Peterson's lock: A derivation

- Combines ideas from Like-to and Selfless locks
- Two variables:
  - in_i : thread T_i is executing in CS, or trying to do so
  - $turn$: id of thread allowed to enter CS under contention ($turn = i \Leftrightarrow victim = 1 - i$)

Establishing mutual exclusion

- Claim: If the following invariant holds when T_i is in CS, so does mutual exclusion

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

in_i wants to enter CS

in_j does not desire to enter CS

in_j wants to enter CS, but it is in_i's turn

Establishing mutual exclusion

- Claim: If the following invariant holds when T_i is in CS, so does mutual exclusion

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

in_i wants to enter CS

in_j does not desire to enter CS

in_j wants to enter CS, but it is in_i's turn

- Why?
  - If we instantiate the invariant for both T_1 and T_2...
  - ...and take the conjunction of the two invariants (i.e., assume both threads in CS)...
  - ...it boils down to $(turn = 0 \wedge turn = 1) = false$

Towards a solution

- The problem then reduces to establishing the following:

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

- How can we do that?

Towards a solution

- The problem then boils down to establishing the following:

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

- How can we do that?

$lock.acquire()$: $in_i := true$
while $(in_j \wedge turn \neq i)$;

if we can get past this loop, $(\neg in_j \vee turn \neq i)$ must hold!

Establishing the invariant

Thread T_0

while(!terminate) {
$in_0 := true$
$\{in_0\}$

while $(in_1 \wedge turn \neq 0)$;
$\{in_0 \wedge (\neg in_1 \vee turn = 0)\}$
CS_0
...
$in_0 := false$
NCS_0
}

Thread T_1

while(!terminate) {
$in_1 := true$
$\{in_1\}$

while $(in_0 \wedge turn \neq 1)$;
$\{in_1 \wedge (\neg in_0 \vee turn = 1)\}$
CS_1
...
$in_1 := false$
NCS_1
}

Establishing the invariant

- Add assignment to $turn$ to establish second disjunct

Thread T_0

while(!terminate) {
$in_0 := true$
$\{in_0\}$

while $(in_1 \wedge turn \neq 0)$;
$\{in_0 \wedge (\neg in_1 \vee turn = 0)\}$
CS_0
...
$in_0 := false$
NCS_0
}

Thread T_1

while(!terminate) {
$in_1 := true$
$\{in_1\}$
$turn := 0$
while $(in_0 \wedge turn \neq 1)$;
$\{in_1 \wedge (\neg in_0 \vee turn = 1)\}$
CS_1
...
$in_1 := false$
NCS_1
}

Establishing the invariant

- Add assignment to $turn$ to establish second disjunct

Thread T_0

while(!terminate) {
$in_0 := true$
$\{in_0\}$
$turn := 1$
while $(in_1 \wedge turn \neq 0)$;
$\{in_0 \wedge (\neg in_1 \vee turn = 0)\}$
CS_0
...
$in_0 := false$
NCS_0
}

Thread T_1

while(!terminate) {
$in_1 := true$
$\{in_1\}$
$turn := 0$
while $(in_0 \wedge turn \neq 1)$;
$\{in_1 \wedge (\neg in_0 \vee turn = 1)\}$
CS_1
...
$in_1 := false$
NCS_1
}

Are we there yet?

Thread T_0

while(!terminate) {
$in_0 := true$
$\{in_0\}$
$turn := 1$
while $(in_1 \wedge turn \neq 0)$;
$\{in_0 \wedge (\neg in_1 \vee turn = 0 \vee at(\alpha_1))\}$
CS_0
...
$in_0 := false$
NCS_0
}

Thread T_1

T_1's PC →
auxiliary variable
while(!terminate) {
$in_1 := true$
$\{in_1\}$
α_1 $turn := 0$
while $(in_0 \wedge turn \neq 1)$;
$\{in_1 \wedge (\neg in_0 \vee turn = 1)\}$
CS_1
...
$in_1 := false$
NCS_1
}

...and we are gold!

Thread T_0

while(!terminate) {
$in_0 := true$
$\{in_0\}$
α_0 $turn := 1$
while $(in_1 \wedge turn \neq 0)$;
$\{in_0 \wedge (\neg in_1 \vee turn = 0 \vee at(\alpha_1))\}$
CS_0
...
$in_0 := false$
NCS_0
}

Thread T_1

while(!terminate) {
$in_1 := true$
$\{in_1\}$
α_1 $turn := 0$
while $(in_0 \wedge turn \neq 1)$;
$\{in_1 \wedge (\neg in_0 \vee turn = 1 \vee at(\alpha_0))\}$
CS_1
...
$in_1 := false$
NCS_1
}

Peterson: Safety

Thread T_0

while(!terminate) {
$in_0 := true$
$\{in_0\}$
α_0 $turn = 1$
while $(in_1 \wedge turn \neq 0)$;
$\{in_0 \wedge (\neg in_1 \vee turn = 0 \vee at(\alpha_1))\}$
CS_0
...
$in_0 = false$
NCS_0
}

Thread T_1

while(!terminate) {
$in_1 := true$
$\{in_1\}$
α_1 $turn = 0$
while $(in_0 \wedge turn \neq 1)$
$\{in_1 \wedge (\neg in_0 \vee turn = 1 \vee at(\alpha_0))\}$
CS_1
...
$in_1 := false$
NCS_1
}

If both in the critical section, then:

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

Peterson: Non-blocking

while(!terminate) {
$\{R_1 : \neg in_0 \wedge (turn = 1 \vee turn = 0)\}$
$in_0 = true$
$\{R_2 : in_0 \wedge (turn = 1 \vee turn = 0)\}$
α_0 $turn = 1$
$\{R_2\}$
while $(in_1 \wedge turn \neq 0)$;
$\{R_3 : in_0 \wedge (\neg in_1 \vee turn = 0 \vee at(\alpha_1))\}$
CS_0
$\{R_3\}$
$in_0 = false$
$\{R_1\}$
T_0's PC → NCS_0
}

T_1's PC →
while(!terminate) {
$\{S_1 : \neg in_1 \wedge (turn = 1 \vee turn = 0)\}$
$in_1 := true$
$\{S_2 : in_1 \wedge (turn = 1 \vee turn = 0)\}$
α_1 $turn := 0$
$\{S_2\}$
while $(in_0 \wedge turn \neq 1)$;
$\{S_3 : in_1 \wedge (\neg in_0 \vee turn = 1 \vee at(\alpha_0))\}$
CS_1
$\{S_3\}$
$in_1 = false$
$\{S_1\}$
NCS_1
}

Blocking Scenario: T_0 before NCS_0, T_1 stuck at while loop

$$R_1 \wedge S_2 \wedge in_0 \wedge (turn = 0) = \neg in_0 \wedge in_1 \wedge in_0 \wedge (turn = 0) = false$$

Peterson: Deadlock-free

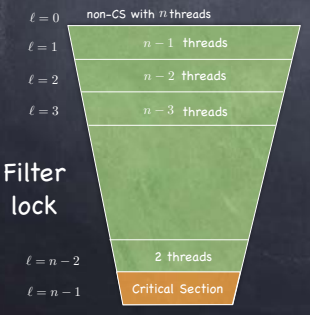
```
while(!terminate) {                    while(!terminate) {
  {R1 : ¬in0 ∧ (turn = 1 ∨ turn = 0)}    {S1 : ¬in1 ∧ (turn = 1 ∨ turn = 0)}
   in0 = true                             in1 := true
   {R2 : in0 ∧ (turn = 1 ∨ turn = 0)}     {S2 : in1 ∧ (turn = 1 ∨ turn = 0)}
α0 turn = 1                            α1 turn := 0
   {R2}                                   {S2}
T0's PC → while(in1 ∧ turn ≠ 0);       T1's PC → while(in0 ∧ turn ≠ 1);
   {R3 : in0 ∧ (¬in1 ∨ turn = 0 ∨ at(α1))}   {S3 : in1 ∧ (¬in0 ∨ turn = 1 ∨ at(α0))}
   CS0                                    CS1
   {R3}                                   {S3}
   in0 = false                            in1 = false
   {R1}                                   {S1}
   NCS0                                   NCS1
}                                      }
```

Blocking Scenario: T_0 and T_1 at the while loop, before entering critical section

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

Taking Stock - I

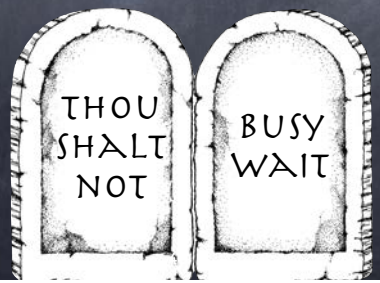
- The critical section problem is very subtle...
- ...and our solution only covers two threads!
  - with multiple threads it gets even harder!



Bakery algorithm

Taking Stock - II

- Our current solutions implement spinlocks
  - threads busy-wait while waiting for lock to be released
    - wastes cycle
    - can lead to priority inversion



A Better Way

- How can we do better?
  - Use hardware to support atomic operations beyond load and store
  - Define higher-level programming abstractions that leverage hardware support

Only on

Disabling Interrupts for Mutual Exclusion

uni-processors

```
lock.acquire()   { disable interrupts}
lock.release()   { enable interrupts}
```

- Simple, but flawed
  - thread may never give up CPU!
  - even if it does, it could it take too long to respond to an interrupt

Only on

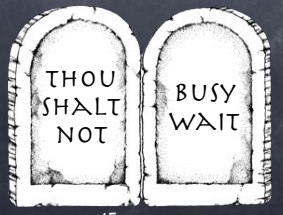
Disabling Interrupts: A Refinement

uni-processors

- Use a variable to implement the lock; enforce mutual exclusion only on the operations that test and set that variable

```
class lock { int value := FREE }

lock.acquire() {
   Disable interrupts;
   while (value == BUSY) {
      Enable interrupts;
      Disable interrupts;
   }
   value := BUSY;
   Enable interrupts;
}
```




```
lock.release() {
   Disable interrupts;
   value := FREE;
   Enable interrupts;
}
```


Only on

Disabling Interrupts: No Busy-Waiting

uni-processors

- Keep a queue of threads waiting for the lock

```
class lock { int value := FREE }

lock.acquire() {
   Disable interrupts;
   if (value == BUSY) {
      Put on queue of threads
        waiting for lock;
      Go to sleep;
   } else {
      value := BUSY;
   }
   Enable interrupts;
}
```

```
lock.release() {
   Disable interrupts;
   if anyone on wait queue {
      Take a waiting thread off wait queue
      and put it at the front of ready queue;
   } else {
      value := FREE;
   }
   Enable interrupts;
}
```


Only on

Disabling Interrupts: No Busy-Waiting

uni-processors

- Keep a queue of threads waiting for the lock

```
class lock { int value := FREE }

lock.acquire() {
   Disable interrupts;
   if (value == BUSY) {
      Put on queue of threads
        waiting for lock;
```

```
lock.release() {
   Disable interrupts;
   if anyone on wait queue {
      Take a waiting thread off wait queue
      and put it at the front of ready queue;
   } else {
      value := FREE;
   }
   Enable interrupts;
}
```

Disabling Interrupts: No Busy-Waiting

Only on uni-processors

- Keep a queue of threads waiting for the lock

class lock { int value := FREE }

```
lock.acquire() {
  Disable interrupts;
  if (value == BUSY) {
    Enable interrupts?
    Put on queue of threads
      waiting for lock;
```

```
lock.release() {
  Disable interrupts;
  if anyone on wait queue {
    Take a waiting thread off wait queue
    and put it at the front of ready queue;
  } else {
    value := FREE;
  }
  Enable interrupts;
}
```


Disabling Interrupts: No Busy-Waiting

Only on uni-processors

- Keep a queue of threads waiting for the lock

class lock { int value := FREE }

```
lock.acquire() {
  Disable interrupts;
  if (value == BUSY) {
    Enable interrupts?
    Put on queue of threads
      waiting for lock;
```

NO!

Suppose T_2 concurrently calls lock.release

- T_2 could check the queue before T_1 is placed on the queue
- T_1 would be waiting on the queue even if lock is free

Disabling Interrupts: No Busy-Waiting

Only on uni-processors

- Keep a queue of threads waiting for the lock

class lock { int value := FREE }

```
lock.acquire() {
  Disable interrupts;
  if (value == BUSY) {
    Put on queue of threads
      waiting for lock;
    Enable interrupts?
    Go to sleep;
  } else {
    value := BUSY;
  }
  Enable interrupts;
}
```

```
lock.release() {
  Disable interrupts;
  if anyone on wait queue {
    Take a waiting thread off wait queue
    and put it at the front of ready queue;
  } else {
    value := FREE;
  }
  Enable interrupts;
}
```


Disabling Interrupts: No Busy-Waiting

Only on uni-processors

- Keep a queue of threads waiting for the lock

class lock { int value := FREE }

```
lock.acquire() {
  Disable interrupts;
  if (value == BUSY) {
    Put on queue of threads
      waiting for lock;
    Enable interrupts?
    Go to sleep;
```

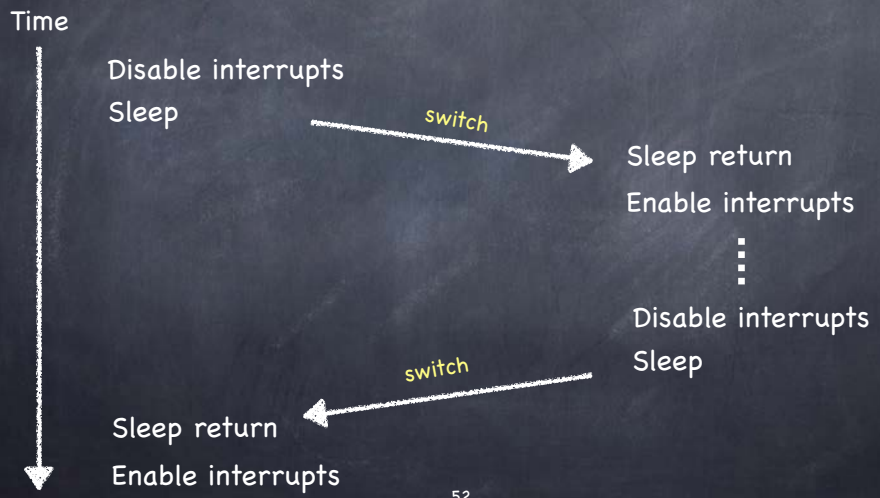
NO!

Suppose T_2 put T_1 on the ready queue

- When T_1 runs it will immediately go to sleep!!

When are interrupts re-enabled??

Passing the baton



Atomic Read/Modify/Write

- On a multiprocessor, disabling interrupts does not ensure atomicity
  - other CPUs could still enter the critical section
  - costly to disable interrupts on all CPUs
- Hardware provides special machine instructions
  - Test-and-Set (TAS)
    - TAS reads value of a memory location, writes back 1 in its place
  - Compare-and-Swap (CAS)
    - CAS compares contents of a memory location with a given value; if they are the same, set the memory location to a new given value
  - Load link/Store Conditional (LL/SC)
    - LL reads value of a memory location; SC writes a new value to that location only if it has not been updated since LL

Hey! What's wrong with this?

```
lock value := 0
lock.acquire()   {while (value = 1);}
lock.release()   value := 0;
```

These new hardware-supported instructions can change atomically the variable that implements the lock

TAS Locks

- Simple, but flawed

```
lock value := 0
lock.acquire()   { while (TAS(value) = 1); }
lock.release()   { value := 0; }
```

well, certainly not for time needed to complete an arbitrarily long CS!

TAS Locks: Try 2

- Going meta!
  - Use TAS (on a second variable guard) to build a new "secondary CS" where the variable value that stores status of the lock associated with the original ("primary") CS can be atomically read and set.
- Why bother?
  - Instead on spinning on TAS for the unknown length of the primary CS, we are now spinning at worst for the known, short length of the secondary CS!

TAS Locks: Try 2

```
lock.acquire() {
   while (TAS(guard)) ;
   if (value == BUSY) {
      Put on queue of threads
      waiting for lock;
      Set guard to 0;
      Go to sleep;
   } else {
      value := BUSY;
      guard := 0
   }
}
```

```
lock.release() {
   while (TAS(guard)) ;
   if anyone on wait queue {
      Take a waiting thread off lock's
      wait queue and put it at the
      front of ready queue;
   } else {
      value := FREE;
   }
   guard := 0
}
```