Atomicity

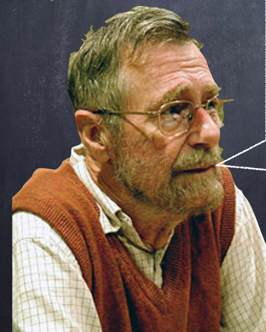
Atomicity and Virtualization

- Virtualizing a resource requires managing concurrent accesses
  - data structures must transition between consistent states
  - atomic actions transform state indivisibly
    - can be implemented by executing actions within a critical section
    - critical section becomes itself a multiplexed resource, though not a physical one

Critical section

- Two ways to implement atomicity
  - Rule out preemption
    - design system calls that prevent preemption
    - disable interrupts
  - Require threads to
    - execute an entry protocol before executing CS
      - lock.acquire()
    - execute an exit protocol after executing CS
      - lock.release()
    - entry/exit set who's next in time-multiplexing CS

Edsger's perspective



"Given in this paper is a solution to a problem which, to the knowledge of the author, has been an open question since at least 1962, irrespective of the solvability. [...]
Although the setting of the problem might seem somewhat academic at first, the author trusts that anyone familiar with the logical problems that arise in computer coupling will appreciate the significance of the fact that this problem indeed can be solved."

Solution to a problem in Concurrent Programming Control, 1965

Critical section

- Mutual Exclusion: At most one thread in CS (Safety)
  - critical sections of different threads do not overlap
- No deadlock: If some thread attempts to acquire the lock, some thread will eventually succeed (Liveness)
- No starvation: Every thread that attempts to acquire the lock eventually succeeds (Liveness)
  - bounded waiting (Safety)

Critical section

Thread T_0

```
while(!terminate) {
  lock.acquire()
  CS_0
  lock.release()
  NCS_0
}
```

Thread T_1

```
while(!terminate) {
  lock.acquire()
  CS_1
  lock.release()
  NCS_1
}
```

Critical section: Like-to lock

Thread T_0

```
while(!terminate) {
  lock.acquire()
  CS_0
  lock.release()
  NCS_0
}
```

Thread T_1

```
while(!terminate) {
  lock.acquire()
  CS_1
  lock.release()
  NCS_1
}
```

Critical section: Like-to lock

Thread T_0

```
while(!terminate) {
  in_0 := true
  while (in_1)   {}
  CS_0
  lock.release()
  NCS_0
}
```

Thread T_1

```
while(!terminate) {
  in_1 := true
  while (in_0)   {}
  CS_1
  lock.release()
  NCS_1
}
```

Critical section: Like-to lock

Thread T_0

```
while(!terminate) {
  in_0 := true
  while (in_1)   {}
  CS_0
  in_0 := false
  NCS_0
}
```

Thread T_1

```
while(!terminate) {
  in_1 := true
  while (in_0)   {}
  CS_1
  in_1 := false
  NCS_1
}
```

Critical section: Like-to lock

Thread T_0

```
while(!terminate) {
  in_0 := true
  while (in_1)   {}
  CS_0
  in_0 := false
  NCS_0
}
```

Thread T_1

```
while(!terminate) {
  in_1 := true
  while (in_0)   {}
  CS_1
  in_1 := false
  NCS_1
}
```

- Guarantees mutual exclusion
- If threads interleave, can deadlock
- But fine if threads execute sequentially!

Critical section: Selfless lock

Thread T_0

```
while(!terminate) {
  lock.acquire()
  CS_0
  lock.release()
  NCS_0
}
```

Thread T_1

```
while(!terminate) {
  lock.acquire()
  CS_1
  lock.release()
  NCS_1
}
```

Critical section: Selfless lock

Thread T_0

```
while(!terminate) {
  victim := 0
  while (victim = 0)   {}
  CS_0
  lock.release()
  NCS_0
}
```

Thread T_1

```
while(!terminate) {
  victim := 1
  while (victim = 1)   {}
  CS_1
  lock.release()
  NCS_1
}
```

Critical section: Selfless lock

Thread T_0

```
while(!terminate) {
  victim := 0
  while (victim = 0)   {}
  CS_0
    {}
  NCS_0
}
```

Thread T_1

```
while(!terminate) {
  victim := 1
  while (victim = 1)   {}
  CS_1
    {}
  NCS_1
}
```

Critical section: Selfless lock

Thread T_0

```
while(!terminate) {
  victim := 0
  while (victim = 0)   {}
  CS_0
    {}
  NCS_0
}
```

Thread T_1

```
while(!terminate) {
  victim := 1
  while (victim = 1)   {}
  CS_1
    {}
  NCS_1
}
```

- Guarantees mutual exclusion
- If threads execute sequentially, can deadlock
- But fine if threads interleave!

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 $(turn = i \Leftrightarrow victim = 1 - i)$
- Claim: If the following holds when T_i enters 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

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?

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

A first fix

- 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
$\dots$
$in_0 = false$
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
$\dots$
$in_1 = false$
NCS_1
}

but these invariants do not hold!

A dirty trick

- To establish the invariant, we add an auxiliary variable α that tracks the position of the PC

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
$\dots$
$in_0 = false$
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
$\dots$
$in_1 = false$
NCS_1
}

Is Peterson's lock safe?

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
$\dots$
$in_0 = false$
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
$\dots$
$in_1 = false$
NCS_1
}

If both in the critical section, then:

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

Live: Deadlock-free

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
$\{R_3\}$
$in_0 = false$
$\{R_1\}$
NCS_0
}

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 and T_1 at the while loop, before entering critical section

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

Live: Starvation free

```
while(!terminate) {
  {R1 : ¬in0 ∧ (turn = 1 ∨ turn = 0)}
  in0 = true
  {R2 : in0 ∧ (turn = 1 ∨ turn = 0)}
α0 turn = 1
  {R2}
  while(in1 ∧ turn ≠ 0);
  {R3 : in0 ∧ (¬in1 ∨ turn = 0 ∨ at(α1))}
  CS0
  {R3}
  in0 = false
  {R1}
  NCS0
}
```

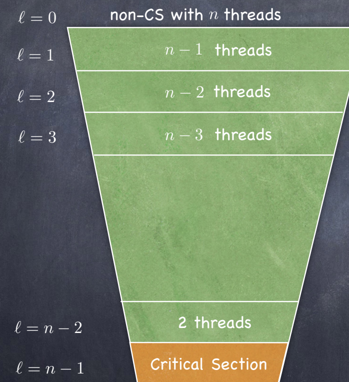
```
while(!terminate){
  {S1 : ¬in1 ∧ (turn = 1 ∨ turn = 0)}
  in1 := true
  {S2 : in1 ∧ (turn = 1 ∨ turn = 0)}
α1 turn := 0
  {S2}
  while(in0 ∧ turn ≠ 1);
  {S3 : in1 ∧ (¬in0 ∨ turn = 1 ∨ at(α0))}
  CS1
  {S3}
  in1 = false
  {S1}
  NCS1
}
```

Blocking Scenario 1: T_0 after lock.release(), T_1 stuck at while loop

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

Blocking Scenario 2: T_0 gets back to while loop, sets $turn = 1$, blocking itself and allowing T_1 to proceed

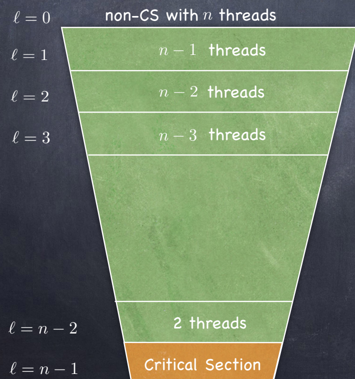
Filter lock: when 2 threads aren't enough



- n -level Peterson
  - level 0: non CS
  - level : CS
- Each level filters out one "victim" thread

Filter lock: when 2 threads aren't enough

```
class Filter implements lock {
  int[] level;
  int[] victim;
  ...
}
```




```
public Filter (int n) {
  level := new int[n];
  victim := new int[n];
  for (int i := 0; i < n; i++) {
    level[i] := 0;
  }
}

public void acquire() {
  int me := ThreadID.get();
  for (int i := 1; i < n; i++) {
    level[me] := i;
    victim[i] := me;
    while ((∃k ≠ me) (level[k] ≥ i ∧ victim[i] = me)) {};
  }
}

public void release() {
  int me := ThreadID.get();
  level[me] := 0;
}
```

Filter lock: when 2 threads aren't enough

```
public void acquire() {
  int me := ThreadID.get();
  for (int i := 1; i < n; i++) {
    level[me] := i;
    victim[i] := me;
    while ((∃k ≠ me) (level[k] ≥ i ∧ victim[i] = me)) {};
  }
}

public void release() {
  int me := ThreadID.get();
  level[me] := 0;
}
```

Lemma 1

For ℓ between 0 and $n-1$, there are at most $n-\ell$ threads at level ℓ

Corollary 1

The Filter lock algorithm satisfies mutual exclusion

Lemma 2

The Filter lock algorithm is starvation-free

Corollary 2

The Filter lock algorithm is deadlock-free

Fairness

- Threads have no guarantees of entering CS in the order they called $acquire()$
- Towards that goal, we split $acquire()$ in two sections:
  - doorway: an interval D consisting of a bounded number of steps
  - waiting: an interval W that may take an unbounded number of steps

FIFO lock:	if T_1 finishes doorway before T_2, then T_1 acquires CS before T_2

Lamport's Bakery algorithm

- Each thread that wants to enter CS, acquires a ticket
- New ticket number is higher than that any ticket previously acquired
- Threads enter CS in increasing ticket number
- Acquiring a ticket is not an atomic action...

The Bakery lock

```
class Bakery implements lock {
 boolean[] flag;
 Ticket[] ticket;
 public Bakery (int n) {
    flag := new boolean[n];
    ticket := new Ticket[n];
    for (int  i := 0;  i < n;  i++) {
        flag[i] := false;  ticket[i] := 0;
    }
 }
 public void acquire() {
   int  i := ThreadID.get();
   flag[i] := true;
   ticket[i] := max(ticket[0], ..., ticket[n − 1]) + 1;
   while ((∃k ≠ i) (flag[k] ∧ (ticket[k], k ≪ ticket[i], i)))  {};
 }
 public void release() {
   flag[ThreadID.get()] := false;
 }
}
```

The Bakery lock


```
class Bakery implements lock {
 boolean[] flag;
 Ticket[] ticket;
 public Bakery (int n) {
    flag := new boolean[n];
    ticket := new Ticket[n];
    for (int  i := 0;  i < n;  i++) {
        flag[i] := false;  ticket[i] := 0;
    }
 }
 public void acquire() {
   int  i := ThreadID.get();
   flag[i] := true;
   ticket[i] := max(ticket[0], ..., ticket[n − 1]) + 1;
   while ((∃k ≠ i) (flag[k] ∧ (ticket[k], k ≪ ticket[i], i)))  {};
 }
 public void release() {
   flag[ThreadID.get()] := false;
 }
}
```


Lemma 1

Bakery-lock is deadlock free

Proof Some waiting thread has the lowest ticket

Lemma 2

Bakery-lock satisfies mutual exclusion

Proof Suppose T_1 and T_2 in mutual exclusion, and that
$(ticket[T_1], T_1) \ll ticket[T_2], T_2)$ ●

- when T_2 entered CS, $flag[T_1]$ must have been false.
- T_1 computed its ticket after T_2
- contradiction with ●

Lemma 3

Bakery-lock is a FIFO lock

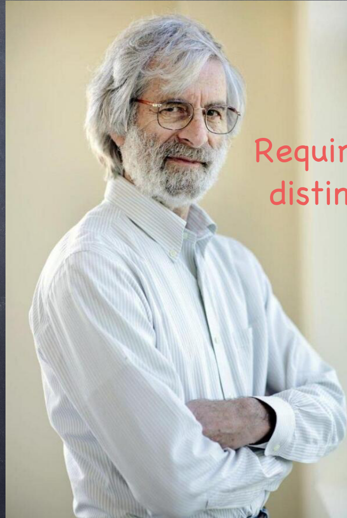
Proof If $D_{T_1} \to D_{T_2}$, $ticket[T_2] > ticket[T_1]$ so T_2 cannot enter CS while $flag[T_1]$.

Corollary

Bakery-lock is starvation-free

Why isn't everyone using the Bakery lock?

- Elegant
- Concise
- Fair



Requires to read N distinct variables

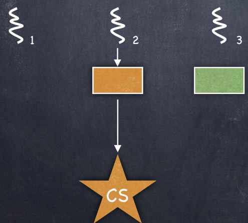
Surely we can do better...

Theorem Deadlock-free mutual exclusion among N threads requires at least N multi-reader/single-writer (MRSW) registers.

Surely we can do better...

Theorem Deadlock-free mutual exclusion among N threads requires at least N multi-reader/single-writer (MRSW) registers.

Proof. Consider two executions

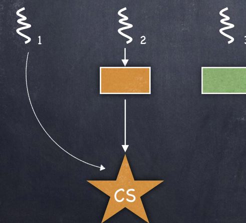


CS is empty. T_2 wants to enter CS. Since lock is deadlock-free, T_2 must be able to enter CS

Surely we can do better...

Theorem Deadlock-free mutual exclusion among N threads requires at least N multi-reader/single-writer (MRSW) registers.

Proof. Consider two executions



CS is empty. T_2 wants to enter CS. Since lock is deadlock-free, T_2 must be able to enter CS

T_1 is in CS. T_2 wants to enter CS. Since T_1 has not modified the state by writing to a register, this execution looks to T_2 indistinguishable from the first. Since lock is deadlock-free, T_2 must be able to enter CS, violating mutual exclusion

What if we use MRMW registers?

Theorem Deadlock-free mutual exclusion among N threads requires at least N multi-reader/multi-writer (MRMW) registers.

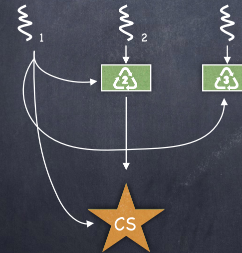
We'll prove it for N = 3

Definition A covering state for a lock object is one where at least a thread is about to write to each shared register, and the state of the shared registers is consistent with an empty CS.

A thread covers the register it is about to write

3 Threads, 2 MRMW Registers

- Suppose T_2 and T_3 cover the two registers



T_1 runs solo, possibly writing one or both registers

Because the lock is deadlock free, T_1 eventually enters the CS

Let T_2 and T_3 overwrite whatever T_1 wrote

the new state is indistinguishable from one without T_1 in the CS

Since the lock is deadlock free, it must be possible for one of T_2, T_3 to enter the CS

Must prove: Covering state reachable from any state with empty CS

Getting to a covering state

Suppose T_3 enters CS three times. Some register (say, B) must twice be the first to be written

Consider first time T_3 is about to write to B

if T_2 runs now, it must be able to enter CS

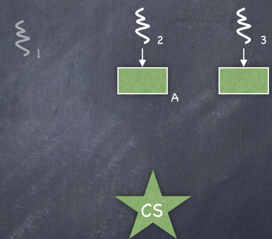
To do so, T_2 must definitely write to A — why?

Run T_2 until it is about to write to A – done?

NO! T_2 could have first written to B, informing T_1 of its intention to enter CS!

So, run T_3 to obliterate content of B, and let it enter and exiting CS, until T_3 is about to write B (first) for the second time

Voilà!



Take away

- Can't achieve n-thread, deadlock-free mutual exclusion with fewer than O(n) read/write registers
  - what a thread writes may be overwritten before any thread gets to see it
- We'll need something better than read/write
  - hardware support
  - but to understand which support, we need to take a deeper dive

Concurrent objects

Sequential Objects

Thanks to Maurice Herlihy
"The Art of Multiprocessor Programming"

- Each object has a state
  - Register: the value it stores
  - Queue: the sequence of objects it holds

Sequential Objects

Thanks to Maurice Herlihy
"The Art of Multiprocessor Programming"

- Each object has a state
  - Register: the value it stores
  - Queue: the sequence of objects it holds
- Each object has a set of methods
  - Register: Read/Write
  - Queue: Enq/Deq/Head

Sequential Specifications

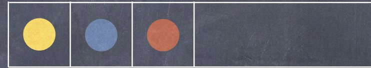
Thanks to Maurice Herlihy

- If (precondition)
  - the object is in such-and-such-state before method is called
- Then (postcondition)
  - the method will return a particular value
  - or throw a particular exception
- and (postcondition continued)
  - the object will be in some other state when method returns

Pre and Postconditions for Deq

Thanks to Maurice Herlihy

- Precondition
  - Queue is non-empty



- Postcondition
  - Returns first item in queue
- Postcondition
  - Removes first item in queue

Pre and Postconditions for Deq

Thanks to Maurice Herlihy

- Precondition
  - Queue is non-empty

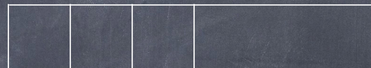
- Postcondition
  - Returns first item in queue

- Postcondition
  - Removes first item in queue

Pre and Postconditions for Deq

Thanks to Maurice Herlihy

- Precondition
  - Queue is empty



- Postcondition
  - Throws Empty exception
- Postcondition
  - Queue state unchanged

Sequential specifications are AWESOME

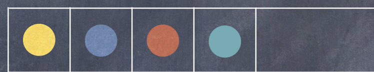
So is Maurice Herlihy

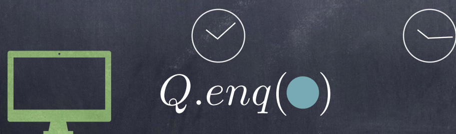
- Interactions among methods captured by side-effects on object state
  - State between method calls is meaningful
- Documentation size linear in the number of methods
  - Separation of concerns: each method described in isolation
- Can add new methods
  - Without changing description of old methods

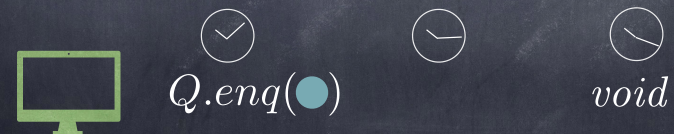
What about concurrent specifications?

- Methods?
- Documentation?
- Adding new methods?

Methods take time







Methods take time

- if you are Sequential
  - Really? Never noticed!
- ...but if you are Concurrent
  - Method call is not an event
  - Method call is an interval
    - concurrent method calls overlap!

What does it mean for correctness?

- Sequential
  - Object needs meaningful states only between method calls
- Concurrent
  - Because method calls overlap, object may never be between method calls

What does it mean for correctness?

- Sequential
  - Each method described in isolation
- Concurrent
  - Must consider all possible interactions between concurrent calls
    - What if two enq() overlap?
    - What if enq() and deq() overlap?

What DOES IT mean for correctness?

- Sequential
  - New methods do not affect existing methods
- Concurrent
  - Everything can potentially interact with everything else!

Registers

- Sequential specification
  - A read returns the result of the latest completed write

Registers

- Sequential specification
  - A read returns the result of the latest completed write
- What if reads and writes can be concurrent?

Registers

- Sequential specification
  - A read returns the result of the latest completed write
- What if reads and writes can be concurrent?
  - A read not concurrent with a write returns the result of the latest completed write

Registers

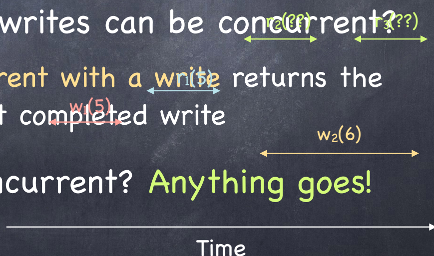
- Sequential specification
  - A read returns the result of the latest completed write
- What if reads and writes can be concurrent?
  - A read not concurrent with a write returns the result of the latest completed write
- And if they are concurrent?

SAFE Registers

- Sequential specification
  - A read returns the result of the latest completed write
- What if reads and writes can be concurrent?
  - A read not concurrent with a write returns the result of the latest completed write
- And if they are concurrent? Anything goes!

SAFE Registers

- Sequential specification
  - A read returns the result of the latest completed write
- What if reads and writes can be concurrent?
  - A read not concurrent with a write returns the result of the latest completed write
- And if they are concurrent? Anything goes!



Regular Registers

- Sequential specification
  - A read returns the result of the latest completed write
- What if reads and writes can be concurrent?
  - A read not concurrent with a write returns the result of the latest completed write
- And if they are concurrent?

  A read overlapping with a write returns either the old or the new value!