2000), p 95.

Deadlock vs livelock

A process 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 **`WaitFor…Object()`**
  - Process does **not** consume CPU time
- a **livelocked** process is alive and waiting on the condition, e.g, busy waiting
  - Process **does** consume CPU time

Reminder: Spin locks

- Sit in a tight loop waiting for a variable to take specific values, e.g.:

  `while ( turn != 1 ) ;`

- Variable usually needs to be volatile
  - No point doing this if another thread is not going to alter it
- Thread will always be busy
  - Constantly checking the value
  - Wastes a lot of CPU time

Full round-robin algorithm

Variable: *turn*: integer variable, initialised to 1, **volatile**

- **Thread 1:**

init

Entry protocol:

while (*turn* != 1) ;

crit

Exit protocol:

turn = 2;

rem

- **Thread 2:**

init

Entry protocol:

while (*turn* != 2) ;

crit

Exit protocol:

turn = 1;

rem

Round-robin properties

- Mutual exclusion?
- Absence of deadlock / livelock?
- Absence of unnecessary delay?
- Eventual entry?

Round-robin properties

- Mutual exclusion?
  - Yes, it actually works
- Absence of deadlock / livelock?
  - Yes? as long as nothing stops: turn variable says which can act
  - But strictly speaking: No. If one process dies, the other gets stuck
- Absence of unnecessary delay?
  - No, each must wait for the other to have acted before it can act, so if you run at double speed you wait a lot
  - Also, if one thread dies the other waits unnecessarily forever
- Eventual entry?
  - As long as nothing dies then yes, as long as the other takes finite time
  - Strictly speaking: No, because livelock can occur (see above)

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;
}
```

Simple Spin-Lock Properties

- Mutual exclusion?
- Absence of deadlock / livelock?
- Absence of unnecessary delay?
- Eventual entry?

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

init1;
while(true)

}
```

```
// Process 2

init2;
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`

Simple Spin-Lock Properties

- Mutual exclusion?
  - No – it doesn’t work!
  - There are interleavings which allow both processes to pass their entry protocols
- Absence of deadlock / livelock?
  - Yes.
  - 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?
  - Yes
- Eventual entry?
  - is guaranteed only if the scheduling policy is *strongly fair*.

Simple Spin-Lock Properties

- Mutual exclusion?
  - No – it doesn't work!
  - There are interleavings which allow both processes to pass their entry protocols
- Absenc
  - Yes.
  - if all pr ... ust be *false*, a ... owed to ente
- Absenc
  - 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

Test-and-Set instruction

The Test-and-Set (atomic) instruction effectively executes the function

```
bool TS(bool lock)
{
    bool v = lock;
    lock = true; // Set true
    return v; // Old lock value
}
```

See InterlockedExchange: https://msdn.microsoft.com/en-us/library/windows/desktop/ms683590%28v=vs.85%29.aspx

Spin lock using Test-and-Set

```
// Process i

initᵢ;
while(true) {
  while (TS(lock)) {}  // entry protocol
  critᵢ;
  lock = false;        // exit protocol
  remᵢ;
}
```

```
// 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

init1;
while(true)

}
```

```
// Process 2

init2;
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

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

}
```

```
// Process 2

init2;
while(true) {
    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

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

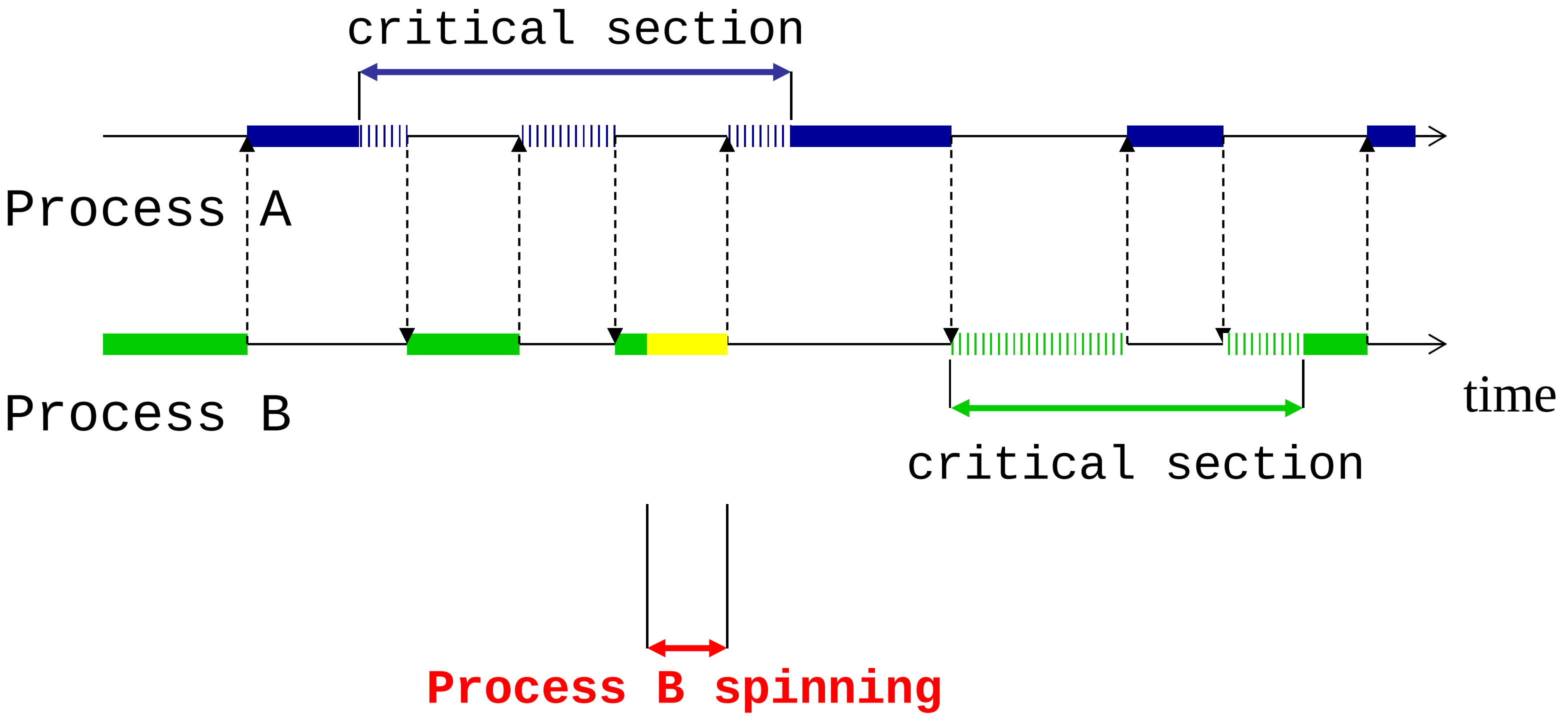
```
// Process 2

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


}
```

`lock == true`

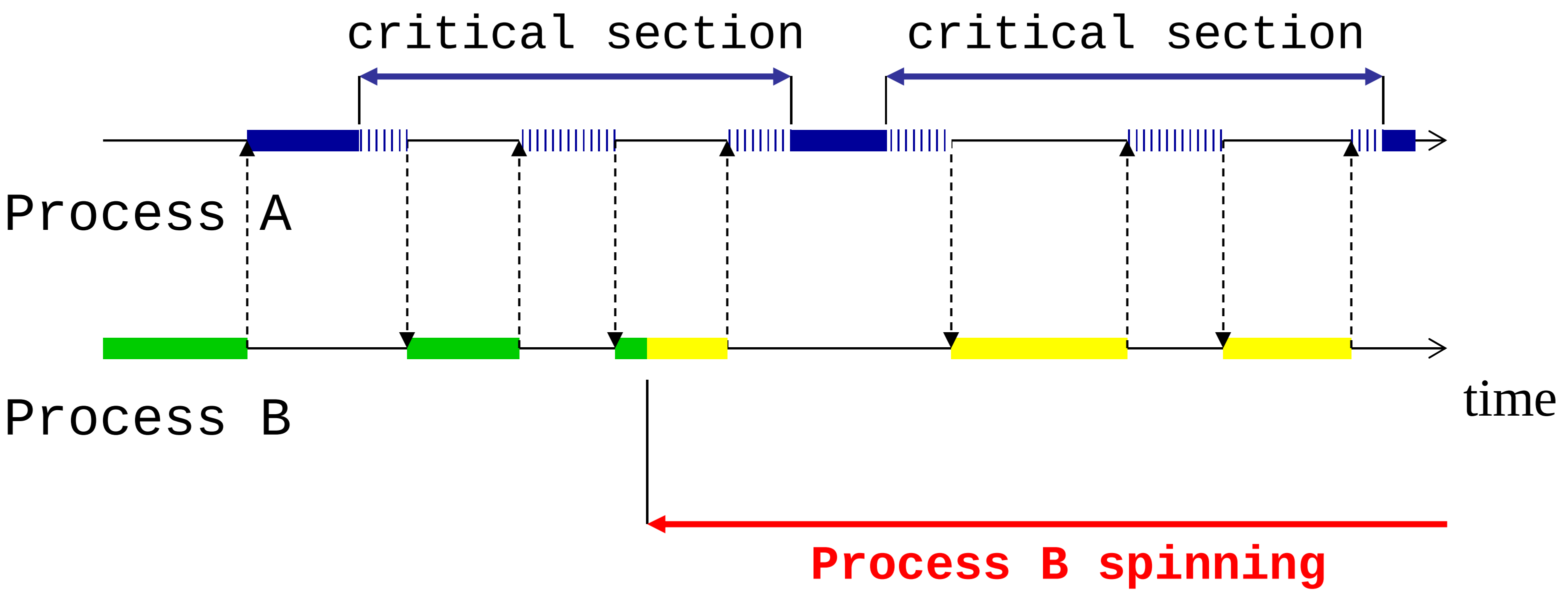
Properties of the Test-and-Set solution

- 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*.

Overhead of spin locks



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 (with enough tries it gets a go)
- Since all processes execute the same protocol it works for any number of processes

Multi-variable non-atomic

```
// Process 1
init1;

while(true)
{
    c1 = 0; // entry protocol
    while (c2 == 0)
        ;
    crit1;
    c1 = 1; // exit protocol
    rem1;
}
```

```
// Process 2
init2;

while(true)
{
    c2 = 0; // entry protocol
    while (c1 == 0)
        ;
    crit2;
    c2 = 1; // exit protocol
    rem2;
}
```

`c1 == 1`   `c2 == 1`

2 variables, c1 and c2. Each thread has its own variable and is the only thread to alter that variable. It sets it to 0 when wanting to enter the critical section, and 1 on leaving.

Multi-variable non-atomic (1)

```
// Process 1
init1;

while(true)
{
    c1 = 0; // entry protocol
    while (c2 == 0)
        ;
    crit1;
    c1 = 1; // exit protocol
    rem1;
}
```

```
// Process 2
init2;

while(true)
{
    c2 = 0; // entry protocol
    while (c1 == 0)
        ;
    crit2;
    c2 = 1; // exit protocol
    rem2;
}
```

`c1 == 0`   `c2 == 1`

2 variables, c1 and c2. Each thread has its own variable and is the only thread to alter that variable. It sets it to 0 when wanting to enter the critical section, and 1 on leaving.

Multi-variable non-atomic (2)

```
// Process 1                    // Process 2
init1;                          init2;

while(true)                     while(true)
{                               {
    c1 = 0; // entry protocol       c2 = 0; // entry protocol
    while (c2 == 0)                 while (c1 == 0)
        ;                               ;
    crit1;                          crit2;
    c1 = 1; // exit protocol        c2 = 1; // exit protocol
    rem1;                           rem2;
}                               }
```

```
c1 == 0      c2 == 1
```

2 variables, c1 and c2. Each thread has its own variable and is the only thread to alter that variable. It sets it to 0 when wanting to enter the critical section, and 1 on leaving.

Multi-variable non-atomic (3)

```
// Process 1
init1;

while(true)
{
    c1 = 0; // entry protocol
    while (c2 == 0)
        ;
    crit1;
    c1 = 1; // exit protocol
    rem1;
}
```

```
// Process 2
init2;

while(true)
{
    c2 = 0; // entry protocol
    while (c1 == 0)
        ;
    crit2;
    c2 = 1; // exit protocol
    rem2;
}
```

`c1 == 0`   `c2 == 0`

2 variables, c1 and c2. Each thread has its own variable and is the only thread to alter that variable. It sets it to 0 when wanting to enter the critical section, and 1 on leaving.

Multi-variable non-atomic (4)

```
// Process 1
init1;

while(true)
{
    c1 = 0; // entry protocol
    while (c2 == 0)
        ;
    crit1;
    c1 = 1; // exit protocol
    rem1;
}
```

```
// Process 2
init2;

while(true)
{
    c2 = 0; // entry protocol
    while (c1 == 0)
        ;
    crit2;
    c2 = 1; // exit protocol
    rem2;
}
```

`c1 == 0`   `c2 == 0`

2 variables, c1 and c2. Each thread has its own variable and is the only thread to alter that variable. It sets it to 0 when wanting to enter the critical section, and 1 on leaving.

Multi-variable non-atomic (5)

```
// Process 1
init1;

while(true)
{
    c1 = 0; // entry protocol
    while (c2 == 0)
        ;
    crit1;
    c1 = 1; // exit protocol
    rem1;
}
```

```
// Process 2
init2;

while(true)
{
    c2 = 0; // entry protocol
    while (c1 == 0)
        ;
    crit2;
    c2 = 1; // exit protocol
    rem2;
}
```

`c1 == 1`   `c2 == 0`

2 variables, c1 and c2. Each thread has its own variable and is the only thread to alter that variable. It sets it to 0 when wanting to enter the critical section, and 1 on leaving.

Multi-variable non-atomic (6)

```
// Process 1
init1;

while(true)
{
    c1 = 0; // entry protocol
    while (c2 == 0)
        ;
    crit1;
    c1 = 1; // exit protocol
    rem1;
}
```

```
// Process 2
init2;

while(true)
{
    c2 = 0; // entry protocol
    while (c1 == 0)
        ;
    crit2;
    c2 = 1; // exit protocol
    rem2;
}
```

`c1 == 1`   `c2 == 0`

2 variables, c1 and c2. Each thread has its own variable and is the only thread to alter that variable. It sets it to 0 when wanting to enter the critical section, and 1 on leaving.

Multi-variable non-atomic (7)

```
// Process 1
init1;

while(true)
{
    c1 = 0; // entry protocol
    while (c2 == 0)
        ;
    crit1;
    c1 = 1; // exit protocol
    rem1;
}
```

```
// Process 2
init2;

while(true)
{
    c2 = 0; // entry protocol
    while (c1 == 0)
        ;
    crit2;
    c2 = 1; // exit protocol
    rem2;
}
```

```
c1 == 0     c2 == 0
```

2 variables, c1 and c2. Each thread has its own variable and is the only thread to alter that variable. It sets it to 0 when wanting to enter the critical section, and 1 on leaving.

Multi-variable non-atomic (8)

```
// Process 1
init1;

while(true)
{
    c1 = 0; // entry protocol
    while (c2 == 0)
        ;
    crit1;
    c1 = 1; // exit protocol
    rem1;
}
```

```
// Process 2
init2;

while(true)
{
    c2 = 0; // entry protocol
    while (c1 == 0)
        ;
    crit2;
    c2 = 1; // exit protocol
    rem2;
}
```

`c1 == 0`   `c2 == 0`

2 variables, c1 and c2. Each thread has its own variable and is the only thread to alter that variable. It sets it to 0 when wanting to enter the critical section, and 1 on leaving.

Multi-variable non-atomic (9)

```
// Process 1
init1;

while(true)
{
    c1 = 0; // entry protocol
    while (c2 == 0)
        ;
    crit1;
    c1 = 1; // exit protocol
    rem1;
}
```

```
// Process 2
init2;

while(true)
{
    c2 = 0; // entry protocol
    while (c1 == 0)
        ;
    crit2;
    c2 = 1; // exit protocol
    rem2;
}
```

```
c1 == 0        c2 == 1
```

2 variables, c1 and c2. Each thread has its own variable and is the only thread to alter that variable. It sets it to 0 when wanting to enter the critical section, and 1 on leaving.

Multi-variable non-atomic (10)

```
// Process 1
init1;

while(true)
{
    c1 = 0; // entry protocol
    while (c2 == 0)
        ;
    crit1;
    c1 = 1; // exit protocol
    rem1;
}
```

```
// Process 2
init2;

while(true)
{
    c2 = 0; // entry protocol
    while (c1 == 0)
        ;
    crit2;
    c2 = 1; // exit protocol
    rem2;
}
```

`c1 == 0`   `c2 == 1`

2 variables, c1 and c2. Each thread has its own variable and is the only thread to alter that variable. It sets it to 0 when wanting to enter the critical section, and 1 on leaving.

Multi-variable non-atomic (11)

```
// Process 1
init1;

while(true)
{
    c1 = 0; // entry protocol
    while (c2 == 0)
        ;
    crit1;
    c1 = 1; // exit protocol
    rem1;
}
```

```
// Process 2
init2;

while(true)
{
    c2 = 0; // entry protocol
    while (c1 == 0)
        ;
    crit2;
    c2 = 1; // exit protocol
    rem2;
}
```

```
c1 == 1      c2 == 0
```

2 variables, c1 and c2. Each thread has its own variable and is the only thread to alter that variable. It sets it to 0 when wanting to enter the critical section, and 1 on leaving.

Multi-variable non-atomic (12)

```
// Process 1
init1;

while(true)
{
    c1 = 0; // entry protocol
    while (c2 == 0)
        ;
    crit1;
    c1 = 1; // exit protocol
    rem1;
}
```

```
// Process 2
init2;

while(true)
{
    c2 = 0; // entry protocol
    while (c1 == 0)
        ;
    crit2;
    c2 = 1; // exit protocol
    rem2;
}
```

`c1 == 0`   `c2 == 0`

2 variables, c1 and c2. Each thread has its own variable and is the only thread to alter that variable. It sets it to 0 when wanting to enter the critical section, and 1 on leaving.

Multi-variable non-atomic (13)

```
// Process 1
init1;

while(true)
{
    c1 = 0; // entry protocol
    while (c2 == 0)
        ;
    crit1;
    c1 = 1; // exit protocol
    rem1;
}
```

```
// Process 2
init2;

while(true)
{
    c2 = 0; // entry protocol
    while (c1 == 0)
        ;
    crit2;
    c2 = 1; // exit protocol
    rem2;
}
```

`c1 == 0` `c2 == 0`

2 variables, c1 and c2. Each thread has its own variable and is the only thread to alter that variable. It sets it to 0 when wanting to enter the critical section, and 1 on leaving.

Problem

- Both processes/threads got stuck
  - Live-lock – using a lot of CPU time
- This process stopped both entering the critical section at once
- BUT! You can get live lock with neither entering
- Dekker’s algorithm solves this problem by building a turn order on top of this basic system
  - If they both get stuck then the turn order says who can go now – the other one releases its lock

Dekker's algorithm

```
// Process 1
init1;
while(true) {
    c1 = 0;    // entry protocol
    while (c2 == 0) {
        if (turn == 2) {
            c1 = 1;
            while (turn == 2) ;
            c1 = 0;
        }
    }
    crit1;
    turn = 2; // exit protocol
    c1 = 1;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    c2 = 0;    // entry protocol
    while (c1 == 0) {
        if (turn == 1) {
            c2 = 1;
            while (turn == 1) ;
            c2 = 0;
        }
    }
    crit2;
    turn = 1; // exit protocol
    c2 = 1;
    rem2;
}
```

`c1 == 1 c2 == 1 turn == 1`

Don’t do this at home…

… or using VS2013 in the labs

- Just a warning: if you try Dekker’s algorithm or Peterson’s algorithm (next lecture):
- Making the variables volatile is not enough to make it work on modern computers
- Modern processors may re-order the code
  - May no longer work
- Need a LOT of memory barriers to fix this
- It would work in Java (see later lectures)

Next lecture

- Tomorrow:
  - Peterson's algorithm
  - Operating system support
    - Mutex and CriticalSection objects
  - Disadvantages of critical sections