# DataCollider: Effective Data-Race Detection for the Kernel

John Erickson, Madanlal Musuvathi,

Sebastian Burckhardt, Kirk Olynyk

*Microsoft Windows and Microsoft Research*

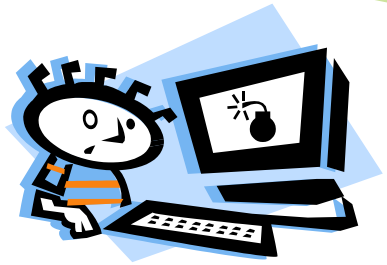
{jerick, madanm, sburckha, kirko}@microsoft.com

"Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism."
— From "The Problem with Threads," by Edward A. Lee, *IEEE Computer*, vol. 25, no. 5, May 2006

# Windows case study #1

Thread A

```
RunContext(...)
{
  pctxt->dwfCtxt &=
    ~CTXTF_RUNNING;
}
```



Thread B

```
RestartCtxtCallback(...)
{
  pctxt->dwfCtxt |=
    CTXTF_NEED_CALLBACK;
}
```

- The OR'ing in of the CTXTF_NEED_CALLBACK flag can be swallowed by the AND'ing out of the CTXTF_RUNNING flag!
- Results in system hang.

# Case study #1, assembled

Thread A

```
mov eax, [pctxt->dwfCtxt]

and eax, NOT 10h

mov [pctxt->dwfCtxt], eax
```

```
EAX = ??
```

Thread B

```
mov eax, [pctxt->dwfCtxt]

or eax, 20h

mov [pctxt->dwfCtxt], eax
```

```
EAX = ??
```

```
pctxt->dwfCtxt = 11h
```

# Case study #1, assembled

| | Thread A | Thread B |
|---|---|---|
| 1 | `mov eax, [pctxt->dwfCtxt]`<br>`and eax, NOT 10h`<br>`mov [pctxt->dwfCtxt], eax` | `mov eax, [pctxt->dwfCtxt]`<br>`or eax, 20h`<br>`mov [pctxt->dwfCtxt], eax` |
| | `EAX = 11h` | `EAX = ??` |

`pctxt->dwfCtxt = 11h`

# Case study #1, assembled

| | Thread A | Thread B |
|---|---|---|
| 1 | `mov eax, [pctxt->dwfCtxt]` | `mov eax, [pctxt->dwfCtxt]` |
| 2 | `and eax, NOT 10h` | `or eax, 20h` |
| | `mov [pctxt->dwfCtxt], eax` | `mov [pctxt->dwfCtxt], eax` |
| | `EAX = 01h` | `EAX = ??` |

`pctxt->dwfCtxt = 11h`

# Case study #1, assembled

| | Thread A | Thread B |
|---|---|---|
| 1 | `mov eax, [pctxt->dwfCtxt]` | `mov eax, [pctxt->dwfCtxt]` |
| 2 | `and eax, NOT 10h` | `or eax, 20h` |
| | `/* CONTEXT SWITCH */` | |
| | `mov [pctxt->dwfCtxt], eax` | `mov [pctxt->dwfCtxt], eax` |
| | `EAX = 01h` | `EAX = ??` |

`pctxt->dwfCtxt = 11h`

# Case study #1, assembled

| Thread A | Thread B |
| --- | --- |
| 1 `mov eax, [pctxt->dwfCtxt]` | `mov eax, [pctxt->dwfCtxt]` 3 |
| 2 `and eax, NOT 10h` | `or eax, 20h` |
| `/* CONTEXT SWITCH */` | |
| `mov [pctxt->dwfCtxt], eax` | `mov [pctxt->dwfCtxt], eax` |
| `EAX = 01h` | `EAX = 11h` |

`pctxt->dwfCtxt = 11h`

# Case study #1, assembled

| Thread A | Thread B |
| --- | --- |
| 1 `mov eax, [pctxt->dwfCtxt]` | `mov eax, [pctxt->dwfCtxt]` 3 |
| 2 `and eax, NOT 10h` | `or eax, 20h` 4 |
| `/* CONTEXT SWITCH */` | |
| `mov [pctxt->dwfCtxt], eax` | `mov [pctxt->dwfCtxt], eax` |
| `EAX = 01h` | `EAX = 31h` |

`pctxt->dwfCtxt = 11h`

# Case study #1, assembled

| Thread A | Thread B |
| --- | --- |
| 1 `mov eax, [pctxt->dwfCtxt]` | `mov eax, [pctxt->dwfCtxt]` 3 |
| 2 `and eax, NOT 10h` | `or eax, 20h` 4 |
| `/* CONTEXT SWITCH */` | |
| `mov [pctxt->dwfCtxt], eax` | `mov [pctxt->dwfCtxt], eax` 5 |
| `EAX = 01h` | `EAX = 31h` |

`pctxt->dwfCtxt = 31h`

# Case study #1, assembled

| Thread A | Thread B |
| --- | --- |
| 1 `mov eax, [pctxt->dwfCtxt]` | `mov eax, [pctxt->dwfCtxt]` 3 |
| 2 `and eax, NOT 10h` | `or eax, 20h` 4 |
| `/* CONTEXT SWITCH */` | |
| 6 `mov [pctxt->dwfCtxt], eax` | `mov [pctxt->dwfCtxt], eax` 5 |
| `EAX = 01h` | `EAX = 31h` |

`pctxt->dwfCtxt = 01h`

# Case study #1, assembled

Thread A

Thread B

1 `mov eax, [pctxt->dwfCtxt]`

3 `mov eax, [pctxt->dwfCtxt]`

2 `and eax, NOT`

4

`/* CONTEXT S`

6 `mov [pctxt->`

`dwfCtxt], eax` 5

CTXTF_NEED_CALLBACK disappeared!

(pctxt->dwfCtxt & 0x20 == 0)

EAX = 01h

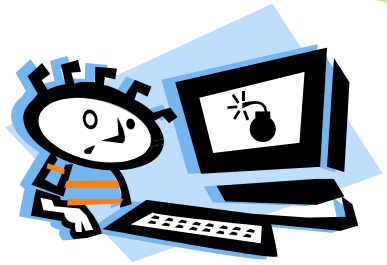
EAX = 31h

pctxt->dwfCtxt = 01h

# Windows case study #1

Thread A

```
RunContext(...)
{
 pctxt->dwfCtxt &=
    ~CTXTF_RUNNING;
 and [ecx+40], ~10h
}
```



Thread B

```
RestartCtxtCallback(...)
{
 pctxt->dwfCtxt |=
    CTXTF_NEED_CALLBACK;
 or [ecx+40], 20h
}
```

- Instructions appear atomic, but they are not!

# Data race definition

* By our definition, a data race is a pair of memory accesses that satisfy all the below:
  * The accesses can happen concurrently
  * There is a non-zero overlap in the physical address ranges specified by the two accesses
  * At least one access modifies the contents of the memory location

# Importance

* Very hard to reproduce
  * Timings can be very tight
* Hard to debug
  * Very easy to mistake as a hardware error “bit flip”

* To support scalability, code is moving away from monolithic locks
  * Fine-grained locks
  * Lock-free approaches

# Previous Techniques

* Happens-before and lockset algorithms have significant overhead
  * Intel Thread Checker has 200x overhead
  * Log all synchronizations
  * Instrument all memory accesses
* High overhead can prevent usage in the field
  * Causes false failures due to timeouts

# Challenges

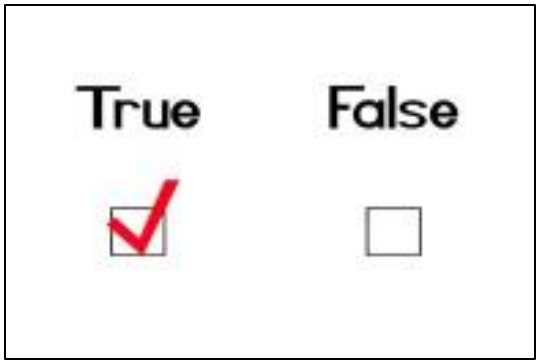
* Prior schemes require a complete knowledge and logging of all locking semantics

* Locking semantics in kernel-mode can be homegrown, complicated and convoluted.
  * e.g. DPCs, interrupts, affinities

# DataCollider: Goals

# DataCollider: Goals

**1. No false data races**




* Tradeoff between having false positives and reporting fewer data races

# False vs. Benign

* **False data race**
  * A data race that cannot actually occur
* **Benign data race**
  * A data race that can and does occur, but is intended to happen as part of normal program execution

# False vs. benign example

Thread A

```
MyLockAcquire();

gReferenceCount++;

MyLockRelease();

gStatisticsCount++;
```

Thread B

```
MyLockAcquire();

gReferenceCount++;

MyLockRelease();

gStatisticsCount++;
```

# False vs. Benign

* **False data race**
  * A data race that cannot actually occur

* **Benign data race**
  * A data race that can and does occur, but is intended to happen as part of normal program execution

# False vs. **benign** example

Thread A

```
MyLockAcquire();

gReferenceCount++;

MyLockRelease();

gStatisticsCount++;
```

Thread B

```
MyLockAcquire();

gReferenceCount++;

MyLockRelease();

gStatisticsCount++;
```

# DataCollider: Goals

**2. User-controlled overhead**

* Give user full control of overhead – from 0.0x up

* Fast vs. more races found

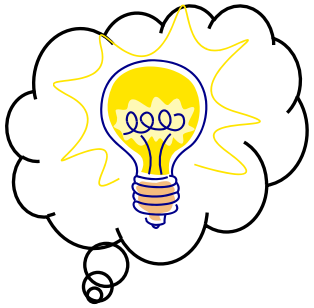
# DataCollider: Goals

**3. Actionable data**

* Contextual information is key to analysis and debugging

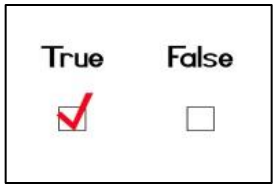
# Insights

# Insights

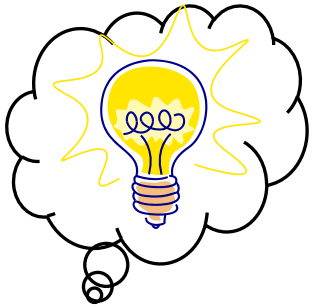


1. **Instead of *inferring*** if a data race *could* have occurred, **let's cause it to actually happen!**

   * No locksets, no happens-before

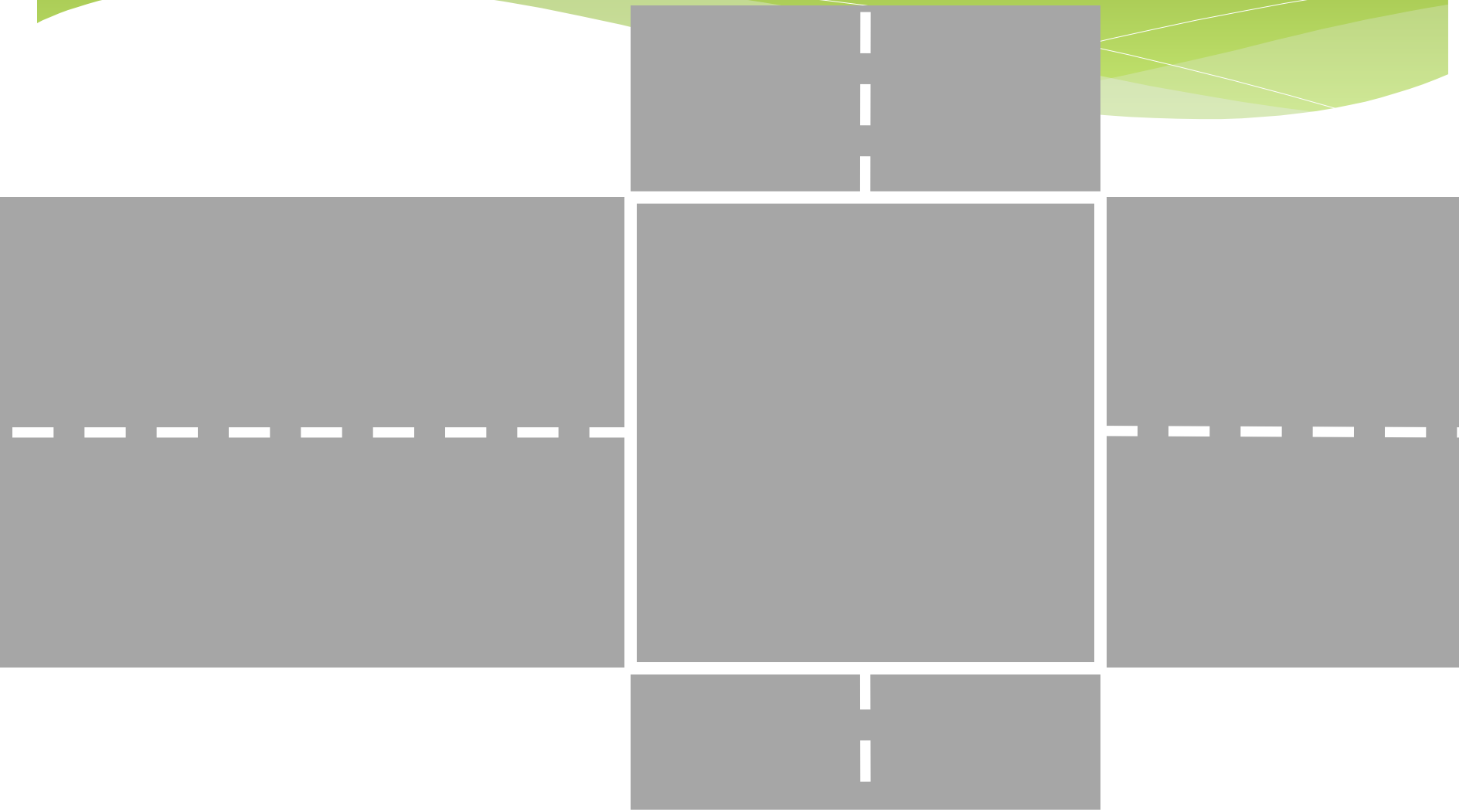



# Insights

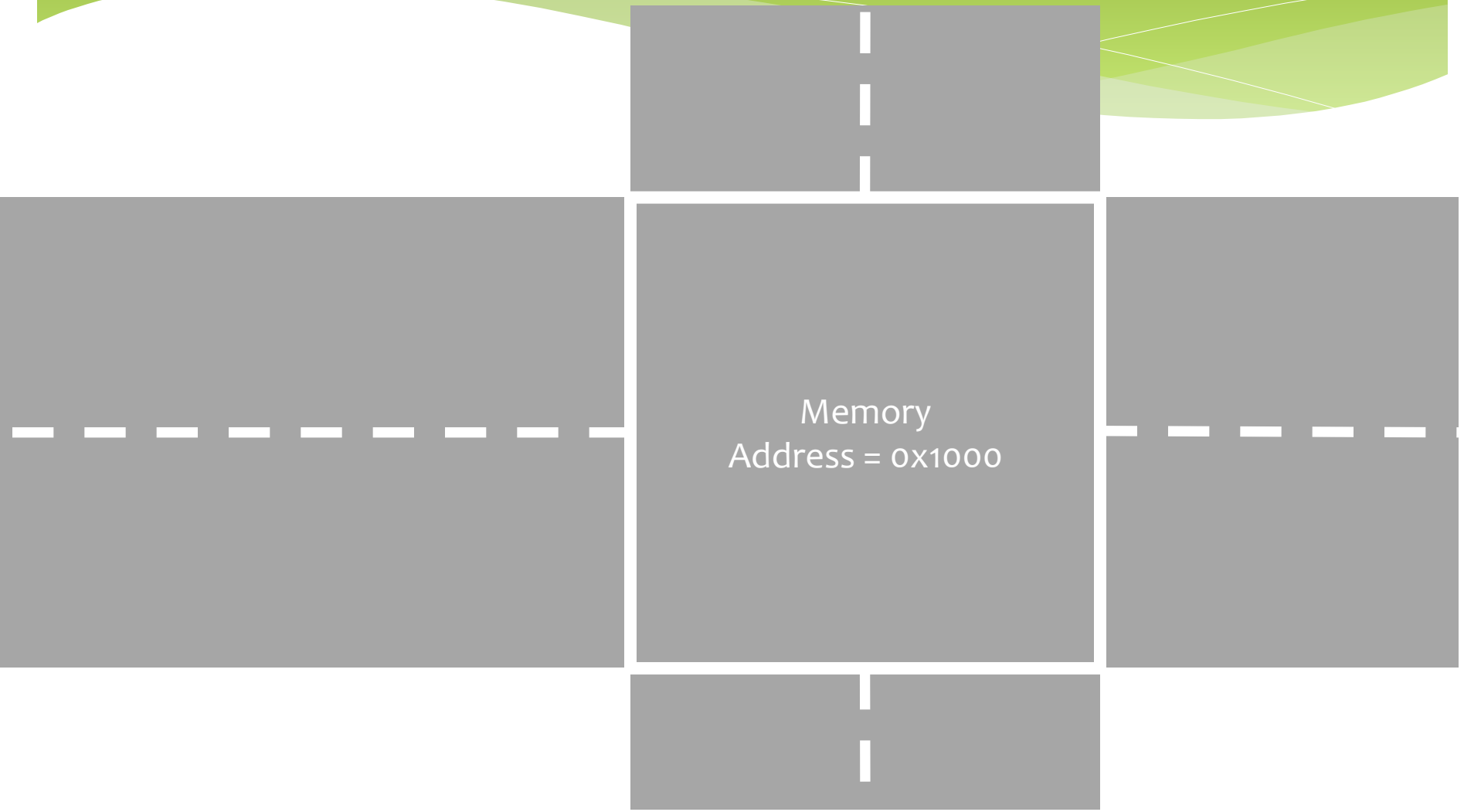


2. **Sample memory accesses**
   * *No* binary instrumentation
     * *No* synchronization logging
     * *No* memory access logging
   * Use code and data breakpoints
   * Randomly selection for uniform coverage

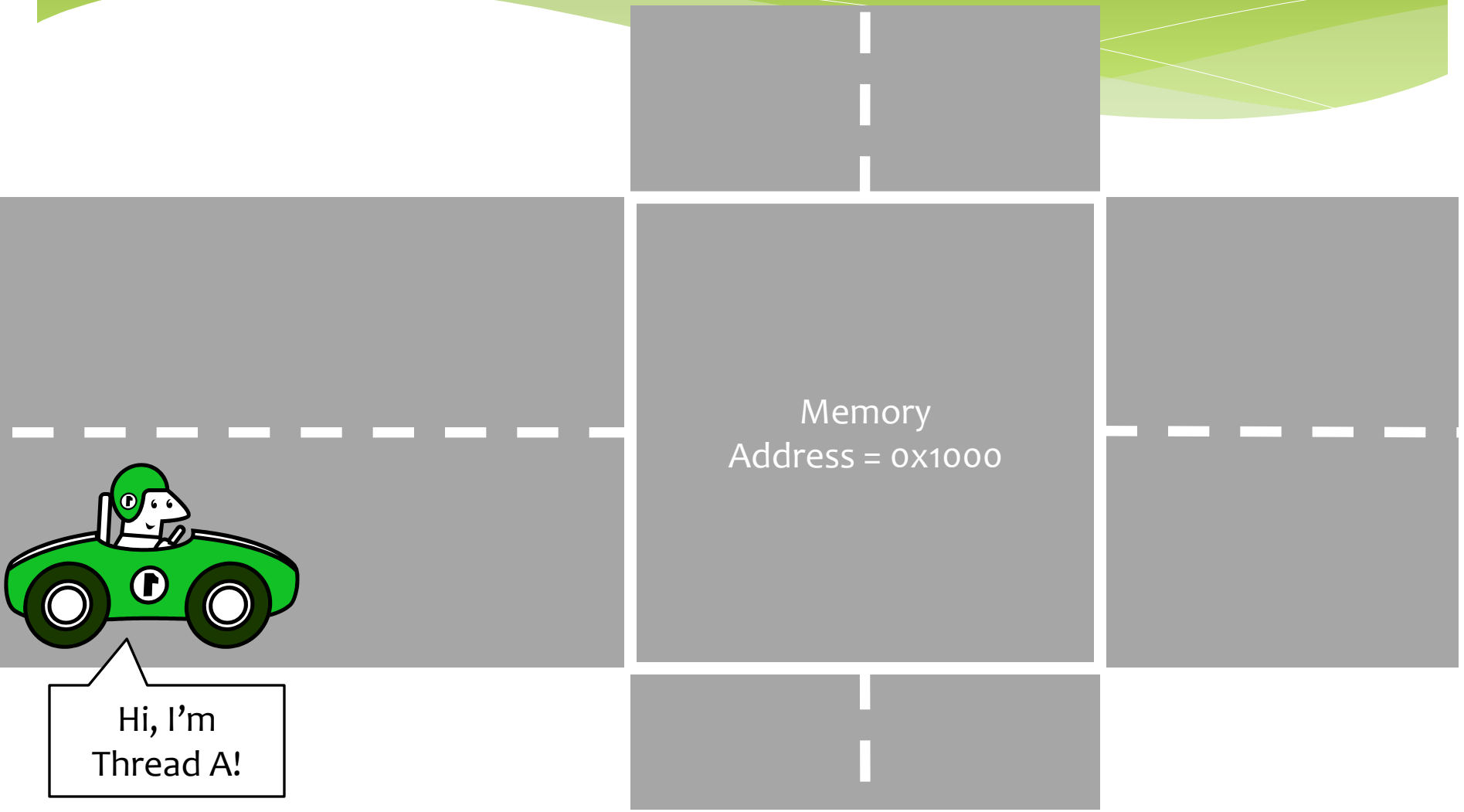
# Intersection Metaphor



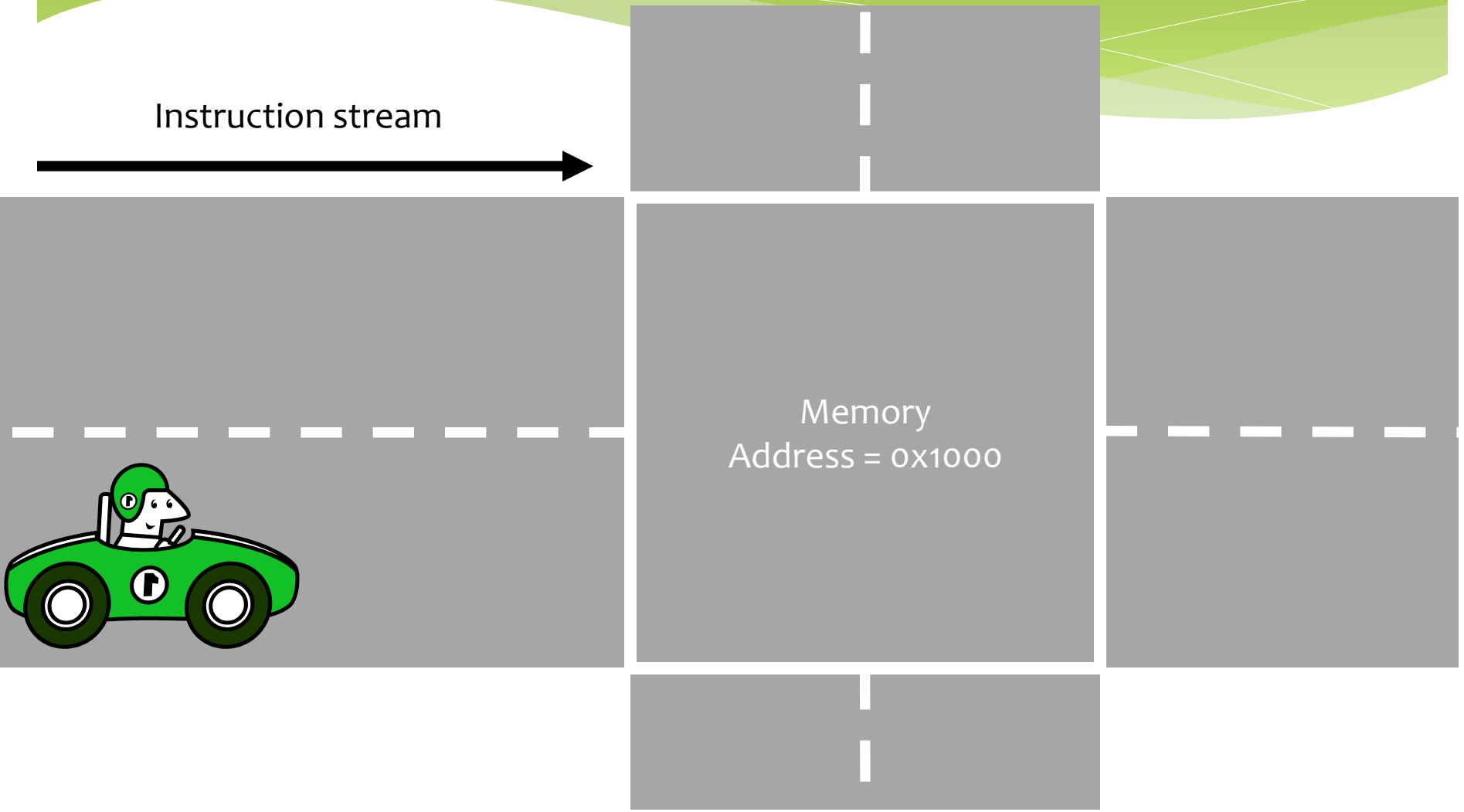
# Intersection Metaphor



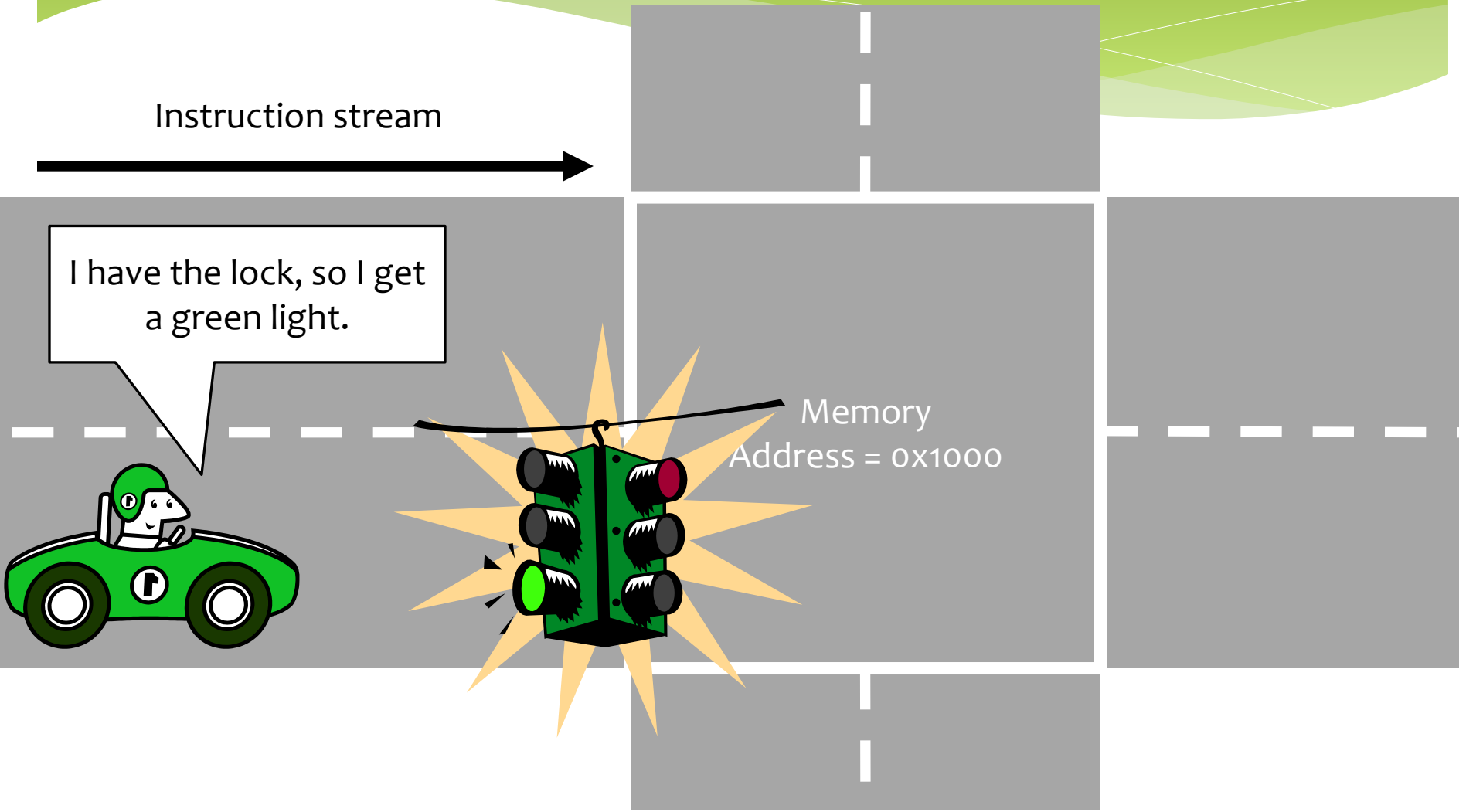
# Intersection Metaphor



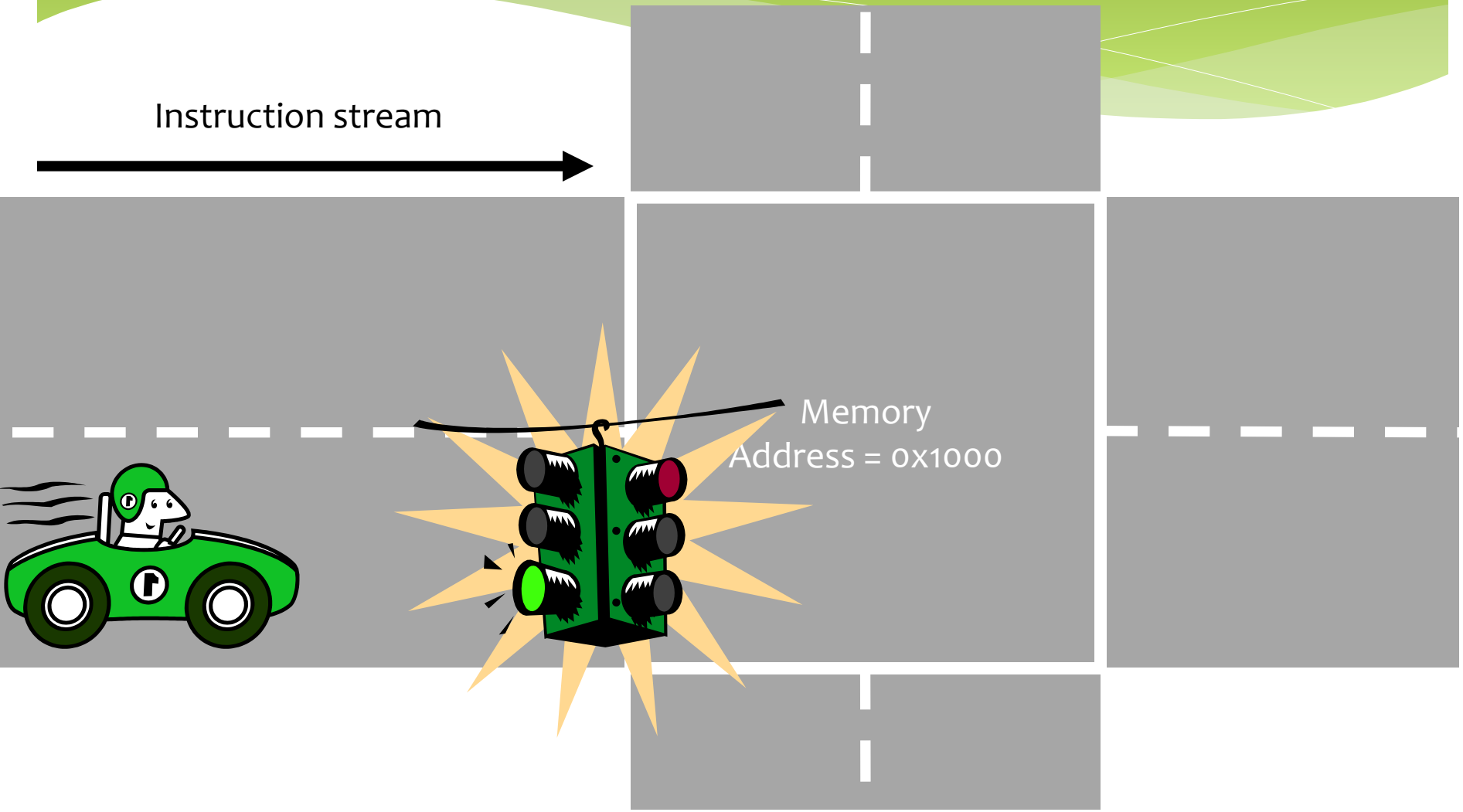
# Intersection Metaphor



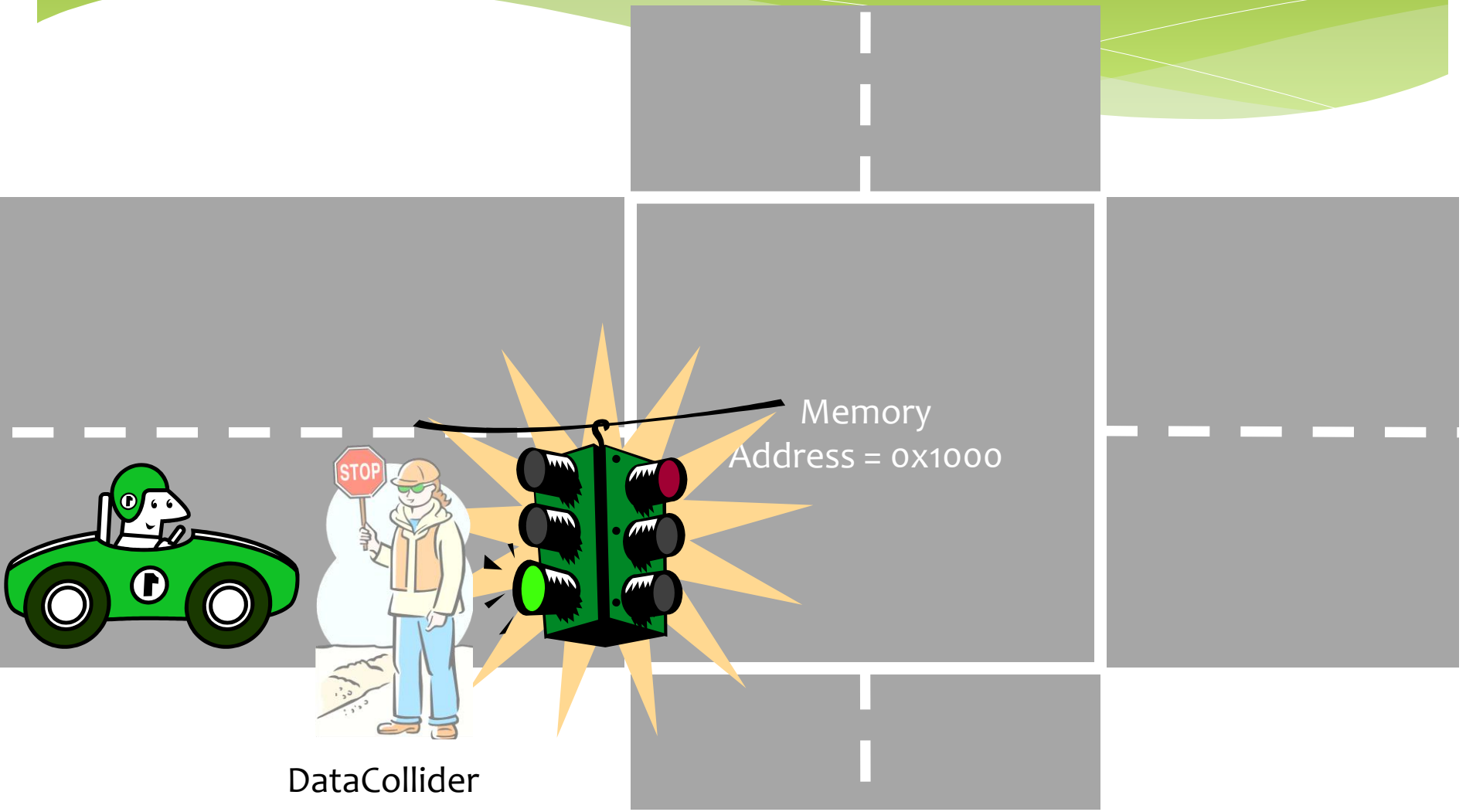
# Intersection Metaphor



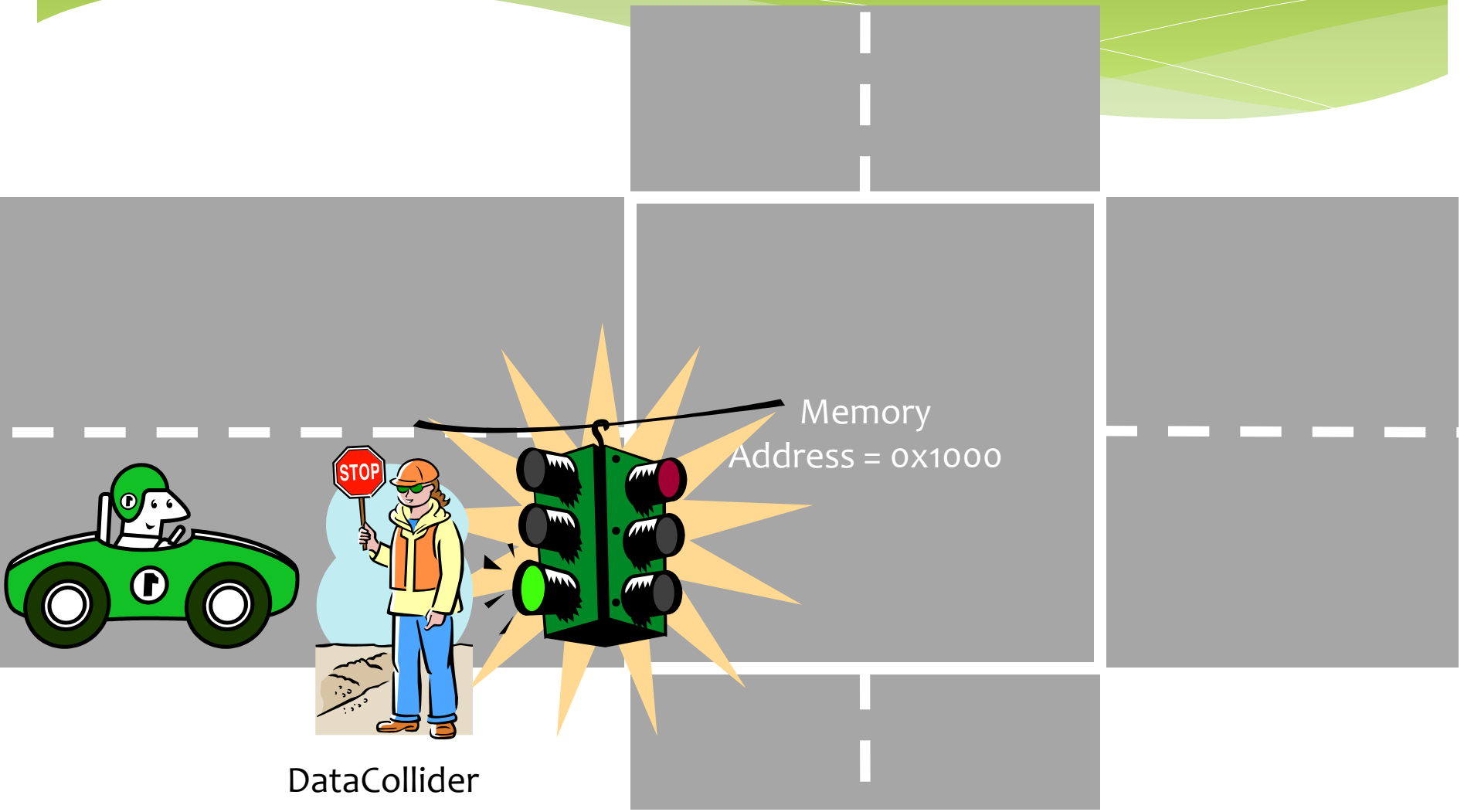
# Intersection Metaphor



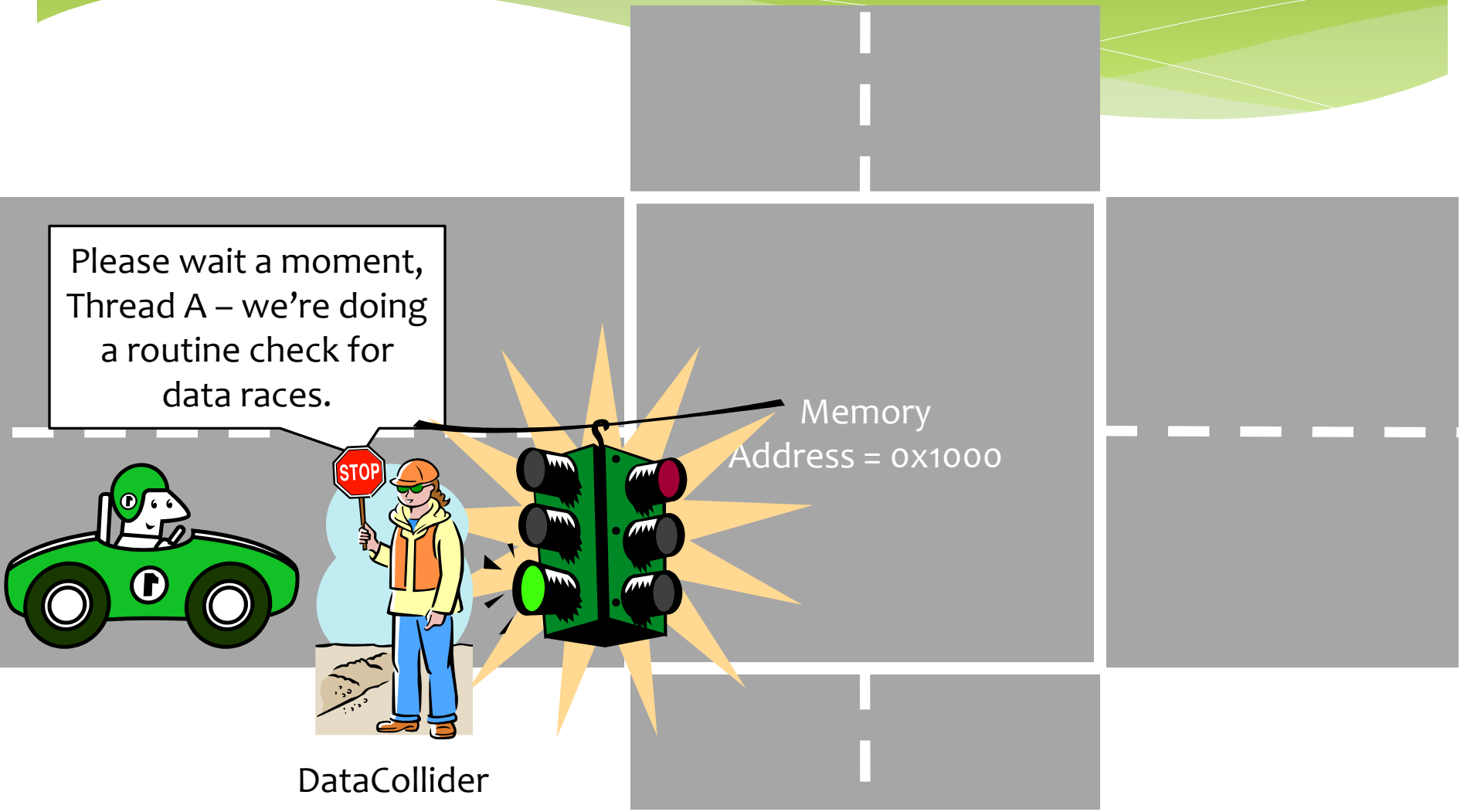
# Intersection Metaphor



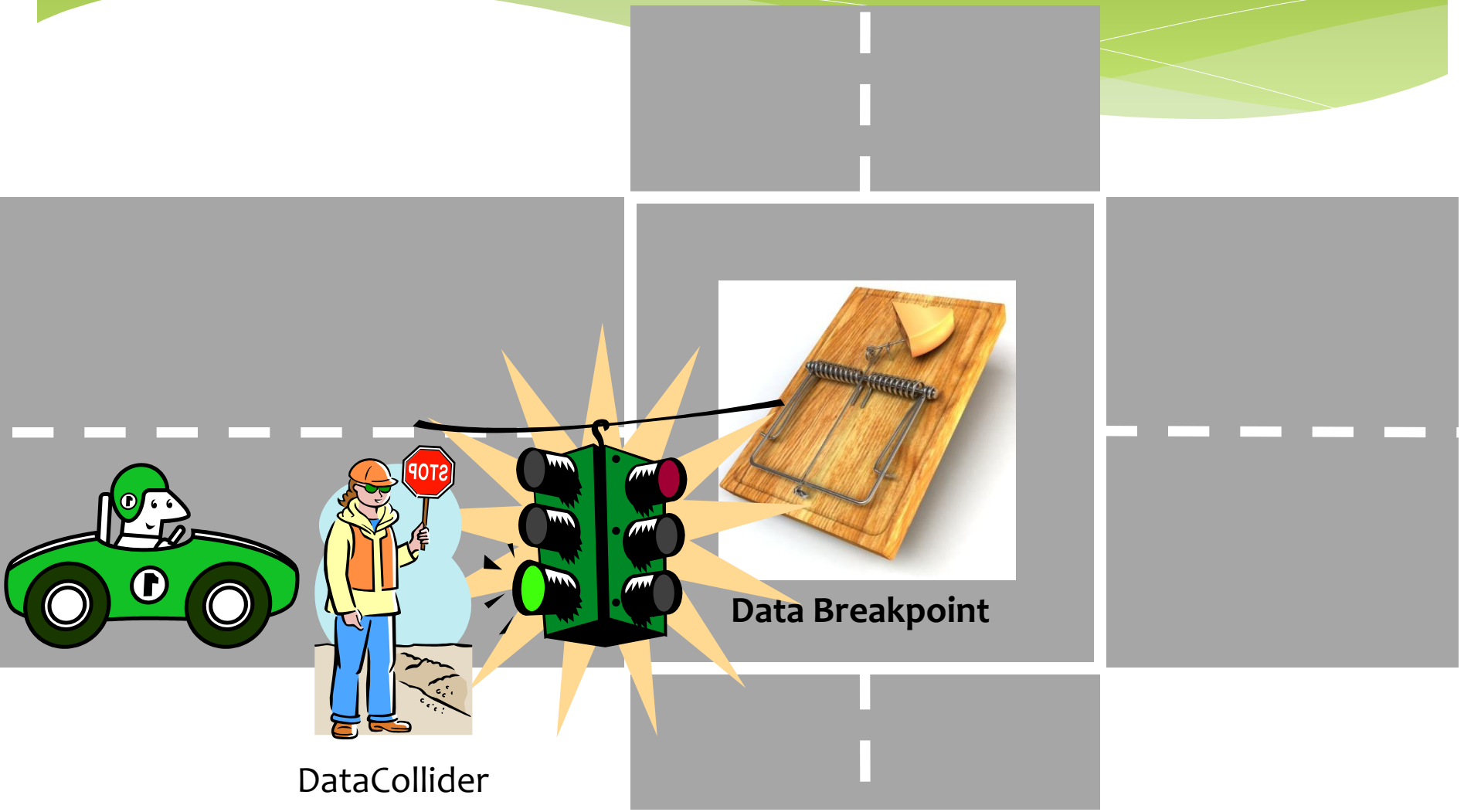
# Intersection Metaphor



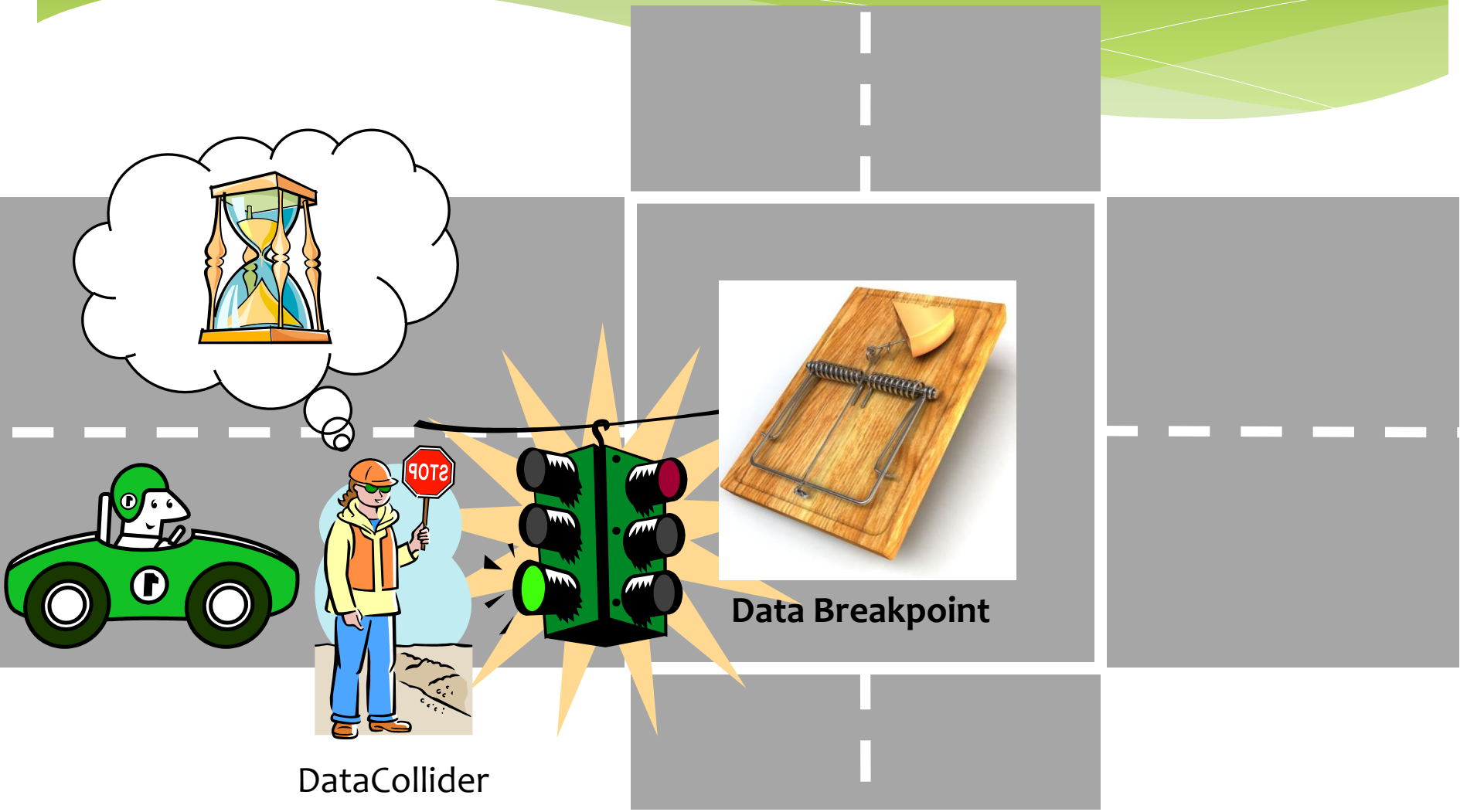
# Intersection Metaphor



# Intersection Metaphor

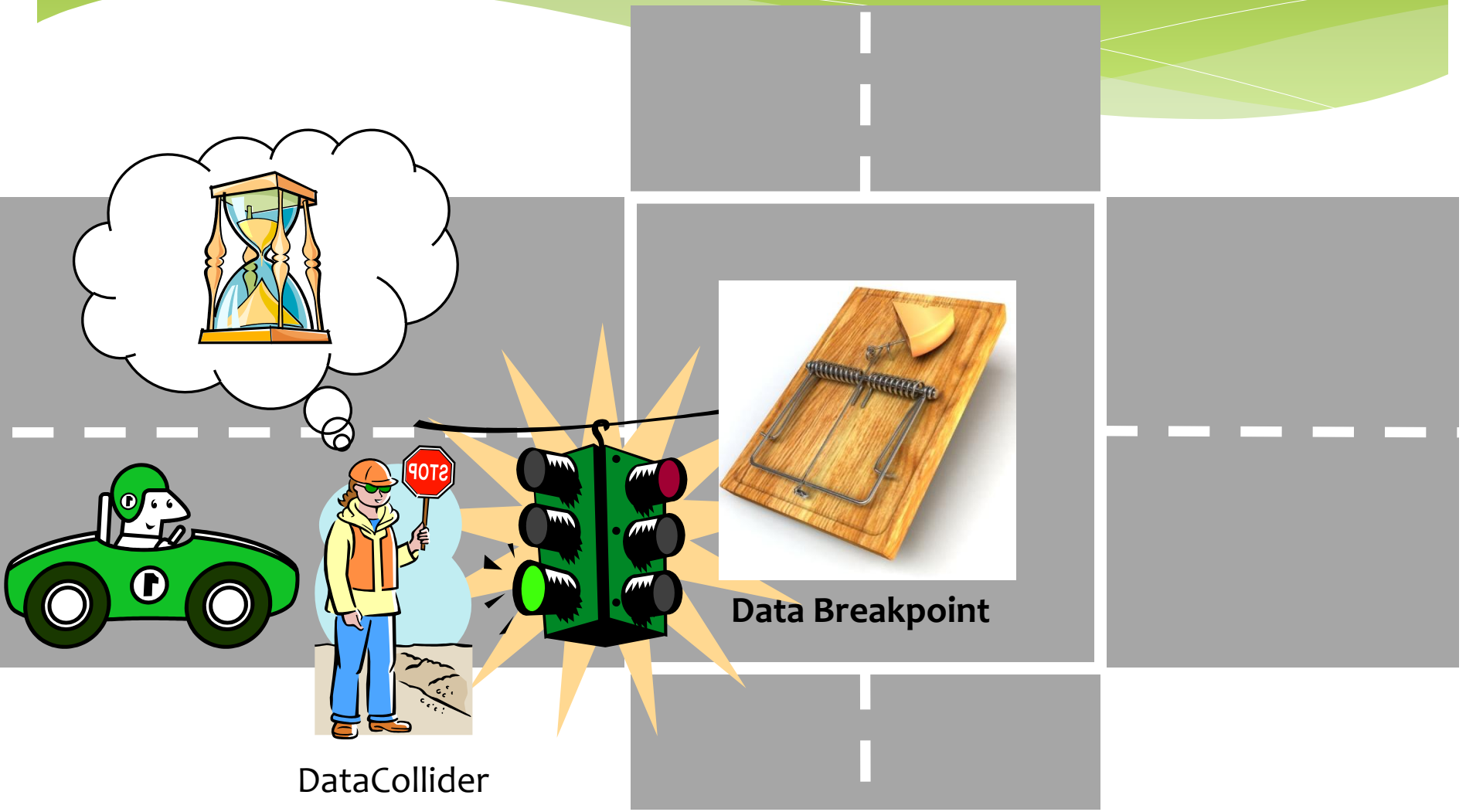


# Intersection Metaphor

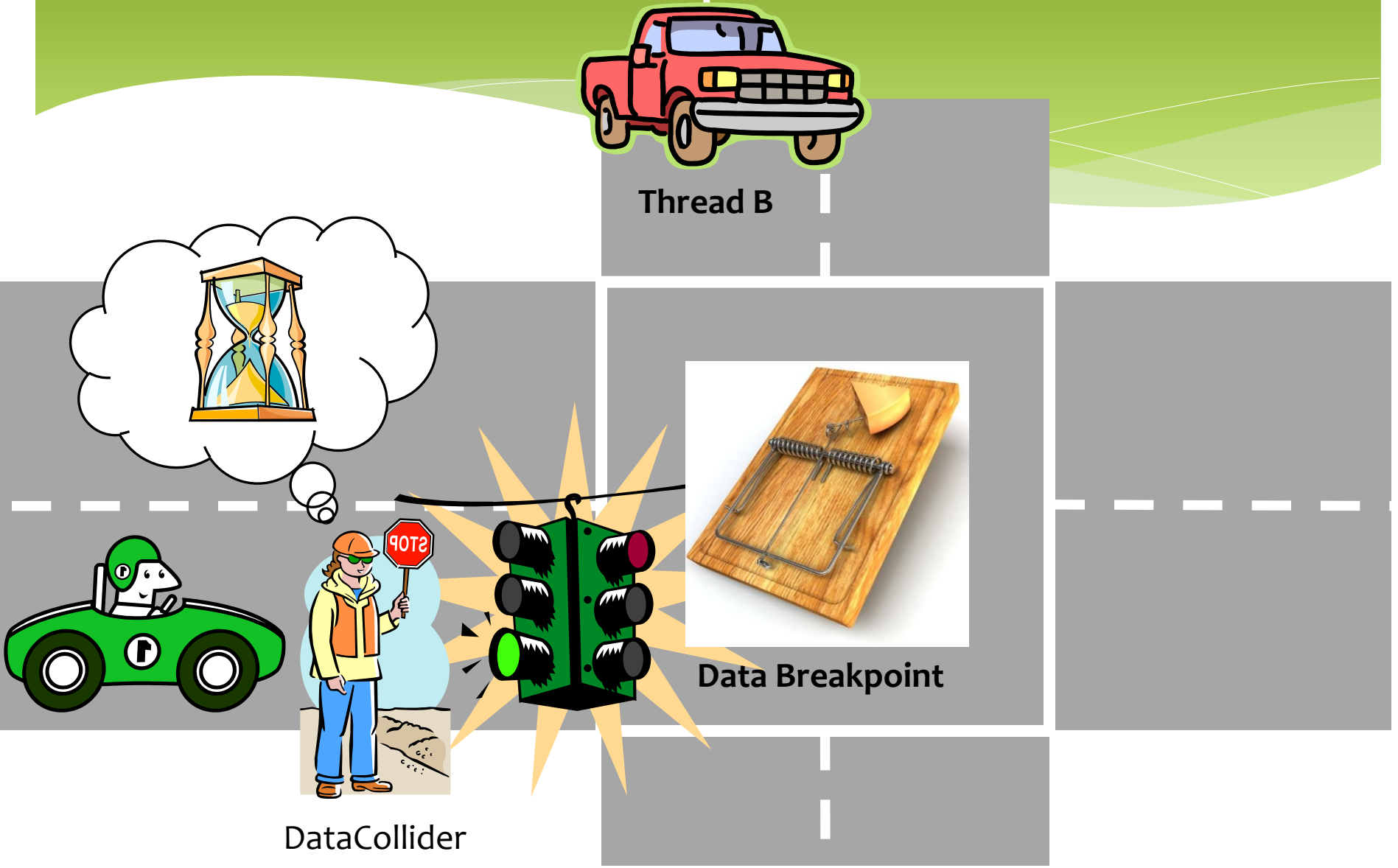


# Intersection Metaphor: Normal Case

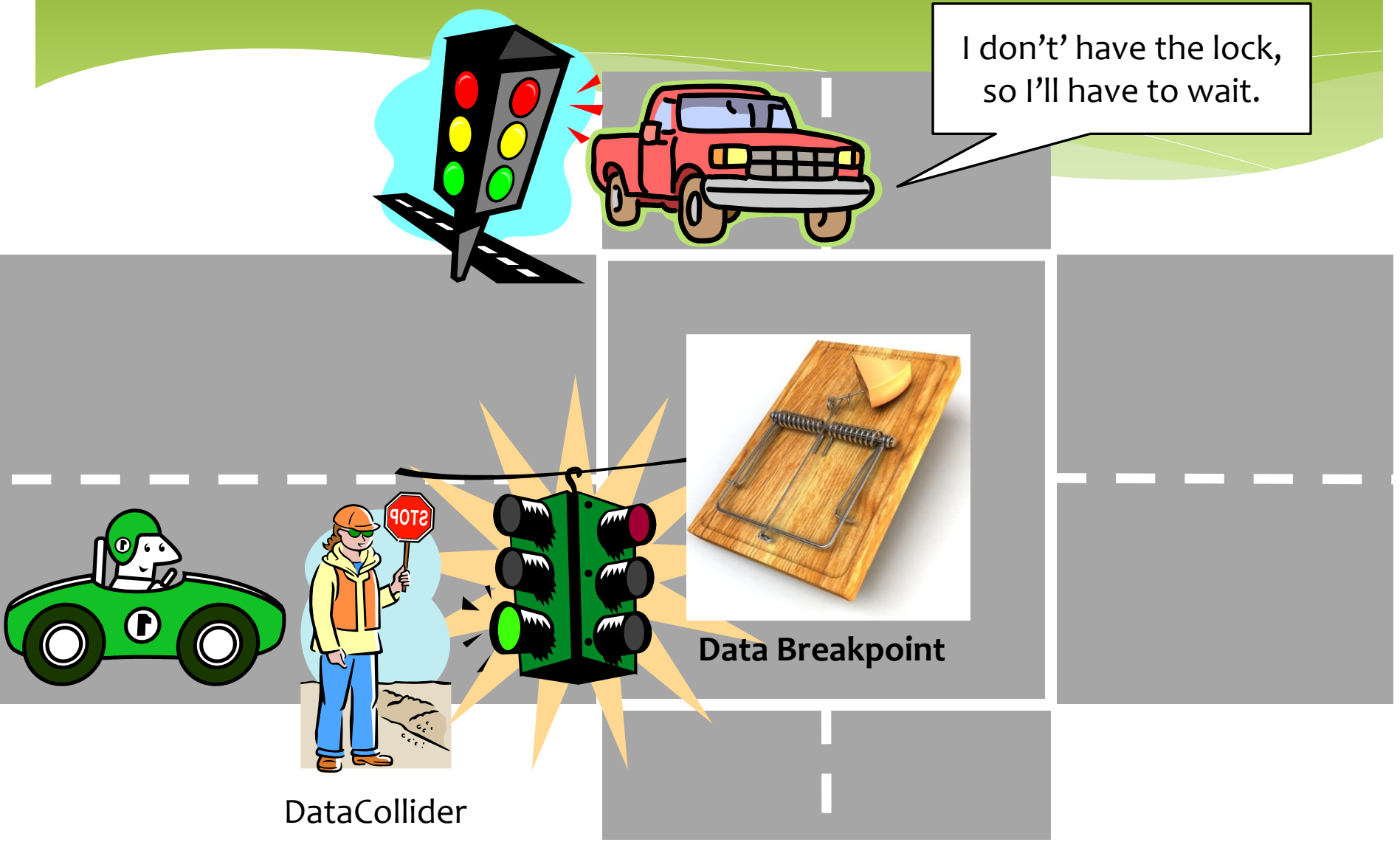
# Intersection Metaphor: Normal Case



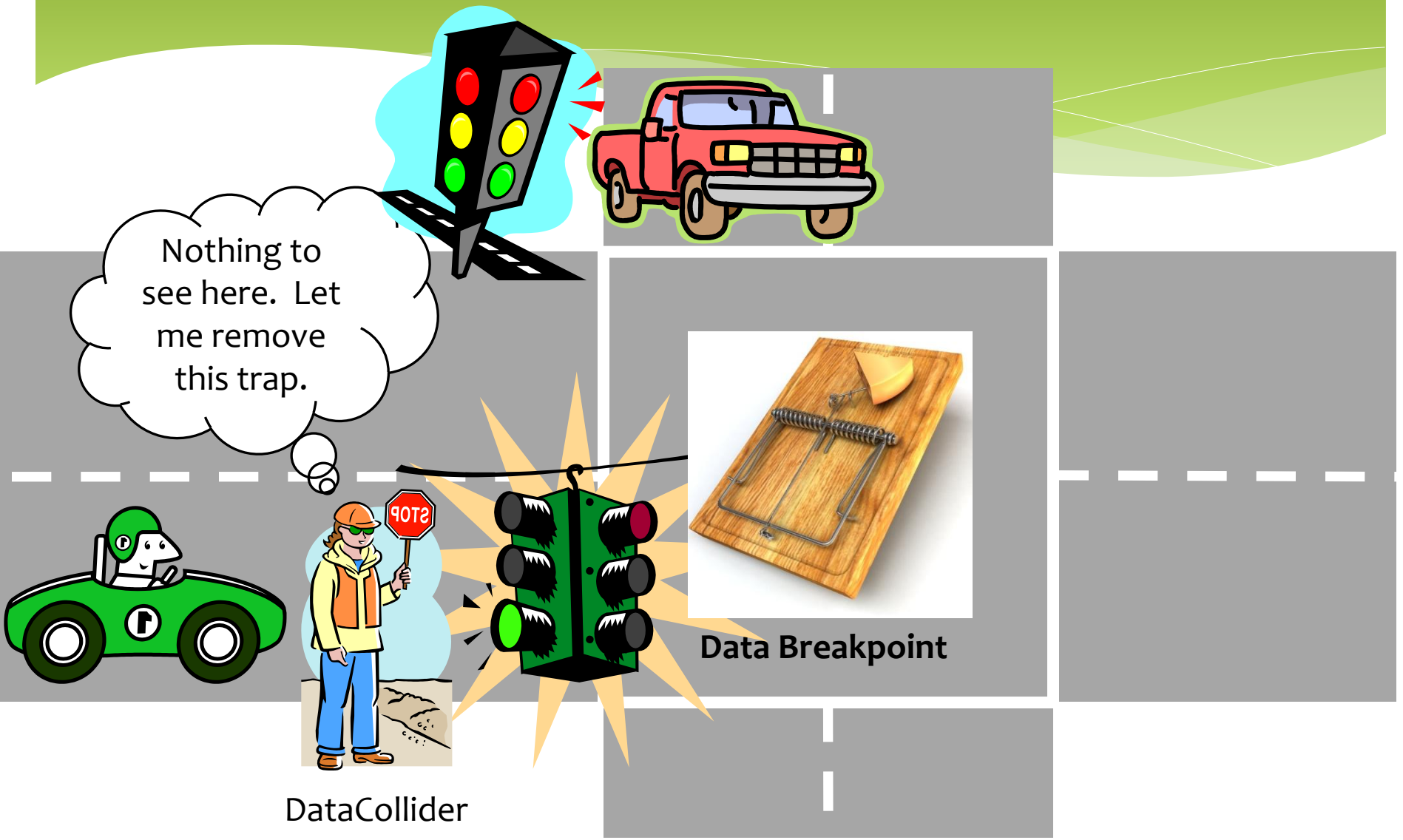
# Intersection Metaphor: Normal Case



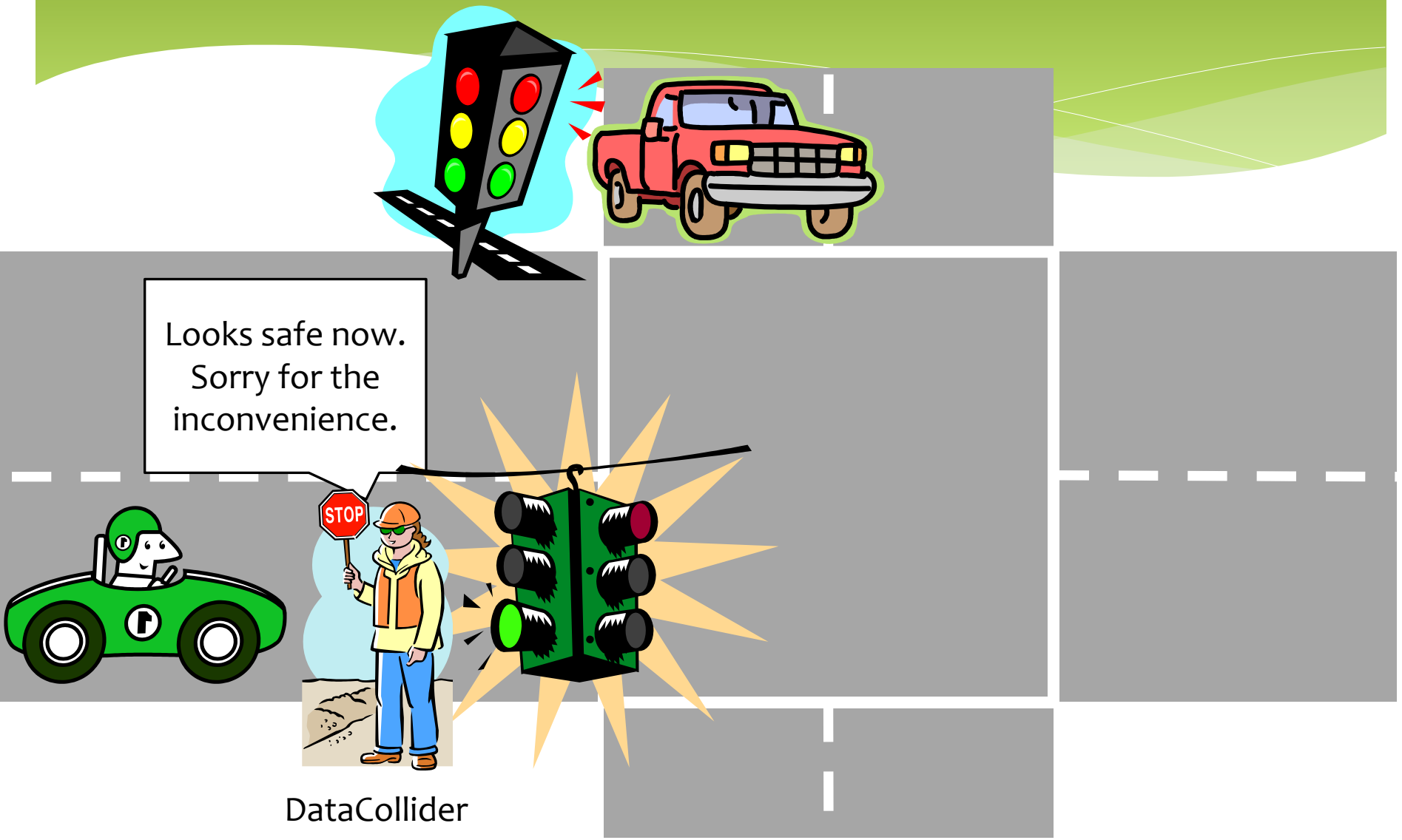
# Intersection Metaphor: Normal Case



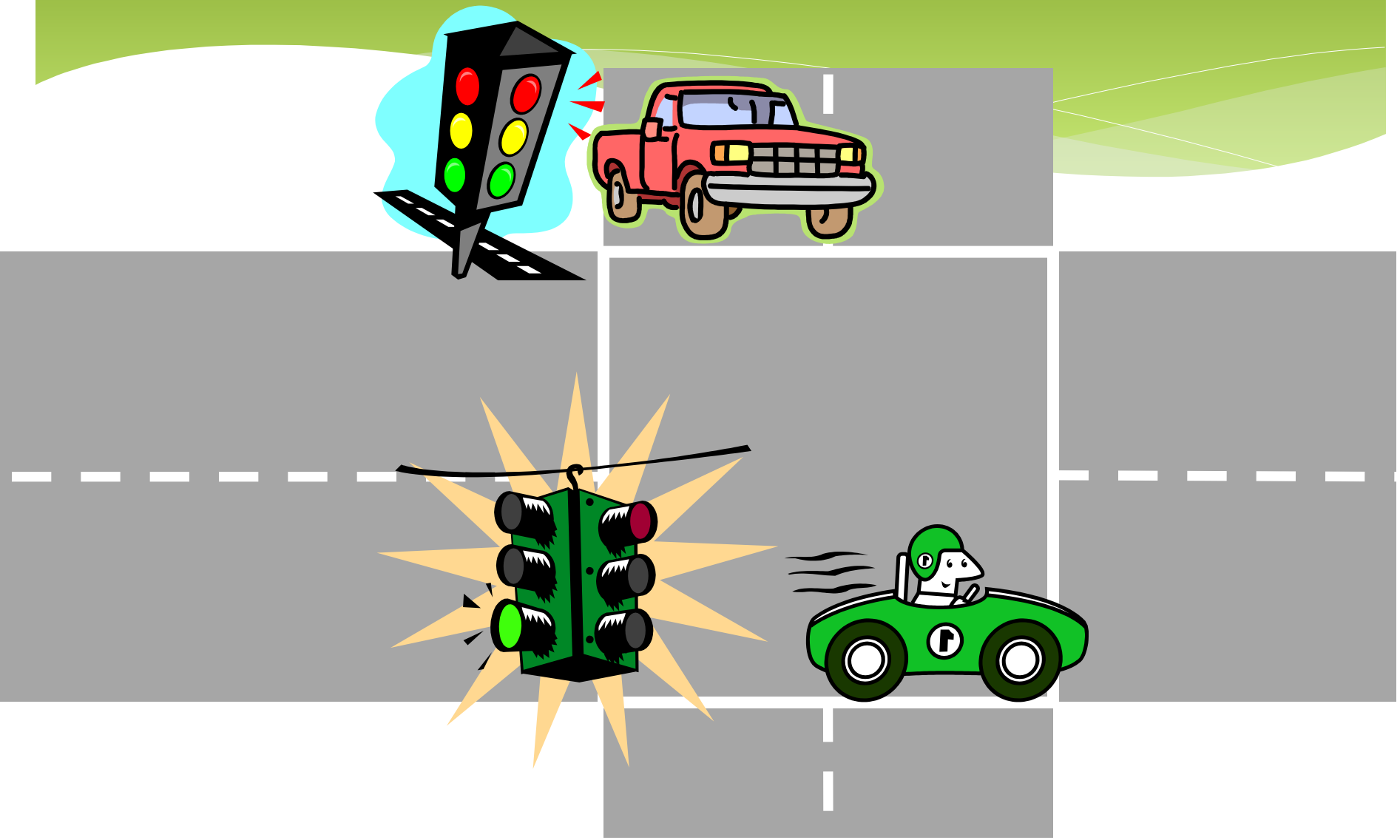
# Intersection Metaphor: Normal Case



# Intersection Metaphor: Normal Case

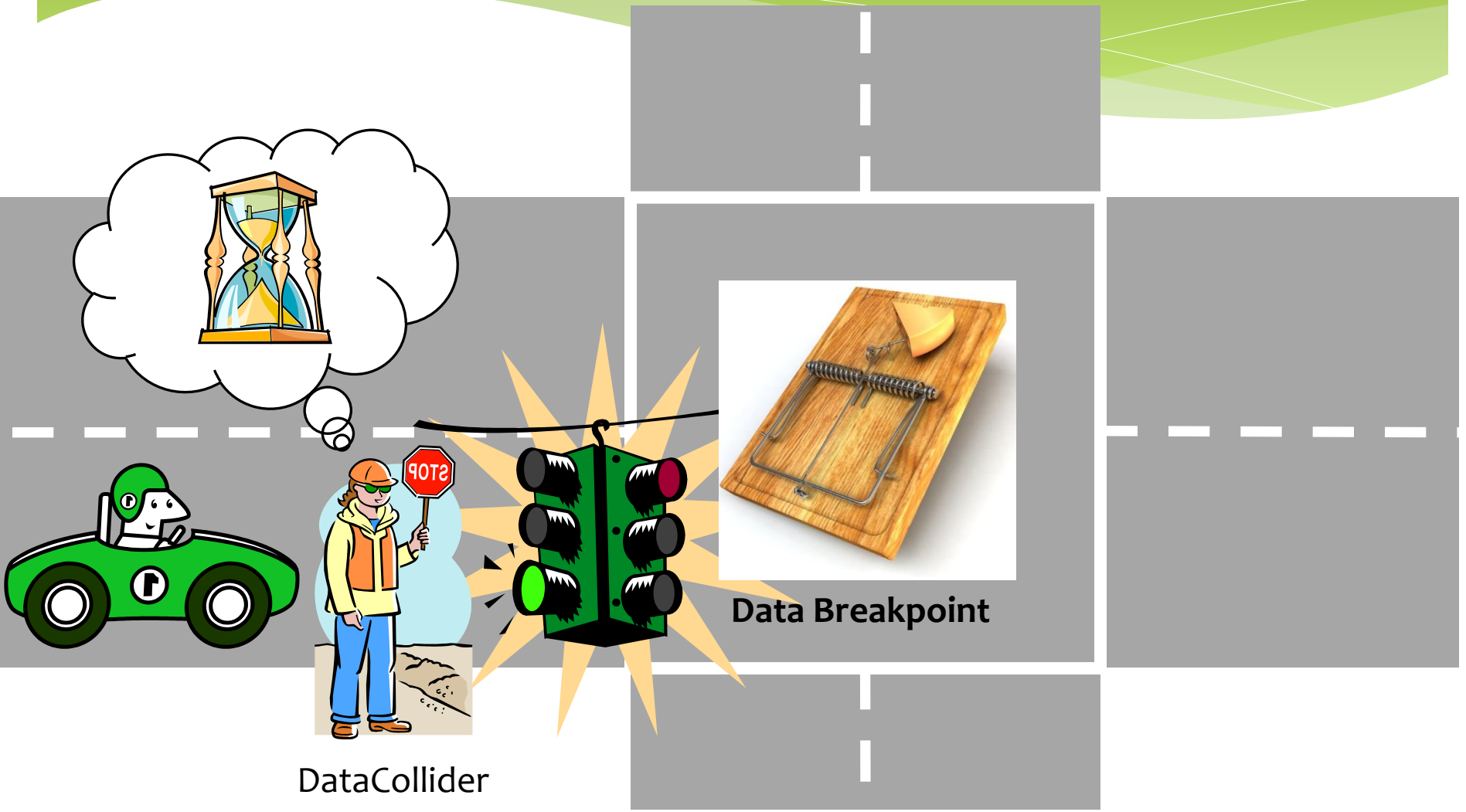


# Intersection Metaphor: Normal Case



# Intersection Metaphor: Data Race

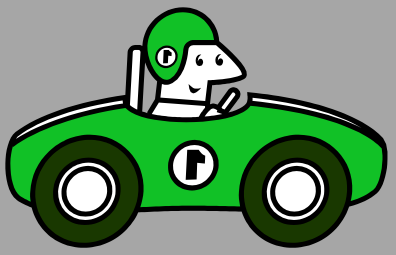
# Intersection Metaphor: Data Race



# Intersection Metaphor: Data Race

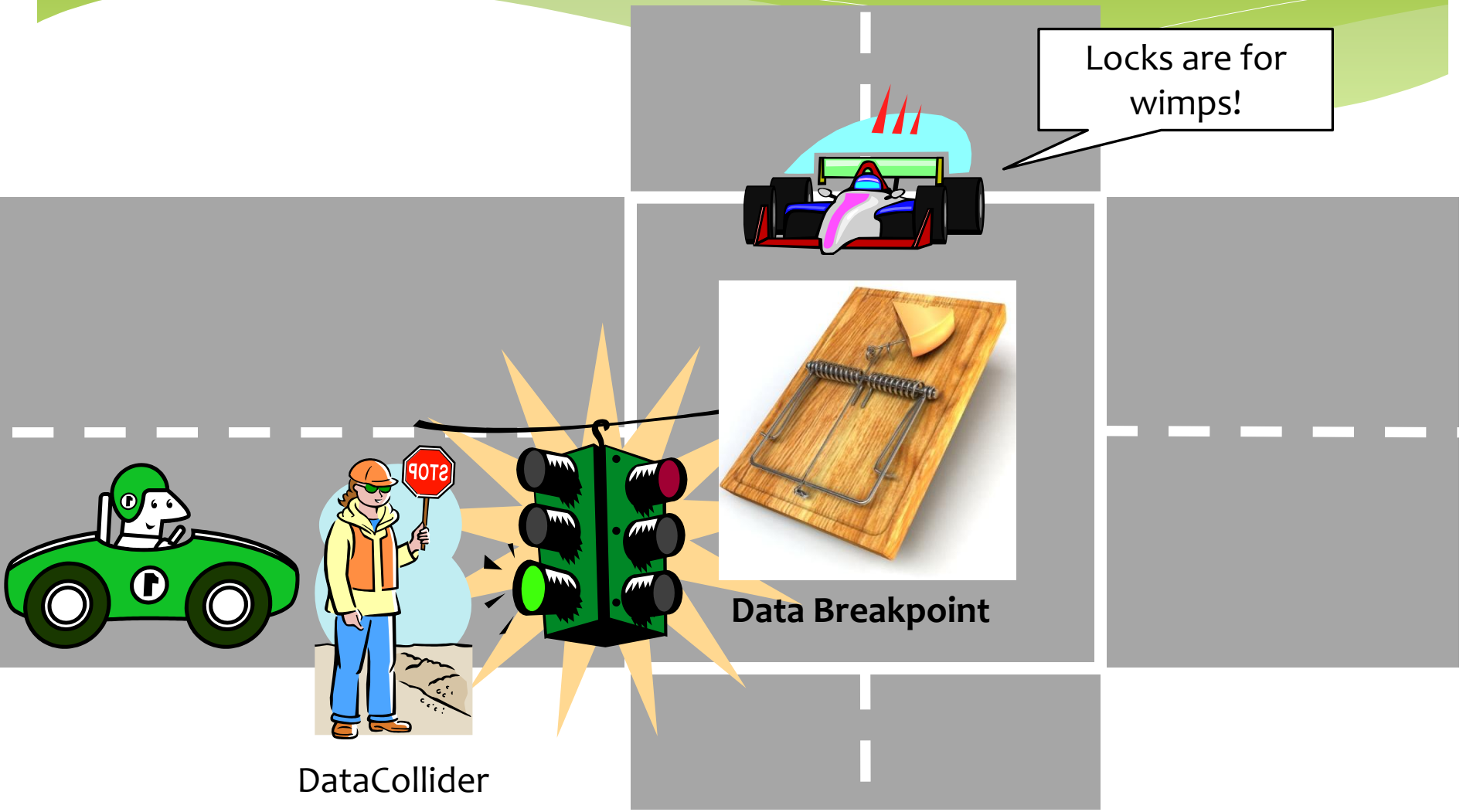
Thread B

Data Breakpoint

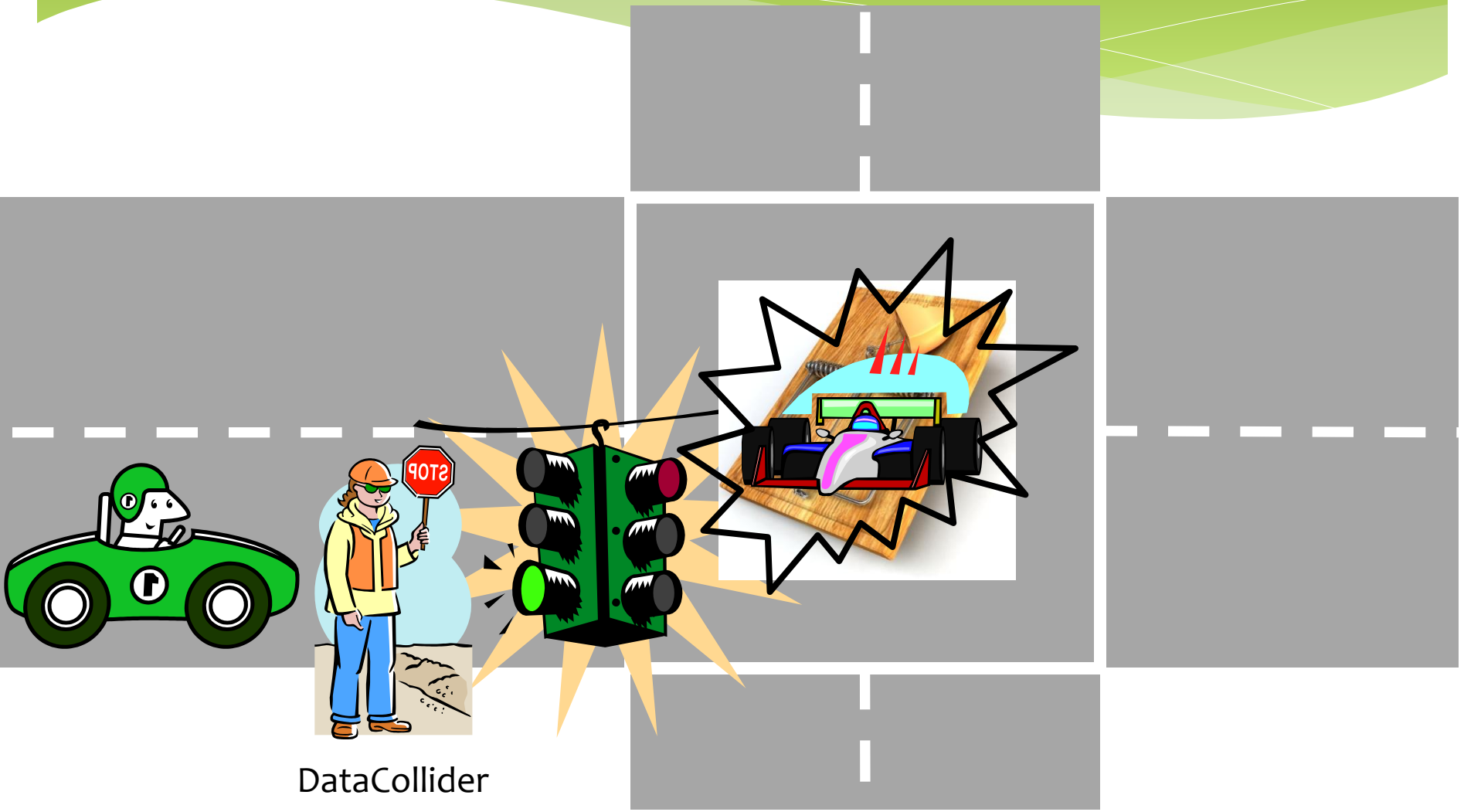



DataCollider

# Intersection Metaphor: Data Race

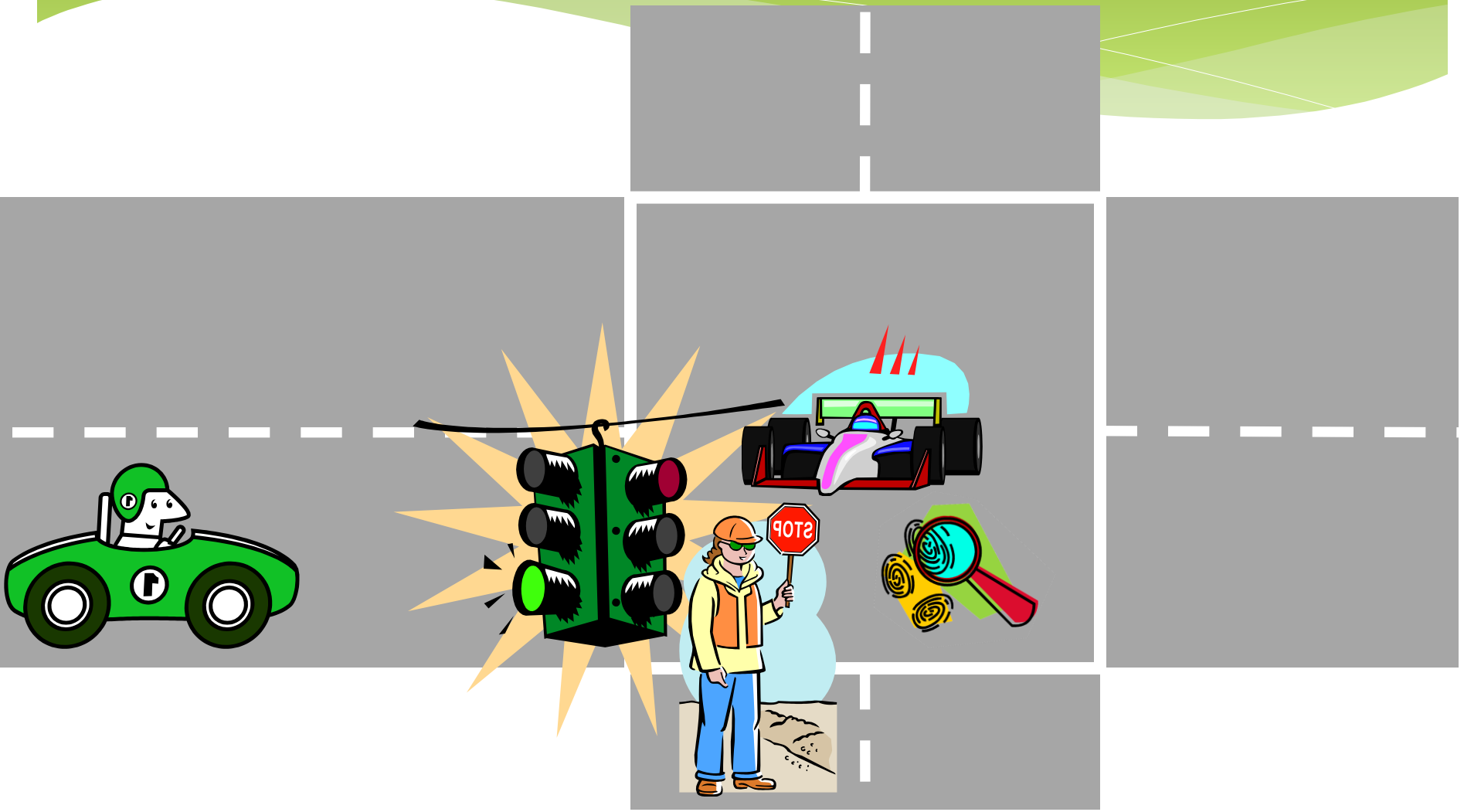


# Intersection Metaphor: Data Race

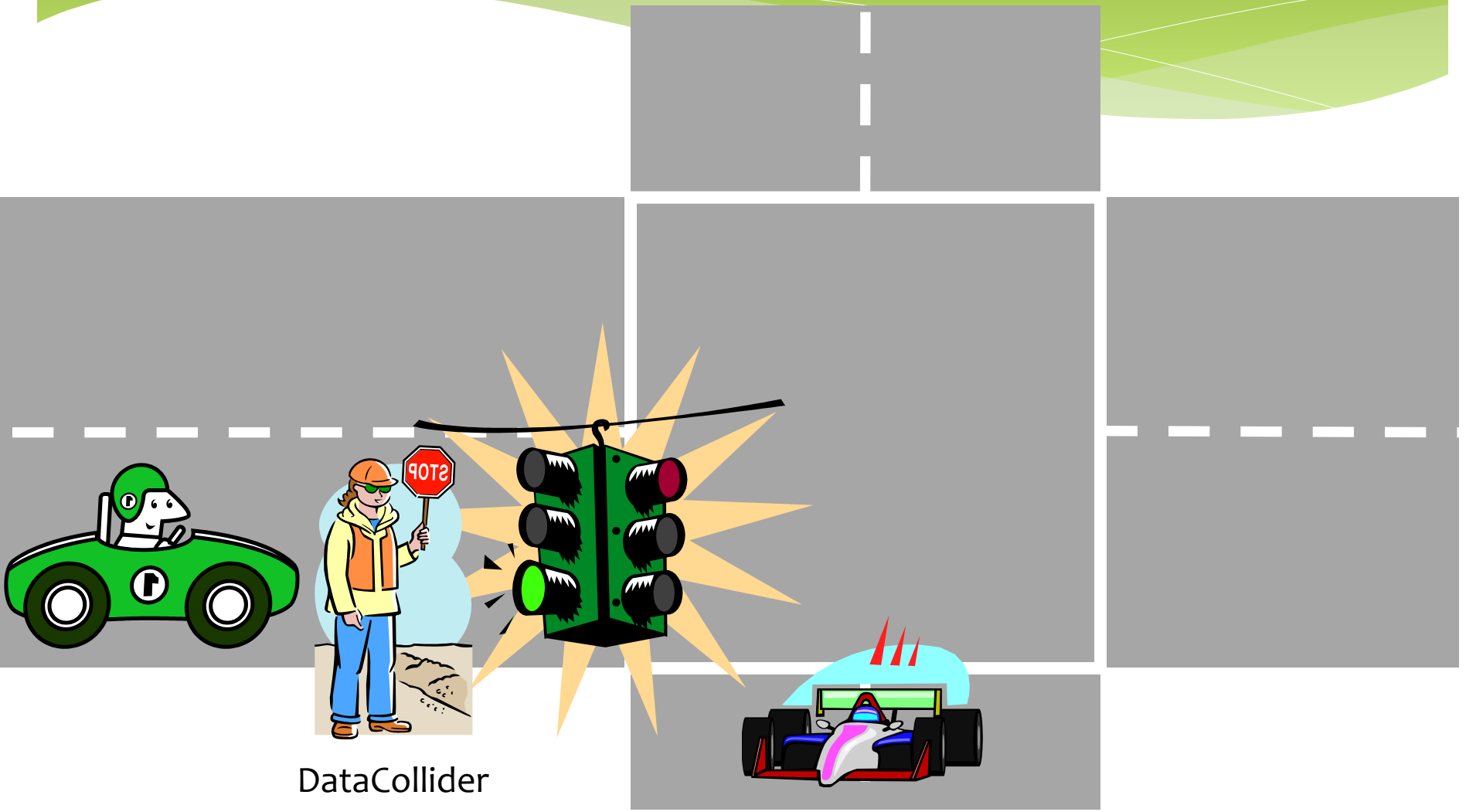



DataCollider

# Intersection Metaphor: Data Race

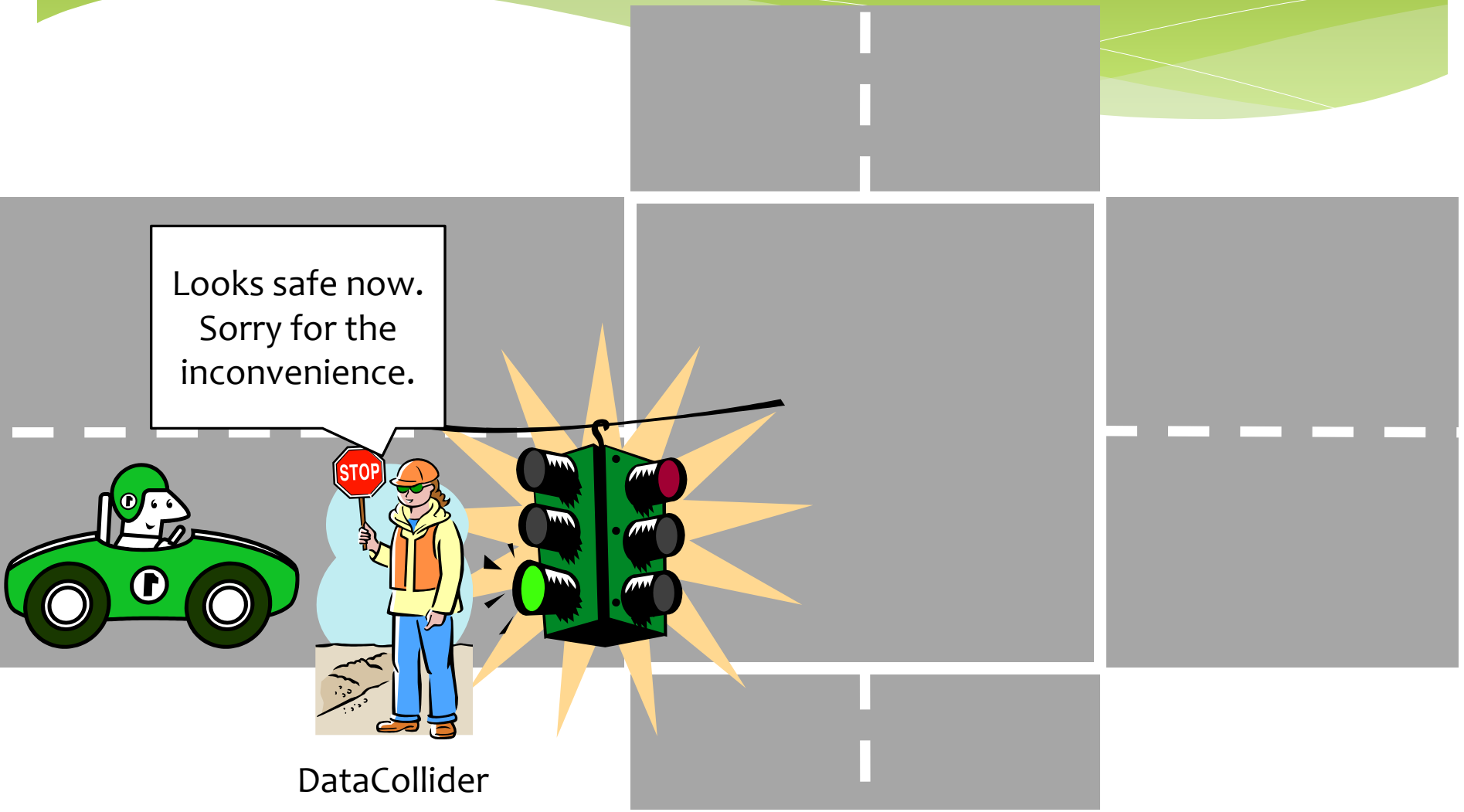


# Intersection Metaphor: Data Race

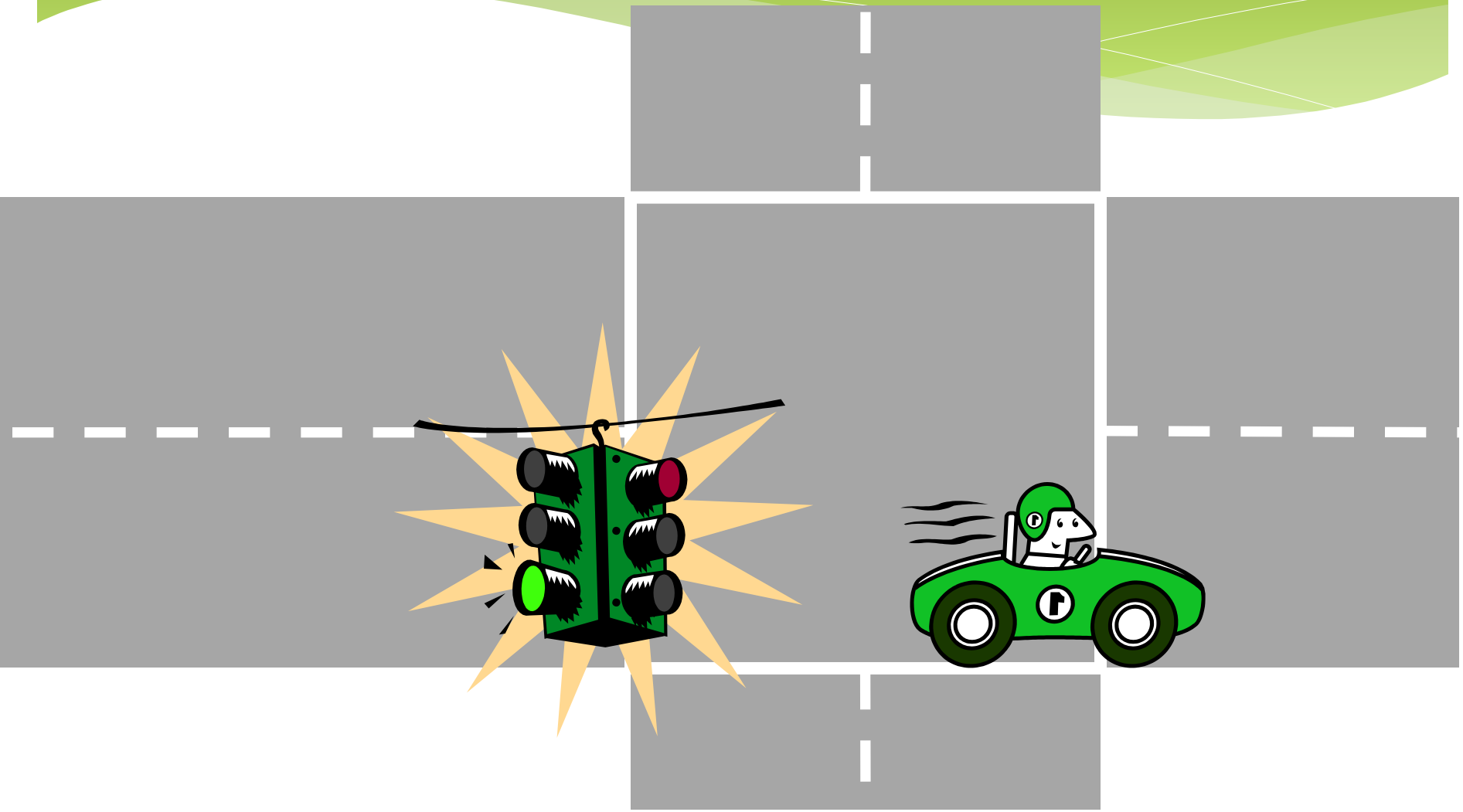



DataCollider

# Intersection Metaphor: Data Race



# Intersection Metaphor: Data Race



# Implementation

# Sampling memory accesses with code breakpoints; part 1

## Process

1. Analyze target binary for memory access instructions.
2. Hook the breakpoint handler.
3. Set code breakpoints at a sampling of the memory access instructions.
4. Begin execution.

## Advantages

* Zero base-overhead– no code breakpoints means only the original code is running.
* No annotations required – only symbols.

# Sampling memory accesses with code breakpoints, part 2

```
OnCodeBreakpoint( pc ) {

  // disassemble the instruction at pc
  (loc, size, isWrite) = disasm( pc );

  DetectConflicts(loc, size, isWrite);

  temp = read( loc, size );
  if ( isWrite )
    SetDataBreakpointRW( loc, size );
  else
    SetDataBreakpointW( loc, size );

  delay();

  ClearDataBreakpoint( loc, size );

  temp' = read( loc, size );
  if(temp != temp' || data breakpoint hit)
     ReportDataRace( );
}
```

## Advantages

- Setting the data breakpoint will catch the colliding thread in the act.
- This provides much more actionable debugging information.

# Sampling memory accesses with code breakpoints, part 2

```
OnCodeBreakpoint( pc ) {

  // disassemble the instruction at pc
  (loc, size, isWrite) = disasm( pc );

  DetectConflicts(loc, size, isWrite);

  temp = read( loc, size );
  if ( isWrite )
    SetDataBreakpointRW( loc, size );
  else
    SetDataBreakpointW( loc, size );

  delay();

  ClearDataBreakpoint( loc, size );

  temp' = read( loc, size );
  if(temp != temp' || data breakpoint hit)
     ReportDataRace( );
}
```

## Advantages

- The additional re-read approach helps detect races caused by:
  - Hardware interaction via DMA
  - Physical memory that has multiple virtual mappings

# Results

# Results: bucketization of races

* Most of dynamic data races are benign
* Many have the potential to be heuristically pruned
* Much room to investigate and develop in this area

| Data Race Category | | Count |
|---|---|---|
| Benign – Heuristically Pruned | Statistic Counter | 52 |
| | Safe Flag Update | 29 |
| | Special Variable | 5 |
| | Subtotal | 86 |
| Benign – Manually Pruned | Double-check locking | 8 |
| | Volatile | 8 |
| | Write Same Value | 1 |
| | Other | 1 |
| | Subtotal | 18 |
| Real | Confirmed | 5 |
| | Investigating | 4 |
| | Subtotal | 9 |
| Total | | 113 |

# Results: bugs found

* 25 confirmed bugs in the Windows OS have been found
* 8 more are still pending investigation

| Data Races Reported | Count |
|---|---|
| Fixed | 12 |
| Confirmed and Being Fixed | 13 |
| Under Investigation | 8 |
| Harmless | 5 |
| Total | 38 |

# Windows case study #2

Thread A

```
Connection->Initialized = TRUE;

or byte ptr [esi+70h],1
```

Thread B

```
Connection->QueuedForClosing = 1;

or byte ptr [esi+70h],2
```

```
struct CONNECTION {
  UCHAR Initialized : 1;
  UCHAR QueuedForClosing : 1;
};
```

This data race was found by using DataCollider on a test machine that was running a multi-threaded fuzzing test. It has been fixed.

# Windows case study #3

Thread A (owns SpinLock)

```
parentFdoExt->idleState = newState;
```

Thread B

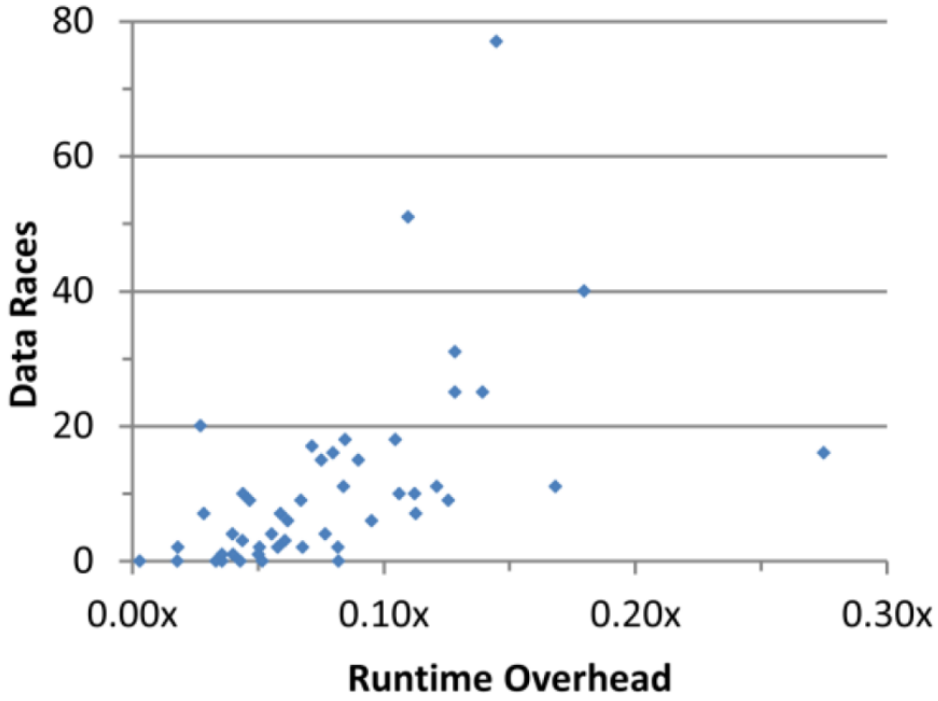
```
parentFdoExt->idleState = newState;
```

```
VOID ChangeIdleState(
  FDO_IDLE_STATE newState,
  BOOLEAN acquireLock);
```

This data race was found by using DataCollider on a test machine that was running a PnP stress test. In certain circumstances, ChangeIdleState was being called with acquireLock==FALSE even though the lock was not already acquired.

# Results: Scalability

* By using the code breakpoint method, we can see that data races can be found with as little as 5% overhead
* The user can effectively adjust the balance between races found and overhead incurred



# Future Work

* Better methods for prioritizing benign vs. non-benign races
  * Statistical analysis? Frequency?
* Apply algorithm to performance issues
  * True data sharing
  * False data sharing = data race “near miss”

# Demo

# Summary

* **DataCollider** can detect data races
  * with no false data races,
  * with zero base-overhead,
  * in kernel mode,
  * and find real product bugs.

We're hiring! ☺ jerick@microsoft.com

# DataCollider Original Prototype

## Original Algorithm

```
OnMemoryAccess( byte* Addr)
{
   if(rand() % 50 != 0) return;
   byte b = *Addr;
   int count = rand() % 1000;
   while(count--) {
       if(b != *Addr) Breakpoint();
   }
}
```

* *"If the memory a thread is accessing changes, then a data race could have occurred."*
* Used an internal tool to inject code into existing binaries
* Written without knowledge of lockset or happens-before approaches

# False vs. **benign** example

```
MyLockAcquire() {
  while(0 !=
    InterlockedExchange(&gLock, 1)
  );
}
```

```
MyLockAcquire() {
  while(0 !=
    InterlockedExchange(&gLock, 1)
  );
}
```

Thread A

```
MyLockAcquire();

gReferenceCount++;

MyLockRelease();

gStatisticsCount++;
```

Thread B

```
MyLockAcquire();

gReferenceCount++;

MyLockRelease();

gStatisticsCount++;
```

# Improvements: Actionable data

* Issue:
  * Fixing a bug when one only has knowledge of one side of the race can be very time consuming because it would often require deep code review to find what the colliding culprit could be.
* Solution:
  * Make use of the hardware debug registers to cause a processor trap to occur on race.

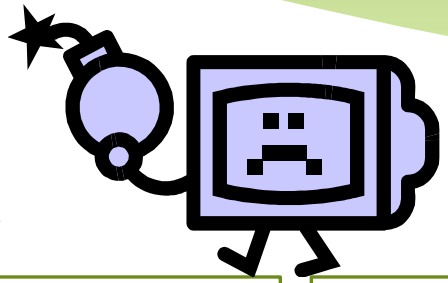
# Improvements: Highly scalable

* **Issue:**
  * Injecting code into a binary introduced an unavoidable non-trivial base overhead.
* **Solution:**
  * Dispose of injecting code into binaries entirely. Sample memory accesses via code breakpoints instead.

# False vs. benign vs. real definitions

* **False data race**
  * A data race that is claimed to exist by a data race detection tool, but, in reality, cannot occur.
* **Benign data race**
  * A data race that can and does occur, but is intended to happen as part of normal program execution. E.g. synchronization primitives usually have benign data races as the key to their operation.
* **Real data race**
  * A data race that is not intended or causes unintended consequences. If the developer were to write the code again, he/she would do so differently.

# False vs. benign vs. **real** example



Thread A

```
MyLockAcquire();

gReferenceCount++;

MyLockRelease();

gStatisticsCount++;
```

Thread B

```
MyLockAcquire();

gReferenceCount++;

MyLockRelease();

gStatisticsCount++;

gReferenceCount++;
```