

*Readers & Writers, multicasting, permissions – oh,
just loads of stuff!*

The East London Massive
plus CMU and Korea and Matthew Parkinson

19th April 2004



Be ye never so low, the hardware is beneath you!



Be ye never so low, the hardware is beneath you!

- ▶ digging tunnels



Be ye never so low, the hardware is beneath you!

- ▶ digging tunnels
- ▶ that cave in.



Isn't it a specification issue?



Isn't it a specification issue?

- ▶ Dangerous code, well-specified, can be used naked.



Isn't it a specification issue?

- ▶ Dangerous code, well-specified, can be used naked.
- ▶ Do we need a new programming language?



Isn't it a specification issue?

- ▶ Dangerous code, well-specified, can be used naked.
- ▶ Do we need a new programming language?
- ▶ Can we specify assembler programs?



Isn't it a specification issue?

- ▶ Dangerous code, well-specified, can be used naked.
- ▶ Do we need a new programming language?
- ▶ Can we specify assembler programs?
- ▶ Of course we can!



Isn't it a specification issue?

- ▶ Dangerous code, well-specified, can be used naked.
- ▶ Do we need a new programming language?
- ▶ Can we specify assembler programs?
- ▶ Of course we can!
- ▶ Who needs kernel mode?



Isn't it a specification issue?

- ▶ Dangerous code, well-specified, can be used naked.
- ▶ Do we need a new programming language?
- ▶ Can we specify assembler programs?
- ▶ Of course we can!
- ▶ Who needs kernel mode?
- ▶ Systems with back doors have burglars.



Great formalist victories



Great formalist victories

- ▶ compiling



Great formalist victories

- ▶ compiling
- ▶ types



Great formalist victories

- ▶ compiling
- ▶ types
- ▶ resource accounting (the next battle)



Great formalist victories

- ▶ compiling
- ▶ types
- ▶ resource accounting (the next battle)
- ▶ verification (we won't admit defeat!)



Great formalist victories

- ▶ compiling
- ▶ types
- ▶ resource accounting (the next battle)
- ▶ verification (we won't admit defeat!)
- ▶ refinement (ongoing)



Great formalist victories

- ▶ compiling
- ▶ types
- ▶ resource accounting (the next battle)
- ▶ verification (we won't admit defeat!)
- ▶ refinement (ongoing)

Resource accounting covers power, space, aliasing, side-effects, space leaks, race conditions, non-allocated buffers, double frees, dereferencing nil, etc., etc.



Great formalist victories

- ▶ compiling
- ▶ types
- ▶ resource accounting (the next battle)
- ▶ verification (we won't admit defeat!)
- ▶ refinement (ongoing)

Resource accounting covers power, space, aliasing, side-effects, space leaks, race conditions, non-allocated buffers, double frees, dereferencing nil, etc., etc.

Our work, led by O'Hearn, tracks Dijkstra's work in the 1960s, which showed that partitioning variables between threads and protecting access to shared variables (resources) was the secret of safe concurrency.



Great formalist victories

- ▶ compiling
- ▶ types
- ▶ resource accounting (the next battle)
- ▶ verification (we won't admit defeat!)
- ▶ refinement (ongoing)

Resource accounting covers power, space, aliasing, side-effects, space leaks, race conditions, non-allocated buffers, double frees, dereferencing nil, etc., etc.

Our work, led by O'Hearn, tracks Dijkstra's work in the 1960s, which showed that partitioning variables between threads and protecting access to shared variables (resources) was the secret of safe concurrency.

Ah, but how to do it?



Separation logic

... is a resource logic grafted on to Hoare logic, using integer pointers
(just like real life!)



Separation logic

... is a resource logic grafted on to Hoare logic, using integer pointers
(just like real life!)

\mapsto describes a singleton heap; \star is for separation; \wedge is for alternate descriptions.



Separation logic

... is a resource logic grafted on to Hoare logic, using integer pointers
(just like real life!)

\mapsto describes a singleton heap; \star is for separation; \wedge is for alternate descriptions.

$$(x \mapsto - \star y \mapsto -) \rightarrow x \neq y$$

$$(x \mapsto E \wedge x \mapsto E') \rightarrow E = E'$$



Separation logic

... is a resource logic grafted on to Hoare logic, using integer pointers (just like real life!)

\mapsto describes a singleton heap; \star is for separation; \wedge is for alternate descriptions.

$$(x \mapsto - \star y \mapsto -) \rightarrow x \neq y$$

$$(x \mapsto E \wedge x \mapsto E') \rightarrow E = E'$$

The ‘frame property’ is roughly that if a program doesn’t crash with a particular heap, it won’t crash in a larger heap.



Separation logic

... is a resource logic grafted on to Hoare logic, using integer pointers (just like real life!)

\mapsto describes a singleton heap; \star is for separation; \wedge is for alternate descriptions.

$$(x \mapsto - \star y \mapsto -) \rightarrow x \neq y$$

$$(x \mapsto E \wedge x \mapsto E') \rightarrow E = E'$$

The ‘frame property’ is roughly that if a program doesn’t crash with a particular heap, it won’t crash in a larger heap.

The frame rule:
$$\frac{\{Q\}C\{R\}}{\{P \star Q\}C\{P \star R\}} \quad (\text{modifies } C \cap \text{vars } P = \emptyset)$$



Separation logic

... is a resource logic grafted on to Hoare logic, using integer pointers (just like real life!)

\mapsto describes a singleton heap; \star is for separation; \wedge is for alternate descriptions.

$$(x \mapsto - \star y \mapsto -) \rightarrow x \neq y$$
$$(x \mapsto E \wedge x \mapsto E') \rightarrow E = E'$$

The ‘frame property’ is roughly that if a program doesn’t crash with a particular heap, it won’t crash in a larger heap.

$$\frac{\{Q\}C\{R\}}{\{P \star Q\}C\{P \star R\}} \quad (\text{modifies } C \cap \text{vars } P = \emptyset)$$

The frame rule:

Magic with allocation: $\{\mathbf{emp}\}x := \text{new}()\{x \mapsto -\};$
 $\{x \mapsto -\} \text{dispose } x\{\mathbf{emp}\}.$



Ownership transfer (O'Hearn, 2002)



Ownership transfer (O'Hearn, 2002)

Resource r : Vars $full, b$;

$x := \text{new}();$ with r when $\neg full$ do $b := x;$ $full := \text{true}$ od	$\text{with } r \text{ when } full \text{ do}$ $y := b;$ $full := \text{false}$ od; dispose y
--	---



Ownership transfer (O'Hearn, 2002)

Resource r : Vars $full, b$;

Invariant $(full \wedge b \mapsto _) \vee (\neg full \wedge \mathbf{emp})$

Initially $\neg full$

$x := \mathbf{new}();$

with r when $\neg full$ do

$b := x;$

$full := \mathbf{true}$

od

with r when $full$ do

$y := b;$

$full := \mathbf{false}$

od;

dispose y



Ownership transfer (O'Hearn, 2002)

Resource r : Vars $full, b$;

Invariant $(full \wedge b \mapsto _) \vee (\neg full \wedge \mathbf{emp})$

Initially $\neg full$

<p>$\{\mathbf{emp}\}$ $x := \text{new}();$</p> <p>with r when $\neg full$ do</p> <p style="padding-left: 2em;">$b := x;$</p> <p style="padding-left: 2em;">$full := \text{true}$</p> <p>od</p> <p>$\{\mathbf{emp}\}$</p>	<p>$\{\mathbf{emp}\}$ with r when $full$ do</p> <p style="padding-left: 2em;">$y := b;$</p> <p style="padding-left: 2em;">$full := \text{false}$</p> <p>od;</p> <p>dispose y</p> <p>$\{\mathbf{emp}\}$</p>
---	---



Ownership transfer (O'Hearn, 2002)

Resource r : Vars $full, b$;

Invariant $(full \wedge b \mapsto _) \vee (\neg full \wedge \mathbf{emp})$

Initially $\neg full$

<p>$\{\mathbf{emp}\}$ $x := \mathbf{new}()$; $\{x \mapsto _ \}$ with r when $\neg full$ do</p> <p style="padding-left: 40px;">$b := x$;</p> <p style="padding-left: 40px;">$full := \mathbf{true}$</p> <p>od $\{\mathbf{emp}\}$</p>	<p>$\{\mathbf{emp}\}$ with r when $full$ do</p> <p style="padding-left: 40px;">$y := b$;</p> <p style="padding-left: 40px;">$full := \mathbf{false}$</p> <p>od;</p> <p>dispose y $\{\mathbf{emp}\}$</p>
--	--



Ownership transfer (O'Hearn, 2002)

Resource r : Vars $full, b$;

Invariant $(full \wedge b \mapsto _) \vee (\neg full \wedge \mathbf{emp})$

Initially $\neg full$

<p>$\{\mathbf{emp}\}$ $x := \mathbf{new}();$ $\{x \mapsto _ \}$ with r when $\neg full$ do $\{\neg full \wedge \mathbf{emp} \star x \mapsto _ \}$ $b := x;$ $full := \mathbf{true}$ od $\{\mathbf{emp}\}$</p>	<p>$\{\mathbf{emp}\}$ with r when $full$ do $y := b;$ $full := \mathbf{false}$ od; dispose y $\{\mathbf{emp}\}$</p>
--	--



Ownership transfer (O'Hearn, 2002)

Resource r : Vars $full, b$;

Invariant $(full \wedge b \mapsto _) \vee (\neg full \wedge \mathbf{emp})$

Initially $\neg full$

<p>$\{\mathbf{emp}\}$ $x := \mathbf{new}();$ $\{x \mapsto _ \}$ with r when $\neg full$ do $\{\neg full \wedge \mathbf{emp} \star x \mapsto _ \}$ $b := x;$ $\{\neg full \wedge \mathbf{emp} \star x \mapsto _ \wedge b = x \}$ $full := \mathbf{true}$ od $\{\mathbf{emp}\}$</p>	<p>$\{\mathbf{emp}\}$ with r when $full$ do $y := b;$ $full := \mathbf{false}$ od; dispose y $\{\mathbf{emp}\}$</p>
--	--



Ownership transfer (O'Hearn, 2002)

Resource r : Vars $full, b$;

Invariant $(full \wedge b \mapsto _) \vee (\neg full \wedge \mathbf{emp})$

Initially $\neg full$

$\{ \mathbf{emp} \}$ $x := \mathbf{new}();$ $\{ x \mapsto _ \}$ with r when $\neg full$ do $\{ \neg full \wedge \mathbf{emp} \star x \mapsto _ \}$ $b := x;$ $\{ \neg full \wedge \mathbf{emp} \star x \mapsto _ \wedge b = x \}$ $full := \mathbf{true}$ $\{ full \wedge b \mapsto _ \star \mathbf{emp} \}$ od $\{ \mathbf{emp} \}$	$\{ \mathbf{emp} \}$ with r when $full$ do $y := b;$ $full := \mathbf{false}$ od; dispose y $\{ \mathbf{emp} \}$
--	--



Ownership transfer (O'Hearn, 2002)

Resource r : Vars $full, b$;

Invariant $(full \wedge b \mapsto _) \vee (\neg full \wedge \mathbf{emp})$

Initially $\neg full$

$\{ \mathbf{emp} \}$ $x := \mathbf{new}();$ $\{ x \mapsto _ \}$ with r when $\neg full$ do $\{ \neg full \wedge \mathbf{emp} \star x \mapsto _ \}$ $b := x;$ $\{ \neg full \wedge \mathbf{emp} \star x \mapsto _ \wedge b = x \}$ $full := \mathbf{true}$ $\{ full \wedge b \mapsto _ \star \mathbf{emp} \}$ od $\{ \mathbf{emp} \}$	$\{ \mathbf{emp} \}$ with r when $full$ do $\{ full \wedge b \mapsto _ \star \mathbf{emp} \}$ $y := b;$ $full := \mathbf{false}$ od; dispose y $\{ \mathbf{emp} \}$
--	---



Ownership transfer (O'Hearn, 2002)

Resource r : Vars $full, b$;

Invariant $(full \wedge b \mapsto _) \vee (\neg full \wedge \mathbf{emp})$

Initially $\neg full$

$\{ \mathbf{emp} \}$ $x := \mathbf{new}();$ $\{ x \mapsto _ \}$ with r when $\neg full$ do $\{ \neg full \wedge \mathbf{emp} \star x \mapsto _ \}$ $b := x;$ $\{ \neg full \wedge \mathbf{emp} \star x \mapsto _ \wedge b = x \}$ $full := \mathbf{true}$ $\{ full \wedge b \mapsto _ \star \mathbf{emp} \}$ od $\{ \mathbf{emp} \}$	$\{ \mathbf{emp} \}$ with r when $full$ do $\{ full \wedge b \mapsto _ \star \mathbf{emp} \}$ $y := b;$ $\{ full \wedge b \mapsto _ \star \mathbf{emp} \wedge y = b \}$ $full := \mathbf{false}$ od; dispose y $\{ \mathbf{emp} \}$
--	---



Ownership transfer (O'Hearn, 2002)

Resource r : Vars $full, b$;

Invariant $(full \wedge b \mapsto _) \vee (\neg full \wedge \mathbf{emp})$

Initially $\neg full$

$\{ \mathbf{emp} \}$ $x := \mathbf{new}();$ $\{ x \mapsto _ \}$ with r when $\neg full$ do $\{ \neg full \wedge \mathbf{emp} \star x \mapsto _ \}$ $b := x;$ $\{ \neg full \wedge \mathbf{emp} \star x \mapsto _ \wedge b = x \}$ $full := \mathbf{true}$ $\{ full \wedge b \mapsto _ \star \mathbf{emp} \}$ od $\{ \mathbf{emp} \}$	$\{ \mathbf{emp} \}$ with r when $full$ do $\{ full \wedge b \mapsto _ \star \mathbf{emp} \}$ $y := b;$ $\{ full \wedge b \mapsto _ \star \mathbf{emp} \wedge y = b \}$ $full := \mathbf{false}$ $\{ \neg full \wedge \mathbf{emp} \star y \mapsto _ \}$ od; dispose y $\{ \mathbf{emp} \}$
--	---



Ownership transfer (O'Hearn, 2002)

Resource r : Vars $full, b$;

Invariant $(full \wedge b \mapsto _) \vee (\neg full \wedge \mathbf{emp})$

Initially $\neg full$

$\{ \mathbf{emp} \}$ $x := \mathbf{new}();$ $\{ x \mapsto _ \}$ with r when $\neg full$ do $\{ \neg full \wedge \mathbf{emp} \star x \mapsto _ \}$ $b := x;$ $\{ \neg full \wedge \mathbf{emp} \star x \mapsto _ \wedge b = x \}$ $full := \mathbf{true}$ $\{ full \wedge b \mapsto _ \star \mathbf{emp} \}$ od $\{ \mathbf{emp} \}$	$\{ \mathbf{emp} \}$ with r when $full$ do $\{ full \wedge b \mapsto _ \star \mathbf{emp} \}$ $y := b;$ $\{ full \wedge b \mapsto _ \star \mathbf{emp} \wedge y = b \}$ $full := \mathbf{false}$ $\{ \neg full \wedge \mathbf{emp} \star y \mapsto _ \}$ od; $\{ y \mapsto _ \}$ dispose y $\{ \mathbf{emp} \}$
--	---



Sharing (collective, 2003)



Sharing (collective, 2003)

Brookes' (CMU) model showed that it's safe (no races) to reason with separation logic and ownership transfer – provided variables are partitioned read/write to threads and 'resources' / read shared.



Sharing (collective, 2003)

Brookes' (CMU) model showed that it's safe (no races) to reason with separation logic and ownership transfer – provided variables are partitioned read/write to threads and 'resources' / read shared.

Can we share read-only buffers? (Brookes suggests yes ...)



Sharing (collective, 2003)

Brookes' (CMU) model showed that it's safe (no races) to reason with separation logic and ownership transfer – provided variables are partitioned read/write to threads and 'resources' / read shared.

Can we share read-only buffers? (Brookes suggests yes ...)

This program is ok:

```
x := new(); [x] := 1;
(y := [x] || z := [x] + 1);
dispose x
```



Sharing (collective, 2003)

Brookes' (CMU) model showed that it's safe (no races) to reason with separation logic and ownership transfer – provided variables are partitioned read/write to threads and 'resources' / read shared.

Can we share read-only buffers? (Brookes suggests yes ...)

This program is ok:

$$\begin{array}{l} x := \text{new}(); [x] := 1; \\ (y := [x] \parallel z := [x] + 1); \\ \text{dispose } x \end{array}$$

This program is not:

$$\begin{array}{l} x := \text{new}(); [x] := 1; \\ \left(y := [x]; \parallel [x] := 2; \right); \\ \left(\text{dispose } x \parallel z := [x] + 1 \right); \\ [x] := y + z \end{array}$$


Sharing (collective, 2003)

Brookes' (CMU) model showed that it's safe (no races) to reason with separation logic and ownership transfer – provided variables are partitioned read/write to threads and 'resources' / read shared.

Can we share read-only buffers? (Brookes suggests yes ...)

This program is ok:
$$\begin{array}{l} x := \text{new}(); [x] := 1; \\ (y := [x] \parallel z := [x] + 1); \\ \text{dispose } x \end{array}$$

This program is not:
$$\begin{array}{l} x := \text{new}(); [x] := 1; \\ \left(y := [x]; \parallel [x] := 2; \right); \\ \left(\text{dispose } x \parallel z := [x] + 1 \right); \\ [x] := y + z \end{array}$$

Boyland (Wisconsin) told us that $\frac{1}{2} + \frac{1}{2} = 1$. If 1 means ownership, a fraction means read access (and no writing, no disposing).



Properties of fractional permissions



Properties of fractional permissions

$$E \vdash_z E' \rightarrow 0 < z \leq 1$$

$$E \vdash_{z+z'} E' \wedge z > 0 \wedge z' > 0 \iff E \vdash_z E' \star E \vdash_{z'} E'$$



Properties of fractional permissions

$$E \vdash_z E' \rightarrow 0 < z \leq 1$$

$$E \vdash_{z+z'} E' \wedge z > 0 \wedge z' > 0 \iff E \vdash_z E' \star E \vdash_{z'} E'$$

$\{R_E^x\}$	$x := E$	$\{R\}$	
$\{E' \vdash_1 -\}$	$[E'] := E$	$\{E' \vdash_1 E\}$	
$\{E' \vdash_z E\}$	$x := [E']$	$\{E' \vdash_z E \wedge x = E\}$	(x not free in E, E')
$\{\mathbf{emp}\}$	$x := \mathbf{new}(E)$	$\{x \vdash_1 E\}$	
$\{E \vdash_1 -\}$	$\mathbf{dispose } E$	$\{\mathbf{emp}\}$	



A proof with fractional permissions

{emp}

$x := \text{new}();$

$[x] := 1;$

$\left(\begin{array}{c} y := [x] \\ \parallel \\ z := [x] + 1 \end{array} \right);$

dispose x

{emp $\wedge y = 1 \wedge z = 2$ }



A proof with fractional permissions

$\{\mathbf{emp}\}$

$x := \mathbf{new}();$

$\{x \mapsto_{\frac{1}{1}} -\}$

$[x] := 1;$

$\left(\begin{array}{c} y := [x] \\ \parallel \\ z := [x] + 1 \end{array} \right);$

dispose x

$\{\mathbf{emp} \wedge y = 1 \wedge z = 2\}$



A proof with fractional permissions

$\{\mathbf{emp}\}$

$x := \mathbf{new}();$

$\{x \mapsto -\}$

$[x] := 1;$

$\{x \mapsto 1\}$

$\left(\begin{array}{c} y := [x] \\ z := [x] + 1 \end{array} \right);$

dispose x

$\{\mathbf{emp} \wedge y = 1 \wedge z = 2\}$



A proof with fractional permissions

$\{\mathbf{emp}\}$

$x := \mathbf{new}();$

$\{x \mapsto -\}$

$[x] := 1;$

$\{x \mapsto 1\} \cdot \{x \xrightarrow{0.5} 1 \star x \xrightarrow{0.5} 1\}$

$\left(\begin{array}{c} y := [x] \\ \parallel \\ z := [x] + 1 \end{array} \right);$

dispose x

$\{\mathbf{emp} \wedge y = 1 \wedge z = 2\}$



A proof with fractional permissions

$$\begin{array}{l} \{\mathbf{emp}\} \\ x := \mathbf{new}(); \\ \left\{ x \overset{1}{\mapsto} - \right\} \\ [x] := 1; \\ \left\{ x \overset{1}{\mapsto} 1 \right\} \cdot \left\{ x \overset{0.5}{\mapsto} 1 \star x \overset{0.5}{\mapsto} 1 \right\} \\ \left(\begin{array}{c} \left\{ x \overset{0.5}{\mapsto} 1 \right\} \\ y := [x] \end{array} \parallel \begin{array}{c} \left\{ x \overset{0.5}{\mapsto} 1 \right\} \\ z := [x] + 1 \end{array} \right); \end{array}$$

dispose x

$$\{\mathbf{emp} \wedge y = 1 \wedge z = 2\}$$



A proof with fractional permissions

$\{\mathbf{emp}\}$
 $x := \mathbf{new}();$
 $\{x \mapsto_{1} -\}$
 $[x] := 1;$
 $\{x \mapsto_{1} 1\} \cdot \{x \mapsto_{0.5} 1 \star x \mapsto_{0.5} 1\}$
 $\left(\begin{array}{c} \{x \mapsto_{0.5} 1\} \\ y := [x] \\ \{x \mapsto_{0.5} 1 \wedge y = 1\} \end{array} \parallel \begin{array}{c} \{x \mapsto_{0.5} 1\} \\ z := [x] + 1 \end{array} \right);$

dispose x

$\{\mathbf{emp} \wedge y = 1 \wedge z = 2\}$



A proof with fractional permissions

$\{\mathbf{emp}\}$

$x := \mathbf{new}();$

$\{x \mapsto_{1} -\}$

$[x] := 1;$

$\{x \mapsto_{1} 1\} \cdot \{x \mapsto_{0.5} 1 \star x \mapsto_{0.5} 1\}$

$\left(\begin{array}{c} \{x \mapsto_{0.5} 1\} \\ y := [x] \\ \{x \mapsto_{0.5} 1 \wedge y = 1\} \end{array} \parallel \begin{array}{c} \{x \mapsto_{0.5} 1\} \\ z := [x] + 1 \\ \{x \mapsto_{0.5} 1 \wedge z = 2\} \end{array} \right);$

dispose x

$\{\mathbf{emp} \wedge y = 1 \wedge z = 2\}$



A proof with fractional permissions

$\{\mathbf{emp}\}$
 $x := \mathbf{new}();$
 $\{x \mapsto_{1} -\}$
 $[x] := 1;$
 $\{x \mapsto_{1} 1\} \cdot \{x \mapsto_{0.5} 1 \star x \mapsto_{0.5} 1\}$
 $\left(\begin{array}{c} \{x \mapsto_{0.5} 1\} \\ y := [x] \\ \{x \mapsto_{0.5} 1 \wedge y = 1\} \end{array} \parallel \begin{array}{c} \{x \mapsto_{0.5} 1\} \\ z := [x] + 1 \\ \{x \mapsto_{0.5} 1 \wedge z = 2\} \end{array} \right);$
 $\{(x \mapsto_{0.5} 1 \wedge y = 1) \star (x \mapsto_{0.5} 1 \wedge z = 2)\}$
 $\mathbf{dispose } x$
 $\{\mathbf{emp} \wedge y = 1 \wedge z = 2\}$



A proof with fractional permissions

$\{\mathbf{emp}\}$
 $x := \mathbf{new}();$
 $\{x \mapsto_{\frac{1}{1}} -\}$
 $[x] := 1;$
 $\{x \mapsto_{\frac{1}{1}} 1\} \cdot \{x \mapsto_{\frac{0.5}{0.5}} 1 \star x \mapsto_{\frac{0.5}{0.5}} 1\}$
 $\left(\begin{array}{c} \{x \mapsto_{\frac{0.5}{0.5}} 1\} \\ y := [x] \\ \{x \mapsto_{\frac{0.5}{0.5}} 1 \wedge y = 1\} \end{array} \parallel \begin{array}{c} \{x \mapsto_{\frac{0.5}{0.5}} 1\} \\ z := [x] + 1 \\ \{x \mapsto_{\frac{0.5}{0.5}} 1 \wedge z = 2\} \end{array} \right);$
 $\{(x \mapsto_{\frac{0.5}{0.5}} 1 \wedge y = 1) \star (x \mapsto_{\frac{0.5}{0.5}} 1 \wedge z = 2)\} \cdot \{x \mapsto_{\frac{1}{1}} 1 \wedge y = 1 \wedge z = 2\}$
 $\mathbf{dispose } x$
 $\{\mathbf{emp} \wedge y = 1 \wedge z = 2\}$



Multicasting – a problem



Multicasting – a problem

Intel's IXP uses pipeline processing, with queues between components.



Multicasting – a problem

Intel's IXP uses pipeline processing, with queues between components.

Singlecast is no problem:

1. a reader thread assembles a packet into a buffer;
2. transfers ownership into a queue for further processing, and forgets about it;
3. intermediate threads get ownership from one queue and put it in another;
4. a final writer thread transmits the packet and disposes the buffer.



Multicasting – a problem

Intel's IXP uses pipeline processing, with queues between components.

Singlecast is no problem:

1. a reader thread assembles a packet into a buffer;
2. transfers ownership into a queue for further processing, and forgets about it;
3. intermediate threads get ownership from one queue and put it in another;
4. a final writer thread transmits the packet and disposes the buffer.

We'd like end-of-the-line disposes in multicasting too, where a single buffer is shared between multiple pipelines. But with fractional permissions??



Multicasting – a problem

Intel's IXP uses pipeline processing, with queues between components.

Singlecast is no problem:

1. a reader thread assembles a packet into a buffer;
2. transfers ownership into a queue for further processing, and forgets about it;
3. intermediate threads get ownership from one queue and put it in another;
4. a final writer thread transmits the packet and disposes the buffer.

We'd like end-of-the-line disposes in multicasting too, where a single buffer is shared between multiple pipelines. But with fractional permissions??

The standard programming technique is *permission counting* (not reference counting!). But with fractional permissions??



Semaphores as resource-holders (O'Hearn 2004)



Semaphores as resource-holders (O'Hearn 2004)

- ▶ Building on the ownership idea, we give semaphores invariants.



Semaphores as resource-holders (O'Hearn 2004)

- ▶ Building on the ownership idea, we give semaphores invariants.
- ▶ Executing P usually releases resources into the program;



Semaphores as resource-holders (O'Hearn 2004)

- ▶ Building on the ownership idea, we give semaphores invariants.
- ▶ Executing P usually releases resources into the program;
- ▶ executing V usually takes the resources back;



Semaphores as resource-holders (O'Hearn 2004)

- ▶ Building on the ownership idea, we give semaphores invariants.
- ▶ Executing P usually releases resources into the program;
- ▶ executing V usually takes the resources back;
- ▶ the quantity of resource can depend on the semaphore value.



Semaphores as resource-holders (O'Hearn 2004)

- ▶ Building on the ownership idea, we give semaphores invariants.
- ▶ Executing P usually releases resources into the program;
- ▶ executing V usually takes the resources back;
- ▶ the quantity of resource can depend on the semaphore value.
- ▶ In place of locks and exclusion we have permission!



A permission-counting program

```
P(read);  
  count++;  
  if count = 1 then P(write);  
V(read);
```

... reading happens here ...;

```
P(read);  
  count --;  
  if count = 0 then V(write);  
V(read)
```

```
P(write);
```

... writing happens here ...

```
V(write)
```



Counting permissions



Counting permissions

- ▶ We have a “block” (which counts) and “chips” (which give read-only access).



Counting permissions

- ▶ We have a “block” (which counts) and “chips” (which give read-only access).

$$E \vdash^n E' \rightarrow n \geq 0$$

$$E \vdash^n E' \iff E \vdash^{n+1} E' \star E \mapsto E'$$



Counting permissions

- ▶ We have a “block” (which counts) and “chips” (which give read-only access).

$$E \vdash^n E' \rightarrow n \geq 0$$

$$E \vdash^n E' \iff E \vdash^{n+1} E' \star E \mapsto E'$$

- ▶ The block gives read access, plus write/discard if $n = 0$.



Counting permissions

- ▶ We have a “block” (which counts) and “chips” (which give read-only access).

$$E \vdash^n E' \rightarrow n \geq 0$$

$$E \vdash^n E' \iff E \vdash^{n+1} E' \star E \mapsto E'$$

- ▶ The block gives read access, plus write/dispose if $n = 0$.

$$\begin{array}{ll}
 \{R_E^x\} & x := E \quad \{R\} \\
 \{E' \xrightarrow{0} _ \} & [x] := E \quad \{E' \xrightarrow{0} E\} \\
 \{E' \mapsto E\} & x := [E'] \quad \{E' \mapsto E \wedge x = E\} \text{ (} x \text{ not free in } E, E') \\
 \{\mathbf{emp}\} & x := \mathbf{new}(E) \quad \{x \xrightarrow{0} E\} \\
 \{E \xrightarrow{0} _ \} & \mathbf{dispose } E \quad \{\mathbf{emp}\}
 \end{array}$$



A proof: resource safety of readers and writers

$write : z \xrightarrow{0} _$

$read : (count) \text{ if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \xrightarrow{count} _$

$\{\mathbf{emp}\}$

$P(read);$

$count+ := 1;$

$\text{if } count = 1 \text{ then}$

$P(write)$

else

$;$

$V(read);$

$\{z \mapsto _ \}$



A proof: resource safety of readers and writers

$write : z \xrightarrow{0} _$

$read : (count) \text{ if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \xrightarrow{count} _$

$\{\mathbf{emp}\}$

$P(read);$

$\left\{ \text{if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \xrightarrow{count} _ \right\}$

$count+ := 1;$

$\text{if } count = 1 \text{ then}$

$P(write)$

else

$;$

$V(read);$

$\{z \mapsto _ \}$



A proof: resource safety of readers and writers

$write : z \xrightarrow{0} _$

$read : (count) \text{ if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \xrightarrow{count} _$

$\{\mathbf{emp}\}$

$P(read);$

$\left\{ \text{if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \xrightarrow{count} _ \right\}$

$count+ := 1;$

$\left\{ \text{if } count - 1 = 0 \text{ then } \mathbf{emp} \text{ else } z \xrightarrow{count-1} _ \right\}$

$\text{if } count = 1 \text{ then } \quad P(write)$

else

$;$

$V(read);$

$\{z \mapsto _ \}$



A proof: resource safety of readers and writers

$write : z \xrightarrow{0} _$

$read : (count) \text{ if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \xrightarrow{count} _$

$\{\mathbf{emp}\}$

$P(read);$

$\left\{ \text{if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \xrightarrow{count} _ \right\}$

$count+ := 1;$

$\left\{ \text{if } count - 1 = 0 \text{ then } \mathbf{emp} \text{ else } z \xrightarrow{count-1} _ \right\}$

$\text{if } count = 1 \text{ then } \{\mathbf{emp}\} P(write)$

else

;

$V(read);$

$\{z \mapsto _ \}$



A proof: resource safety of readers and writers

$write : z \xrightarrow{0} _$

$read : (count) \text{ if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \xrightarrow{count} _$

$\{\mathbf{emp}\}$

$P(read);$

$\left\{ \text{if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \xrightarrow{count} _ \right\}$

$count+ := 1;$

$\left\{ \text{if } count - 1 = 0 \text{ then } \mathbf{emp} \text{ else } z \xrightarrow{count-1} _ \right\}$

$\text{if } count = 1 \text{ then } \{\mathbf{emp}\} P(write) \left\{ z \xrightarrow{0} _ \right\}$

else

;

$V(read);$

$\{z \mapsto _ \}$



A proof: resource safety of readers and writers

$write : z \xrightarrow{0} _$

$read : (count) \text{ if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \xrightarrow{count} _$

$\{\mathbf{emp}\}$

$P(read);$

$\left\{ \text{if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \xrightarrow{count} _ \right\}$

$count+ := 1;$

$\left\{ \text{if } count - 1 = 0 \text{ then } \mathbf{emp} \text{ else } z \xrightarrow{count-1} _ \right\}$

$\text{if } count = 1 \text{ then } \{\mathbf{emp}\} P(write) \left\{ z \xrightarrow{0} _ \right\}$
 $\text{else } \left\{ z \xrightarrow{count-1} _ \right\};$

$V(read);$

$\{z \mapsto _ \}$



A proof: resource safety of readers and writers

$write : z \xrightarrow{0} -$

$read : (count) \text{ if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \xrightarrow{count} -$

$\{\mathbf{emp}\}$

$P(read);$

$\left\{ \text{if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \xrightarrow{count} - \right\}$

$count+ := 1;$

$\left\{ \text{if } count - 1 = 0 \text{ then } \mathbf{emp} \text{ else } z \xrightarrow{count-1} - \right\}$

$\text{if } count = 1 \text{ then } \{\mathbf{emp}\} P(write) \left\{ z \xrightarrow{0} - \right\}$
else $\left\{ z \xrightarrow{count-1} - \right\};$

$\left\{ z \xrightarrow{count-1} - \right\}$

$V(read);$

$\{z \mapsto -\}$



A proof: resource safety of readers and writers

$write : z \xrightarrow{0} -$

$read : (count) \text{ if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \xrightarrow{count} -$

$\{\mathbf{emp}\}$

$P(read);$

$\left\{ \text{if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \xrightarrow{count} - \right\}$

$count+ := 1;$

$\left\{ \text{if } count - 1 = 0 \text{ then } \mathbf{emp} \text{ else } z \xrightarrow{count-1} - \right\}$

$\text{if } count = 1 \text{ then } \{\mathbf{emp}\} P(write) \left\{ z \xrightarrow{0} - \right\}$
else $\left\{ z \xrightarrow{count-1} - \right\};$

$\left\{ z \xrightarrow{count-1} - \right\} \therefore \left\{ z \xrightarrow{count} - \star z \mapsto - \right\}$

$V(read);$

$\{z \mapsto -\}$



Multicast pipeline stage 1

```
while true do
   $x := \text{new}()$ ;
   $\text{fill}(x)$ ;  $i := 0$ ;  $\text{lim} := \text{addresscount}(x)$ ;
  while  $i < \text{lim}$  do
    if  $i + 1 \neq \text{lim}$  then  $\text{split } x$  fi;
     $\text{enqueue}(x, \text{address}_i(x))$ ;  $i++$ 
  od
od
```



Multicast pipeline stage 1

```
while true do
  x := new();
  fill(x); i := 0; lim := addresscount(x);
  while i < lim do
    if i + 1 ≠ lim then split x fi;
    enqueue(x, addressi(x)); i++
  od
od
```

- ▶ new allocates a buffer and, just next to it, a semaphore with value 1 and invariant if $n = 1$ then **emp** else $x \xrightarrow{n} _$ fi.



Multicast pipeline stage 1

```
while true do
  x := new();
  fill(x); i := 0; lim := addresscount(x);
  while i < lim do
    if i + 1 ≠ lim then split x fi;
    enqueue(x, addressi(x)); i++
  od
od
```

- ▶ new allocates a buffer and, just next to it, a semaphore with value 1 and invariant if $n = 1$ then **emp** else $x \xrightarrow{n} _$ fi.
- ▶ ‘split’ is actually $V(\&(x - 1))$ (cf. readers and writers ‘read’ semaphore).



Multicast end-of-pipeline

```
while true do  
  x := dequeue();  
  empty(x); free(x)  
od
```



Multicast end-of-pipeline

```
while true do
  x := dequeue();
  empty(x); free(x)
od
```

- ▶ ‘free’ is actually $P(\&(x - 1))$; the corresponding critical region is $n - -$; if $n = 0$ then $\text{dispose}(x - 1, \text{bufsiz} + 1)$ fi.



Multicast end-of-pipeline

```
while true do
  x := dequeue();
  empty(x); free(x)
od
```

- ▶ ‘free’ is actually $P(\&(x - 1))$; the corresponding critical region is $n - -$; if $n = 0$ then $\text{dispose}(x - 1, \text{bufsiz} + 1)$ fi.
- ▶ (The dispose gets rid of the semaphore as well.)



Multicast end-of-pipeline

```
while true do
  x := dequeue();
  empty(x); free(x)
od
```

- ▶ ‘free’ is actually $P(\&(x - 1))$; the corresponding critical region is $n - -$; if $n = 0$ then $\text{dispose}(x - 1, \text{bufsiz} + 1)$ fi.
- ▶ (The dispose gets rid of the semaphore as well.)
- ▶ You can’t write the precondition of ‘free’ without mentioning n . Maybe you can’t write it at all without a race condition ... The postcondition is definitely **emp**, though.



Disclaimer

This work is already over six weeks old. Things move fast!

Matthew Parkinson has already pointed out that we didn't know how to handle shared recursive data structures (and provided a correction).

There are models for fractional and counting permissions; there are moves (Parkinson again) to unify them.

Watch this space!

