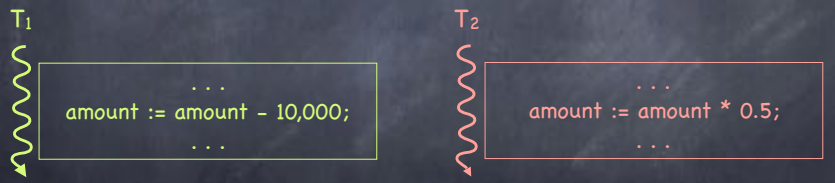
Thread Synchronization: Foundations

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%

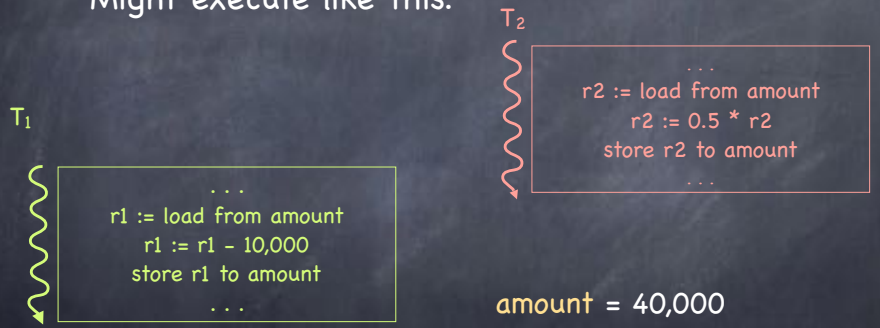


amount = 100,000

What happens when T_1 and T_2 execute concurrently?

Two Theads, One Shared Variable

Might execute like this:

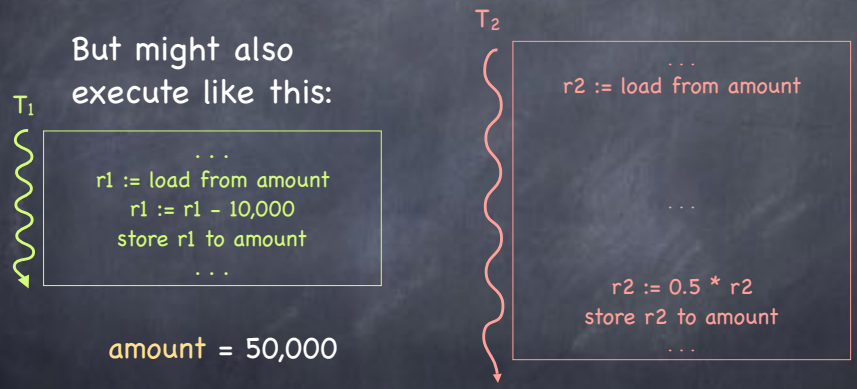


amount = 40,000

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

Two Theads, One Shared Variable

But might also execute like this:



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

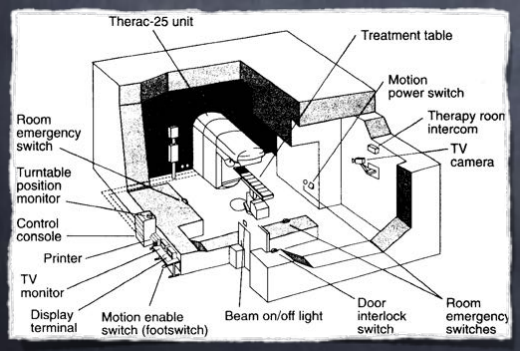
All possible interleavings should be safe!

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
- Compiler and processor hardware can reorder instructions

Therac-25 [1982]

Computer-controlled radiation therapy machine



- Safety critical system with software interlocks
  - they let state of element A determine allowed states for element B
    - Ex: elevator cannot move with doors open
- Beam controlled entirely through a custom OS

Therac-25

- Old system used a hardware interlock
  - Lever either in the "electron-beam" or "x-ray" position
- New system was computer controlled
- Much went wrong:
  - A synchronization failure triggered when competent nurses used back arrow to change the data on the screen "too quickly"
  - Engineers reused software from older models
    - it was buggy, but hardware interlocks masked the bugs
  - The system noted a problem and halted X-beam, displaying "MALFUNCTION" followed by obscure error code 54
    - technician resumed treatment

Therac-25 Outcome

- Patients received over 100x the recommended dose of radiation
  - Three patients died of radiation overdose
  - Many cancer patients received inadequate treatment
- People died because a programmer could not write correct code for a concurrent system
- 38 Year Later…. Now what?

Aye, there's the rub…

- OS virtualizes resources
- 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

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

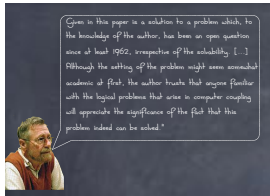
Liveness properties

- "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 data shared by N threads
  - Used to protect data structures (e.g., queues, shared variables, lists, ...)
- Must be executed atomically
- Key requirements:
  - Solution must be symmetrical for the N threads
  - Nothing can be assumed about the speed of the N threads, but that their speed inside the CS is not zero
  - A thread that stops outside CS must not impede access to CS for other threads
  - "Italians at a door syndrome" (mutual blocking) unacceptable

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

Thread T_0

while(!terminate) {

$entry_0$

CS_0

lock.release()

NCS_0

}

Thread T_1

while(!terminate) {

$entry_1$

CS_1

lock.release()

NCS_1

}

Critical section

Thread T_0

while(!terminate) {

$entry_0$

CS_0

$exit_0$

NCS_0

}

Thread T_1

while(!terminate) {

$entry_1$

CS_1

$exit_1$

NCS_1

}

Critical Section

- Mutual Exclusion: At most one thread in CS (Safety)
  - $in(CS_i) \wedge in(CS_j)$ must be false
- 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)
  - If $at(entry_i)$, then eventually $at(CS_i)$
  - When $in(NCS_i)$, thread i cannot block other threads from entering CS
- Assumption: if $in(CS_i)$, then eventually $after(CS_i)$

Critical Section: Like-to Lock (unless you do too)

Thread T_0

while(!terminate) {

$entry_0$

CS_0

$exit_0$

NCS_0

}

Thread T_1

while(!terminate) {

$entry_1$

CS_1

$exit_1$

NCS_1

}

Critical Section: Like-to Lock (unless you do too)

Thread T_0

while(!terminate) {

$in_0 := true$

await $\neg in_1$

CS_0

$exit_0$

NCS_0

}

Thread T_1

while(!terminate) {

$in_1 := true$

await $\neg in_0$

CS_1

$exit_1$

NCS_1

}

Critical Section: Like-to Lock (unless you do too)

Thread T_0

while(!terminate) {

$in_0 := true$

while (in_1) {}

CS_0

$exit_0$

NCS_0

}

Thread T_1

while(!terminate) {

$in_1 := true$

while (in_0) {}

CS_1

$exit_1$

NCS_1

}

Critical Section: Like-to Lock (unless you do too)

Thread T_0

while(!terminate) {
$in_0 := true$
await $\neg in_1$
CS_0
$exit_0$
NCS_0
}

Thread T_1

while(!terminate) {
$in_1 := true$
await $\neg in_0$
CS_1
$exit_1$
NCS_1
}

Critical Section: Like-to Lock (unless you do too)

Thread T_0

while(!terminate) {
$in_0 := true \ \{in_0\}$
await $\neg in_1 \ \{in_0 \wedge \neg in_1\}$
CS_0
$in_0 := false \ \{\neg in_0\}$
NCS_0
}

Thread T_1

while(!terminate) {
$in_1 := true \ \{in_1\}$
await $\neg in_0 \ \{in_1 \wedge \neg in_0\}$
CS_1
$in_1 := false \{\neg in_1\}$
NCS_1
}

Critical Section: Like-to Lock (unless you do too)

Thread T_0

while(!terminate) {
$in_0 := true \ \{in_0\}$
await $\neg in_1 \ \{in_0 \wedge \neg in_1\}$
CS_0
$in_0 := false \ \{\neg in_0\}$
NCS_0
}

Thread T_1

while(!terminate) {
$in_1 := true \ \{in_1\}$
await $\neg in_0 \ \{in_1 \wedge \neg in_0\}$
CS_1
$in_1 := false \{\neg in_1\}$
NCS_1
}

Mutual exclusion?

$in(CS_0) \Rightarrow in_0 \wedge \neg in_1$
$in(CS_1) \Rightarrow in_1 \wedge \neg in_0$

$in(CS_0) \wedge in(CS_1) =$
$in_0 \wedge \neg in_1 \wedge in_1 \wedge \neg in_0 = false$

✓

Critical Section: Like-to Lock (unless you do too)

Thread T_0

while(!terminate) {
$in_0 := true \ \{in_0\}$
await $\neg in_1 \ \{in_0 \wedge \neg in_1\}$
CS_0
$in_0 := false \ \{\neg in_0\}$
NCS_0
}

Thread T_1

while(!terminate) {
$in_1 := true \ \{in_1\}$
await $\neg in_0 \ \{in_1 \wedge \neg in_0\}$
CS_1
$in_1 := false \{\neg in_1\}$
NCS_1
}

Non Blocking?

Blocked $in(entry_0) \Rightarrow in_0 \wedge \neg(\neg in_1)$ (1)
Blocked $in(entry_1) \Rightarrow in_1 \wedge \neg(\neg in_0)$ (2)

(1) $\wedge$ (2) =
$(in_0 \wedge in_1) \wedge (in_1 \wedge in_0) =$
$(in_0 \wedge in_1) \neq false$

✗

Once More unto the Breach: Taking Turns

Thread T_0	Thread T_1
$in_0 := true$	$in_1 := true$
await $\neg in_1$	await $\neg in_0$

The above condition for entering CS_i is too strong: we weaken it by adding turns

Even if in_1, if it is T_0's turn, then T_0 is allowed to enter CS_0

Invariant I: $turn = 0 \vee turn = 1$

The new entry code then is

Thread T_0	Thread T_1
$in_0 := true$	$in_1 := true$
await $\neg in_1 \vee (turn = 0)$	await $\neg in_0 \vee (turn = 1)$

Critical Section: Taking Turns

Thread T_0

while(!terminate) {

$in_0 := true \ \{in_0 \wedge I\}$

await $\neg in_1 \vee (turn = 0)$

$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$

CS_0

$in_0 := false \ \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

$in_1 := true \ \{in_1 \wedge I\}$

await $\neg in_0 \vee (turn = 1)$

$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1

$in_1 := false \{\neg in_1 \wedge I\}$

NCS_1

}

Critical Section: Taking Turns

Thread T_0

while(!terminate) {

$in_0 := true \ \{in_0 \wedge I\}$

while $(in_1 \wedge turn \neq 0)$;

$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$

CS_0

$in_0 := false \ \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

$in_1 := true \ \{in_1 \wedge I\}$

while $(in_0 \wedge turn \neq 1)$;

$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1

$in_1 := false \{\neg in_1 \wedge I\}$

NCS_1

}

Critical Section: Taking Turns

Thread T_0

while(!terminate) {

$in_0 := true \ \{in_0 \wedge I\}$

while $(in_1 \wedge turn \neq 0)$;

$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$

CS_0

$in_0 := false \ \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

$in_1 := true \ \{in_1 \wedge I\}$

while $(in_0 \wedge turn \neq 1)$;

$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1

$in_1 := false \{\neg in_1 \wedge I\}$

NCS_1

}

Interference Freedom

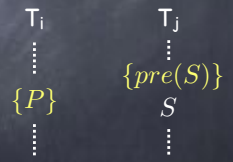
- By executing $in_1 := true$, T_1 can interfere on the truth of T_0's assertion! (and symmetrically for T_0)

$$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$$

- In general, interference freedom requires to establish

$$\{pre(S) \wedge P\} \;\; S \;\; \{P\}$$

for all S in one thread and P in the other



Establishing Interference Freedom

Thread T_0

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

while $(in_1 \wedge turn \neq 0)$;
$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$

CS_0
$in_0 := false \;\; \{\neg in_0 \wedge I\}$

NCS_0
}

Thread T_1

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

while $(in_0 \wedge turn \neq 1)$;
$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1
$in_1 := false \; \{\neg in_1 \wedge I\}$

NCS_1
}

Establishing Interference Freedom

Thread T_0

while(!terminate) {
$in_0 := true \;\; \{in_0 \wedge I\}$ ✓

while $(in_1 \wedge turn \neq 0)$;
$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$ ✘

CS_0
$in_0 := false \;\; \{\neg in_0 \wedge I\}$ ✓

NCS_0
}

Thread T_1

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

while $(in_0 \wedge turn \neq 1)$;
$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1
$in_1 := false \; \{\neg in_1 \wedge I\}$

NCS_1
}

Establishing Interference Freedom

Thread T_0

while(!terminate) {
$in_0 := true \;\; \{in_0 \wedge I\}$ ✓ ✓

while $(in_1 \wedge turn \neq 0)$;
$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$ ✘ ✓

CS_0
$in_0 := false \;\; \{\neg in_0 \wedge I\}$ ✓ ✓

NCS_0
}

Thread T_1

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

while $(in_0 \wedge turn \neq 1)$;
$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1
$in_1 := false \; \{\neg in_1 \wedge I\}$

NCS_1
}

Establishing Interference Freedom

Thread T_0

while(!terminate) {

$\langle in_0 := true \;\; \{in_0 \wedge I\}$
$\;\; turn = 1 \;\;\; \{in_0 \wedge I\} \rangle$

while $(in_1 \wedge turn \neq 0)$;
$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$

CS_0

$in_0 := false \;\; \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

$\langle in_1 := true \;\; \{in_1 \wedge I\}$
$\;\; turn = 0 \;\;\; \{in_1 \wedge I\} \rangle$

Operations execute atomically

while $(in_0 \wedge turn \neq 1)$;
$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1

$in_1 := false \{\neg in_1 \wedge I\}$

NCS_1

}

Taking stock

- We solved the critical section problem, as long as we know how to execute multiple operations atomically
  - in other words, we can solve the CS problem as long as we can solve the CS problem... sigh...
  - besides, no machine instruction allows for those operations to execute atomically...
- But what if we don't execute the entry code atomically? Where is the problem?

Establishing Interference Freedom

Thread T_0

while(!terminate) {

$\langle in_0 := true \;\; \{in_0 \wedge I\}$
$\;\; turn = 1 \;\;\; \{in_0 \wedge I\} \rangle$

while $(in_1 \wedge turn \neq 0)$;
$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$

CS_0

$in_0 := false \;\; \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

$\langle in_1 := true \;\; \{in_1 \wedge I\}$
$\;\; turn = 0 \;\;\; \{in_1 \wedge I\} \rangle$

Operations execute atomically

while $(in_0 \wedge turn \neq 1)$;
$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1

$in_1 := false \{\neg in_1 \wedge I\}$

NCS_1

}

Establishing Interference Freedom

Thread T_0

while(!terminate) {

$in_0 := true \;\; \{in_0 \wedge I\}$
$turn = 1 \;\;\; \{in_0 \wedge I\}$

while $(in_1 \wedge turn \neq 0)$;
$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$

CS_0

$in_0 := false \;\; \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

PC_{T_1} → $in_1 := true \;\; \{in_1 \wedge I\}$ No problem!
$turn = 0 \;\;\; \{in_1 \wedge I\}$

while $(in_0 \wedge turn \neq 1)$;
$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1

$in_1 := false \{\neg in_1 \wedge I\}$

NCS_1

}

Establishing Interference Freedom

Thread T_0

while(!terminate) {

$in_0 := true \quad \{in_0 \wedge I\}$

$turn = 1 \quad \{in_0 \wedge I\}$

while $(in_1 \wedge turn \neq 0)$;

$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$

CS_0

$in_0 := false \quad \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

$in_1 := true \quad \{in_1 \wedge I\}$

$turn = 0 \quad \{in_1 \wedge I\}$

PC_{T_1} → while $(in_0 \wedge turn \neq 1)$; No problem!

$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1

$in_1 := false \{\neg in_1 \wedge I\}$

NCS_1

}

Establishing Interference Freedom

Thread T_0

while(!terminate) {

$in_0 := true \quad \{in_0 \wedge I\}$

$turn = 1 \quad \{in_0 \wedge I\}$

while $(in_1 \wedge turn \neq 0)$;

$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$

CS_0

$in_0 := false \quad \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

$in_1 := true \quad \{in_1 \wedge I\}$

PC_{T_1} → $turn = 0 \quad \{in_1 \wedge I\}$ Problem!

We weaken the assertion

while $(in_0 \wedge turn \neq 1)$;

$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1

$in_1 := false \{\neg in_1 \wedge I\}$

NCS_1

}

Establishing Interference Freedom

Thread T_0

while(!terminate) {

$in_0 := true \quad \{in_0 \wedge I\}$

$turn = 1 \quad \{in_0 \wedge I\}$

while $(in_1 \wedge turn \neq 0)$;

$\{in_0 \wedge (\neg in_1 \vee turn = 0 \vee at(turn = 0)) \wedge I\}$

CS_0

$in_0 := false \quad \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

$in_1 := true \quad \{in_1 \wedge I\}$

PC_{T_1} → $turn = 0 \quad \{in_1 \wedge I\}$ No problem!

We weaken the assertion

while $(in_0 \wedge turn \neq 1)$;

$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1

$in_1 := false \{\neg in_1 \wedge I\}$

NCS_1

}

Establishing Interference Freedom

Thread T_0

while(!terminate) {

$in_0 := true \quad \{in_0 \wedge I\}$

$turn = 1 \quad \{in_0 \wedge I\}$

while $(in_1 \wedge turn \neq 0)$;

$\{in_0 \wedge (\neg in_1 \vee turn = 0 \vee at(turn = 0)) \wedge I\}$

CS_0

$in_0 := false \quad \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

$in_1 := true \quad \{in_1 \wedge I\}$

PC_{T_1} → $turn = 0 \quad \{in_1 \wedge I\}$ No problem!

We weaken the assertion

while $(in_0 \wedge turn \neq 1)$;

$\{in_1 \wedge (\neg in_0 \vee turn = 1 \vee at(turn = 1)) \wedge I\}$

CS_1

$in_1 := false \{\neg in_1 \wedge I\}$

NCS_1

}

Peterson's Algorithm

Thread T_0

while(!terminate) {

$in_0 := true \ \{in_0 \wedge I\}$

$turn = 1 \quad \{in_0 \wedge I\}$

while $(in_1 \wedge turn \neq 0);$

$\{in_0 \wedge (\neg in_1 \vee turn = 0 \vee at(turn = 0)) \wedge I\}$

CS_0

$in_0 := false \ \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

$in_1 := true \ \{in_1 \wedge I\}$

$turn = 0 \quad \{in_1 \wedge I\}$

while $(in_0 \wedge turn \neq 1);$

$\{in_1 \wedge (\neg in_0 \vee turn = 1 \vee at(turn = 1)) \wedge I\}$

CS_1

$in_1 := false \ \{\neg in_1 \wedge I\}$

NCS_1

}

Peterson's Algorithm: Safety

Thread T_0

while(!terminate) {

$in_0 := true \ \{in_0 \wedge I\}$

$turn = 1 \quad \{in_0 \wedge I\}$

while $(in_1 \wedge turn \neq 0);$

$\{in_0 \wedge (\neg in_1 \vee turn = 0 \vee at(turn = 0)) \wedge I\}$

CS_0

$in_0 := false \ \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

$in_1 := true \ \{in_1 \wedge I\}$

$turn = 0 \quad \{in_1 \wedge I\}$

while $(in_0 \wedge turn \neq 1);$

$\{in_1 \wedge (\neg in_0 \vee turn = 1 \vee at(turn = 1)) \wedge I\}$

CS_1

$in_1 := false \ \{\neg in_1 \wedge I\}$

NCS_1

}

Mutual exclusion?

$$\begin{array}{ll} in(CS_0) \Rightarrow & \{in_0 \wedge (\neg in_1 \vee turn = 0 \vee at(turn = 0)) \wedge I\} \wedge \\ \wedge & \neg at(turn = 1) \wedge \\ in(CS_1) \Rightarrow & \{in_1 \wedge (\neg in_0 \vee turn = 1 \vee at(turn = 1)) \wedge I\} \wedge \\ & \neg at(turn = 0) = \\ & in_1 \wedge in_2 \wedge turn = 0 \wedge turn = 1 = false \end{array}$$ ✓

Peterson: Non-blocking

while(!terminate) {

$\{R_1 : \neg in_0 \wedge (turn = 1 \vee turn = 0)\}$

$in_0 = true$

$\{R_2 : in_0 \wedge (turn = 1 \vee turn = 0)\}$

$\alpha_0 \ turn = 1$

$\{R_2\}$

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

}

T0's PC →

while(!terminate) {

$\{S_1 : \neg in_1 \wedge (turn = 1 \vee turn = 0)\}$

$in_1 := true$

$\{S_2 : in_1 \wedge (turn = 1 \vee turn = 0)\}$

$\alpha_1 \ turn := 0$

$\{S_2\}$

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

}

T1's PC →

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) {

$\{R_1 : \neg in_0 \wedge (turn = 1 \vee turn = 0)\}$

$in_0 = true$

$\{R_2 : in_0 \wedge (turn = 1 \vee turn = 0)\}$

$\alpha_0 \ turn = 1$

$\{R_2\}$

T0's PC → 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) {

$\{S_1 : \neg in_1 \wedge (turn = 1 \vee turn = 0)\}$

$in_1 := true$

$\{S_2 : in_1 \wedge (turn = 1 \vee turn = 0)\}$

$\alpha_1 \ turn := 0$

$\{S_2\}$

T1's PC → 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

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