IN5170/IN9170 – Models of concurrency

Fall 2021

Compiled: October 7, 2021

Abstract

This is the "handout" version of the slides for the lecture (i.e., it's a rendering of the content of the slides in a way that does not waste so much paper when printing out). The material is found in [**?**]. Being a handout-version of the slides, some figures and graph overlays may not be rendered in full detail, I remove most of the overlays, especially the long ones, because they don't make sense much on a handout/paper. Scroll through the real slides instead, if one needs the overlays.

This handout version also contains more remarks and footnotes, which would clutter the slides, and which typically contains remarks and elaborations, which may be given orally in the lecture.

Contents

6. 10. 2021

1 Program Analysis: Part II Concurrency (week 6)

Today

Analysis Part I: Reasoning about sequential programs.

Analysis Part II: How to reason about concurrent systems.

- Reptition of Rules
- await statements
- shared variables and interference
  - cobegin
  - interference freedom
- examples
  - Producer/Consumer (await language)
- monitor invariants
- signal/wait
- Examples: reader/writer
- Examples: semphores

1.1 Repetition

Program Logic (PL)

- PL lets us *express* and *prove* properties about programs
- *Formulas* are on the form

  "triple" (or "Hoare triple")

  $$\{ P \}\ S\ \{ Q \}$$

  - S: program statement(s)
  - P and Q: assertions over program states
  - P: Pre-condition
  - Q: Post-condition

If we can use PL to prove some property of a program, then this property will hold for all executions of the program

PL rules from part I: structural inference rules

$$\frac{P' \Rightarrow P \qquad \{ P \}\ S\ \{ Q \} \qquad Q \Rightarrow Q'}{\{ P' \}\ S\ \{ Q' \}}\ \textsc{Consequence}$$

$$\frac{\{ P \}\ S\ \{ Q \} \qquad \{ P' \}\ S\ \{ Q' \}}{\{ P \wedge P' \}\ S\ \{ Q \wedge Q' \}}\ \textsc{Conjunction}$$

$$\frac{\{ P \}\ S_1\ \{ R \} \qquad \{ R \}\ S_2\ \{ Q \}}{\{ P \}\ S_1; S_2\ \{ Q \}}\ \textsc{Seq}$$

- Red: logical proof obligations

PL rules from part I: if/while/assignment

$$\frac{\{\, P \wedge B \,\}\ S\ \{\, Q \,\} \qquad P \wedge \neg B \Rightarrow Q}{\{\, P \,\}\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{fi}\ \{\, Q \,\}}\ \textsc{Cond1}$$

$$\frac{\{\, P \wedge B \,\}\ S\ \{\, Q \,\} \qquad \{\, P \wedge \neg B \,\}\ S'\ \{\, Q \,\}}{\{\, P \,\}\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ S'\ \{\, Q \,\}}\ \textsc{Cond2}$$

$$\frac{\{\, I \wedge B \,\}\ S\ \{\, I \,\}}{\{\, I \,\}\ \textbf{while}\ B\ \textbf{do}\ S\ \{\, I \wedge \neg B \,\}}\ \textsc{While}$$

$$\{\, P[e/x] \,\}\ x := e\ \{\, P \,\} \quad \textsc{Assign}$$

Restrictions

- no side-effects (in expressions)
- no remote access by pointers
- no pointer arithmetic
- so far, we did not consider arrays

And so far no concurrency!

1.2 Usage

How to actually *use* the while rule?

- Cannot control the execution in the same manner as for **if**-statements
  - Cannot tell *from the code* how many times the loop body will be executed (not a "syntax-directed" rule)

    $$\{\, y \geq 0 \,\}\ \textbf{while}\ (y > 0)\ y := y - 1$$

  - Cannot speak about the state after the first, second, third ... iteration
- Solution: Find an assertion I that is maintained by the loop body
  - *Loop invariant:* express a property preserved by the loop
- Often hard to find suitable loop invariants
  - The course is *not* an exercise in finding complicated invariants
  - "suitable:
    1. must be preserved by the body, i.e., it must be actually an invariant
    2. must be strong enough to imply the desired post-condition
    3. Note: both "true" and "false" are loop invariants for partial correctness! Both typically fail to be suitable (i.e. they are basically useless invariants).

While rule

$$\frac{\{\ I \wedge B\ \}\ S\ \{\ I\ \}}{\{\ I\ \}\ \textbf{while}\ B\ \textbf{do}\ S\ \{\ I \wedge \neg B\ \}}\ \textsc{While}$$

Can use this rule to reason about the general situation:

$$\{\ P\ \}\ \textbf{while}\ B\ \textbf{do}\ S\ \{\ Q\ \}$$

where

- P need not be the loop invariant
- Q need not match $(I \wedge \neg B)$ syntactically

Combine While-rule with Consequence-rule to prove:

- **Entry:** $P \Rightarrow I$
- **Loop:** $\{\ I \wedge B\ \}\ S\ \{\ I\ \}$
- **Exit:** $I \wedge \neg B \Rightarrow Q$

While rule: example

$$\{\ 0 \leq n\ \}\ k := 0;\ \{\ k \leq n\ \}\ \textbf{while}\ \ (k < n)\ k := k + 1;\ \{\ k = n\ \}$$

Composition rule splits a proof in two: assignment and loop. Let $k \leq n$ be the loop invariant

- **Entry:** $k \leq n$ follows from itself
- **Loop:**

$$\frac{k < n \Rightarrow k + 1 \leq n}{\{\ k \leq n \wedge k < n\ \}\ k := k + 1\ \{\ k \leq n\ \}}$$

- **Exit:** $(k \leq n \wedge \neg(k < n)) \Rightarrow k = n$

Await statement

Rule for await

$$\frac{\{\ P \wedge B\ \}\ S\ \{\ Q\ \}}{\{\ P\ \}\ \langle \textbf{await}\ (B)\ S \rangle\ \{\ Q\ \}}\ \textsc{Await}$$

Remember: we are reasoning about safety properties/partial correctness

- termination is assumed/ignored
- the rule does not ensure waiting nor progress

1.3 Concurrent systems and interference

Concurrent execution and interference

Assume two statements S_1 and S_2 such that:

$$\{\ P_1\ \}\ \langle S_1 \rangle\ \{\ Q_1\ \} \quad \text{and} \quad \{\ P_2\ \}\ \langle S_2 \rangle\ \{\ Q_2\ \}$$

To simplify, we let S_1 and S_2 be atomic regions.

First attempt for a **co** . . . **oc** rule in PL:

$$\frac{\{\ P_1\ \}\ \langle S_1 \rangle\ \{\ Q_1\ \} \qquad \{\ P_2\ \}\ \langle S_2 \rangle\ \{\ Q_2\ \}}{\{\ P_1 \wedge P_2\ \}\ \textbf{co}\ \langle S_1 \rangle \parallel \langle S_2 \rangle\ \textbf{oc}\ \{\ Q_1 \wedge Q_2\ \}}\ \textsc{Par}$$

Example 1 (Problem with this rule).

$$\frac{\{\ x = 0\ \}\ \langle x := x + 1\rangle\ \{\ x = 1\ \} \qquad \{\ x = 0\ \}\ \langle x := x + 2\rangle\ \{\ x = 2\ \}}{\{\ x = 0\ \}\ \mathbf{co}\langle x := x + 1\rangle \parallel \langle x = x + 2\rangle\ \mathbf{oc}\ \{\ x = 1 \wedge x = 2\ \}}$$

but this conclusion is not true: the postcondition should be $x = 3$!

1>This first attempt may seem plausible. One has two programs, both with its "own" precondition. Therefore, if they run in parallel, they start in a common state, obviously. That may be characterized by the conjunction. Alternatively, one may use the *same* precondition. There is not much difference between those two ways of thinking (due to strengthening of preconditions). Indeed, the precondition in this line of reasoning is not problematic. Note however, that conceptually we are thinking in a *forward* way, we are not currently reason like "assume you are in a given post-state, ...". But the forward reasoning fits better to the following illustrating example. 2>Different ways to analyze what's exactly wrong. But the important observation is: that it's plain wrong. Remember Soundness. The break of soundness is illustrated by the following example. Linear logic, resources: "I win 100 dollar ∧ I win 100 dollar". The rule, if it were true, wo be nice: compositionality. For independent variables (i.e., local ones) it would be true. So, the reason, why concurrency is hard/compositional reasoning does not work, are shared variables. The absence of such problem will later be called interference free. It will not be defined for processes alone, but for specifications insofar: an annotated program is interference free if the *conditions* of parallel processes are not disturbed. It's like an invariant.

Interference problem

$$S_1 \quad \{\ x = 0\ \}\ \langle x := x + 1\rangle\ \{\ x = 1\ \}$$

$$S_2 \quad \{\ x = 0\ \}\ \langle x := x + 2\rangle\ \{\ x = 2\ \}$$

- execution of S_2 interferes with pre- and postconditions of S_1
  - The assertion $x = 0$ need not hold when S_1 starts execution
- execution of S_1 interferes with pre- and postconditions of S_2
  - The assertion $x = 0$ need not hold when S_2 starts execution

Solution: *weaken* the assertions to account for the other process:

$$S_1 \quad \{\ x = 0 \vee x = 2\ \}\ \langle x := x + 1\rangle\ \{\ x = 1 \vee x = 3\ \}$$

$$S_2 \quad \{\ x = 0 \vee x = 1\ \}\ \langle x := x + 2\rangle\ \{\ x = 2 \vee x = 3\ \}$$

Interference problem

Apply the previous "parallel-composition-is-conjunction"rule again:

$$\frac{\begin{array}{c}\{\ x = 0 \vee x = 2\ \}\ \langle x := x + 1\rangle\ \{\ x = 1 \vee x = 3\ \} \\ \{\ x = 0 \vee x = 1\ \}\ \langle x := x + 2\rangle\ \{\ x = 2 \vee x = 3\ \}\end{array}}{\{\ PRE\ \}\ \mathbf{co}\langle x := x + 1\rangle \parallel \langle x := x + 2\rangle\ \mathbf{oc}\ \{\ POST\ \}}$$

where:

$$\begin{array}{rl} PRE & : (x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1) \\ POST & : (x = 1 \vee x = 3) \wedge (x = 2 \vee x = 3)\end{array}$$

which gives:

$$\{\ x = 0\ \}\ \mathbf{co}\ \ \parallel x := x + 1 \parallel \langle x := x + 2\rangle\ \ \mathbf{oc}\ \{\ x = 3\ \}$$

Concurrent execution

Assume $\{\ P_i\ \}\ S_i\ \{\ Q_i\ \}$ **for all** $S_1, \ldots, S_n$

$$\frac{\{\ P_i\ \}\ S_i\ \{\ Q_i\ \} \quad \text{are } \textit{interference free}}{\{\ P_1 \wedge \ldots \wedge P_n\ \}\ \mathbf{co}\ S_1 \parallel \ldots \parallel S_n\ \mathbf{oc}\ \{\ Q_1 \wedge \ldots \wedge Q_n\ \}}\ \textsc{Cooc}$$

Interference freedom

A process interferes with (the specification of) another process, if its execution changes the assertions[1] of

[1] Only "critical assertions" considered, i.e. outside mutex-protected sections

the other process.

- assertions inside awaits: not endangered
- *critical assertions* or critical conditions: assertions outside await statement bodies.

Interference freedom

Interference freedom

- S: statement in some process, with pre-condition $pre(S)$
- C: critical assertion in another process
- S does not interfere with C, if

$$\vdash \{\ C \wedge pre(S)\ \}\ S\ \{\ C\ \}$$

is derivable in PL (= theorem).

"C is *invariant* under the execution of the other process"

$$\frac{\{\ P_1\ \}\ S_1\ \{\ Q_1\ \} \qquad \{\ P_2\ \}\ S_2\ \{\ Q_2\ \}}{\{\ P_1 \wedge P_2\ \}\ \mathbf{co}\ S_1 \parallel S_2\ \mathbf{oc}\ \{\ Q_1 \wedge Q_2\ \}}$$

Four interference freedom requirements (for 2 statements in co-oc):

$$\begin{array}{ll} \{P_2 \wedge P_1\}\ S_1\ \{P_2\} & \{P_1 \wedge P_2\}\ S_2\ \{P_1\} \\ \{Q_2 \wedge P_1\}\ S_1\ \{Q_2\} & \{Q_1 \wedge P_2\}\ S_2\ \{Q_1\} \end{array}$$

"Avoiding" interference: Weakening assertions

$$\begin{array}{ll} S_1: & \{\ x = 0\ \}\ <x := x + 1;>\ \{\ x = 1\ \} \\ S_2: & \{\ x = 0\ \}\ <x := x + 2;>\ \{\ x = 2\ \} \end{array}$$

Here we have interference, for instance the precondition of S_1 is not maintained by execution of S_2:

$$\{\ (x = 0) \wedge (x = 0)\ \}\ x := x + 2\ \{\ x = 0\ \}$$

is not true

However, after weakening:

$$\begin{array}{ll} S_1: & \{\ x = 0 \vee x = 2\ \}\ \langle x := x + 1\rangle\ \{\ x = 1 \vee x = 3\ \} \\ S_2: & \{\ x = 0 \vee x = 1\ \}\ \langle x := x + 2\rangle\ \{\ x = 2 \vee x = 3\ \} \end{array}$$

$$\{\ (x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1)\ \}\ x := x + 2\ \{\ x = 0 \vee x = 2\ \}$$

(Correspondingly for the other three critical conditions)

Avoiding interference: Disjoint variables

- V *set:* global variables referred to (i.e. read or written) by a process
- W *set:* global variables written to by a process
- *Reference set:* global variables in critical (Hoare) assertions/conditions of one process

S_1 and S_2: in 2 different processes. No interference, if:

- W set of S_1 is disjoint from reference set of S_2
- W set of S_2 is disjoint from reference set of S_1

Alas: variables in a critical condition of one process will often be among the written variables of another

Avoiding interference: Global invariants

Global inductive invariants

- An assertion that only refers to global (shared) variables
- Holds initially.
- Preserved by all assignments/transitions ("inductive")

"Separation of concerns: We avoid interference if critical conditions are on the form $\{ I \wedge L \}$ where:

- I is a global invariant
- L only refers to local variables of the considered process

Avoiding interference: Synchronization

- Hide critical conditions
- MUTEX to critical sections

$$\textbf{co} \ldots; S; \ldots \parallel \ldots; S_1; \{ C \} S_2; \ldots \textbf{oc}$$

S might interfere with C Hide the critical condition by a critical region:

$$\textbf{co} \ldots; S; \ldots \parallel \ldots; \langle S_1; \{ C \} S_2 \rangle; \ldots \textbf{oc}$$

Example: Producer/ consumer synchronization

Let process `Producer` deliver data to a `Consumer` process

$PC : c \leq p \leq c + 1 \wedge (p = c + 1 \Rightarrow buf = a[p - 1])$

PC a *global inductive invariant* of the producer/consumer?

```
            int buf, p := 0; c := 0;

process Producer {                  process Consumer {
  int a[N];...                      int b[N];...
  while (p < N) {                   while (c < N) {
    < await (p = c) ; >              < await (p > c) ; >
    buf := a[p];                      b[c] := buf;
      p := p+1;                          c := c+1;
  }                                   }
}                                   }
```

Example: Producer

Loop invariant of Producer: $I_P : PC \wedge p \leq n$

```
process Producer {
  int a[n];
  { I_P }                    //  entering loop
  while (p < n) {            { I_P ∧ p < n }
    < await (p = c); >       { I_P ∧ p < n ∧ p = c }
                             { I_P[p + 1/p][a[p]/buf] }
    buf := a[p];             { I_P[p + 1/p] }
    p := p + 1;              {I_P}
  } {I_P ∧ ¬(p < n)}         //  exit loop
     ⇔ {PC ∧ p = n}
}
```

Proof obligation: $I_P \wedge p < n \wedge p = c \Rightarrow I_P[p + 1/p][a[p]/buf]$

Example: Consumer

Loop invariant of Consumer: $I_C : PC \wedge c \leq n \wedge b[0:c-1] = a[0:c-1]$

```
process Consumer {
  int b[n];
  {I_C}                      // entering loop
  while (c < n) {            {I_C ∧ c < n}
    < await (p > c) ; >      {I_C ∧ c < n ∧ p > c}
                             {I_C[c+1/c][buf/b[c]]}
    b[c] := buf;             {I_C}[c+1/c]
    c := c + 1;              {I_C}
  } {I_C ∧ ¬(c < n)}         // exit loop
     ⇔ {PC ∧ c = n ∧ b[0:c-1] = a[0:c-1]}
}
```

Proof Obligation: $I_C \wedge c < n \wedge p > c \;\Rightarrow\; I_C[c+1/c][buf/b[c]]$

Example: Producer/Consumer

The final state of the program satisfies:

$$PC \wedge p = n \wedge c = n \wedge b[0:c-1] = a[0:c-1]$$

which ensures that all elements in `a` are received and occur in the same order in `b`

Interference freedom is ensured by the global/local discipline, except for the assertions right after the await tests (which are talking also about shared variables).

The cointerference freedom requirements for these conditions give:
Combining the two `await` assertions, we get:

$$I_P \wedge p < n \wedge p = c \wedge I_C \wedge c < n \wedge p > c$$

which gives *false*! *At any time, only one process can be after the await statement!*

Thus the cointerference freedom requirements are satisfied.

1.4 Monitor invariants

Monitor invariant

```
monitor name {
  monitor variables
  initialization
  procedures
}
```

- A monitor invariant (I): describe the monitor's inner state
- Express relationship between monitor variables
- Maintained by execution of procedures:
  - Must hold after initialization
  - Must hold when a procedure terminates
  - Must hold when we suspend execution due to a call to **wait**
  - Can assume that the invariant holds *after* `wait` and when a procedure starts
- Should be as *strong* as possible!

Axioms for signal and continue (1)

Assume that the monitor invariant I and predicate P *does not* mention `cv`. Then we can set up the following axioms:

$\{ I \}\ wait(cv)\ \{ I \}$
$\{ P \}\ signal(cv)\ \{ P \}$ for arbitrary P
$\{ P \}\ signal_all(cv)\ \{ P \}$ for arbitrary P

Monitor solution to reader/writer problem

Verification of the invariant over `request_read`

$$I : (nr = 0 \vee nw = 0) \wedge nw \leq 1$$

```
procedure request_read() {
        { I }
        while (nw > 0) {        { I ∧ nw > 0 }
              { I } wait(oktoread); { I }
        }    { I ∧ nw = 0 }
        { I[nr + 1/nr] }
        nr := nr + 1;
        { I }
}
```

Verification conditions:

$(I \wedge nw > 0) \Rightarrow I$ $(I \wedge nw = 0) \Rightarrow I[nr+1/nr]$ 1>The invariant we had earlier already, it's the obvious one.

Axioms for Signal and Continue (2)

Assume that the invariant can mention the number of processes in the queue to a condition variable.

- Let $\#cv$ be the number of proc's waiting in the queue to cv.
- The test **empty**(cv) thus corresponds to $\#cv = 0$

wait(cv) is *modelled* as an extension of the queue followed by processor release:

$$wait(cv) : \{?\}\ \#cv := \#cv + 1; \{I\}\ \text{“}sleep''\{I\}$$

by assignment axiom:

$$wait(cv) : \{I[\#cv + 1/\#cv]; \#cv := \#cv + 1; \{\ I\ \}\ \text{“}sleep''\{\ I\ \}$$

Axioms for Signal and Continue (3)

$signal(cv)$ can be modelled as a reduction of the queue, if the queue is not empty:

$$\textbf{signal}(cv) : \{\ ?\ \}\ if\ (\#cv \neq 0)\ \#cv := \#cv - 1\ \{\ P\ \}$$

$$\begin{aligned} \textbf{signal}(cv) : \quad & \{((\#cv = 0) \Rightarrow P) \wedge ((\#cv \neq 0) \Rightarrow P[\#cv - 1/\#cv])\} \\ & if\ (\#cv \neq 0)\ \#cv := \#cv - 1 \\ & \{P\} \end{aligned}$$

- $signal_all(cv)$: $\{\ P[0/\#cv]\ \}\ \ \#cv := 0\ \{P\}$

Axioms for Signal and Continue (4)

Together this gives:

Axioms for monitor communication

$\{\ I[\#cv + 1/\#cv]\ \}$ **wait**(cv) $\{\ I\ \}$ WAIT

$\{\ ((\#cv = 0) \Rightarrow P) \wedge ((\#cv \neq 0) \Rightarrow P[\#cv - 1/\#cv])\ \}$ **signal**(cv) $\{\ P\ \}$ SIGNAL

$\{\ P[0/\#cv]\ \}$ **signal_all**(cv) $\{\ P\ \}$ SIGNALALL

If we know that $\#cv \neq 0$ whenever we signal, then the axiom for `signal(cv)` be simplified to:

$$\{\ P[\#cv - 1/\#cv]\ \}\ \textbf{signal}(cv)\ \{\ P\ \}$$

Note! *$\#cv$ is not allowed in statements!* Only used for reasoning.

Example: FIFO semaphore verification (1)

```
monitor Semaphore        { # monitor invariant: s ≥ 0
  int s := 0;              # value of the semaphore
  cond pos;                # wait condition

  procedure Psem() {
    if   (s=0)
         wait (pos);
    else
         s := s - 1
  }

  procedure Vsem() {
    if   empty(pos)
         s := s + 1
    else
         signal(pos);
  }
}
```

Consider the following monitor invariant:

$$s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

No process is waiting if the semaphore value is positive!

1>The example is from the monitor chapter. This is a monitor solution for FIFO-semaphores, even under the weak s&c signalling discipline. It's "forwarding the condition"

Example: FIFO semaphore verification: Psem

$$I: \quad s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

```
procedure Psem() {
{I}
   if (s=0) {I ∧ s = 0}
       {I[#pos + 1/#pos]} wait(pos); {I}
   else {I ∧ s ≠ 0}
      {I[s - 1/s]} s := s-1; {I}
{I}
}
```

Two proof obligations:

- Then-branch: $(I \wedge s = 0) \Rightarrow I[\#pos + 1/\#pos]$
- Else-branch: $(I \wedge s \neq 0) \Rightarrow I[s - 1/s]$

Example: FIFO semaphore verification (3)

$$I: \quad s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

This gives two proof obligations:

Then-branch:

$$\begin{array}{lcl} (I \wedge s = 0) & \Rightarrow & I[\#pos + 1/\#pos] \\ s = 0 & \Rightarrow & s \geq 0 \wedge (s > 0 \Rightarrow \#pos + 1 = 0) \\ s = 0 & \Rightarrow & s \geq 0 \end{array}$$

Else-branch:

$$\begin{array}{lcl} (I \wedge s \neq 0) & \Rightarrow & I[s - 1/s] \\ (s > 0 \wedge \#pos = 0) & \Rightarrow & s - 1 \geq 0 \wedge (s - 1 > 0 \Rightarrow \#pos = 0) \\ (s > 0 \wedge \#pos = 0) & \Rightarrow & s > 0 \wedge \#pos = 0 \end{array}$$

Note: $(s - 1 > 0 \Rightarrow \#pos = 0)$ follows since we have $\#pos = 0$.

Example: FIFO semaphore verification: Vsem

$$I: \quad s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

```
procedure Vsem() {
{I}
   if empty(pos) {I ∧ #pos = 0}
       {I[s + 1/s]}s:=s+1; {I}
   else {I ∧ #pos ≠ 0}
      {I[#pos − 1/#pos]} signal(pos); {I}
{I}
}
```

Example: FIFO semaphore verification (5)

$$I: \quad s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

As above, this gives two proof obligations:

Then-branch:

$$\begin{array}{lcl} (I \wedge \#pos = 0) & \Rightarrow & I[s+1/s] \\ (s \geq 0 \wedge \#pos = 0) & \Rightarrow & s+1 \geq 0 \wedge (s+1 > 0 \Rightarrow \#pos = 0) \\ (s \geq 0 \wedge \#pos = 0) & \Rightarrow & s+1 \geq 0 \wedge \#pos = 0 \end{array}$$

Else-branch:

$$\begin{array}{lcl} (I \wedge \#pos \neq 0) & \Rightarrow & I[\#pos - 1/\#pos] \\ (s = 0 \wedge \#pos \neq 0) & \Rightarrow & s \geq 0 \wedge (s > 0 \Rightarrow \#pos - 1 = 0) \\ s = 0 & \Rightarrow & s \geq 0 \end{array}$$

Summary

- interference proofs tricky – easy to forget something
  - critical regions help (but may reduce program efficiency)
  - global invariants help
    - avoid interference by splitting in global invariant and local condition
- reasoning about monitor invariants much easier
  - like local, sequential reasoning
  - but *signal/wait* need special care
  - can reason about the length of the condition queues
- reasoning about global/local invariants in the *await* language
  - sequential reasoning for each atomic statement/critical section
  - the conditions helps in reasoning (more than with *signal/wait*)
  - verified await programs can be transformed to other languages