Concurrent programming in operating systems

Richard Bornat
Professor of Computer Programming
(scouting for talent)

4th February 2004

Slogans

Slogans

- Call me Richard.

Slogans

- Call me Richard.
- Say “Slow down!”.

Slogans

- Call me Richard.
- Say “Slow down!”.
- Try “Shut up and listen to me!”

Programming versus Operating Systems

Programming versus Operating Systems

- Operating Systems are computer programs.

Programming versus Operating Systems

- Operating Systems are computer programs.
- The job of an OS is Resource Management.

Programming versus Operating Systems

- Operating Systems are computer programs.
- The job of an OS is Resource Management.
- Safety and security are resource management problems.

Programming versus Operating Systems

- Operating Systems are computer programs.
- The job of an OS is Resource Management.
- Safety and security are resource management problems.
- So are fairness, liveness, sharing, race-conditions.

Programming versus Operating Systems

- Operating Systems are computer programs.
- The job of an OS is Resource Management.
- Safety and security are resource management problems.
- So are fairness, liveness, sharing, race-conditions.
- RM problems are also programming problems, even in everyday programming with pointers.

Programming versus Operating Systems

- Operating Systems are computer programs.
- The job of an OS is Resource Management.
- Safety and security are resource management problems.
- So are fairness, liveness, sharing, race-conditions.
- RM problems are also programming problems, even in everyday programming with pointers.
- My research (and one day, perhaps yours too) is in resource logics applied to programming problems.

A simple program

Middlesex University

A simple program

- Here's a fragment of a C/Java program:

```
x = 0; y = x;
if (y==0) print("yes");
else print("no");
```

A simple program

- Here’s a fragment of a C/Java program:

```
x = 0; y = x;
if (y==0) print("yes");
else print("no");
```

- Could this program *ever* print “no”?

A simple program

- Here’s a fragment of a C/Java program:

```
x = 0; y = x;
if (y==0) print("yes");
else print("no");
```

- Could this program *ever* print “no”?
- What could go wrong?

A simple program

- Here's a fragment of a C/Java program:

```
x = 0; y = x;
if (y==0) print("yes");  ||  x = 3;  ||  y = 7;
else print("no");
```

- Could this program *ever* print "no"?
- What could go wrong?

A simple program

- Here's a fragment of a C/Java program:

```
x = 0; y = x;                        ║         ║
if (y==0) print("yes");              ║ x = 3;  ║ y = 7;
else print("no");                    ║         ║
```

- Could this program *ever* print "no"?
- What could go wrong?
- Whoops! processes/threads can have *races* (and sometimes your horse loses).

An aside about caches

An aside about caches

- The word “cache” comes from French backwoodsmen in North America.

An aside about caches

- The word “cache” comes from French backwoodsmen in North America.
- In computing a “cache” goes between small, fast, expensive X and large, slow, cheap Y.

An aside about caches

Middlesex University

- The word “cache” comes from French backwoodsmen in North America.
- In computing a “cache” goes between small, fast, expensive X and large, slow, cheap Y.
- A cache translates a **key** k into a **value** V, if possible without asking Y.

An aside about caches

- The word “cache” comes from French backwoodsmen in North America.
- In computing a “cache” goes between small, fast, expensive X and large, slow, cheap Y.
- A cache translates a **key** k into a **value** V, if possible without asking Y.
- Caches in programs avoid:

An aside about caches

- The word “cache” comes from French backwoodsmen in North America.
- In computing a “cache” goes between small, fast, expensive X and large, slow, cheap Y.
- A cache translates a **key** k into a **value** V, if possible without asking Y.
- Caches in programs avoid:
  - computations (between call and procedure);

An aside about caches

Middlesex University

- The word “cache” comes from French backwoodsmen in North America.
- In computing a “cache” goes between small, fast, expensive X and large, slow, cheap Y.
- A cache translates a **key** k into a **value** V, if possible without asking Y.
- Caches in programs avoid:
  - computations (between call and procedure);
  - memory accesses (between registers and memory);

An aside about caches

- The word “cache” comes from French backwoodsmen in North America.
- In computing a “cache” goes between small, fast, expensive X and large, slow, cheap Y.
- A cache translates a **key** k into a **value** V, if possible without asking Y.
- Caches in programs avoid:
    - computations (between call and procedure);
    - memory accesses (between registers and memory);
    - disc accesses (between memory and disc);

An aside about caches

- The word “cache” comes from French backwoodsmen in North America.
- In computing a “cache” goes between small, fast, expensive X and large, slow, cheap Y.
- A cache translates a **key** k into a **value** V, if possible without asking Y.
- Caches in programs avoid:
  - computations (between call and procedure);
  - memory accesses (between registers and memory);
  - disc accesses (between memory and disc);
  - network accesses ...

An aside about caches

- The word “cache” comes from French backwoodsmen in North America.
- In computing a “cache” goes between small, fast, expensive X and large, slow, cheap Y.
- A cache translates a **key** k into a **value** V, if possible without asking Y.
- Caches in programs avoid:
  - computations (between call and procedure);
  - memory accesses (between registers and memory);
  - disc accesses (between memory and disc);
  - network accesses ...
  - etc., etc., etc.

Anatomy of a cache

Anatomy of a cache

- All caches have the same parts.

Anatomy of a cache

- All caches have the same parts.
- A collection of **buffers** which hold previously-discovered key/value pairs.

Anatomy of a cache

- All caches have the same parts.
- A collection of **buffers** which hold previously-discovered key/value pairs.
- A fast **lookup table** which takes k and points to a buffer containing k/V, if there is one.

Anatomy of a cache

- All caches have the same parts.
- A collection of **buffers** which hold previously-discovered key/value pairs.
- A fast **lookup table** which takes k and points to a buffer containing k/V, if there is one.
- (Slow lookup is ok if the cache is very, very small – less than 6 items.)

Anatomy of a cache

- All caches have the same parts.
- A collection of **buffers** which hold previously-discovered key/value pairs.
- A fast **lookup table** which takes *k* and points to a buffer containing *k*/*V*, if there is one.
- (Slow lookup is ok if the cache is very, very small – less than 6 items.)
-
```
ptr = cache_lookup(k);
if (ptr==NULL) {
  ptr = getbuffer(); cache_forget(ptr.key);
  ptr.key = k; ptr.value = Y(k);
  cache_remember(k, ptr);
}
return ptr.value;
```

A Deadlock horror story

A Deadlock horror story

- 1977; Early Unix; multitasking; small machine; max $\sim$ 6 users; max $\sim$ 50 processes.

A Deadlock horror story

- 1977; Early Unix; multitasking; small machine; max $\sim$ 6 users; max $\sim$ 50 processes.
- Small block cache ($\sim$ 10 buffers), used by disc.

A Deadlock horror story

- 1977; Early Unix; multitasking; small machine; max $\sim$ 6 users; max $\sim$ 50 processes.
- Small block cache ($\sim$ 10 buffers), used by disc.
- – also used by block-addressable magnetic tape.

A Deadlock horror story

- 1977; Early Unix; multitasking; small machine; max $\sim$ 6 users; max $\sim$ 50 processes.
- Small block cache ($\sim$ 10 buffers), used by disc.
- – also used by block-addressable magnetic tape.
- Under heavy use, when mag tape was in use, machine “froze” quite often.

A Deadlock horror story

Middlesex University

- 1977; Early Unix; multitasking; small machine; max $\sim$ 6 users; max $\sim$ 50 processes.
- Small block cache ($\sim$ 10 buffers), used by disc.
- – also used by block-addressable magnetic tape.
- Under heavy use, when mag tape was in use, machine “froze” quite often.
- After a while, we guessed the problem was in the block cache.

A Deadlock horror story

- 1977; Early Unix; multitasking; small machine; max $\sim$ 6 users; max $\sim$ 50 processes.
- Small block cache ($\sim$ 10 buffers), used by disc.
- – also used by block-addressable magnetic tape.
- Under heavy use, when mag tape was in use, machine "froze" quite often.
- After a while, we guessed the problem was in the block cache.
- Luckily, we had the Unix source ...

Some background

Some background

- This is a block cache in a multi-process, multi-device system (no lookup table, for simplicity):

Some background

- This is a block cache in a multi-process, multi-device system (no lookup table, for simplicity):

```
while (true) {
  ptr = find_buffer(dev, b);
  if (ptr==NULL) {
    ptr = getbuffer();
    ptr.device = dev; ptr.block = b;
    start_read(dev, ptr, b);
  } else if (ptr.lock==0)
    return ptr;
  wait(ptr);
}
```

Some background

- This is a block cache in a multi-process, multi-device system (no lookup table, for simplicity):

```
while (true) {
  ptr = find_buffer(dev, b);
  if (ptr==NULL) {
    ptr = getbuffer();
    ptr.device = dev; ptr.block = b;
    start_read(dev, ptr, b);
  } else if (ptr.lock==0)
    return ptr;
  wait(ptr);
}
```

- If there are no free (lock==0) buffers, getbuffer waits.

A block cache with pre-fetch

Middlesex University

```
while (true) {
  ptr = find_buffer(dev, b);
  if (ptr==NULL) {
    ptr = getbuffer(); ...
    start_read(dev, ptr, b);
    if (find_buffer(dev,b+1)==NULL) {
      ptr2 = getbuffer(); ...
      start_read(dev, ptr2, b+1);
    }
  } else if (ptr.lock==0)
    return ptr;
  wait(ptr);
}
```

Abstraction makes things easier to see!

Abstraction makes things easier to see!

- A table ...

Abstraction makes things easier to see!

- A table ...
- Five chairs ...

Abstraction makes things easier to see!

Middlesex University

- A table ...
- Five chairs ...
- A big bowl of spaghetti ...

Abstraction makes things easier to see!

- A table ...
- Five chairs ...
- A big bowl of spaghetti ...
- Five forks ...

Abstraction makes things easier to see!

- A table ...
- Five chairs ...
- A big bowl of spaghetti ...
- Five forks ...
- Five hungry people!

Abstraction makes things easier to see!

- A table ...
- Five chairs ...
- A big bowl of spaghetti ...
- Five forks ...
- Five hungry people!
- The spaghetti is slippery; you need two forks to eat it.

Abstraction makes things easier to see!

- A table ...
- Five chairs ...
- A big bowl of spaghetti ...
- Five forks ...
- Five hungry people!
- The spaghetti is slippery; you need two forks to eat it.
- Everybody sits down together;

Abstraction makes things easier to see!

Middlesex University

- A table ...
- Five chairs ...
- A big bowl of spaghetti ...
- Five forks ...
- Five hungry people!
- The spaghetti is slippery; you need two forks to eat it.
- Everybody sits down together;
- everybody reaches for a fork;

Abstraction makes things easier to see!

- A table ...
- Five chairs ...
- A big bowl of spaghetti ...
- Five forks ...
- Five hungry people!
- The spaghetti is slippery; you need two forks to eat it.
- Everybody sits down together;
- everybody reaches for a fork;
- and then for a second fork;

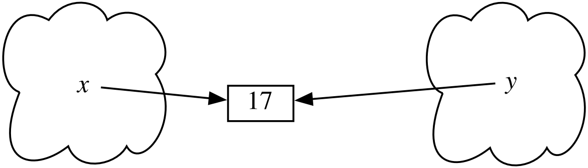
Abstraction makes things easier to see!

- A table ...
- Five chairs ...
- A big bowl of spaghetti ...
- Five forks ...
- Five hungry people!
- The spaghetti is slippery; you need two forks to eat it.
- Everybody sits down together;
- everybody reaches for a fork;
- and then for a second fork;
- ... deadlock! Starvation!

Abstraction makes things easier to see!

- A table ...
- Five chairs ...
- A big bowl of spaghetti ...
- Five forks ...
- Five hungry people!
- The spaghetti is slippery; you need two forks to eat it.
- Everybody sits down together;
- everybody reaches for a fork;
- and then for a second fork;
- ... deadlock! Starvation!
- But if just one person hangs back ...

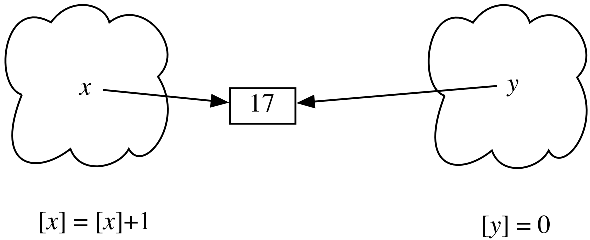
The standard "race condition"



$[x] = [x]+1$

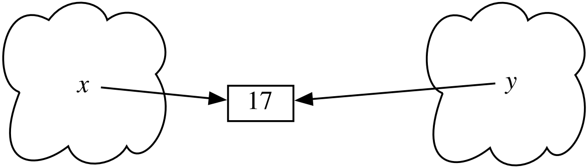
$[y] = 0$

The standard "race condition"



- With *atomic* actions, the outcome is either 1 or 0.

The standard "race condition"

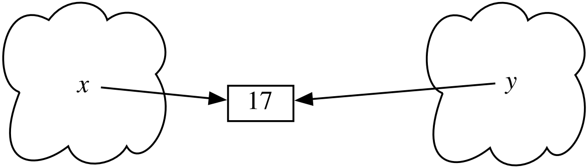


$[x] = [x]{+}1$
(read, inc, write)

$[y] = 0$
(write)

- With *atomic* actions, the outcome is either 1 or 0.

The standard “race condition”

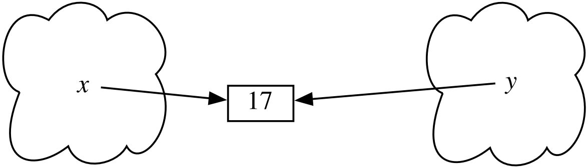


[x] = [x]+1
(read, inc, write)

[y] = 0
(write)

- With *atomic* actions, the outcome is either 1 or 0.
- With *interleaved* actions (but atomic read/write), the outcome is either 0, 1 or 18 (a lost message).

The standard "race condition"



$[x] = [x]+1$
(read, inc, write)

$[y] = 0$
(write)

- With *atomic* actions, the outcome is either 1 or 0.
- With *interleaved* actions (but atomic read/write), the outcome is either 0, 1 or 18 (a lost message).
- If read and write can be subdivided – chaos.

Dijkstra's solution: block signalling

Dijkstra's solution: block signalling

- Semaphores are like railway signals.

Dijkstra's solution: block signalling

Middlesex University

- Semaphores are like railway signals.
- "Critical sections" are like sections of track:
  $\mathrm{P}(m)$; .. critical ..; $\mathrm{V}(m)$.

Dijkstra's solution: block signalling

- Semaphores are like railway signals.
- "Critical sections" are like sections of track:
  $\mathrm{P}(m)$; .. critical ..; $\mathrm{V}(m)$.
- Atomic P and V required special hardware, now universally used.

Dijkstra's solution: block signalling

Middlesex University

- Semaphores are like railway signals.
- "Critical sections" are like sections of track:
  $\mathrm{P}(m)$; .. critical ..; $\mathrm{V}(m)$.
- Atomic P and V required special hardware, now universally used.
- Critical sections with the same semaphore are mutually exclusive, effectively atomic.

Dijkstra's solution: block signalling

Middlesex University

- Semaphores are like railway signals.
- "Critical sections" are like sections of track:
  $\mathrm{P}(m)$; .. critical ..; $\mathrm{V}(m)$.
- Atomic P and V required special hardware, now universally used.
- Critical sections with the same semaphore are mutually exclusive, effectively atomic.
- But semaphores caused waiting, queuing, *stopping*.

Dijkstra's solution: block signalling

Middlesex University

- Semaphores are like railway signals.
- "Critical sections" are like sections of track:
  $\mathrm{P}(m)$; .. critical ..; $\mathrm{V}(m)$.
- Atomic P and V required special hardware, now universally used.
- Critical sections with the same semaphore are mutually exclusive, effectively atomic.
- But semaphores caused waiting, queuing, *stopping*.
- New problems: deadlock, livelock, unfairness, starvation, ...

An early speedup

- Many readers at once, only one writer (and then no readers).

An early speedup

- Many readers at once, only one writer (and then no readers).
- New problems: fairness between readers and writers.

An early speedup

- Many readers at once, only one writer (and then no readers).
- New problems: fairness between readers and writers.
- But still ... (Courtois, Heymans, Parnas; 1971):

An early speedup

- Many readers at once, only one writer (and then no readers).
- New problems: fairness between readers and writers.
- But still ... (Courtois, Heymans, Parnas; 1971):

Reader:

$\mathrm{P}(read);$
$count += 1;$
$\text{if } (count == 1)\ \mathrm{P}(write);$
$\mathrm{V}(read);$

... reading happens here ...;

$\mathrm{P}(read);$
$count -= 1;$
$\text{if } (count == 0)\ \mathrm{V}(write);$
$\mathrm{V}(read)$

Writer:

$\mathrm{P}(write);$

... writing happens here ...

$\mathrm{V}(write)$

Hoare logic

Hoare logic

- Moder comput*ing* arose from a collision between mathematical logic and mechanical calculators during WW2.

Hoare logic

- Moder comput*ing* arose from a collision between mathematical logic and mechanical calculators during WW2.
- Every programming language is a mathematical *formal system* – that is, a *logic*.

Hoare logic

- Moder comput*ing* arose from a collision between mathematical logic and mechanical calculators during WW2.
- Every programming language is a mathematical *formal system* – that is, a *logic*.
- Every computer program is a sketch of a formal proof.

Hoare logic

- Moder comput*ing* arose from a collision between mathematical logic and mechanical calculators during WW2.
- Every programming language is a mathematical *formal system* – that is, a *logic*.
- Every computer program is a sketch of a formal proof.
- The task of computer science is to exploit the links between formal logic and practical programming.

Hoare logic

- Moder comput*ing* arose from a collision between mathematical logic and mechanical calculators during WW2.
- Every programming language is a mathematical *formal system* – that is, a *logic*.
- Every computer program is a sketch of a formal proof.
- The task of computer science is to exploit the links between formal logic and practical programming.
- The best attempt so far is Hoare logic: $\{\text{pre}\}$ command $\{\text{post}\}$.

Hoare logic

Middlesex University

- Moder comput*ing* arose from a collision between mathematical logic and mechanical calculators during WW2.
- Every programming language is a mathematical *formal system* – that is, a *logic*.
- Every computer program is a sketch of a formal proof.
- The task of computer science is to exploit the links between formal logic and practical programming.
- The best attempt so far is Hoare logic: {pre} command {post}.
- Example: $\{y + 1 = z\} x = y + 1 \{x = z\}$.

Hoare logic

- Moder comput*ing* arose from a collision between mathematical logic and mechanical calculators during WW2.
- Every programming language is a mathematical *formal system* – that is, a *logic*.
- Every computer program is a sketch of a formal proof.
- The task of computer science is to exploit the links between formal logic and practical programming.
- The best attempt so far is Hoare logic: {pre} command {post}.
- Example: $\{y+1=z\}x=y+1\{x=z\}$.
- This derives from a *rule*: $\{R_x^E\}x=E\{R\}$.

Hoare logic

- Moder comput*ing* arose from a collision between mathematical logic and mechanical calculators during WW2.
- Every programming language is a mathematical *formal system* – that is, a *logic*.
- Every computer program is a sketch of a formal proof.
- The task of computer science is to exploit the links between formal logic and practical programming.
- The best attempt so far is Hoare logic: {pre} command {post}.
- Example: $\{y+1=z\}x=y+1\{x=z\}$.
- This derives from a *rule*: $\{R^E_x\}x=E\{R\}$.
- There are rules for every program structure.

Progress is slow

Progress is slow

- Twenty-five years ago, some of us thought that Hoare's "formal methods" would sweep the board.

Progress is slow

- Twenty-five years ago, some of us thought that Hoare's "formal methods" would sweep the board.
- But it is difficult to scale up ...

Progress is slow

- Twenty-five years ago, some of us thought that Hoare's "formal methods" would sweep the board.
- But it is difficult to scale up ...
- The best that has been done so far is a program that runs the safety software on a driverless train line in Paris.

Progress is slow

- Twenty-five years ago, some of us thought that Hoare's "formal methods" would sweep the board.
- But it is difficult to scale up ...
- The best that has been done so far is a program that runs the safety software on a driverless train line in Paris.
- – a few thousand lines, and **no bugs**!

Progress is slow

- Twenty-five years ago, some of us thought that Hoare's "formal methods" would sweep the board.
- But it is difficult to scale up ...
- The best that has been done so far is a program that runs the safety software on a driverless train line in Paris.
- – a few thousand lines, and **no bugs**!
- Until recently, pointers (aka Java "references") were thought to be beyond the scope of Hoare logic ...

Progress is slow

- Twenty-five years ago, some of us thought that Hoare's "formal methods" would sweep the board.
- But it is difficult to scale up ...
- The best that has been done so far is a program that runs the safety software on a driverless train line in Paris.
- – a few thousand lines, and **no bugs**!
- Until recently, pointers (aka Java "references") were thought to be beyond the scope of Hoare logic ...
- ... but we've found a way! $x \mapsto 17$ says that x contains a pointer to a location that contains 17 ...

Progress is slow

- Twenty-five years ago, some of us thought that Hoare's "formal methods" would sweep the board.
- But it is difficult to scale up ...
- The best that has been done so far is a program that runs the safety software on a driverless train line in Paris.
- – a few thousand lines, and **no bugs**!
- Until recently, pointers (aka Java "references") were thought to be beyond the scope of Hoare logic ...
- ... but we've found a way! $x \mapsto 17$ says that x contains a pointer to a location that contains 17 ...
- ... and $x \mapsto E \star y \mapsto E'$ says that there are two separate heap cells, **which we can reason about separately** ...

Progress is slow

- Twenty-five years ago, some of us thought that Hoare's "formal methods" would sweep the board.
- But it is difficult to scale up ...
- The best that has been done so far is a program that runs the safety software on a driverless train line in Paris.
- – a few thousand lines, and **no bugs**!
- Until recently, pointers (aka Java "references") were thought to be beyond the scope of Hoare logic ...
- ... but we've found a way! $x \mapsto 17$ says that x contains a pointer to a location that contains 17 ...
- ... and $x \mapsto E \star y \mapsto E'$ says that there are two separate heap cells, **which we can reason about separately** ...
- ... now we can prove lots of pointer programs.

Thirty years later ...

- The readers-and-writers program obviously works ...

Thirty years later ...

- The readers-and-writers program obviously works ...
- ... and at last we can prove some things about it!

Thirty years later ...

- The readers-and-writers program obviously works ...
- ... and at last we can prove some things about it!
- O'Hearn has inverted semaphores, making them safes which lock away resources, opened by P and locked by V:

Thirty years later ...

- The readers-and-writers program obviously works ...
- ... and at last we can prove some things about it!
- O'Hearn has inverted semaphores, making them safes which lock away resources, opened by P and locked by V:

$$\{\mathbf{emp}\}\ \mathrm{P}(m)\ \{I_m\}$$
$$\{I_m\}\ \mathrm{V}(m)\ \{\mathbf{emp}\}$$

Thirty years later ...

- The readers-and-writers program obviously works ...
- ... and at last we can prove some things about it!
- O'Hearn has inverted semaphores, making them safes which lock away resources, opened by P and locked by V:

$$\begin{array}{r} \{\mathbf{emp}\}\ \mathrm{P}(m)\ \{I_m\} \\ \{I_m\}\ \mathrm{V}(m)\ \{\mathbf{emp}\} \end{array}$$

- Calcagno and I invented read permissions ($\mapsto$) and counting permissions ($\overset{n}{\longmapsto}$, where only $\overset{0}{\longmapsto}$ can write).

Thirty years later ...

- The readers-and-writers program obviously works ...
- ... and at last we can prove some things about it!
- O'Hearn has inverted semaphores, making them safes which lock away resources, opened by P and locked by V:

$$\{\mathbf{emp}\}\ \mathrm{P}(m)\ \{I_m\}$$
$$\{I_m\}\ \mathrm{V}(m)\ \{\mathbf{emp}\}$$

- Calcagno and I invented read permissions ($\mapsto$) and counting permissions ($\overset{n}{\longmapsto}$, where only $\overset{0}{\longmapsto}$ can write).

$$x \overset{n}{\longmapsto} E \iff x \overset{n+1}{\longmapsto} E \star x \mapsto E$$

A proof

$write : z \stackrel{0}{\longmapsto} \_$

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

$\{\mathbf{emp}\}$

$\mathrm{P}(read);$

$\{\text{if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \xmapsto{count} \_\}$

$count{+} := 1;$

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

$\text{if } count = 1 \text{ then } \{\mathbf{emp}\}\ \mathrm{P}(write)\ \{z \stackrel{0}{\longmapsto} \_\}$

$\qquad\qquad \text{else} \qquad\qquad \{z \xmapsto{count-1} \_\};$

$\{z \xmapsto{count-1} \_\}$

$\{z \xmapsto{count} \_ \star z \mapsto \_\}$

$\mathrm{V}(read);$

$\{z \mapsto \_\}$

But only a part of a proof …

But only a part of a proof ...

- There are problems far worse than race conditions.

But only a part of a proof ...

- There are problems far worse than race conditions.
- Starvation, as in “dining philosophers”, is a result of lack of progress.

But only a part of a proof ...

- There are problems far worse than race conditions.
- Starvation, as in “dining philosophers”, is a result of lack of progress.
- $\{\mathbf{emp}\}$ $\mathrm{P}(m)$ $\{I_m\}$ is “partial correctness” - *if* you get through then you collect a prize, but you may never get through.

But only a part of a proof ...

Middlesex University

- There are problems far worse than race conditions.
- Starvation, as in “dining philosophers”, is a result of lack of progress.
- $\{\mathbf{emp}\}\ \mathrm{P}(m)\ \{I_m\}$ is “partial correctness” - *if* you get through then you collect a prize, but you may never get through.
- We can reason about resource ownership, resource leaks, resource safety ... all at the local level.

But only a part of a proof ...

- There are problems far worse than race conditions.
- Starvation, as in “dining philosophers”, is a result of lack of progress.
- $\{\mathbf{emp}\}\ \mathrm{P}(m)\ \{I_m\}$ is “partial correctness” - *if* you get through then you collect a prize, but you may never get through.
- We can reason about resource ownership, resource leaks, resource safety ... all at the local level.
- Reasoning about progress still needs to be global.

But only a part of a proof ...

- There are problems far worse than race conditions.
- Starvation, as in “dining philosophers”, is a result of lack of progress.
- $\{\mathbf{emp}\}\ \mathrm{P}(m)\ \{I_m\}$ is “partial correctness” - *if* you get through then you collect a prize, but you may never get through.
- We can reason about resource ownership, resource leaks, resource safety ... all at the local level.
- Reasoning about progress still needs to be global.
- This is still beyond us in practice.

Summary

Summary

- Right here in Mdx U, world-class research is going on.

Summary

- Right here in Mdx U, world-class research is going on.
- You have a chance to join in.

Summary

- Right here in Mdx U, world-class research is going on.
- You have a chance to join in.
- It will stretch you.

Summary

- Right here in Mdx U, world-class research is going on.
- You have a chance to join in.
- It will stretch you.
- But isn’t that why you came here?