Resources, Concurrency and Local Reasoning

Peter O'Hearn

Queen Mary, University of London

ETAPS, Barcelona, 29 March, 2004

Part I

An Illustrative Example

$$
\begin{array}{lcl}
\texttt{semaphore } \mathit{free} := 1;\ \mathit{busy} := 0 & & \\
\vdots & & \vdots \\
P(\mathit{free}); & & P(\mathit{busy}); \\
[10] := m; & \| & n := [10]; \\
V(\mathit{busy}); & & V(\mathit{free}); \\
\vdots & & \vdots
\end{array}
$$

Recall: $V(s)$ bumps s up by 1; $P(s)$ decrements by 1 if > 0, waiting otherwise

$$
\begin{array}{l}
\texttt{semaphore } \mathit{free} := 1;\ \mathit{busy} := 0 \\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\{\texttt{emp}\} \\
P(\mathit{free}); \qquad\qquad\qquad\qquad\qquad\qquad\quad P(\mathit{busy}); \\
\\
[10] := m; \qquad\qquad \| \qquad\qquad\quad n := [10]; \\
\\
V(\mathit{busy}); \qquad\qquad\qquad\qquad\qquad\quad V(\mathit{free}); \\
\\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots
\end{array}
$$

Recall: $V(s)$ bumps s up by 1; $P(s)$ decrements by 1 if > 0, waiting otherwise

$$
\begin{array}{l}
\texttt{semaphore } \mathit{free} := 1;\ \mathit{busy} := 0 \\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\{\texttt{emp}\} \\
P(\mathit{free}); \qquad\qquad\qquad\qquad\qquad\qquad\quad P(\mathit{busy}); \\
\{10 \mapsto -\} \\
[10] := m; \qquad\qquad \| \qquad\qquad n := [10]; \\
\\
V(\mathit{busy}); \qquad\qquad\qquad\qquad\qquad\quad V(\mathit{free}); \\
\\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots
\end{array}
$$

Recall: $V(s)$ bumps s up by 1; $P(s)$ decrements by 1 if > 0, waiting otherwise

$$
\begin{array}{l}
\texttt{semaphore } \mathit{free} := 1;\ \mathit{busy} := 0 \\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\{\texttt{emp}\} \\
P(\mathit{free}); \qquad\qquad\qquad\qquad\qquad\qquad\quad P(\mathit{busy}); \\
\{10 \mapsto -\} \\
[10] := m; \qquad\qquad \| \qquad\qquad\quad n := [10]; \\
\{10 \mapsto -\} \\
V(\mathit{busy}); \qquad\qquad\qquad\qquad\qquad\quad V(\mathit{free}); \\
\\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots
\end{array}
$$

Recall: $V(s)$ bumps s up by 1; $P(s)$ decrements by 1 if > 0, waiting otherwise

$$
\begin{array}{l}
\texttt{semaphore } \mathit{free} := 1;\ \mathit{busy} := 0 \\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\{\texttt{emp}\} \\
P(\mathit{free}); \qquad\qquad\qquad\qquad\qquad\qquad\quad P(\mathit{busy}); \\
\{10 \mapsto -\} \\
[10] := m; \qquad\qquad \| \qquad\qquad\quad n := [10]; \\
\{10 \mapsto -\} \\
V(\mathit{busy}); \qquad\qquad\qquad\qquad\qquad\quad V(\mathit{free}); \\
\{\texttt{emp}\} \\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots
\end{array}
$$

Recall: $V(s)$ bumps s up by 1; $P(s)$ decrements by 1 if > 0, waiting otherwise

$$
\begin{array}{l}
\texttt{semaphore } \mathit{free} := 1;\ \mathit{busy} := 0 \\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\{\texttt{emp}\} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \{\texttt{emp}\} \\
P(\mathit{free}); \qquad\qquad\qquad\qquad\qquad\qquad\quad\ P(\mathit{busy}); \\
\{10 \mapsto -\} \\
[10] := m; \qquad\qquad \| \qquad\qquad n := [10]; \\
\{10 \mapsto -\} \\
V(\mathit{busy}); \qquad\qquad\qquad\qquad\qquad\qquad\quad\ V(\mathit{free}); \\
\{\texttt{emp}\} \\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots
\end{array}
$$

Recall: $V(s)$ bumps s up by 1; $P(s)$ decrements by 1 if > 0, waiting otherwise

$$
\begin{array}{l}
\texttt{semaphore } \mathit{free} := 1;\ \mathit{busy} := 0 \\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\{\texttt{emp}\} \qquad\qquad\qquad\qquad\qquad\qquad\quad \{\texttt{emp}\} \\
P(\mathit{free}); \qquad\qquad\qquad\qquad\qquad\quad P(\mathit{busy}); \\
\{10 \mapsto -\} \qquad\qquad\qquad\qquad\qquad\quad \{10 \mapsto -\} \\
[10] := m; \qquad\qquad \| \qquad\qquad n := [10]; \\
\{10 \mapsto -\} \\
V(\mathit{busy}); \qquad\qquad\qquad\qquad\qquad\quad V(\mathit{free}); \\
\{\texttt{emp}\} \\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots
\end{array}
$$

Recall: $V(s)$ bumps s up by 1; $P(s)$ decrements by 1 if > 0, waiting otherwise

$$
\begin{array}{l}
\texttt{semaphore } \mathit{free} := 1;\ \mathit{busy} := 0 \\
\vdots \\
\{\texttt{emp}\} \\
P(\mathit{free}); \\
\{10 \mapsto -\} \\
[10] := m; \\
\{10 \mapsto -\} \\
V(\mathit{busy}); \\
\{\texttt{emp}\} \\
\vdots
\end{array}
\quad \| \quad
\begin{array}{l}
\vdots \\
\{\texttt{emp}\} \\
P(\mathit{busy}); \\
\{10 \mapsto -\} \\
n := [10]; \\
\{10 \mapsto -\} \\
V(\mathit{free}); \\
\\
\vdots
\end{array}
$$

Recall: $V(s)$ bumps s up by 1; $P(s)$ decrements by 1 if > 0, waiting otherwise

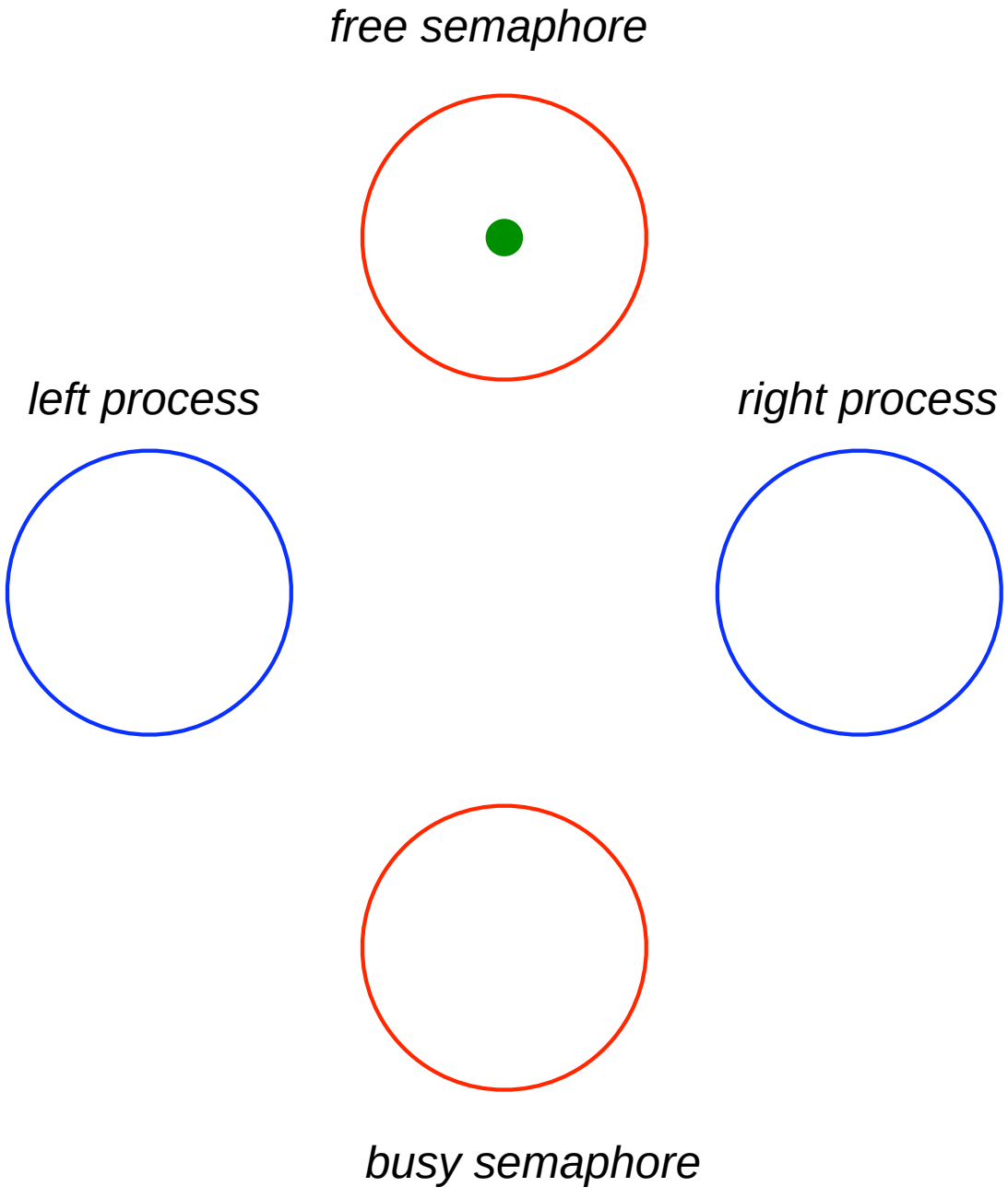
$$
\begin{array}{l}
\texttt{semaphore } \mathit{free} := 1;\ \mathit{busy} := 0 \\
\vdots \\
\{\texttt{emp}\} \\
P(\mathit{free}); \\
\{10 \mapsto -\} \\
[10] := m; \\
\{10 \mapsto -\} \\
V(\mathit{busy}); \\
\{\texttt{emp}\} \\
\vdots
\end{array}
\quad \| \quad
\begin{array}{l}
\\
\vdots \\
\{\texttt{emp}\} \\
P(\mathit{busy}); \\
\{10 \mapsto -\} \\
n := [10]; \\
\{10 \mapsto -\} \\
V(\mathit{free}); \\
\{\texttt{emp}\} \\
\vdots
\end{array}
$$

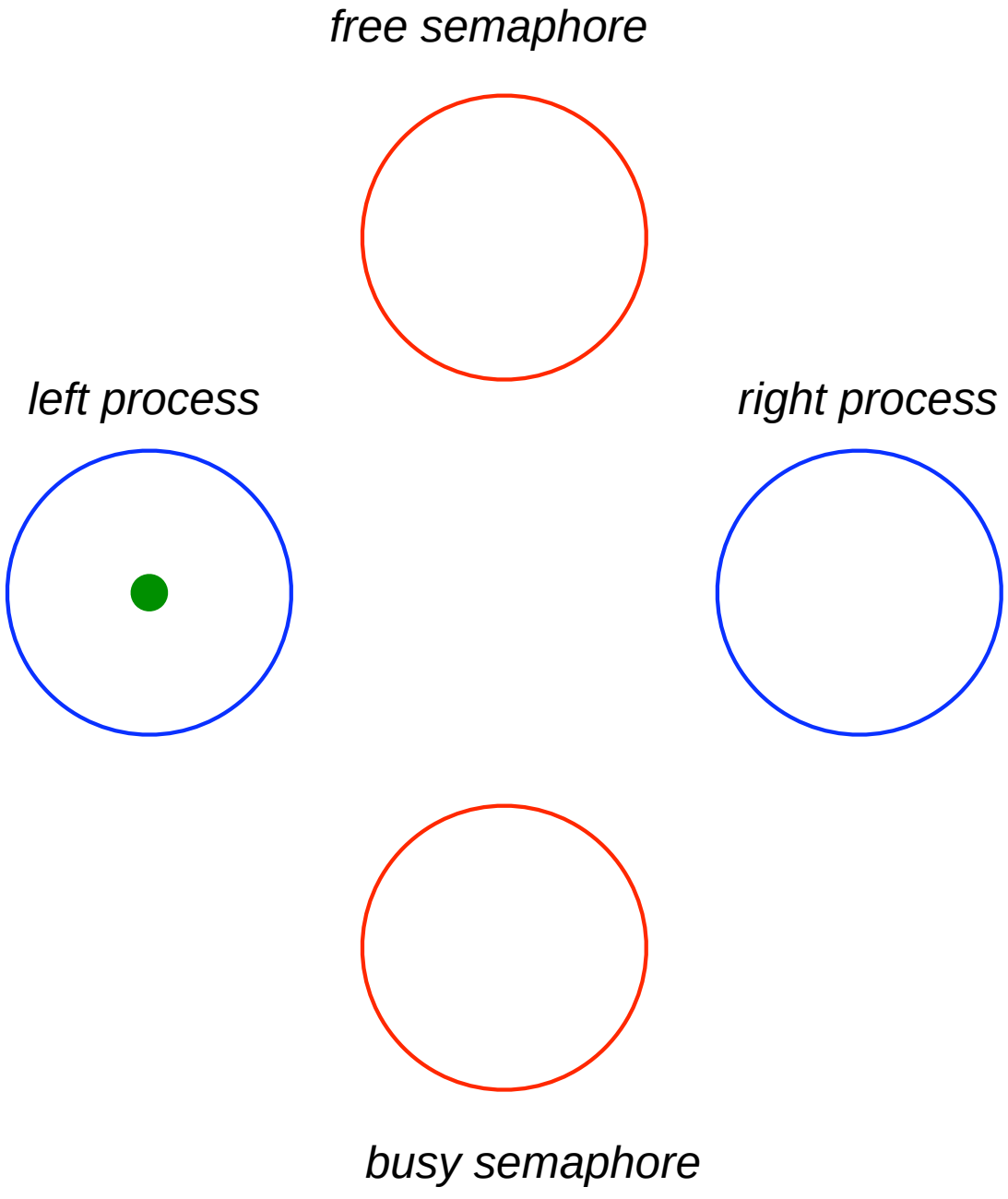
Recall: $V(s)$ bumps s up by 1; $P(s)$ decrements by 1 if > 0, waiting otherwise

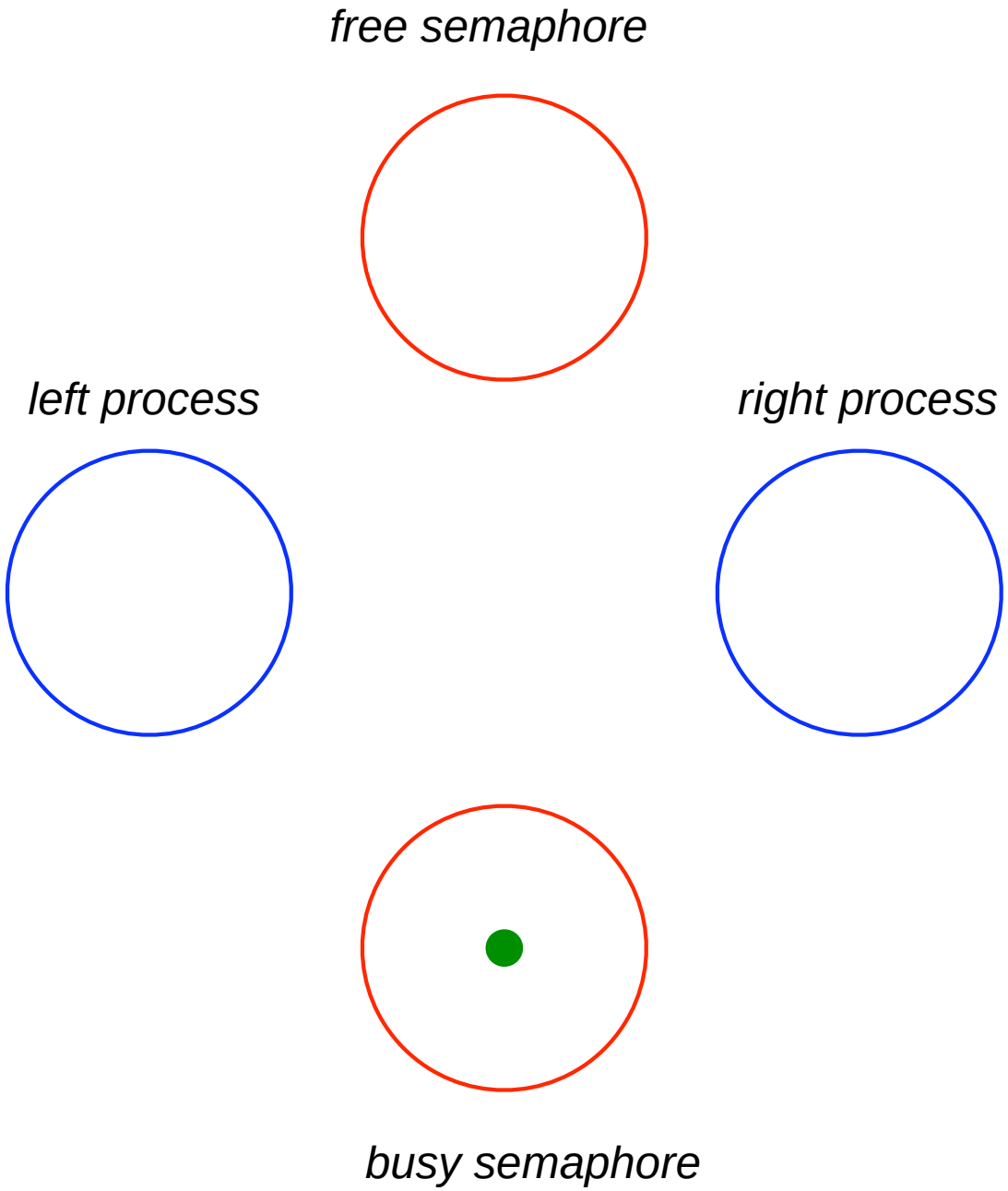
$$
\begin{array}{l}
\texttt{semaphore}\ \mathit{free} := 1;\ \mathit{busy} := 0 \\
\vdots \\
\{\mathtt{emp}\} \\
P(\mathit{free}); \\
\{10 \mapsto -\} \\
[10] := m; \\
\{10 \mapsto -\} \\
V(\mathit{busy}); \\
\{\mathtt{emp}\} \\
\vdots
\end{array}
\quad \| \quad
\begin{array}{l}
\\
\vdots \\
\{\mathtt{emp}\} \\
P(\mathit{busy}); \\
\{10 \mapsto -\} \\
n := [10]; \\
\{10 \mapsto -\} \\
V(\mathit{free}); \\
\{\mathtt{emp}\} \\
\vdots
\end{array}
$$

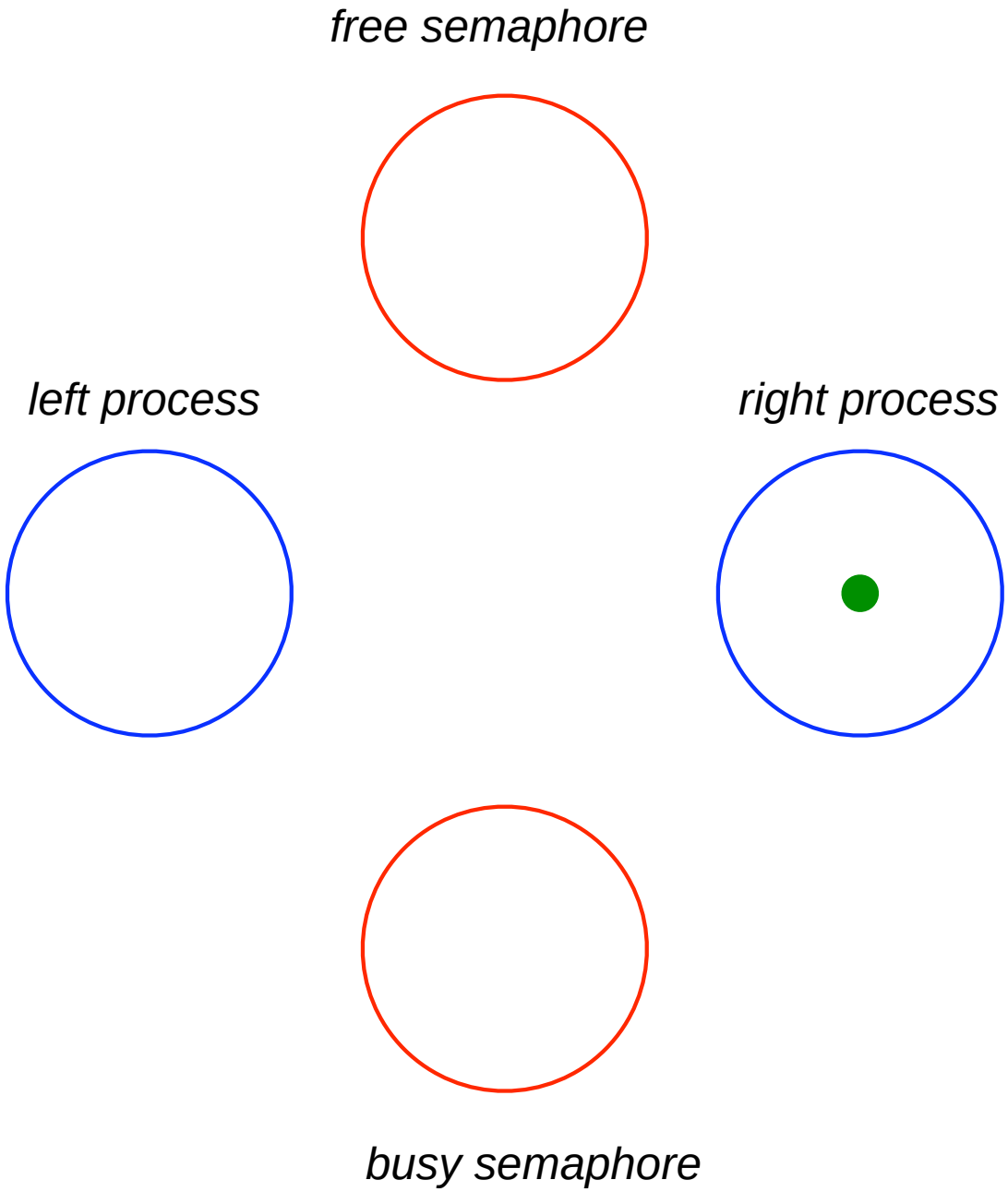
$$RI_{free} \stackrel{\text{def}}{=} (\mathit{free} = 0 \wedge \mathtt{emp}) \vee (\mathit{free} = 1 \wedge 10 \mapsto -)$$

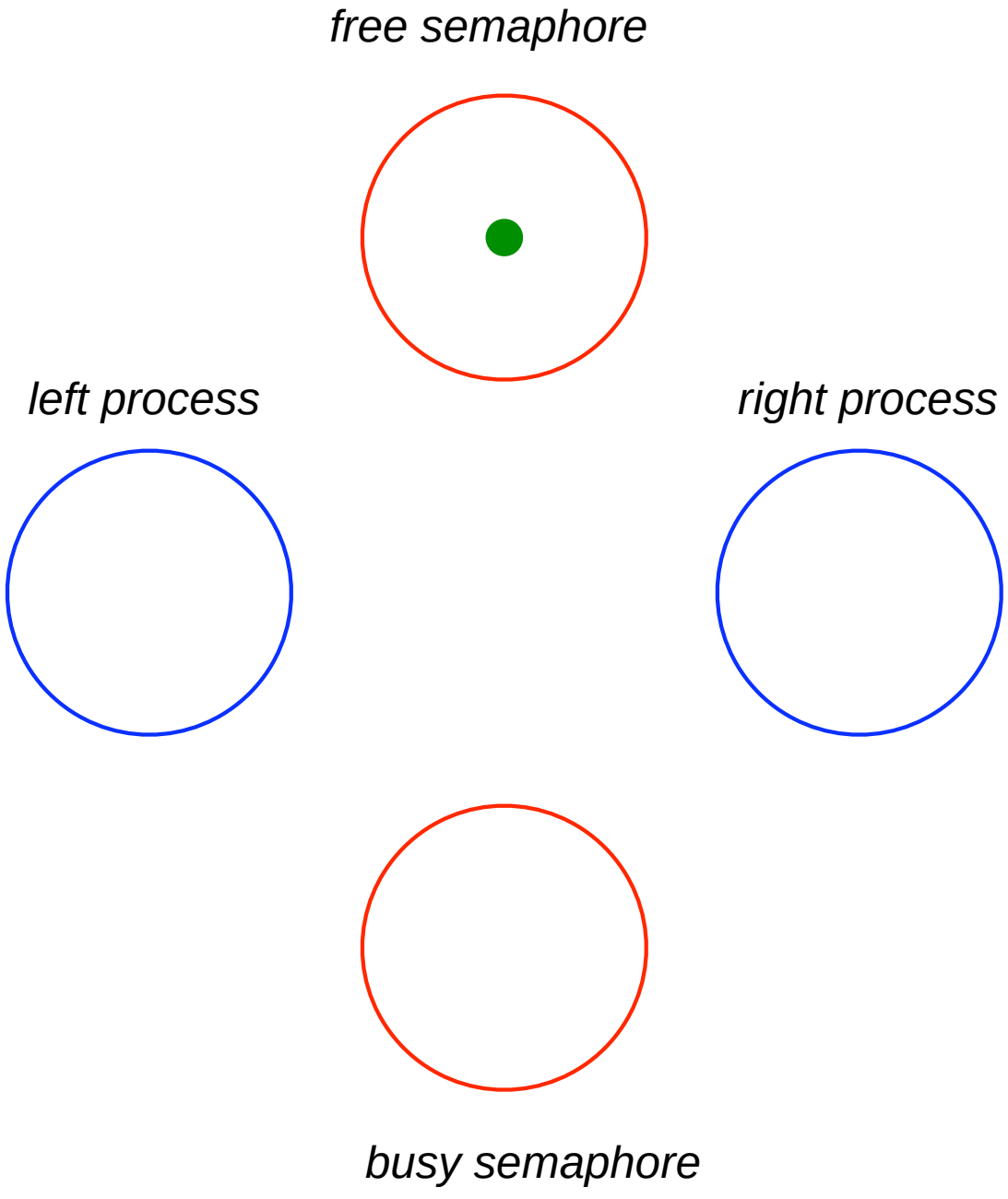
$$RI_{busy} \stackrel{\text{def}}{=} (\mathit{busy} = 0 \wedge \mathtt{emp}) \vee (\mathit{busy} = 1 \wedge 10 \mapsto -)$$


free semaphore
left process
right process
busy semaphore


free semaphore
left process
right process
busy semaphore


free semaphore
left process
right process
busy semaphore


free semaphore
left process
right process
busy semaphore


free semaphore
left process
right process
busy semaphore

$\{10 \mapsto -\}$

semaphore $free := 1,\ busy := 0;$

$\{RI_{free} * RI_{busy} * \mathtt{emp} * \mathtt{emp}\}$

$\{\mathtt{emp}\}$		$\{\mathtt{emp}\}$
`while true do`		`while true do`
produce m		$P(busy);$
$P(free);$		$n := [10];$
$[10] := m;$	$\|$	$V(free);$
$V(busy);$		*consume n*
$\{\mathtt{false}\}$		$\{\mathtt{false}\}$

$\{\mathtt{false}\}$

$$RI_{free} \stackrel{\text{def}}{=} (free = 0 \wedge \mathtt{emp}) \vee (free = 1 \wedge 10 \mapsto -)$$

$$RI_{busy} \stackrel{\text{def}}{=} (busy = 0 \wedge \mathtt{emp}) \vee (busy = 1 \wedge 10 \mapsto -)$$

$\{10 \mapsto -\}$

semaphore $free := 1,\ busy := 0;$

$\{RI_{free} * RI_{busy} * \mathtt{emp} * \mathtt{emp}\}$

$\{\mathtt{emp}\}$	$\{\mathtt{emp}\}$
while true do	while true do
$\{\mathtt{emp} \wedge \mathtt{true}\}$	$\{\mathtt{emp} \wedge \mathtt{true}\}$
produce m	$P(busy);$
$\{\mathtt{emp}\}$	$\{\mathtt{emp}\}$
$P(free);$	$n := [10];$
$\{10 \mapsto -\}$	$\{10 \mapsto -\}$
$[10] := m;$ $\quad \|$	$V(free);$
$\{10 \mapsto -\}$	$\{10 \mapsto -\}$
$V(busy);$	*consume n*
$\{\mathtt{emp}\}$	$\{\mathtt{emp}\}$
$\{\mathtt{false}\}$	$\{\mathtt{false}\}$

$\{\mathtt{false}\}$

$$RI_{free} \stackrel{\text{def}}{=} (free = 0 \wedge \mathtt{emp}) \vee (free = 1 \wedge 10 \mapsto -)$$

$$RI_{busy} \stackrel{\text{def}}{=} (busy = 0 \wedge \mathtt{emp}) \vee (busy = 1 \wedge 10 \mapsto -)$$

Remarks

- Each semaphore invariant talks only about itself, not about the other semaphore or the processes.

Remarks

- Each semaphore invariant talks only about itself, not about the other semaphore or the processes.
- Each assertion within a process talks about only its own state, not the state of the other process or even the semaphores.

Remarks

- Each semaphore invariant talks only about itself, not about the other semaphore or the processes.
- Each assertion within a process talks about only its own state, not the state of the other process or even the semaphores.
- We don't maintain $0 \leq \mathit{free} + \mathit{busy} \leq 1$ as a global invariant.

Remarks

- Each semaphore invariant talks only about itself, not about the other semaphore or the processes.
- Each assertion within a process talks about only its own state, not the state of the other process or even the semaphores.
- We don't maintain $0 \leq \mathit{free} + \mathit{busy} \leq 1$ as a global invariant.
- Semaphores are "logically attached" to resource. P and V are ownership transformers.

Part II

Context

Resource Logics

- Linear Logic (Girard, 1987)
- BI (O'Hearn, Pym, 1999)
- Separation Logic (Reynolds, O'Hearn,... 2000...)

Resource Logics

- Linear Logic (Girard, 1987)
- BI (O'Hearn, Pym, 1999)
- Separation Logic (Reynolds, O'Hearn,... 2000...)

- Resource is a central concern in concurrent programming
- A natural question: Could resource logic be used?

Local Reasoning

- It should be possible for reasoning and specification to be confined to the resources that a program actually accesses (it's *resource footprint*.[1]

[1] O'Hearn, Reynolds, Yang. Local Reasoning about Programs that Alter Date Structures. CSL'01

Local Reasoning

- It should be possible for reasoning and specification to be confined to the resources that a program actually accesses (it's *resource footprint*.[1]

- Separation Logic's *frame rule*

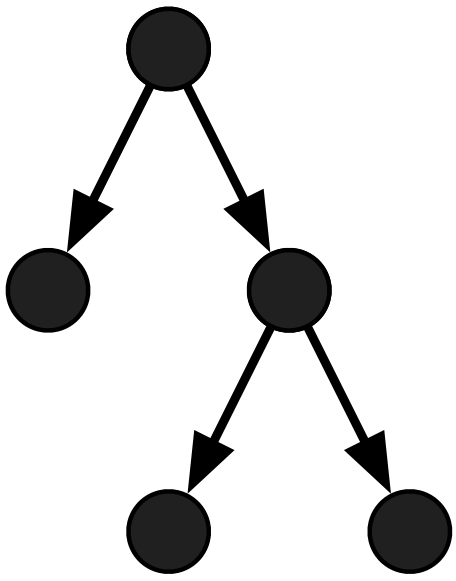
$$\frac{\{P\}\,C\,\{Q\}}{\{P * R\}\,C\,\{Q * R\}}\ \mathrm{Modifies}(C) \cap \mathrm{Free}(R) = \emptyset$$

[1] O'Hearn, Reynolds, Yang. Local Reasoning about Programs that Alter Date Structures. CSL'01

From

{ tree(z) } dispose tree(z) {emp}

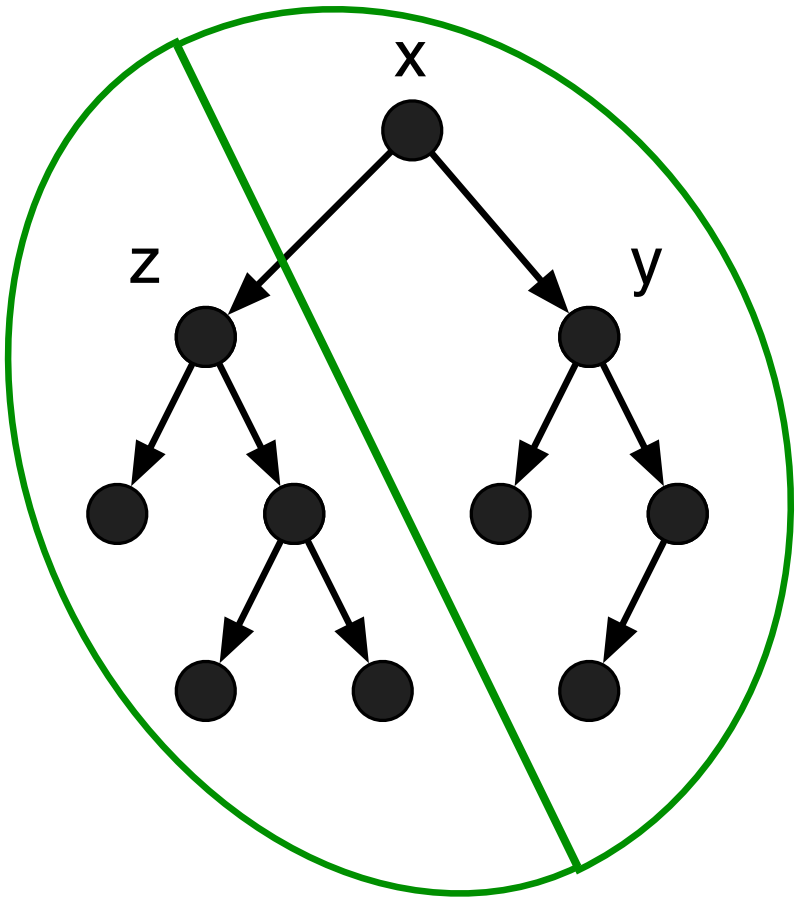
z

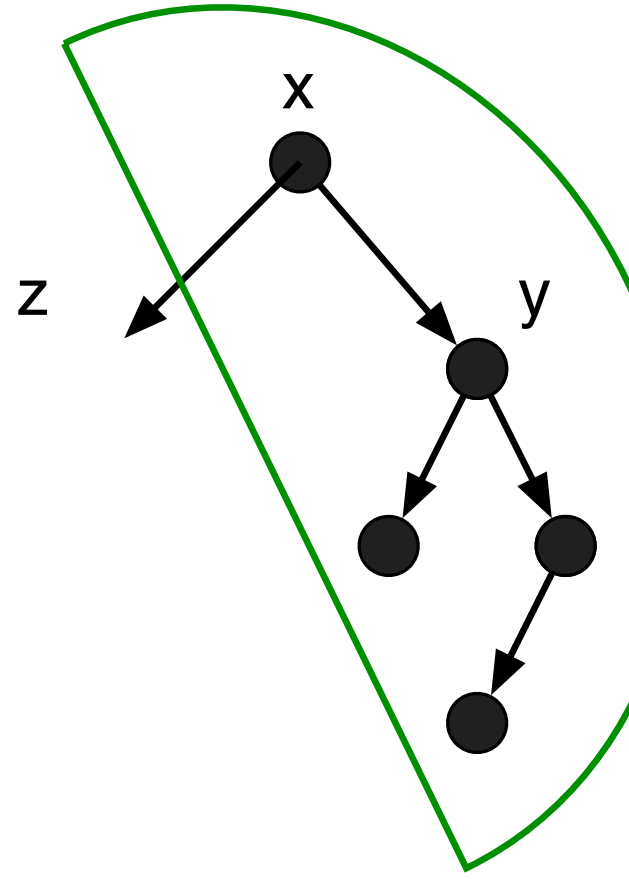


z

We Can Infer

{ tree(z) * tree(y) * xl-y,z} disposetree(z) {tree(y) * xl-y,z}





Parallel Disposetree

- Disjoint Concurrency

$$\frac{\{P_1\}\,C_1\,\{Q_1\} \quad \{P_2\}\,C_2\,\{Q_2\}}{\{P_1 * P_2\}\,C_1 \parallel C_2\,\{Q_1 * Q_2\}}$$

Parallel Disposetree

- Disjoint Concurrency

$$\frac{\{P_1\}\,C_1\,\{Q_1\} \quad \{P_2\}\,C_2\,\{Q_2\}}{\{P_1 * P_2\}\,C_1 \parallel C_2\,\{Q_1 * Q_2\}}$$

- Main Part of Proof

$$\begin{array}{c} \{tree(y) * tree(z)\} \\ \{tree(y)\} \qquad\qquad \{tree(z)\} \\ \texttt{disposetree}(y) \quad \parallel \quad \texttt{disposetree}(z) \\ \{\texttt{emp}\} \qquad\qquad \{\texttt{emp}\} \\ \{\texttt{emp} * \texttt{emp}\} \\ \{\texttt{emp}\} \end{array}$$

- Specification: $\{tree(x)\}\texttt{disposetree}(x)\{\texttt{emp}\}$

Parallel Disposetree

- Disjoint Concurrency

$$\frac{\{P_1\}\,C_1\,\{Q_1\} \quad \{P_2\}\,C_2\,\{Q_2\}}{\{P_1 * P_2\}\,C_1 \parallel C_2\,\{Q_1 * Q_2\}}$$

- Main Part of Proof

$$\begin{array}{c} \{tree(y) * tree(z)\} \\ \{tree(y)\} \qquad\qquad \{tree(z)\} \\ \texttt{disposetree}(y) \quad \parallel \quad \texttt{disposetree}(z) \\ \{\texttt{emp}\} \qquad\qquad \{\texttt{emp}\} \\ \{\texttt{emp} * \texttt{emp}\} \\ \{\texttt{emp}\} \end{array}$$

- Specification: $\{tree(x)\}\texttt{disposetree}(x)\{\texttt{emp}\}$
- No need for a rely (no one touches my tree)

Parallel Disposetree

- Disjoint Concurrency

$$\frac{\{P_1\}\,C_1\,\{Q_1\} \quad \{P_2\}\,C_2\,\{Q_2\}}{\{P_1 * P_2\}\,C_1 \parallel C_2\,\{Q_1 * Q_2\}}$$

- Main Part of Proof

$$\begin{array}{c} \{tree(y) * tree(z)\} \\ \{tree(y)\} \qquad\qquad \{tree(z)\} \\ \texttt{disposetree}(y) \quad \parallel \quad \texttt{disposetree}(z) \\ \{\texttt{emp}\} \qquad\qquad \{\texttt{emp}\} \\ \{\texttt{emp} * \texttt{emp}\} \\ \{\texttt{emp}\} \end{array}$$

- Specification: $\{tree(x)\}\texttt{disposetree}(x)\{\texttt{emp}\}$
- No need for a rely (no one touches my tree)
- Or a guarantee (I touch only my own tree)

Reasoning about (shared-variable) Concurrency

- Owicki-Gries, Rely-Guarantee, Temporal Logic...

Reasoning about (shared-variable) Concurrency

- Owicki-Gries, Rely-Guarantee, Temporal Logic...
- Hoare, 1972: Towards a Theory of Parallel Programming
  - revolves around "spatial separation"
  - extremely modular
  - limited, but...

Cautious and Daring Concurrency

- A program is *cautious* if, whenever concurrent processes access the same piece of state, they do so only within the same grouping of mutual exclusion. Otherwise, the program is *daring*.

Cautious and Daring Concurrency

- A program is *cautious* if, whenever concurrent processes access the same piece of state, they do so only within the same grouping of mutual exclusion. Otherwise, the program is *daring*.

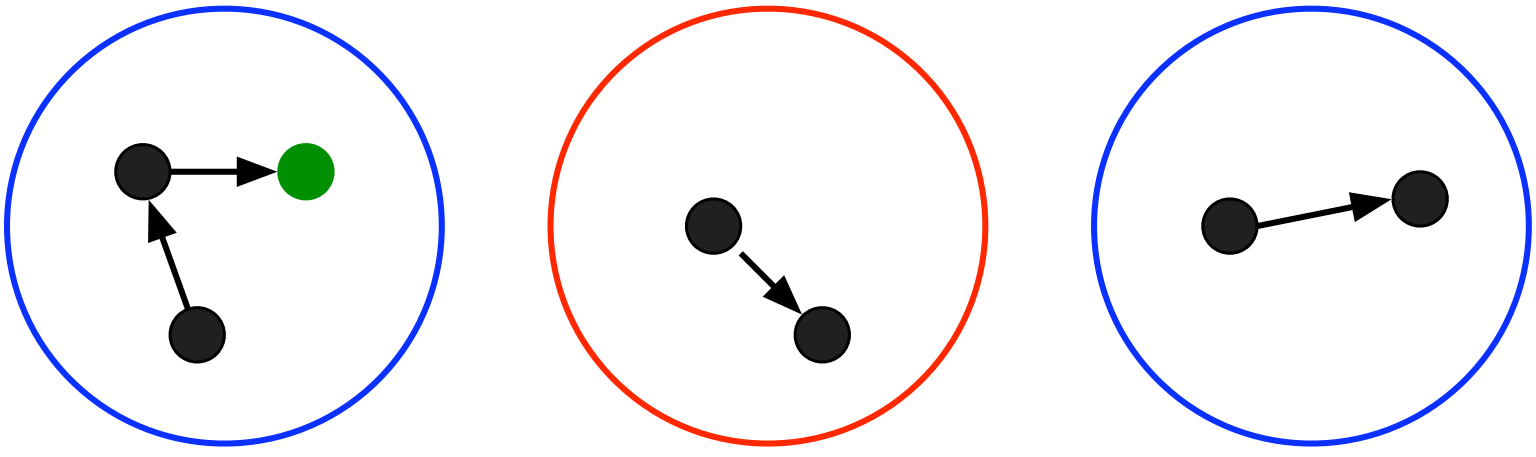
- Daring programs are many
  - Memory Managers, Thread Pools, Connection Pools

Cautious and Daring Concurrency

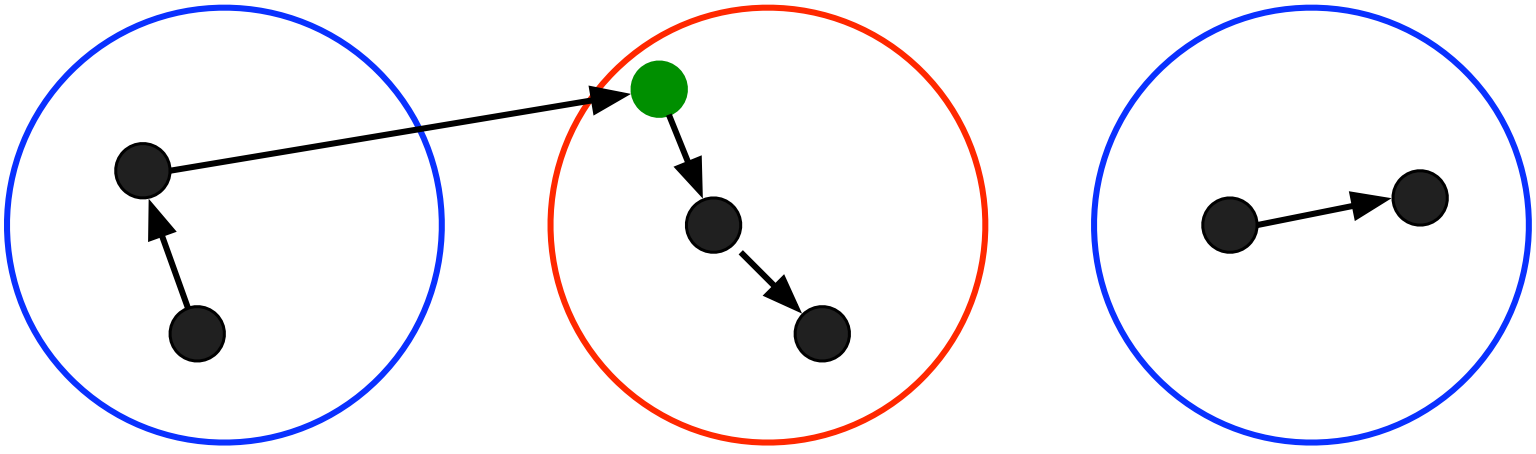
- A program is *cautious* if, whenever concurrent processes access the same piece of state, they do so only within the same grouping of mutual exclusion. Otherwise, the program is *daring*.

- Daring programs are many
  - Memory Managers, Thread Pools, Connection Pools
  - Efficient Message Passing (copy avoiding)
  - Double-buffered I/O
  - Microkernel OS designs
  - Essentially all semaphore programs

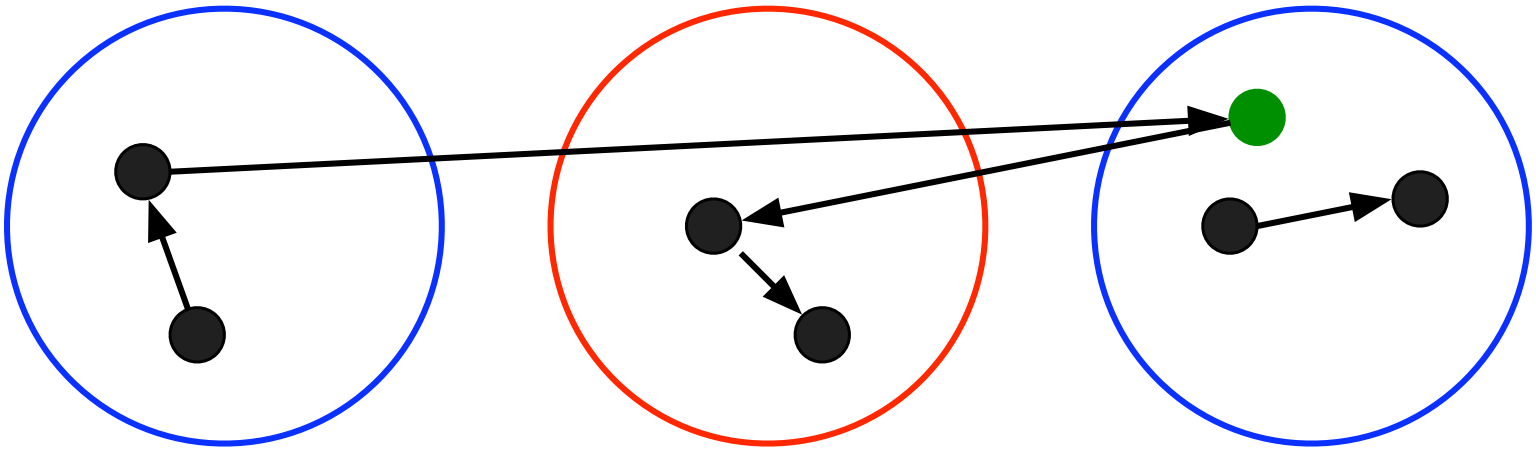
Dynamic Partitioning and Ownership Transfer



Dynamic Partitioning and Ownership Transfer



Dynamic Partitioning and Ownership Transfer



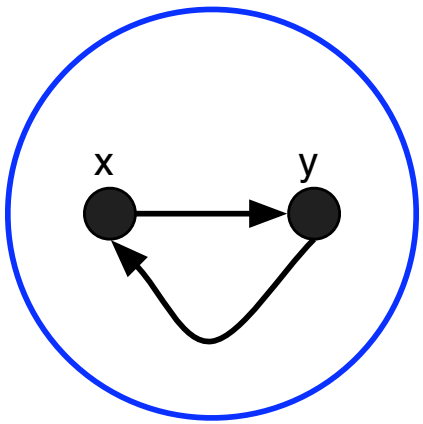
Part III

Separation Logic[2]

[2]Separation Logic: A logic for shared mutable data structure. John Reynolds, LICS'02.. + papers by Reynolds, O'Hearn, Yang...

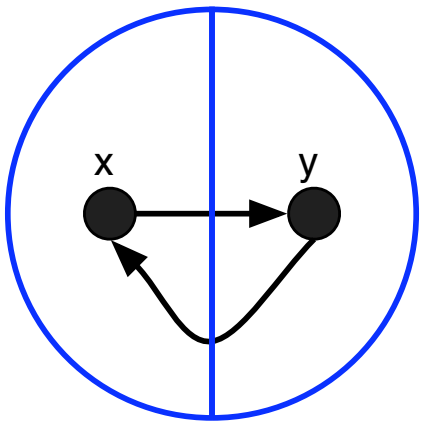
Separation Logic

$$x|\!\!\to y \;*\; y|\!\!\to x$$



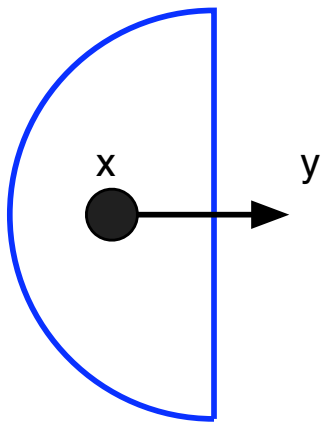
Separation Logic

xl->y * yl-> x



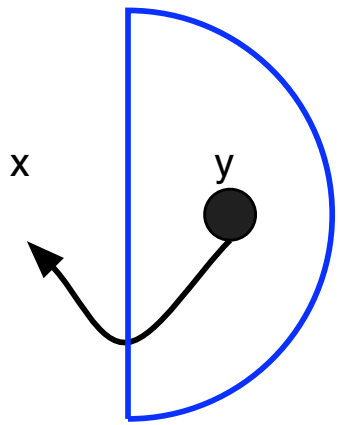
Separation Logic

x|->y



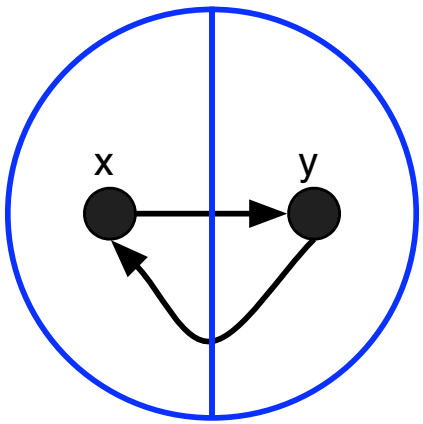
Separation Logic

y|-> x



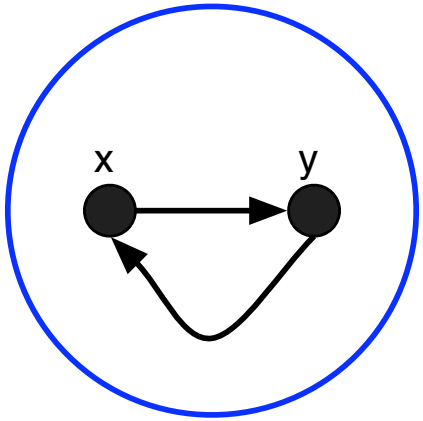
Separation Logic

xl->y * yl-> x



Separation Logic

xl->y * yl-> x



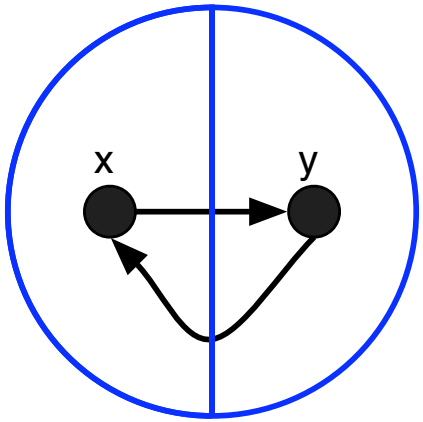
x=10

y=42

10	42
42	10

Separation Logic

xl->y * yl-> x



x=10

y=42

10

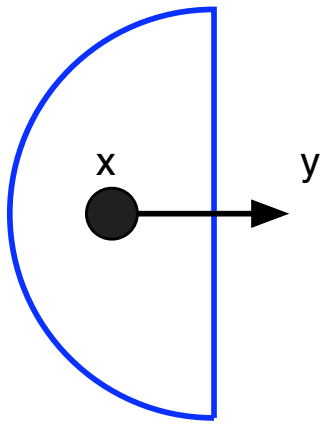
42

42

10

Separation Logic

x|->y



x=10

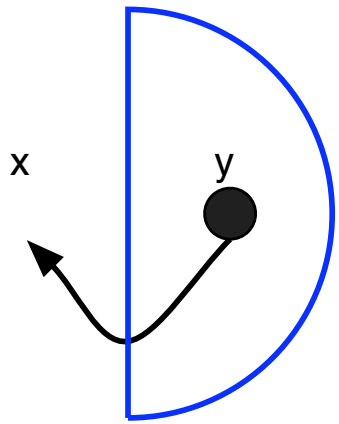
y=42

10

42

Separation Logic

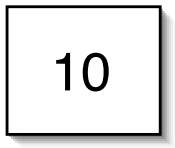
y|-> x



x=10

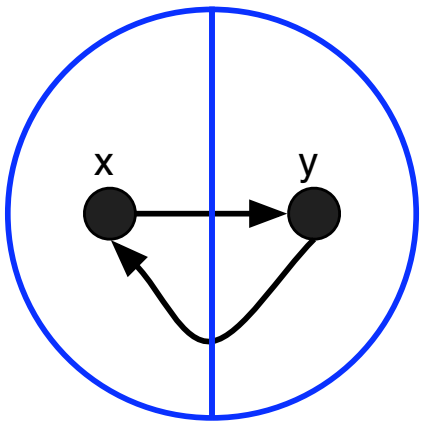
y=42

42



Separation Logic

xl->y * yl-> x



The Assertion Language and Model

Assertions

$$P, Q, R \quad ::= \quad \texttt{emp} \mid E \mapsto F \mid P * Q \mid P \rightarrow Q \mid \forall x.P \mid \cdots$$

Semantics: $s, h \models P$, where

$$s \in S = \texttt{Variables} \rightharpoonup \texttt{Val} \qquad h \in H = \texttt{Addresses} \rightharpoonup \texttt{Val}$$

RAM Model: `Addresses`=Naturals, `Val`=Integers.

$*$ splits h component, unioning heaps with disjoint domains. H has a partial commutative monoid structure.

The powerset of H is a boolean algebra, *and* a residuated monoid. (A Boolean BI Algebra.)[3]

[3] Possible Worlds and Resources, the Semantics of BI. Pym, O'Hearn, Yang, TCS'04. And Pym monograph on BI.

Well specified programs don't go wrong

$\{(x \mapsto -) * P\}\ [x] := 7\ \{(x \mapsto 7) * P\}$

Well specified programs don't go wrong

$\{(x \mapsto -) * P\}\ [x] := 7\ \{(x \mapsto 7) * P\}$

$\{\texttt{true}\}\ [x] := 7\ \{???\}$

Well specified programs don't go wrong

$\{(x \mapsto -) * P\}\ [x]{:=}\,7\ \{(x \mapsto 7) * P\}$

$\{\texttt{true}\}\ [x]{:=}\,7\ \{???\}$

$\{P * (x \mapsto -)\}\ \texttt{dispose}(x)\ \{P\}$

Well specified programs don't go wrong

$\{(x \mapsto -) * P\}\ [x] := 7\ \{(x \mapsto 7) * P\}$

$\{\texttt{true}\}\ [x] := 7\ \{???\}$

$\{P * (x \mapsto -)\}\ \texttt{dispose}(x)\ \{P\}$

$\{\texttt{true}\}\ \texttt{dispose}(x)\ \{???\}$

Well specified programs don't go wrong

$\{(x \mapsto -) * P\}\ [x]{:=}\,7\ \{(x \mapsto 7) * P\}$

$\{\texttt{true}\}\ [x]{:=}\,7\ \{???\}$

$\{P * (x \mapsto -)\}\ \texttt{dispose}(x)\ \{P\}$

$\{\texttt{true}\}\ \texttt{dispose}(x)\ \{???\}$

$\{P\}\ \ x = \texttt{cons}(a, b)\ \ \{P * (x \mapsto a, b)\}\ \ \ (x \notin \mathit{free}(P))$

Part IV

Concurrency Proof Rules[4]

[4] Following Hoare, and Owicki-Gries

Process Interaction:

Conditional Critical Regions (Hoare, 72)

init;
`resource` $r_1(\text{variable list}), \ldots, r_m(\text{variable list})$
$C_1 \parallel \cdots \parallel C_n$

`with` r `when` B `do` C `endwith`.

–if a variable belongs to a resource, it cannot appear in a parallel process except in a critical section for that resource

–if a variable is changed in one process, it cannot appear in another unless it belongs to a resource.

$$\frac{\{P\}\,init\,\{RI_{r_1} * \cdots * RI_{r_m} * P'\} \qquad \{P'\}\,C_1 \parallel \cdots \parallel C_n\,\{Q\}}{}$$

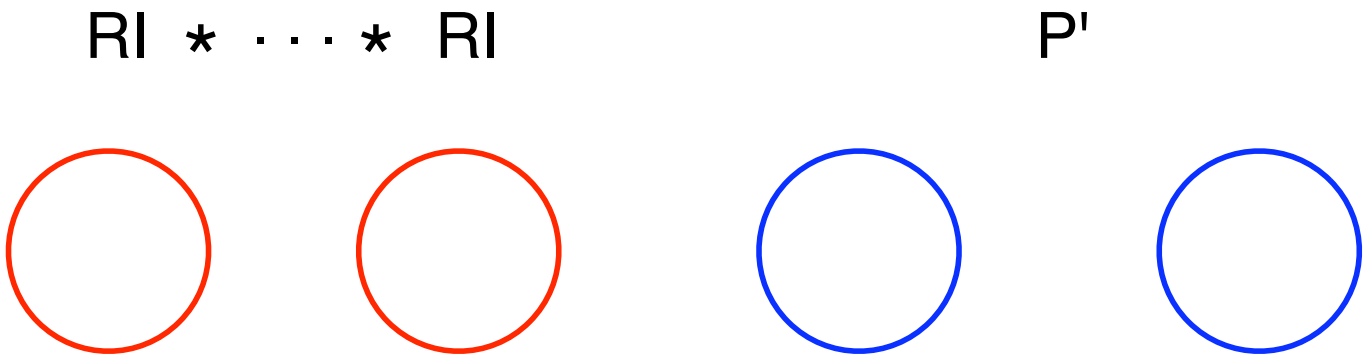
$\{P\}$

init;

`resource` r_1(variable list), . . . , r_m(variable list)

$C_1 \parallel \cdots \parallel C_n$

$\{RI_{r_1} * \cdots * RI_{r_m} * Q\}$



$$\frac{\{P\}\, init\, \{RI_{r_1} * \cdots * RI_{r_m} * P'\} \qquad \{P'\}\, C_1 \parallel \cdots \parallel C_n \{Q\}}{}$$

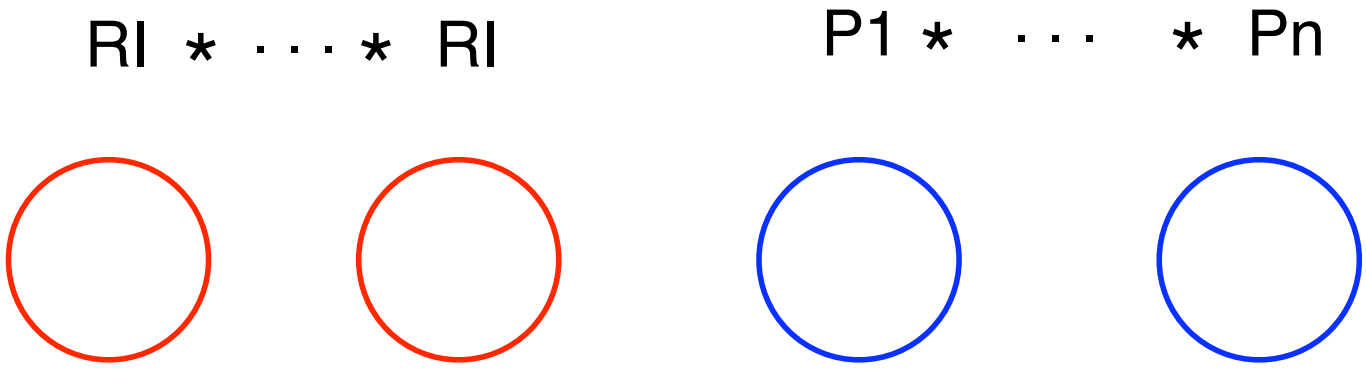
$\{P\}$

init;

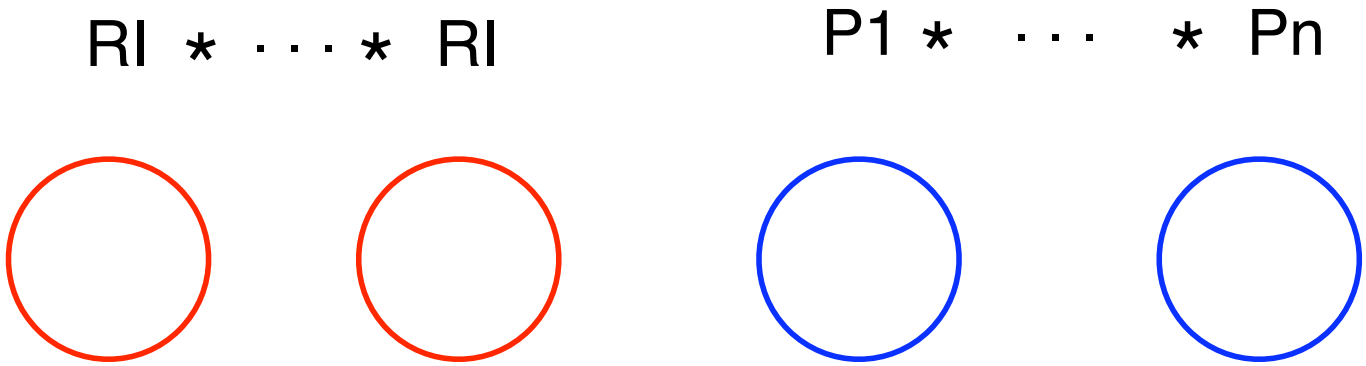
`resource` r_1(variable list), ..., r_m(variable list)

$C_1 \parallel \cdots \parallel C_n$

$\{RI_{r_1} * \cdots * RI_{r_m} * Q\}$



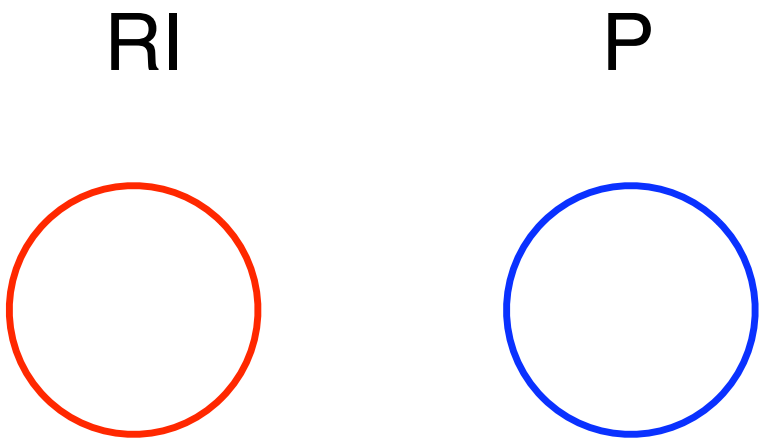
$$\frac{\{P_1\}\, C_1\, \{Q_1\} \;\cdots\; \{P_n\}\, C_n\, \{Q_n\}}{\{P_1 * \cdots * P_n\}\, C_1 \parallel \cdots \parallel C_n\, \{Q_1 * \cdots * Q_n\}}$$



– no variable free in P_i or Q_i is changed in C_j when $j \not= i$.

– any modified variable free in RI_r must occur only within a critical region for r.

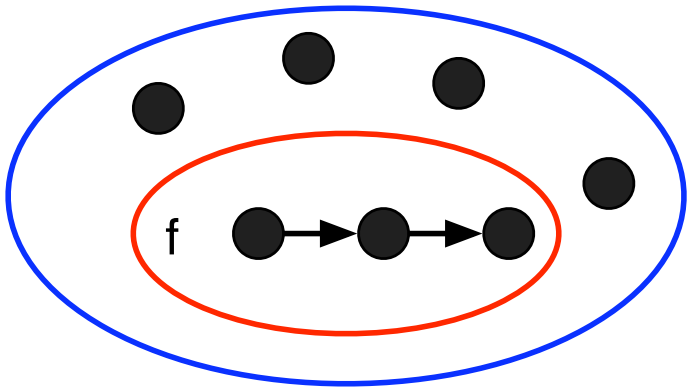
$$\frac{\{(P * RI_r) \wedge B\}\ C\ \{Q * RI_r\}}{\{P\}\ \texttt{with}\ r\ \texttt{when}\ B\ \texttt{do}\ C\ \texttt{endwith}\ \{Q\}}$$



- No other process modifies variables in P or Q

Brookes's Semantic Analysis

- **Soundness Theorem**. The proof rules are sound, as long as resource invariants are *precise*.
- Reynolds counterexample shows unsoundness for imprecise invariants.
- A predicate is precise if, for every state, there is at most one substate that satisfies it.



Brookes's Semantic Analysis, II

- **Safety Theorem**. If $\{P\}C\{Q\}$ holds then there will be *no race condition* and *no memory fault* in any execution starting in a state satisfying P.
- **Slogan**. *Well specified processes mind their own business.*

Part V

Toy Resource Manager

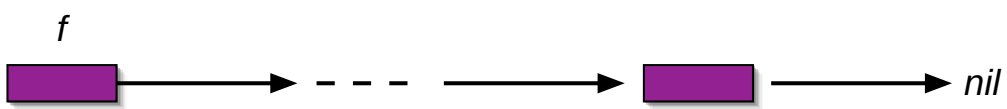
$$\texttt{resource}\ \texttt{mm}(f).$$

$$\texttt{alloc}(x, a, b) \stackrel{\text{def}}{=} \texttt{with}\ \texttt{mm}\ \texttt{when}\ \texttt{true}\ \texttt{do}\ \texttt{if}\ f = \texttt{nil}\ \texttt{then}\ x := \texttt{cons}(a, b)\ \texttt{else}\ x := f;\ f := x.2;\ z.1 := a;\ x.2 := b\ \texttt{endwith}$$

$$\texttt{dealloc}(y) \stackrel{\text{def}}{=} \texttt{with}\ \texttt{mm}\ \texttt{when}\ \texttt{true}\ \texttt{do}\ y.2 := f;\ f := y;\ \texttt{endwith}$$

Resource Invariant: *list*(f)

The manager owns the free list.



$$\mathit{list}\, x \stackrel{\text{def}}{\text{ iff }} (x = \texttt{nil} \wedge \texttt{emp}) \vee (\exists y.\, x \mapsto -, y * \mathit{list}\, y)$$

- Proof Establishing $\{y \mapsto -,-\}\texttt{dealloc}(y)\{\texttt{emp}\}$:

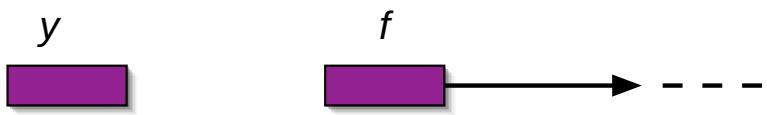
  $\{(y \mapsto -,-) * \mathit{list}\ f\}$

  $y.2 := f;$

  $f := y;$

  $\{\mathit{list}\ f\}$

  $\{\texttt{emp} * \mathit{list}\ f\}$



- Proof Establishing $\{y \mapsto -,-\}\texttt{dealloc}(y)\{\texttt{emp}\}$:

  $\{(y \mapsto -,-) * \mathit{list}\ f\}$

  $y.2 := f;$

  $f := y;$

  $\{\mathit{list}\ f\}$

  $\{\texttt{emp} * \mathit{list}\ f\}$

- Proof Establishing $\{y \mapsto -,-\}\texttt{dealloc}(y)\{\texttt{emp}\}$:

  $\{(y \mapsto -,-) * \mathit{list}\ f\}$
  $y.2 := f;$
  $f := y;$
  $\{\mathit{list}\ f\}$
  $\{\texttt{emp} * \mathit{list}\ f\}$

- $$\frac{\{(P * RI_r) \wedge B\}\ C\ \{Q * RI_r\}}{\{P\}\ \texttt{with}\ r\ \texttt{when}\ B\ \texttt{do}\ C\ \texttt{endwith}\ \{Q\}}$$

- Proof Establishing $\{y \mapsto -,-\}\texttt{dealloc}(y)\{\texttt{emp}\}$:

  $\{(y \mapsto -,-) * \mathit{list}\ f\}$
  $y.2 := f;$
  $f := y;$
  $\{\mathit{list}\ f\}$
  $\{\texttt{emp} * \mathit{list}\ f\}$

- $$\frac{\{(P * RI_r) \wedge B\}\ C\ \{Q * RI_r\}}{\{P\}\ \texttt{with}\ r\ \texttt{when}\ B\ \texttt{do}\ C\ \texttt{endwith}\ \{Q\}}$$

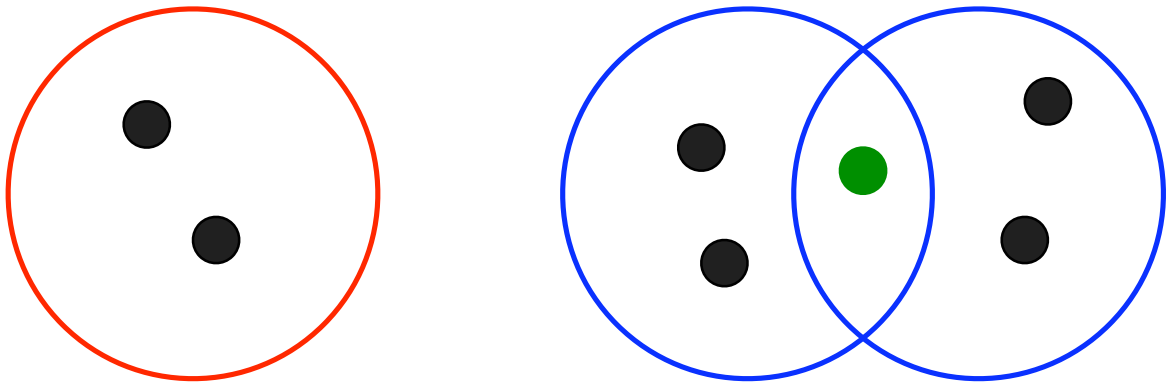
- Similarly, $\{\texttt{emp}\}\texttt{alloc}(x, a, b)\{x \mapsto a, b\}$.

A Safe Example

$$
\begin{array}{l}
\vdots \\
\{x \mapsto -, -\} \\
x.1 := 42; \\
\{x \mapsto -, -\} \\
\texttt{dealloc}(x); \\
\{\texttt{emp}\} \\
\vdots
\end{array}
\quad \| \quad
\begin{array}{l}
\vdots \\
\{\texttt{emp}\} \\
\texttt{alloc}(y, a, b); \\
\{y \mapsto a, b\} \\
y.1 := 7; \\
\{y \mapsto 7, b\} \\
\vdots
\end{array}
$$

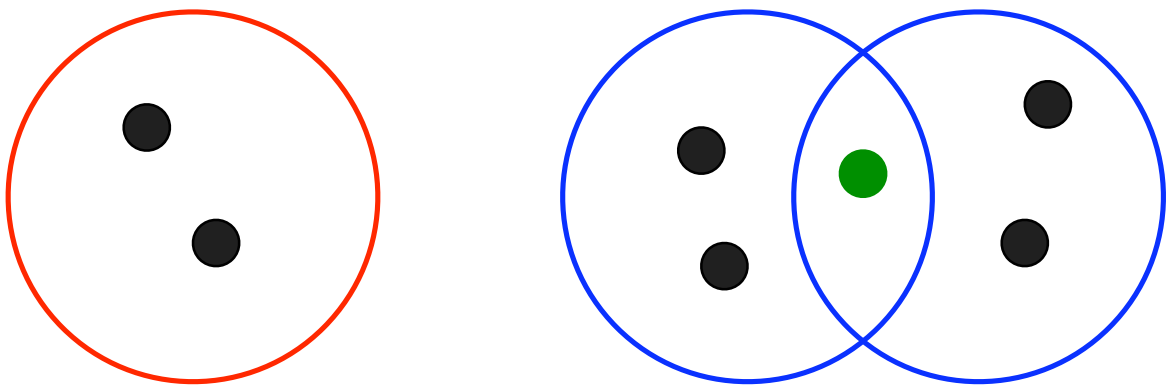
An Unsafe Example

$$
\begin{array}{l}
\vdots \\
\{x \mapsto -, -\} \\
x.1 := 42; \\
\{x \mapsto -, -\} \\
\texttt{dealloc}(x); \\
\{\texttt{emp}\}
\end{array}
\quad \| \quad
\begin{array}{l}
\vdots \\
\{\texttt{emp}\} \\
\texttt{alloc}(y, a, b); \\
\{y \mapsto a, b\} \\
y.2 := 17; \\
\{y \mapsto a, y\} \\
\vdots \\
\vdots
\end{array}
$$



An Unsafe Example

$$
\begin{array}{l}
\vdots \\
\{x \mapsto -, -\} \\
x.1 := 42; \\
\{x \mapsto -, -\} \\
\texttt{dealloc}(x); \\
\{\texttt{emp}\} \\
\\
x.2 := x \\
\\
\{???\}
\end{array}
\quad \| \quad
\begin{array}{l}
\vdots \\
\{\texttt{emp}\} \\
\texttt{alloc}(y, a, b); \\
\{y \mapsto a, b\} \\
y.2 := 17; \\
\{y \mapsto a, y\} \\
\vdots \\
\vdots
\end{array}
$$



Part VI

Modularity

Pointer-transferring Buffer

$full$:= `false`;
`resource` $buf(c, full)$

$$\texttt{put}(x) \stackrel{\text{def}}{=} \texttt{with}\ buf\ \texttt{when}\ \neg full\ \texttt{do}\ c := x;\ full := \texttt{true}\ \texttt{endwith}$$

$$\texttt{get}(y) \stackrel{\text{def}}{=} \texttt{with}\ buf\ \texttt{when}\ full\ \texttt{do}\ y := c;\ full := \texttt{false}\ \texttt{endwith};$$

Pointer-transferring Buffer

$full$:= `false`;
`resource` $buf(c, full)$

$$\texttt{put}(x) \stackrel{\text{def}}{=} \begin{array}{l} \texttt{with } buf \texttt{ when } \neg full \texttt{ do} \\ \quad c := x;\ full := \texttt{true} \\ \texttt{endwith} \end{array}$$

$$\texttt{get}(y) \stackrel{\text{def}}{=} \begin{array}{l} \texttt{with } buf \texttt{ when } full \texttt{ do} \\ \quad y := c;\ full := \texttt{false} \\ \texttt{endwith}; \end{array}$$

$\{x \mapsto -, -\}\texttt{put}(x)\{\texttt{emp}\}$ $\quad$ $\{\texttt{emp}\}\texttt{get}(y)\{y \mapsto -, -\}$

$RI_{buf} \stackrel{\text{def}}{=} (full \wedge c \mapsto -, -) \vee (\neg full \wedge \texttt{emp})$

Code Snippet

$$
\begin{array}{l}
\{\mathtt{emp} * \mathtt{emp}\} \\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\{\mathtt{emp}\} \qquad\qquad\qquad\qquad\qquad\quad \{\mathtt{emp}\} \\
x := \mathtt{cons}(a, b) \qquad \| \qquad \mathtt{get}(y); \\
\{x \mapsto -, -\} \qquad\qquad\qquad\quad \{y \mapsto -, -\} \\
\mathtt{put}(x); \qquad\qquad\qquad\qquad\quad \mathit{use}(y); \\
\{\mathtt{emp}\} \qquad\qquad\qquad\qquad\quad \{y \mapsto -, -\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathtt{dispose}(y); \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \{\mathtt{emp}\} \\
\{\mathtt{emp} * \mathtt{emp}\} \\
\{\mathtt{emp}\}
\end{array}
$$

$$RI_{buf} \stackrel{\text{def}}{=} (\mathit{full} \wedge c \mapsto -, -) \vee (\neg \mathit{full} \wedge \mathtt{emp})$$

Code Snippet

$$
\begin{array}{lcl}
\multicolumn{3}{c}{\{\mathtt{emp} * \mathtt{emp}\}} \\
\vdots & & \vdots \\
\{\mathtt{emp}\} & & \{\mathtt{emp}\} \\
x := \mathtt{cons}(a, b) & \| & \mathtt{get}(y); \\
\{x \mapsto -, -\} & & \{y \mapsto -, -\} \\
\mathtt{put}(x); & & \mathit{use}(y); \\
\{\mathtt{emp}\} & & \{y \mapsto -, -\} \\
\mathtt{x.1} := 44 & & \mathtt{dispose}(y); \\
\{???\} & & \{\mathtt{emp}\} \\
\multicolumn{3}{c}{\{\mathtt{emp} * \mathtt{emp}\}} \\
\multicolumn{3}{c}{\{\mathtt{emp}\}}
\end{array}
$$

$$RI_{buf} \stackrel{\text{def}}{=} (\mathit{full} \wedge c \mapsto -, -) \vee (\neg \mathit{full} \wedge \mathtt{emp})$$

Code Snippet

$$
\begin{array}{l}
\{\mathtt{emp} * \mathtt{emp}\} \\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\{\mathtt{emp}\} \qquad\qquad\qquad\qquad\qquad\quad \{\mathtt{emp}\} \\
x := \mathtt{cons}(a, b) \qquad \| \qquad \mathtt{get}(y); \\
\{x \mapsto -, -\} \qquad\qquad\qquad\quad \{y \mapsto -, -\} \\
\mathtt{put}(x); \qquad\qquad\qquad\qquad\quad \mathit{use}(y); \\
\{\mathtt{emp}\} \qquad\qquad\qquad\qquad\quad \{y \mapsto -, -\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathtt{dispose}(y); \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \{\mathtt{emp}\} \\
\{\mathtt{emp} * \mathtt{emp}\} \\
\{\mathtt{emp}\}
\end{array}
$$

$$RI_{buf} \stackrel{\text{def}}{=} (\mathit{full} \wedge c \mapsto -, -) \vee (\neg \mathit{full} \wedge \mathtt{emp})$$

Combining *buf* and mm

$$
\begin{array}{l}
\{\mathtt{emp}{*}\mathtt{emp}\} \\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\{\mathtt{emp}\} \qquad\qquad\qquad\qquad\qquad\quad \{\mathtt{emp}\} \\
\ \mathtt{alloc}(x,a,b); \quad \| \quad \mathtt{get}(y); \\
\{x \mapsto -,-\} \qquad\qquad\qquad\quad \{y \mapsto -,-\} \\
\mathtt{put}(x); \qquad\qquad\qquad\qquad\quad \mathit{use}(y); \\
\{\mathtt{emp}\} \qquad\qquad\qquad\qquad\quad \{y \mapsto -,-\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \ \mathtt{dealloc}(y); \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \{\mathtt{emp}\} \\
\{\mathtt{emp}{*}\mathtt{emp}\} \\
\{\mathtt{emp}\}
\end{array}
$$

$$RI_{buf} \stackrel{\text{def}}{=} (\mathit{full} \wedge c \mapsto -,-) \vee (\neg \mathit{full} \wedge \mathtt{emp})$$

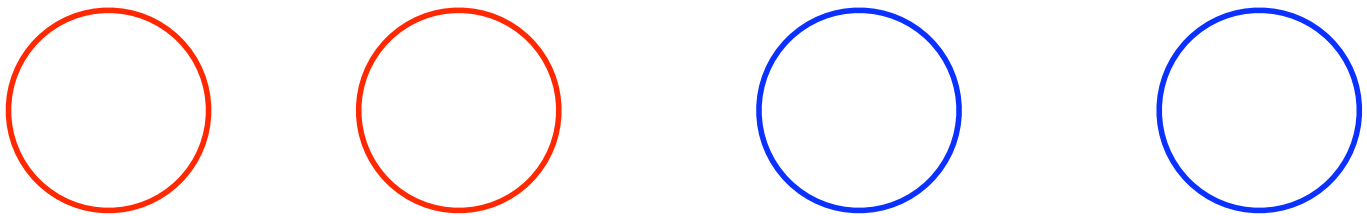
$$RI_{\mathtt{mm}} \stackrel{\text{def}}{=} \mathit{list}(f)$$

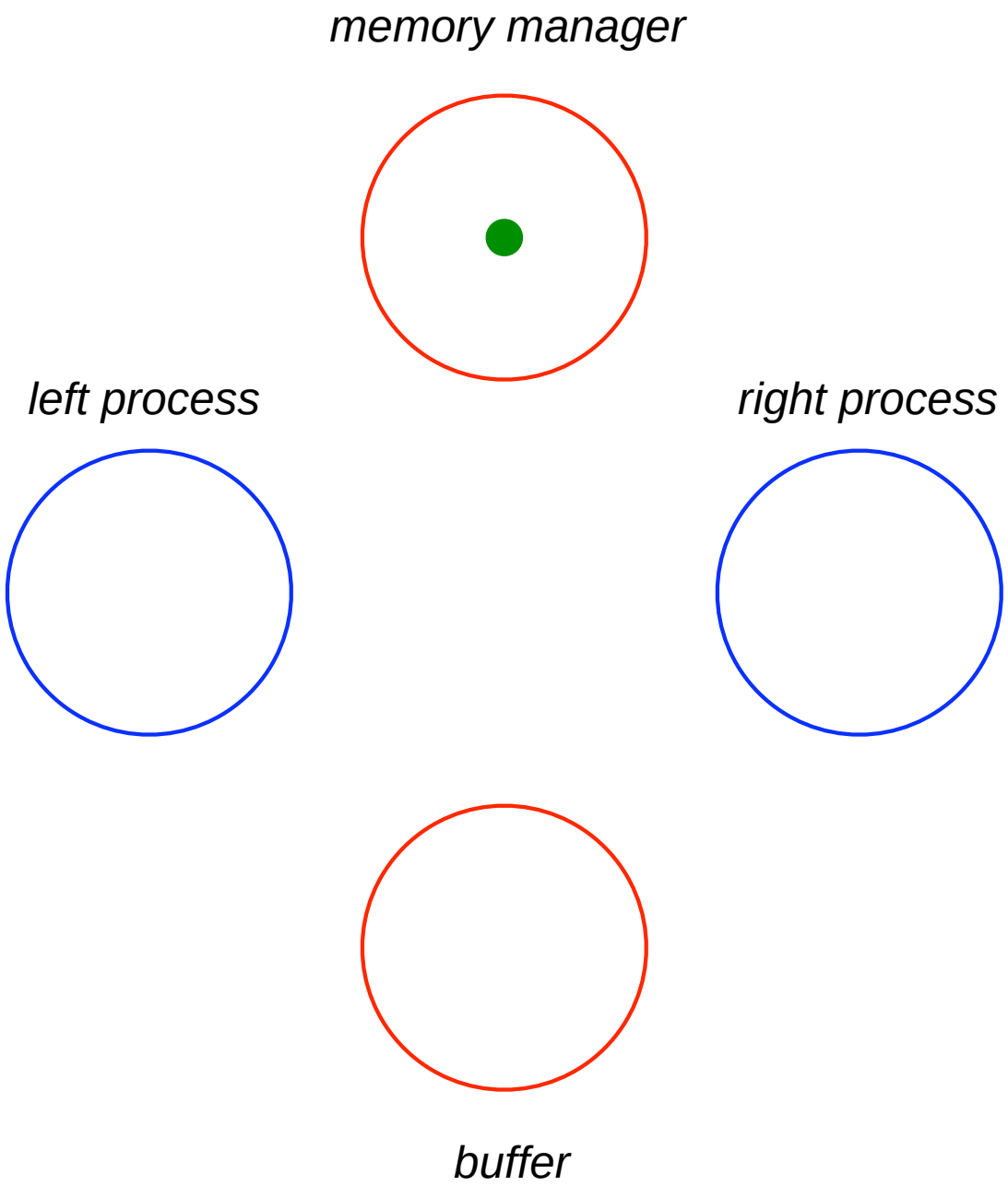
$$\frac{\{P_1\}\, C_1\, \{Q_1\} \;\cdots\; \{P_n\}\, C_n\, \{Q_n\}}{\{P_1 * \cdots * P_n\}\, C_1 \parallel \cdots \parallel C_n\, \{Q_1 * \cdots * Q_n\}}$$

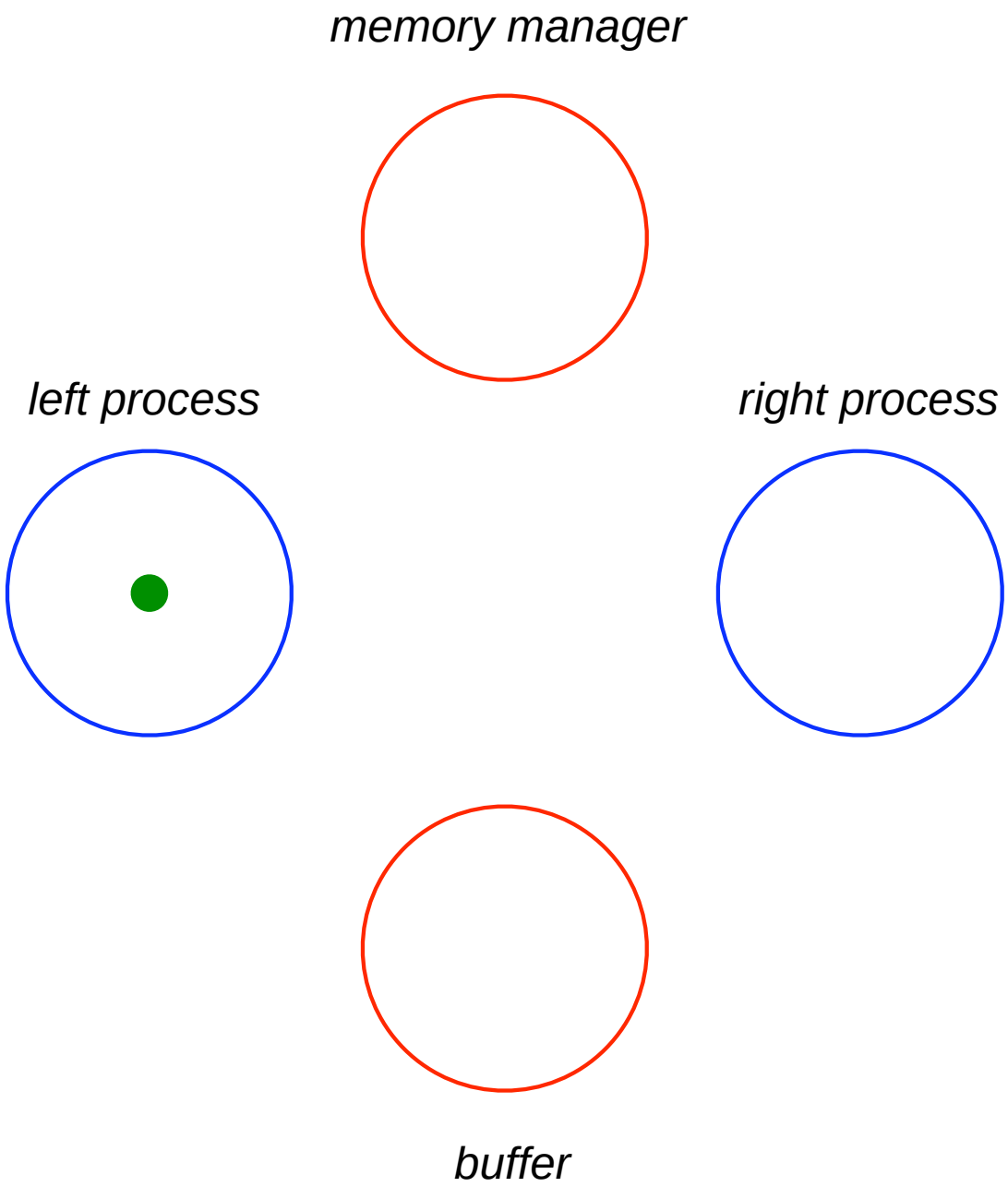
$$\frac{\{(P * RI_r) \wedge B\}\, C\, \{Q * RI_r\}}{\{P\}\, \texttt{with}\ r\ \texttt{when}\ B\ \texttt{do}\ C\ \texttt{endwith}\, \{Q\}}$$

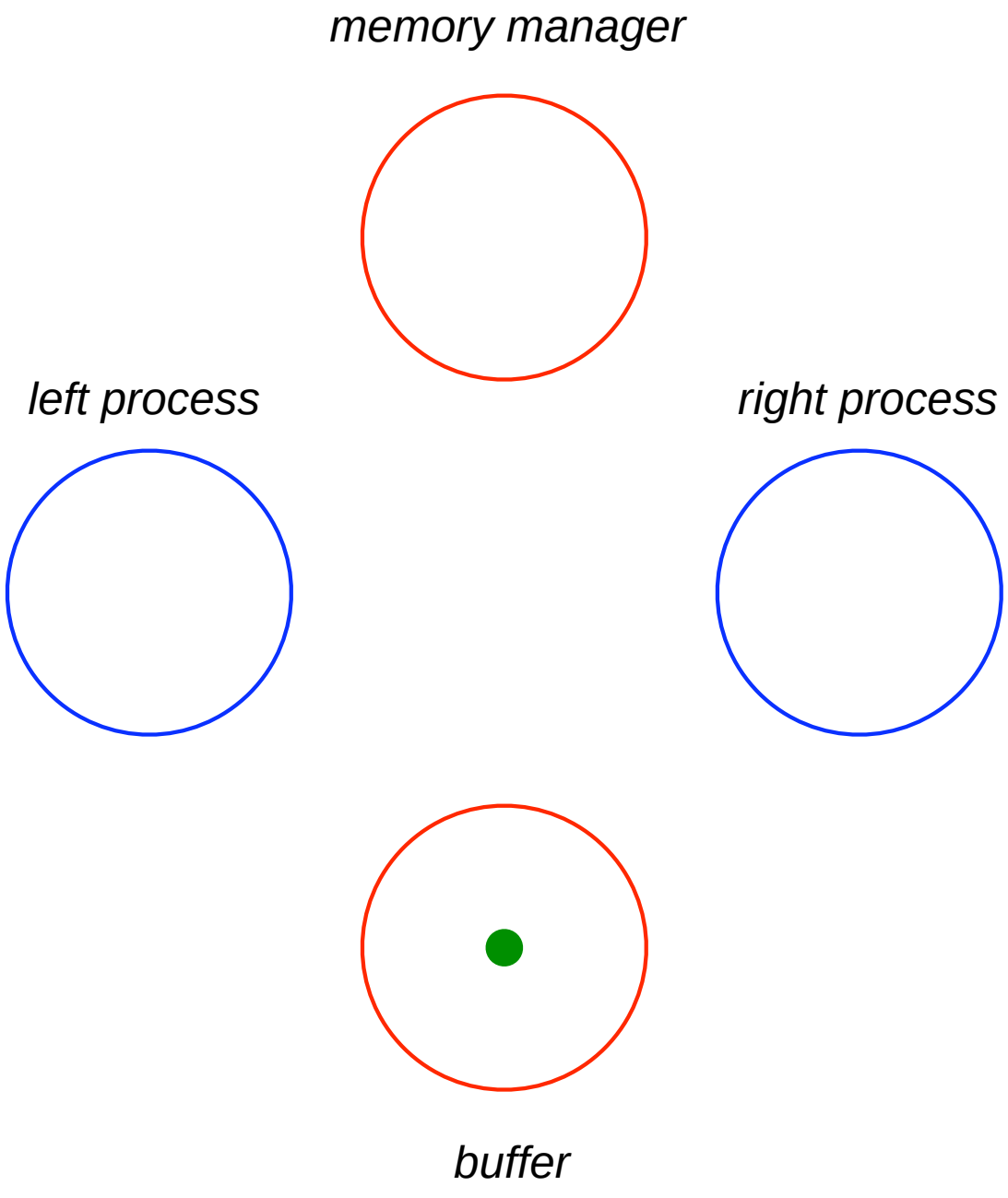
RI * ··· * RI

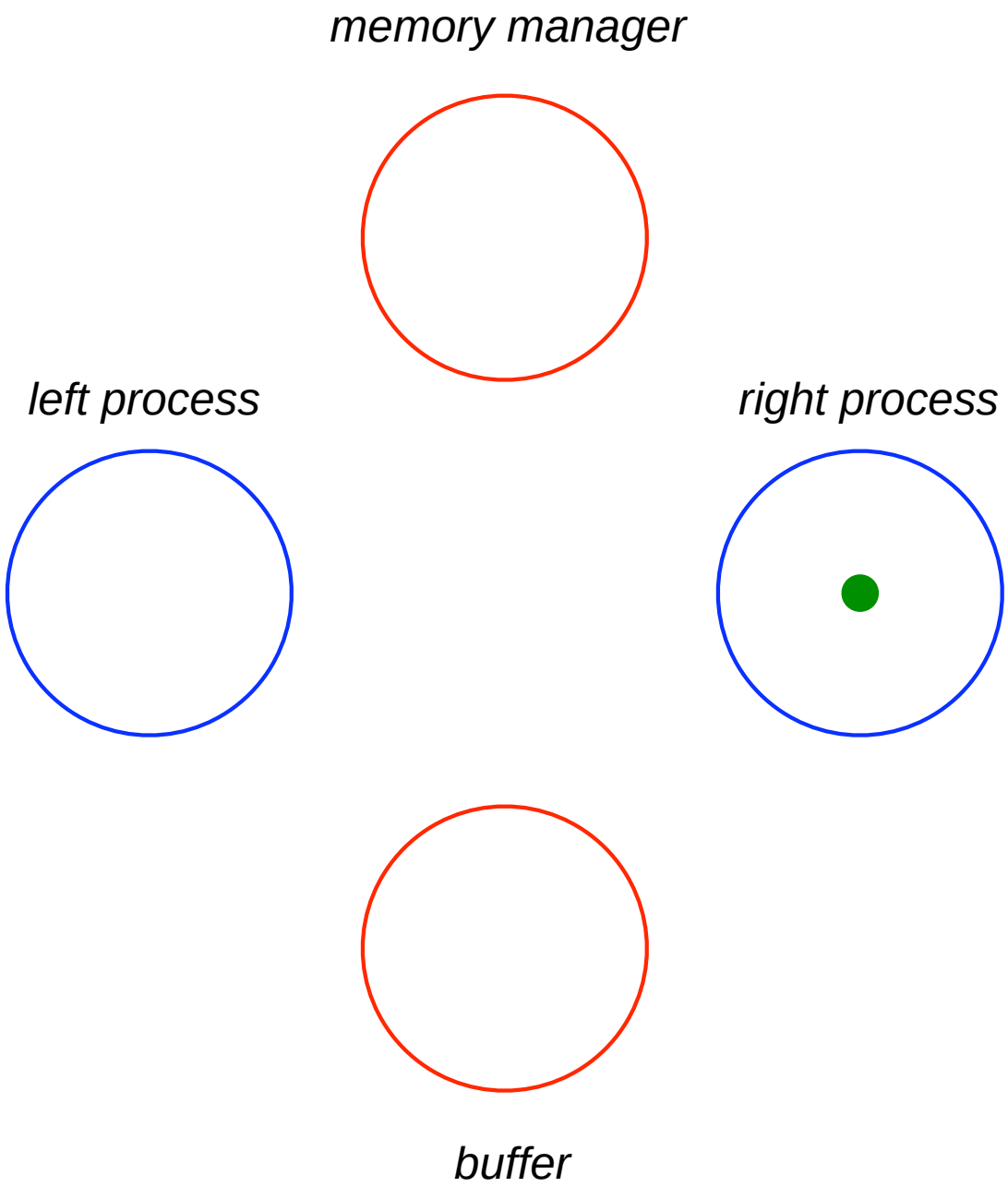
P1 * ··· * Pn

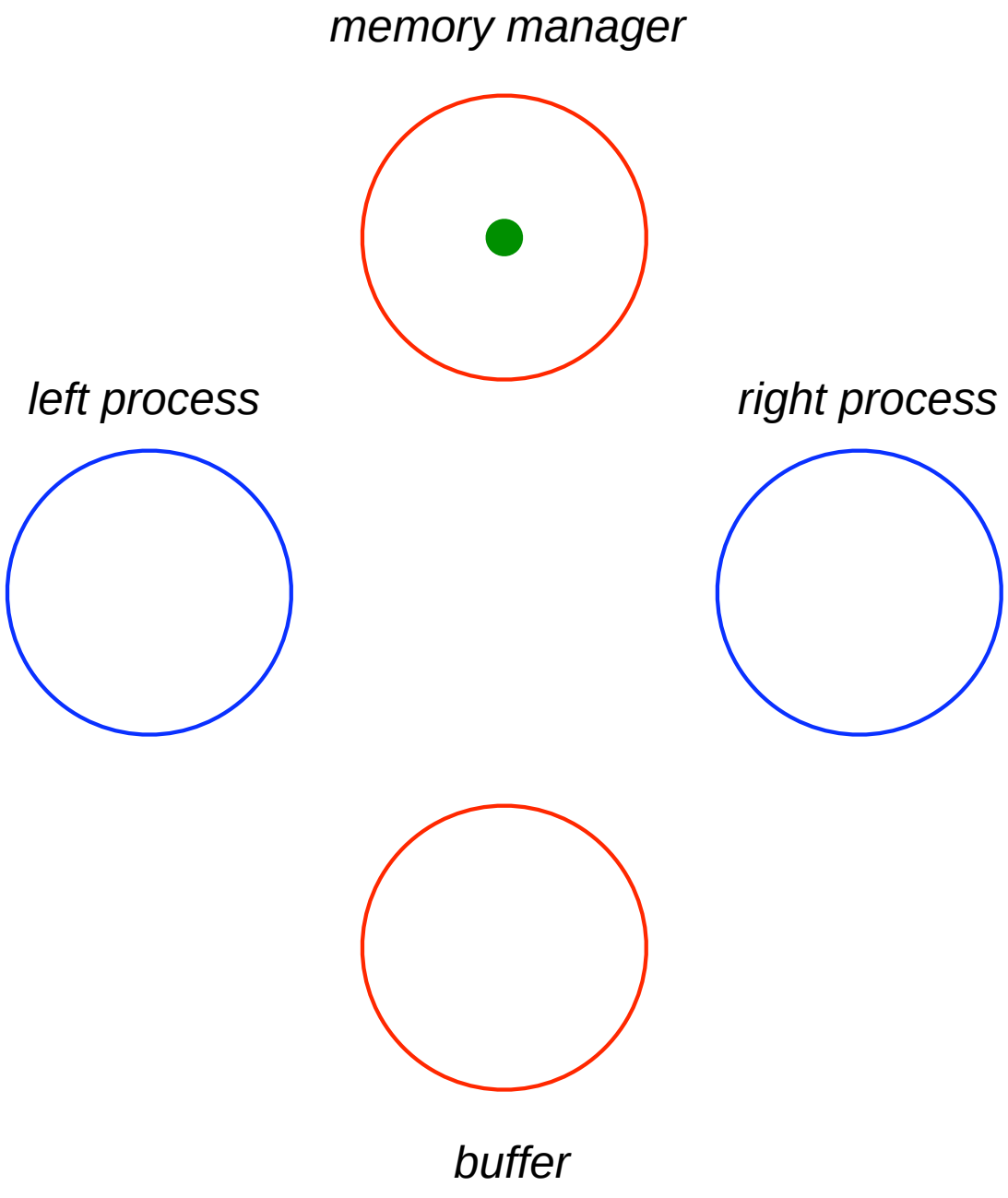



memory manager
left process
right process
buffer


memory manager
left process
right process
buffer


memory manager
left process
right process
buffer


memory manager
left process
right process
buffer


memory manager
left process
right process
buffer

Part VII

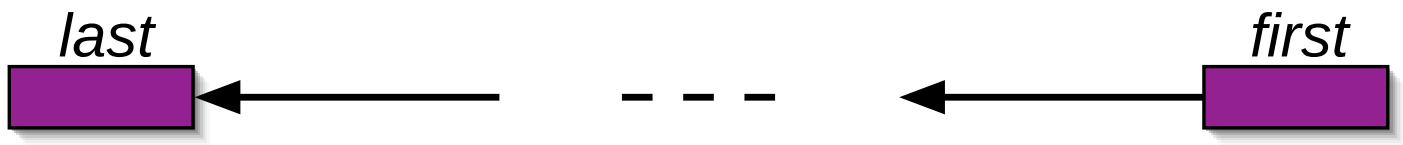
Counting Semaphore Example[5]

[5] From: Cooperating Sequential Processes, Dijkstra, 1968

```
begin integer number of queuing portions;
      number of queuing portions := 0;
      parbegin
      producer:begin
                  again 1: produce the next portion;
                           add portion to buffer;
                           V(number of queuing positions);
                           goto again 1;
                  end;
      consumer: begin
                  again 2: P(number of queuing positions);
                           take portion from buffer;
                           process portion taken;
                           goto again 2;
                  end;
      parend;
end;
```

Add and Take: No Mutexes Needed

We use a non-empty linked list.



The *last* element is a dummy node.

Producer puts message in dummy, then creates new dummy.

Consumer takes *first* element of queue.

Concurrent access is possible.

Relevant Parts of Code

$$
\begin{array}{l}
\texttt{semaphore number:= 0;} \\
\vdots \\
\mathit{produce}(m) \\
\mathit{last}.1 := m; \\
\mathit{last}.2 := \texttt{cons}(-,-); \\
\mathit{last} := \mathit{last}.2 \\
V(\texttt{number}); \\
\vdots
\end{array}
\quad \| \quad
\begin{array}{l}
\vdots \\
P(\texttt{number}); \\
n := \mathit{first}.1; \;\; t := \mathit{first} \\
\mathit{first} := \mathit{first}.2 \\
\texttt{dispose}(t); \\
\mathit{consume}(n); \\
\vdots
\end{array}
$$

Auxiliary Variables

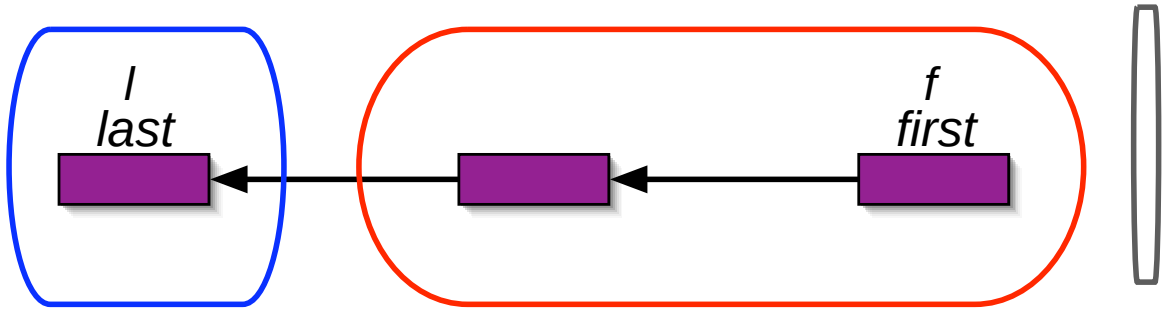
- To describe ownership we need auxiliary variables l and f.
- The semaphore will own a linked list of size `number` from f to l (but not including l).
- We insert auxiliary manipulations into CCRs simulating the semaphores.

$$P'(\texttt{number}) \stackrel{\text{def}}{=} \texttt{with number when number} > 0 \text{ do } f := \mathit{first}.2;\ \texttt{number}--$$

$$V'(\texttt{number}) \stackrel{\text{def}}{=} \texttt{with number when true do } l := \mathit{last};\ \texttt{number}{+}{+}$$

Relevant Parts of Code

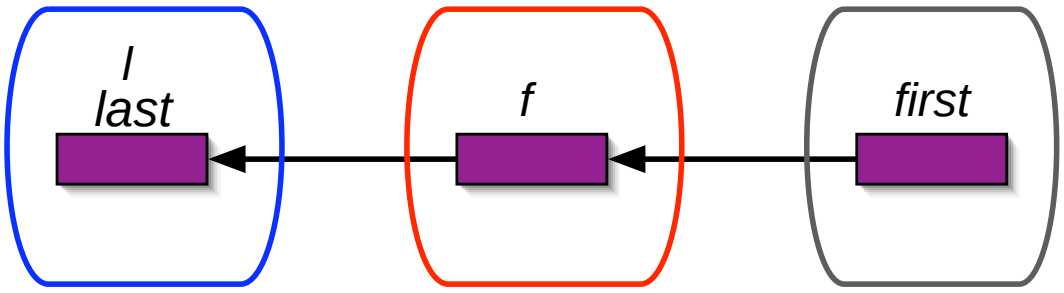
$$
\begin{array}{l}
\texttt{semaphore number} := 0; \\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\mathit{produce}(m) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad P'(\texttt{number}); \\
\mathit{last}.1 := m; \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad n := \mathit{first}.1;\ \ t := \mathit{first} \\
\mathit{last}.2 := \texttt{cons}(-,-); \qquad\qquad \| \qquad\qquad\quad \mathit{first} := \mathit{first}.2 \\
\mathit{last} := \mathit{last}.2 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{dispose}(t); \\
V'(\texttt{number}); \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{consume}(n); \\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots
\end{array}
$$



produce(m) || *P(number)*

producer **semaphore** **consumer**

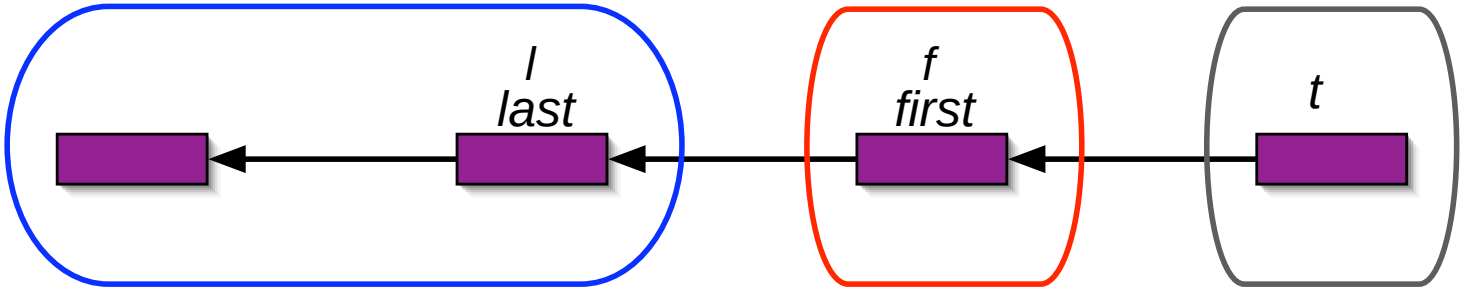
produce(m) || *P(number)*



last.1:=m
last.2:=cons(-,-) ||
n:=first.1; t:=first
first:=first.2

producer **semaphore** **consumer**

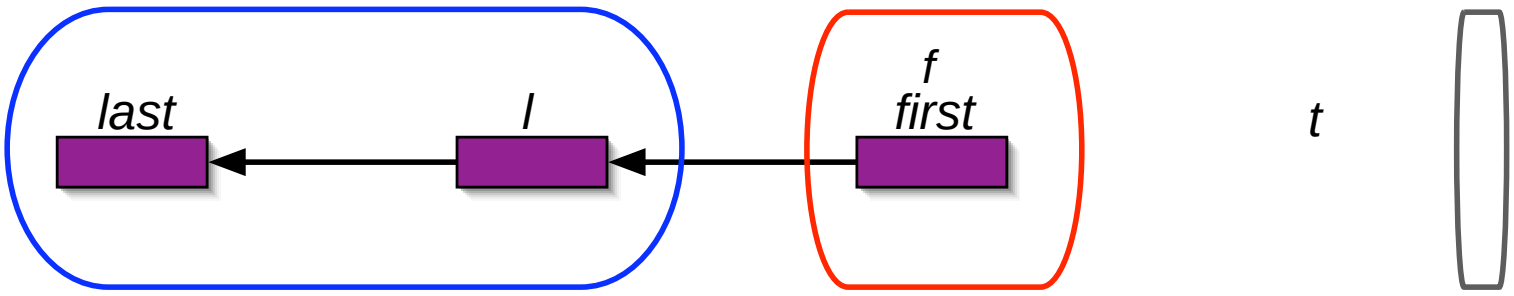
last.1:=m
last.2:=cons(-,-) || *n:=first.1; t:=first*
first:=first.2



last:=last.2 || *dispose(t)*

producer **semaphore** **consumer**

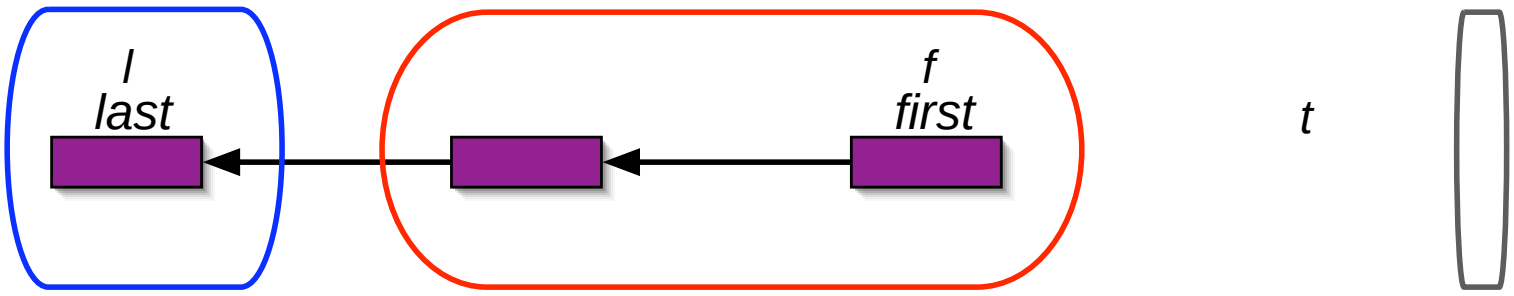
last:=last.2 || *dispose(t)*



V'(number) || *consume(m)*

producer **semaphore** **consumer**

V'(number) || *consume(m)*



producer **semaphore** **consumer**

$\{l = last \wedge last \mapsto -, -\}$
$produce(m)$
$\{l = last \wedge last \mapsto -, -\}$
$last.1 := m;$
$\{l = last \wedge last \mapsto -, -\}$
$last.2 := \mathtt{cons}(-, -);$
$\{l = last \wedge \exists g.(last \mapsto -, g * g \mapsto -, -)\}$
$last.; = last.2$
$\{l \mapsto -, last * last \mapsto -, -)\}$
$V'(\mathtt{number});$
$\{l = last \wedge last \mapsto -, -\}$

$\|$

$\{f = first \wedge \mathtt{emp}\}$
$P'(\mathtt{number});$
$\{first \mapsto -, f\}$
$n := first.1; \;\; t := first$
$\{t = first \wedge t \mapsto -, f\}$
$first := first.2$
$\{f = first \wedge t \mapsto -, f\}$
$\mathtt{dispose}(t);$
$\{f = first \wedge \mathtt{emp}\}$
$consume(n);$
$\{f = first \wedge \mathtt{emp}\}$

$$SI \;=\; \mathtt{ls}[f\ \mathtt{number}\ l]$$

where
$$\mathtt{ls}[x\,n\,z] \overset{\text{def}}{\text{iff}} (x = z \wedge n = 0 \wedge \mathtt{emp}) \vee (x \not= z \wedge n > 0 \wedge \exists y.\, x \mapsto -, y * \mathtt{ls}[y\,(n-1)\,z])$$

No Race

$\{\mathtt{emp}\}$

$\mathtt{semaphore\ number}{:=}\,0;$

$\mathtt{while\ true\ do}$		$\mathtt{while\ true\ do}$
$\quad \mathit{produce}(m)$		$\quad P(\mathtt{number});$
$\quad \mathit{last}.1{:=}\ m;$		$\quad n{:=}\ \mathit{first}.1;\ \ t{:=}\ \mathit{first}$
$\quad \mathit{last}.2{:=}\ \mathtt{cons}(-,-);$	$\parallel$	$\quad \mathit{first}{:=}\ \mathit{first}.2$
$\quad \mathit{last}{:}\ ;{=}\mathit{last}.2$		$\quad \mathtt{dispose}(t);$
$\quad V(\mathtt{number});$		$\quad \mathit{consume}(n);$

$\{\mathtt{false}\}$

Future Directions

- True concurrency model (for proven programs only!)
- Passivity (shared read access)
- Rely/guarantee formalism
- Temporal formalism
- Process calculi (CSP, pi-calculus)
- Mechanized proof
- Model checking, abstract interpretation.