Synchronization

Concurrency Problems

A statement like w-- in C (or C++) is implemented by several machine instructions:

```
ld      r4, #w
add     r4, r4, -1
st      r4, #w
```

Now, imagine the following sequence, what is the value of w?

```
ld      r4, #w          C.S.        _____________
_____________                       ld      r4, #w
_____________                       add     r4, r4, -1
_____________                       st      r4, #w
add     r4, r4, -1
st      r4, #w
```

Concurrency Problems

```
int w = 4;
main() {
    …
    w--;
    …
}
sig_handler() {
    …
    w--;
}
```

```
                ld      r4, #w
signal posted
                add     r4, r4, -1
                st      r4, #w
```

Concurrency Problems

- Concurrency problem can occur in user-programs because of:
  - Multithreading
    - Two threads may modify the same variables concurrently
  - Signal handling (single-threaded or multithreaded)
    - If the program receives a signal in the middle of modifying a global variable and the signal handler modifies that variable
- Operating system kernels
  - The kernel could be multithreaded (as in multiprocessors)
  - Interrupt handling (single threaded or multithreaded)
    - An interrupt may occur in the middle of modifying a kernel data structure

The Fundamental Issue

- In all these cases, what we thought to be an *atomic* operation is not done atomically by the machine

Definition: An atomic operation is one that executes to completion without any interruption or failure

- An atomic operation has "an all or nothing" flavor:
  - Either it executes to completion, or
  - it did not execute at all, and
  - it executes without interruptions

Critical Sections

- A critical section is an abstraction that
  - consists of a number of consecutive program instructions
  - all code within the section executes atomically
- In the previous example, the w-- statement should be placed in a critical section
- Critical sections are used profusely in an OS to protect data structures
  - Queues
  - shared variables
  - interrupt handlers

Solutions

There are many solutions to implement critical sections:

- Hardware *vs.* software
  - some solutions require hardware support, kernel support, a combination thereof, or no special support
- Multithreaded *vs.* single-threaded
  - some solutions are for single-threaded only
- Busy waiting *vs* blocking
  - in busy-waiting (spin-locks), a thread retains the CPU, and loops until it can enter the critical section
  - in blocking, a thread gives up the CPU, and waits on some queue until it gets notified that it can enter the critical section

Solution Desiderata

A critical section implementation must meet these criteria:

- correctness: only one thread can execute in the critical section at any given time
- efficiency: getting into and out of the critical section must be fast
- concurrency control: a good implementation allows maximum concurrency while preserving correctness
- flexibility: a good implementation must have as few restrictions as practically possible

The Problem

Thread T_0

```
while (!terminate) {
    <entry0>
    CS
    <exit0>
    NCS0
}
```

Thread T_1

```
while (!terminate) {
    <entry1>
    CS
    <exit1>
    NCS1
}
```

Safety and Liveness

- *Safety property* : "nothing bad happens"
  - Windows™ never crashes
  - if one general attacks, both do
  - a program never terminates with a wrong answer

  – holds in every finite execution prefix

  – can often been satisfied by doing nothing
- *Liveness property*: "something good eventually happens"
  - Windows™ always reboots
  - both generals eventually attack
  - a program eventually terminates

  – no partial execution is irremediable

Solution Correctness

A correct solution to the critical section problem must satisfy:

Safety: at most one thread may be executing in the critical section (mutual exclusion)

Liveness: If one of more threads are in their `entry` code, then eventually at least one of them enters the critical section.

We can impose a further requirement:

Bounded waiting: If thread i is in $\texttt{entry}_i$, then there is a bound on the number of times that other threads are allowed to enter the critical section before thread i's request is granted

Is bounded waiting a safety or a liveness property?

Solution 1: take turns

Thread T_0

```
while (!terminate) {
    while turn ≠ 0;
    CS
    turn := 1;
    NCS0
}
```

Thread T_1

```
while (!terminate) {
    while turn ≠ 1;
    CS
    turn := 0;
    NCS1
}
```

init: *turn* = 0

Safe?

```
Thread T0                       Thread T1
while (!terminate) {            while (!terminate) {
    while (turn ≠ 0);               while (turn ≠ 1);
    CS0                             CS1
    turn := 1;                      turn:=0;
    NCS0                            NCS1
            }                               }
```

$at(CS_0) \Rightarrow turn = 0$ $\quad$ $at(CS_1) \Rightarrow turn = 1$

$at(CS_0) \wedge at(CS_1) \Rightarrow (turn = 0) \wedge (turn = 1) = FALSE$

Live?

```
Thread T0                       Thread T1
while (!terminate) {            while (!terminate) {
    while (turn ≠ 0);               while (turn ≠ 1);
    CS0                             CS1
    turn := 1;                      turn:=0;
    NCS0                            NCS1
            }                               }
```

What happens if T_1 terminates before T_0?

Bounded waiting?

```
Thread T0                       Thread T1
while (!terminate) {            while (!terminate) {
    while (turn ≠ 0);               while (turn ≠ 1);
    CS0                             CS1
    turn := 1                       turn:=0
    NCS0                            NCS1
          }                               }
```

Yes, and the bound is 1!

Solution 2: is the CS vacant?

```
Thread T0                       Thread T1
while (!terminate) {            while (!terminate) {
    while in;                       while in;
    in:= true                       in:=true
      CS                              CS
    in:= false                      in:=false
    NCS0                            NCS1
}                               }
```

init: *in* = false

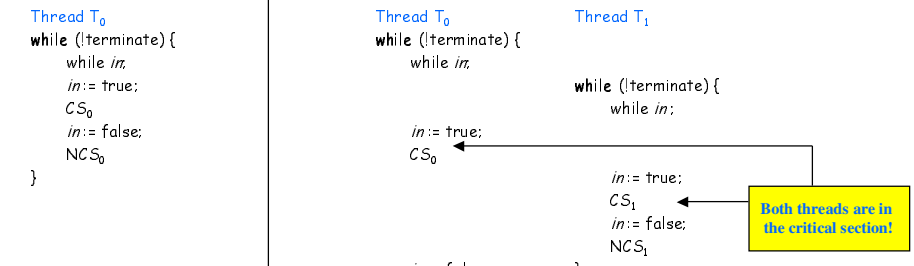
Safe?

Thread T_0
```
while (!terminate) {
    while in;
    in := true;
    CS0
    in := false;
    NCS0
}
```

Thread T_1
```
while (!terminate) {
    while in;
    in := true;
    CS1
    in := false;
    NCS1
}
```

Thread T_0
```
while (!terminate) {
    while in;
    in := true;
    CS0
    in := false;
    NCS0
}
```

Thread T_1
```
while (!terminate) {
    while in;
    in := true;
    CS1
    in := false;
    NCS1
}
```



Live?

Thread T_0
```
while (!terminate) {
    while in;
    in := true;
    CS0
    in := false;
    NCS0
}
```

Thread T_1
```
while (!terminate) {
    while in;
    in := true;
    CS1
    in := false;
    NCS1
}
```

Yes. in = true $\Rightarrow$ eventually in = false

Bounded waiting?

```
Thread T0                    Thread T1
while (!terminate) {         while (!terminate) {
    while in;                    while in;
    in := true;                  in := true;
    CS0                          CS1
    in := false;                 in := false;
    NCS0                         NCS1
}                            }
```

No. T_0 could starve T_1 (or viceversa)

What did go wrong?

- We were setting *in* too late…
- What about

```
Thread T0                    Thread T1
while (!terminate) {         while (!terminate) {
    in := true;                  in := true;
    while in;                    while in;
    CS0                          CS1
    in := false;                 in := false;
    NCS0                         NCS1
}                            }
```

- What about using two different *in* variables?

Solution 2.1: is the CS free? (smart version)

Thread T_0

```
while (!terminate) {
    in0 := true
    while in1;
      CS
    in0 := false
    NCS0
}
```

Thread T_1

```
while (!terminate) {
    in1 := true
    while in0;
      CS
    in1 := false
    NCS1
}
```

init: $in_0 = in_1 =$ false

Safe?

Thread T_0

```
while (!terminate) {
    in0 := true
    while in1;
    CS0
    in0 := false;
    NCS0
}
```

Thread T_1

```
while (!terminate) {
    in1 := true
    while in0;
    CS1
    in1 := false;
    NCS1
}
```

Yes. Informally:

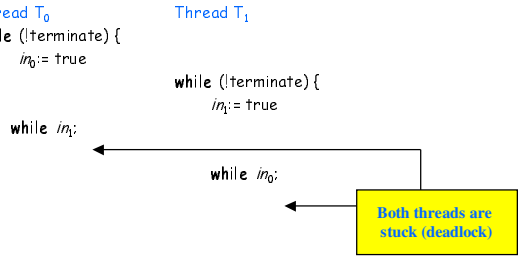
$at(CS_0) \Rightarrow in_0 \Rightarrow T_1$ cannot enter CS_1
$at(CS_1) \Rightarrow in_1 \Rightarrow T_0$ cannot enter CS_0

true in sequential execution, not interfered with

Live?

Thread T_0
```
while (!terminate) {
    in0 := true
    while in1;
    CS0
    in0 := false;
    NCS0
}
```

Thread T_1
```
while (!terminate) {
    in1 := true
    while in0;
    CS1
    in1 := false;
    NCS1
}
```



Bounded waiting?

Thread T_0
```
while (!terminate) {
    in0 := true
    while in1;
    CS0
    in0 := false;
    NCS0
}
```

Thread T_1
```
while (!terminate) {
    in1 := true
    while in0;
    CS1
    in1 := false;
    NCS1
}
```

Not a prayer.

Solution 3: combine ideas of 1 and 2

Variables:

- in_i: thread T_i is executing , or attempting to execute, in CS
- $turn$: id of thread that is allowed to enter CS if multiple would like to

Claim: We can achieve mutual exclusion if the following invariant holds before entering the critical section:

$\{(\neg in_j \vee (in_j \wedge turn = i)) \wedge in_i\}$
CS
……..
$in_i = false$

$((\neg in_0 \vee (in_0 \wedge turn = 1)) \wedge in_1) \wedge$
$((\neg in_1 \vee (in_1 \wedge turn = 0)) \wedge in_0)$
$\Rightarrow$
$((turn = 0) \wedge (turn = 1)) \Rightarrow false$

Towards a solution

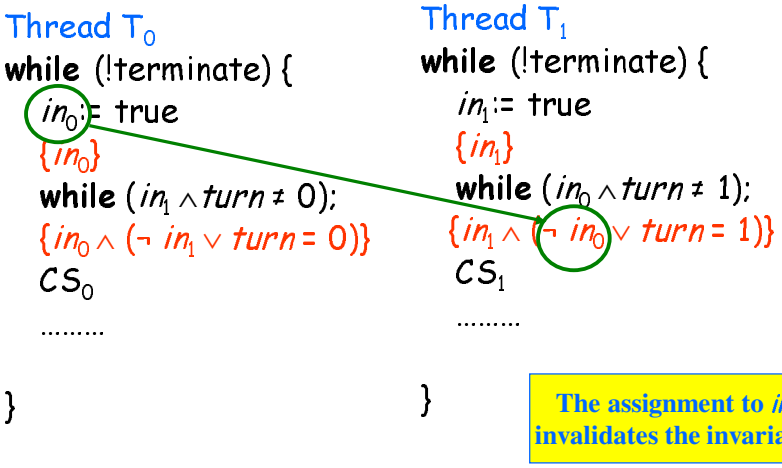
The problem boils down to establishing the following right after $entry_i$

$(\neg in_j \vee (in_j \wedge turn = i)) \wedge in_i \;=\; (\neg in_j \wedge turn = i) \wedge in_i$

How can we do that?

$entry_i =$ in_i := true;
while ($in_j \wedge turn \neq i$);

We hit a snag



What can we do?

Add assignment to *turn* to establish the second disjunct

Thread T_0

```
while (!terminate) {
    in0 := true;
α0  turn := 1;
    {in0}
    while (in1 ∧ turn ≠ 0);
    {in0 ∧ (¬ in1 ∨ turn = 0)}
    CS0
    in0 := false;
    NCS0
}
```

Thread T_1

```
while (!terminate) {
    in1 := true;
α1  turn := 0;
    {in1}
    while (in0 ∧ turn ≠ 1);
    {in1 ∧ (¬ in0 ∨ turn = 1)}
    CS1
    in1 := false;
    NCS1
}
```

Safe?

Thread T_0

```
while (!terminate) {
    in0 := true;
α0  turn := 1;
    {in0}
    while (in1 ∧ turn ≠ 0);
    {in0 ∧ (¬ in1 ∨ turn = 0)}
    CS0
    in0 := false;
    NCS0
}
```

Thread T_1

```
while (!terminate) {
    in1 := true;
α1  turn := 0;
    {in1}
    while (in0 ∧ turn ≠ 1);
  {in1 ∧ (¬ in0 ∨ turn = 1)}
    CS1
    in1 := false;
    NCS1
}
```

If both in CS, then

$in_0 \wedge (\neg in_j \vee at(\alpha_1) \vee turn = 0) \wedge in_1 \wedge (\neg in_0 \vee at(\alpha_0) \vee turn = 1) \wedge$

$\wedge \neg at(\alpha_0) \wedge \neg at(\alpha_1) = (turn = 0) \wedge (turn = 1) = false$

Live?

Thread T_0

```
while (!terminate) {
    {S1: ¬in0 ∧ (turn = 1 ∨ turn = 0)}
    in0 := true;
    {S2: in0 ∧ (turn = 1 ∨ turn = 0)}
α0  turn := 1;
    {S2}
    while (in1 ∧ turn ≠ 0);
    {S3: in0 ∧ (¬ in1 ∨ at(α1) ∨ turn = 0)}
    CS0
    {S3}
    in0 := false;
    {S1}
    NCS0
}
```

Thread T_1

```
while (!terminate) {
    {R1: ¬in0 ∧ (turn = 1 ∨ turn = 0)}
    in1 := true;
    {R2: in0 ∧ (turn = 1 ∨ turn = 0)}
α1  turn := 0;
    {R2}
    while (in0 ∧ turn ≠ 1);
    {R3: in1 ∧ (¬ in0 ∨ at(α0) ∨ turn = 1)}
    CS1
    {R3}
    in1 := false;
    {R1}
    NCS1
}
```

Non-blocking: T_0 before NCS_0, T_1 at $entry_1$

$S_1 \wedge R_2 \wedge in_0 \wedge (turn = 0) = \neg in_0 \wedge in_1 \wedge in_0 \wedge (turn = 0) = false$

Deadlock-free: T_1 and T_0 at **while,** before entering the critical section

$S_2 \wedge R_2 \wedge (in_0 \wedge (turn = 0)) \wedge (in_1 \wedge (turn = 1)) \Rightarrow (turn = 0) \wedge (turn = 1) = false$

Bounded waiting?

Thread T_0

```
while (!terminate) {
    in0 := true;
    turn := 1;
    while (in1 ∧ turn ≠ 0);
    CS0
    in0 := false;
    NCS0
}
```

Thread T_0

```
while (!terminate) {
    in1 := true;
    turn := 0;
    while (in0 ∧ turn ≠ 1);
    CS0
    in1 := false;
    NCS0
}
```

Yup!

Disabling Interrupts

- For a single-threaded system, interrupts are the only source of concurrency problems (hardware or software)
- It follows that a simple solution to implement critical section is:
  - Disable interrupts at beginning of critical section
  - Enable interrupts at the end of critical section
- For OS'es, disabling hardware interrupts is very dangerous
  - Can lead to delaying response to an important event
  - Prevent the OS from being used in a multiprocessor

Critical sections must be kept short

Synchronization Instructions

- Relying on test-and-set instructions
  - implemented in hardware
  - it's a machine instruction that:
    - reads a value from memory
    - tests the value against some constant
    - if the test true it sets the value in memory to a different value
    - reports the result of the test in a flag (e.g. carry or zero flag)
- This instruction executes atomically
  - it locks the memory bus while executing
  - does not scale very well in multiprocessor environments with a shared bus

Example

```
lock = 0;

while(! test-and-set(lock, 0, 1));
w--;
lock = 0;
```

Works because of the direct hardware support for atomicity

- multithreaded or single threaded environments
- must be used with care
- does not scale to large number of threads/processors