Thread Synchronization: Foundations

Two Theads, One Shared Variable

Two threads updating shared variable amount

- T_1 wants to decrement amount by \$10K
- T_2 wants to decrement amount by 50%

T_1

```
. . .
amount := amount - 10,000;
. . .
```

T_2

```
. . .
amount := amount * 0.5;
. . .
```

amount = 100,000

What happens when T_1 and T_2 execute concurrently?

Two Theads, One Shared Variable

Might execute like this:

T_2

```
. . .
r2 := load from amount
r2 := 0.5 * r2
store r2 to amount
. . .
```

T_1

```
. . .
r1 := load from amount
r1 := r1 - 10,000
store r1 to amount
. . .
```

amount = 40,000

Or viceversa: T_1 and then T_2 amount = 45,000

Two Theads, One Shared Variable

But might also execute like this:

T_2

```
. . .
r2 := load from amount

. . .

r2 := 0.5 * r2
store r2 to amount
. . .
```

T_1

```
. . .
r1 := load from amount
r1 := r1 - 10,000
store r1 to amount
. . .
```

amount = 50,000

One update is lost! Wrong – and very hard to debug

Race Conditions

Timing dependent behavior involving shared state

- Behavior of race condition depends on how threads are scheduled!
  - one program can generate exponentially many schedules or interleavings
  - bug if any of them generates an undesirable behavior

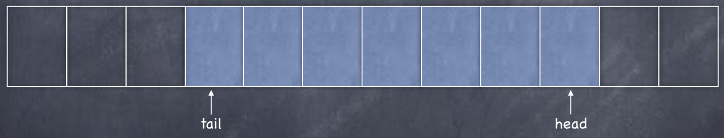
All possible interleavings should be safe!

Race Conditions: Hard to Debug

- Only some interleavings may produce a bug
- But bad interleavings may happen very rarely
  - program may run 100s of times without generating an unsafe interleaving
- Small changes to the program may hide bugs
  - timing dependent (Therac-25)
- Compiler and processor hardware can reorder instructions

Example: Races with Queues

- Two concurrent enqueue() operations?
- Two concurrent dequeue() operations?

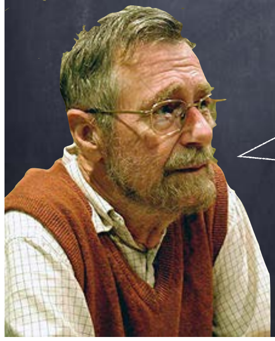


What could possibly go wrong?

There is the rub…

- 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

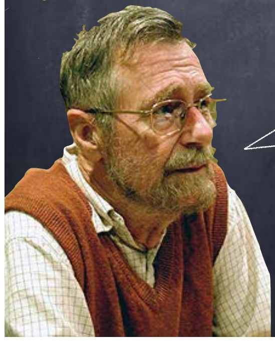
Edsger's perspective



"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

Edsger's Perspective



Testing can only prove the presence of bugs...
...not their absence!

Take a walk on the wild side...

Lou Reed, 1972

Properties

Property: a predicate that is evaluated over a run of the program (a trace)

"every message that is received was previously sent"

Not everything you may want to say about a program is a property:

"the program sends an average of 50 messages in a run"

Safety properties

- "Nothing bad happens"
  - No more than k processes are simultaneously in the critical section
  - Messages that are delivered are delivered in FIFO order
  - No patient is ever given the wrong medication
  - Windows never crashes
- A safety property is "prefix closed":
  - if it holds in a run, it holds in its every prefix

Liveness properties

- "Something good eventually happens"
  - A process that wishes to enter the critical section eventually does so
  - Some message is eventually delivered
  - Medications are eventually distributed to patients
  - Windows eventually boots
- Every run can be extended to satisfy a liveness property
  - if it does not hold in a prefix of a run, it does not mean it may not hold eventually

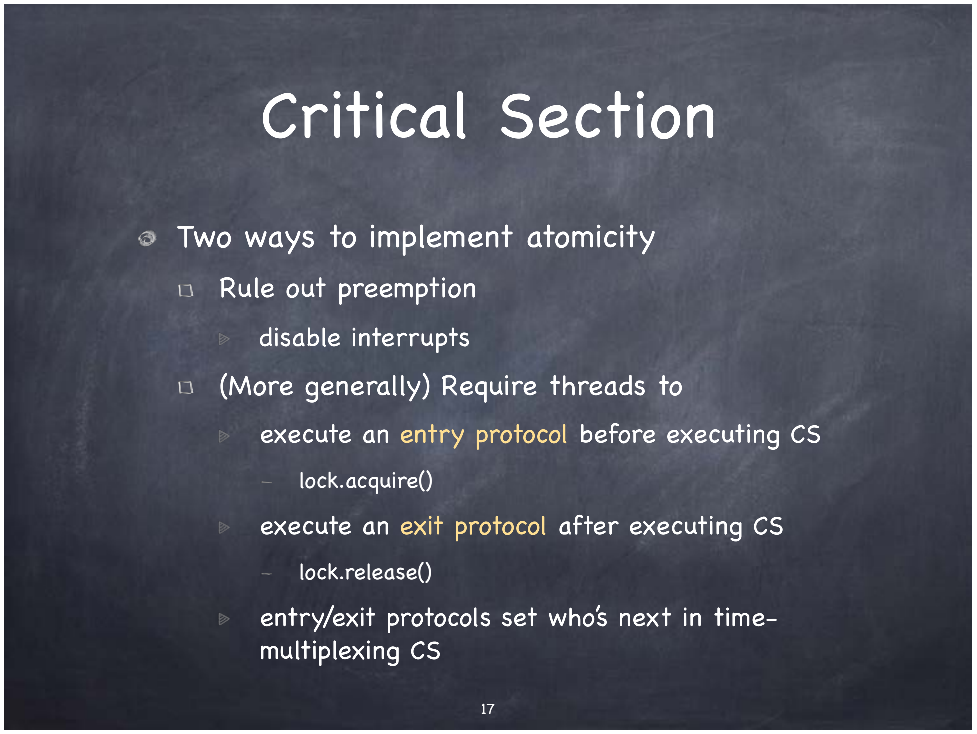
A really cool theorem

Every property is a combination of a safety property and a liveness property

(Alpern & Schneider)

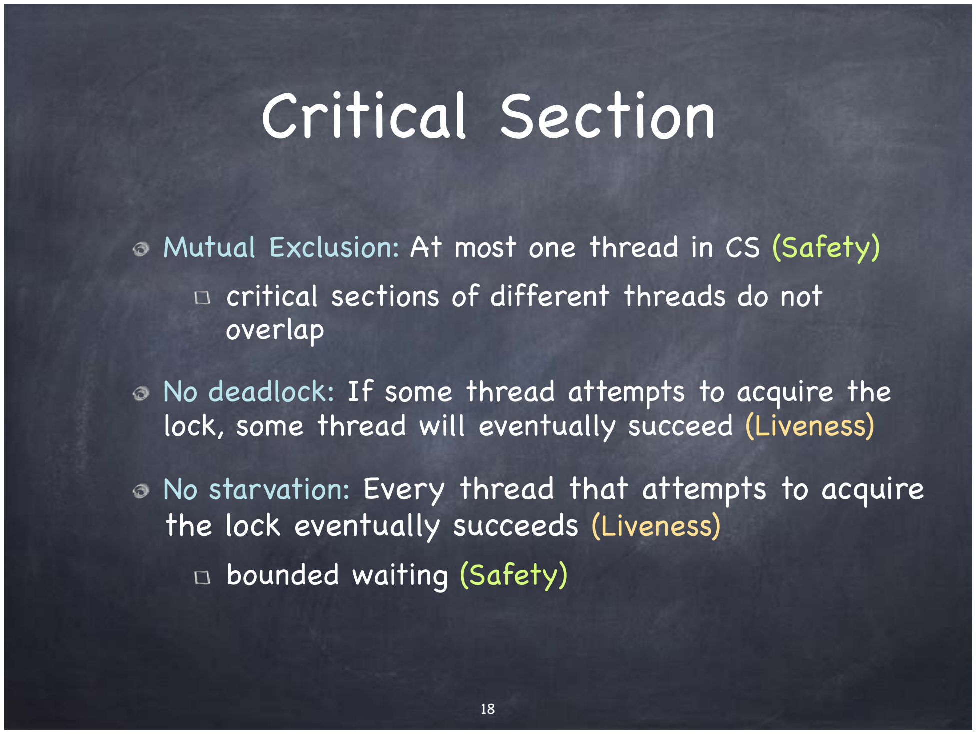
Critical Section

- A segment of code involved in reading and writing a shared data area
  - Used to protect data structures (e.g., queues, shared variables, lists, ...)
- Must be executed atomically
- Key assumptions:
  - Finite Progress Axiom: Processes execute at a finite, positive, but otherwise unknown, speed.
  - Processes can halt only outside of the critical section (by failing, or just terminating)
    - but: wait-free synchronization (Herlihy, 1991)



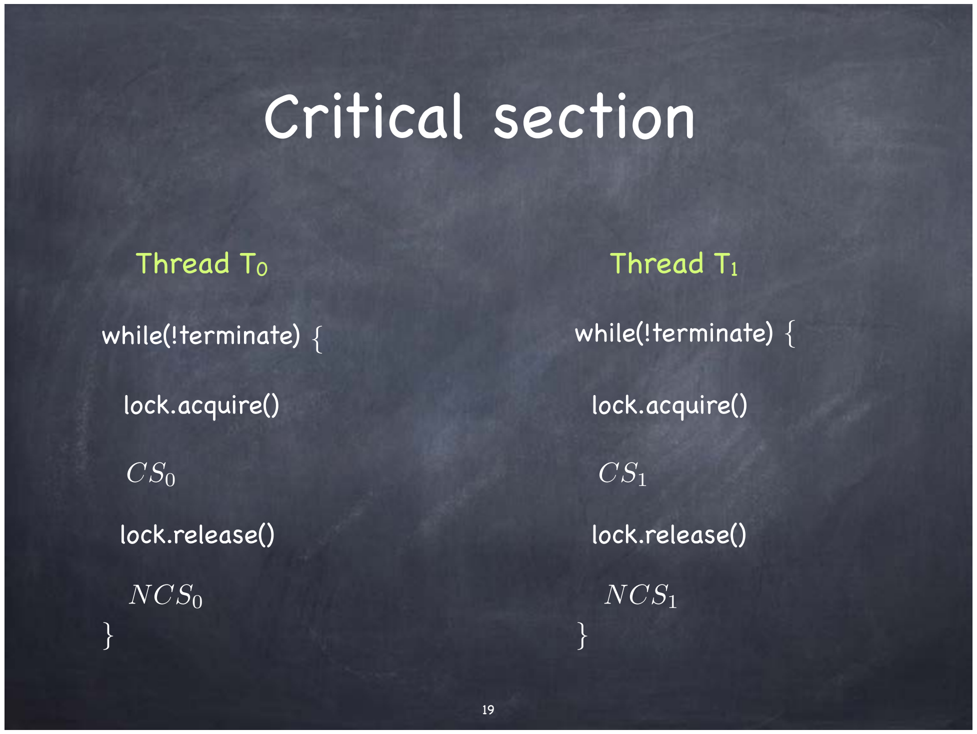
Critical Section

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



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)



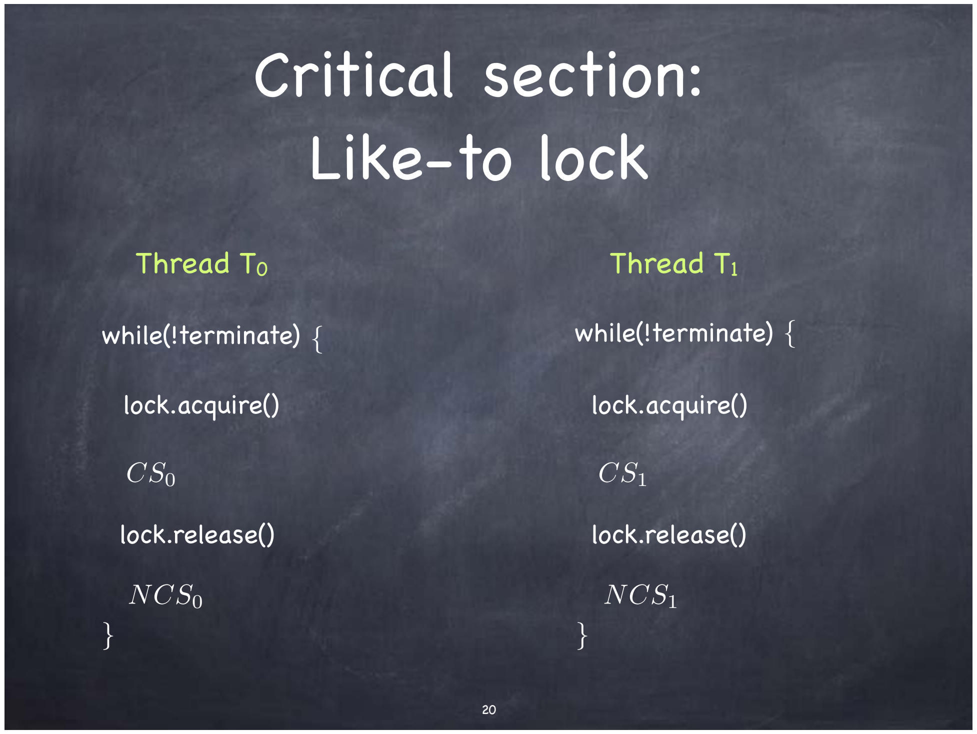
Critical section

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!