Android Concurrency: Challenges of Concurrency

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Module

- Understand key complexities of concurrent software for mobile devices

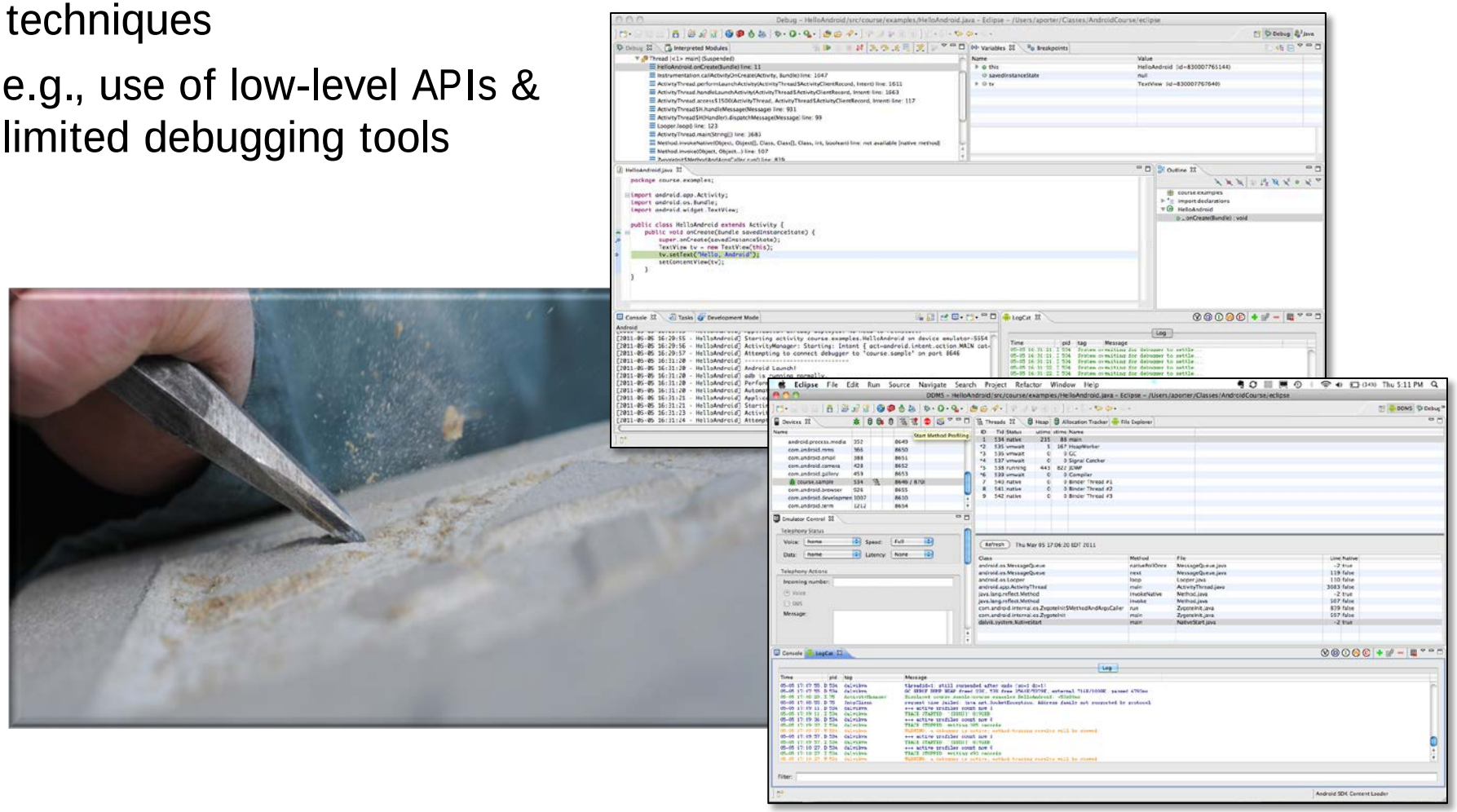
en.wikipedia.org/wiki/Gordian_Knot

Key Complexities in Concurrent Software

- *Accidental complexities* stem from limitations with development tools & techniques

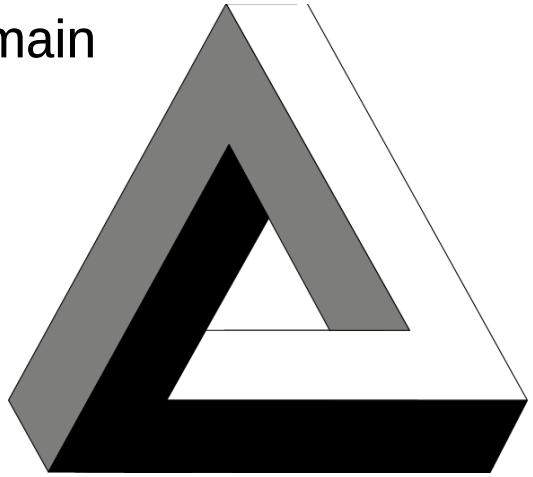
Key Complexities in Concurrent Software

- *Accidental complexities* stem from limitations with development tools & techniques
  - e.g., use of low-level APIs & limited debugging tools



Key Complexities in Concurrent Software

- *Accidental complexities* stem from limitations with development tools & techniques
- *Inherent complexities* stem from fundamental domain challenges
  - e.g., scheduling, synchronization, & deadlock



Key Complexities in Concurrent Software

- *Accidental complexities* stem from limitations with development tools & techniques
- *Inherent complexities* stem from fundamental domain challenges
  - e.g., scheduling, synchronization, & deadlock

Accidental Complexities for Concurrent Software

Accidental Complexities for Concurrent Software

- Low-level APIs
  - Tedious, error-prone, & non-portable

Accidental Complexities for Concurrent Software

- Low-level APIs

```
typedef struct
{ char message_[20]; int thread_id_; } PARAMS;

void *print_hello_world (void *ptr) {
  PARAMS *params = (PARAMS *) ptr;
  printf ("%s from thread %d\n",
          params->message_, params->thread_id_);
}

int main (void) {
  pthread_t thread; PARAMS params;
  params.thread_id_ = 1; strcpy (params.message_, "Hello World");


  pthread_create (&thread, 0, &print_hello_world,
                  (void *) &params);
  /* ... */
  pthread_join(thread, 0);
  return 0;
}
```

bionic/libc/bionic/pthread.c has Android's pthreads implementation

Accidental Complexities for Concurrent Software

- Low-level APIs

```
typedef struct
{ char message_[20]; int thread_id_; } PARAMS;

void *print_hello_world (void *ptr) {
  PARAMS *params = (PARAMS *) ptr;
  printf ("%s from thread %d\n",
          params->message_, params->thread_id_);
}

int main (void) {
  pthread_t thread; PARAMS params;
  params.thread_id_ = 1; strcpy (params.message_, "Hello World");


  pthread_create (&thread, 0, &print_hello_world,
                  (void *) &params);
  /* ... */
  pthread_join(thread, 0);
  return 0;
}
```

Pointer-to-function

Accidental Complexities for Concurrent Software

- Low-level APIs

```
typedef struct
{ char message_[20]; int thread_id_; } PARAMS;

void *print_hello_world (void *ptr) {
  PARAMS *params = (PARAMS *) ptr;
  printf ("%s from thread %d\n",
          params->message_, params->thread_id_);
}

int main (void) {
  pthread_t thread; PARAMS params;
  params.thread_id_ = 1; strcpy (params.message_, "Hello World");


  pthread_create (&thread, 0, &print_hello_world,
                  (void *) &params);
  /* ... */
  pthread_join(thread, 0);
  return 0;
}
```

Cast from void *

Cast to void *

Accidental Complexities for Concurrent Software

- Low-level APIs

```
typedef struct
{ char message_[20]; int thread_id_; } PARAMS;

void *print_hello_world (void *ptr) {
  PARAMS *params = (PARAMS *) ptr;
  printf ("%s from thread %d\n",
          params->message_, params->thread_id_);
}

int main (void) {
  pthread_t thread; PARAMS params;
  params.thread_id_ = 1; strcpy (params.message_, "Hello World");


  pthread_create (&thread, 0, &print_hello_world,
                  (void *) &params);
  /* ... */
  pthread_join(thread, 0);
  return 0;
}
```

"Quasi-typed" thread handle

Accidental Complexities for Concurrent Software

- Low-level APIs

```
typedef struct
{ char message_[20]; int thread_id_; } PARAMS;

void *print_hello_world (void *ptr) {
  PARAMS *params = (PARAMS *) ptr;
  printf ("%s from thread %d\n",
          params->message_, params->thread_id_);
}

int main (void) {
  pthread_t thread; PARAMS params;
  params.thread_id_ = 1; strcpy (params.message_, "Hello World");
```

Not portable to non-POSIX platforms

```
  pthread_create (&thread, 0, &print_hello_world,
                  (void *) &params);
  /* ... */
  pthread_join(thread, 0);
  return 0;
}
```

Accidental Complexities for Concurrent Software

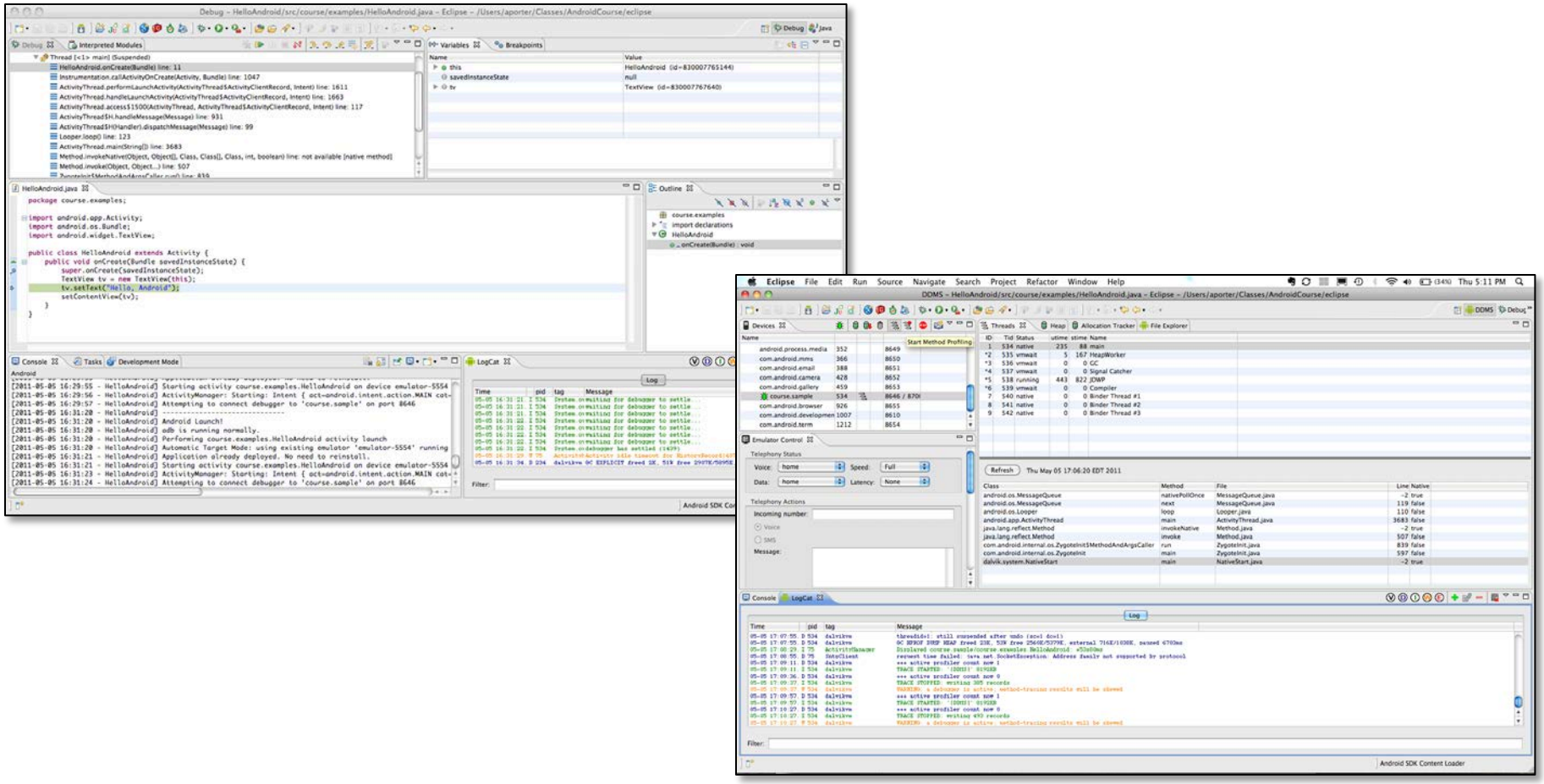
- Low-level APIs

```
typedef struct
{ char message_[20]; int thread_id_; } PARAMS;

void *print_hello_world (void *ptr) {
  PARAMS *params = (PARAMS *) ptr;
  printf ("%s from thread %d\n",
          params->message_, params->thread_id_);
}

int main (void) {
  pthread_t thread; PARAMS params;
  params.thread_id_ = 1; strcpy (params.message_, "Hello World");


  pthread_create (&thread, 0, &print_hello_world,
                  (void *) &params);
  /* ... */
  pthread_join(thread, 0);
  return 0;
}
```

Other C threading APIs have similar accidental complexities

Accidental Complexities for Concurrent Software

- Low-level APIs
- Limited debugging tools



Accidental Complexities for Concurrent Software

- Low-level APIs
- Limited debugging tools

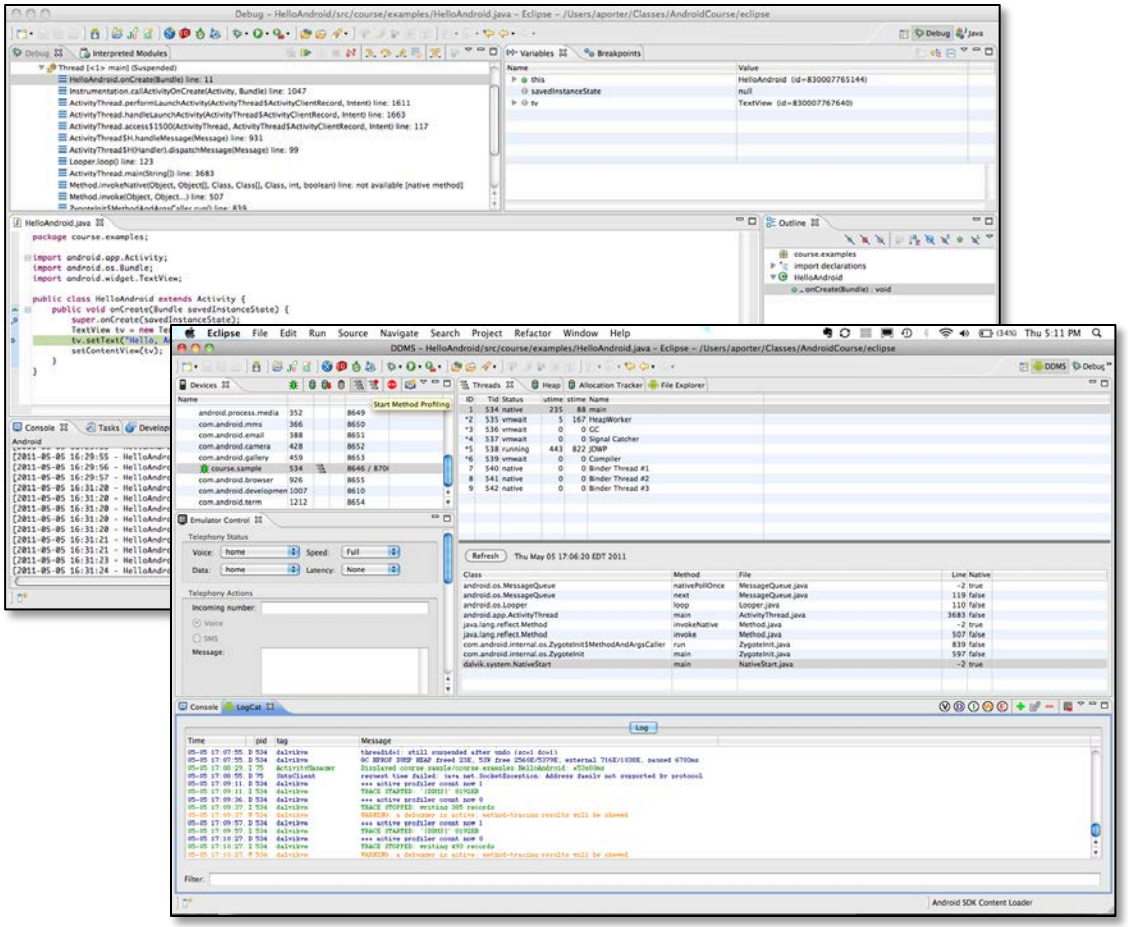
Accidental Complexities for Concurrent Software

- Low-level APIs
- Limited debugging tools

en.wikipedia.org/wiki/Trepanning has more on old school “debugging!”

Accidental Complexities for Concurrent Software

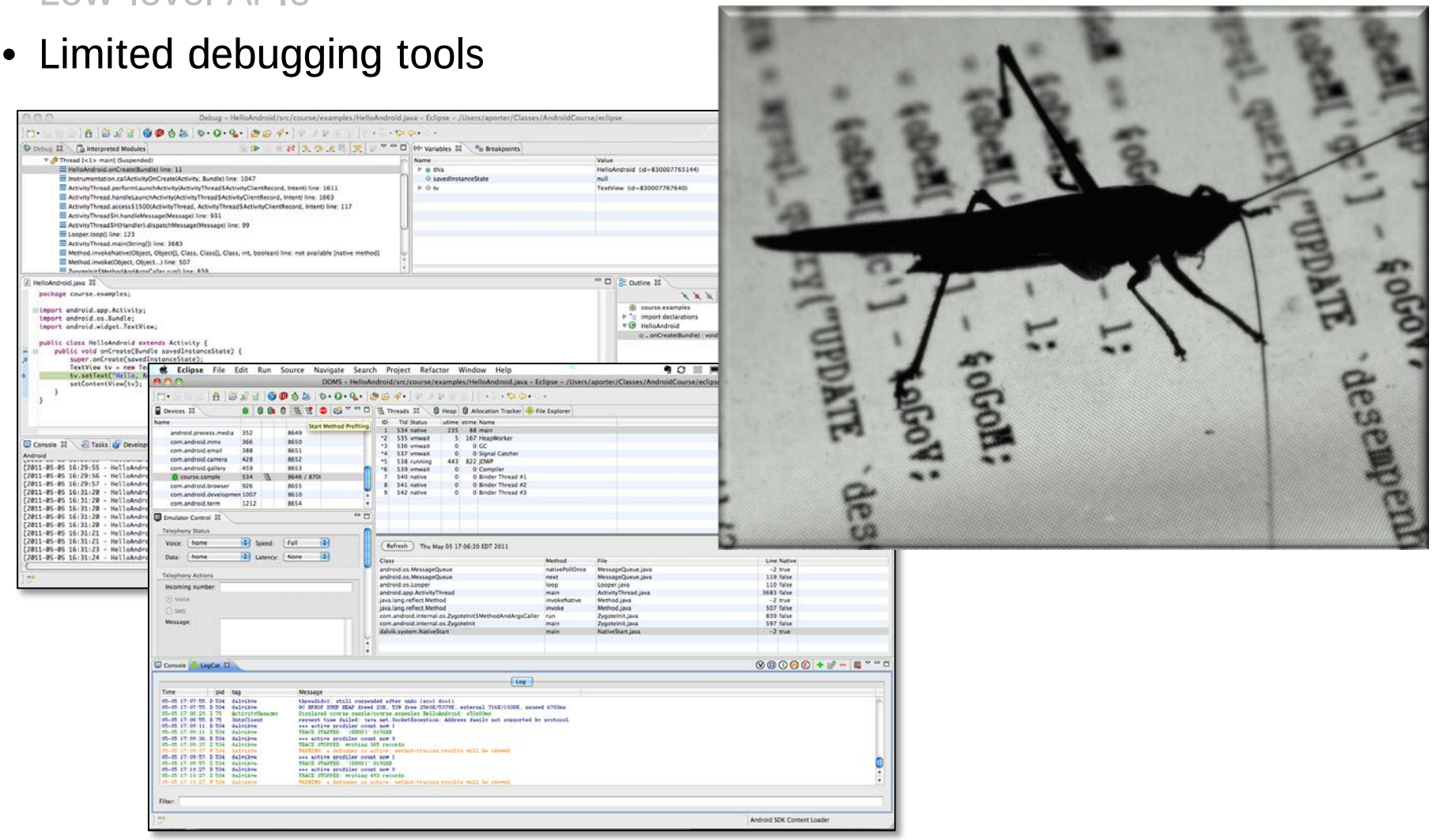
- Low-level APIs
- Limited debugging tools



en.wikipedia.org/wiki/Trepanning has more on old school “debugging!”

Accidental Complexities for Concurrent Software

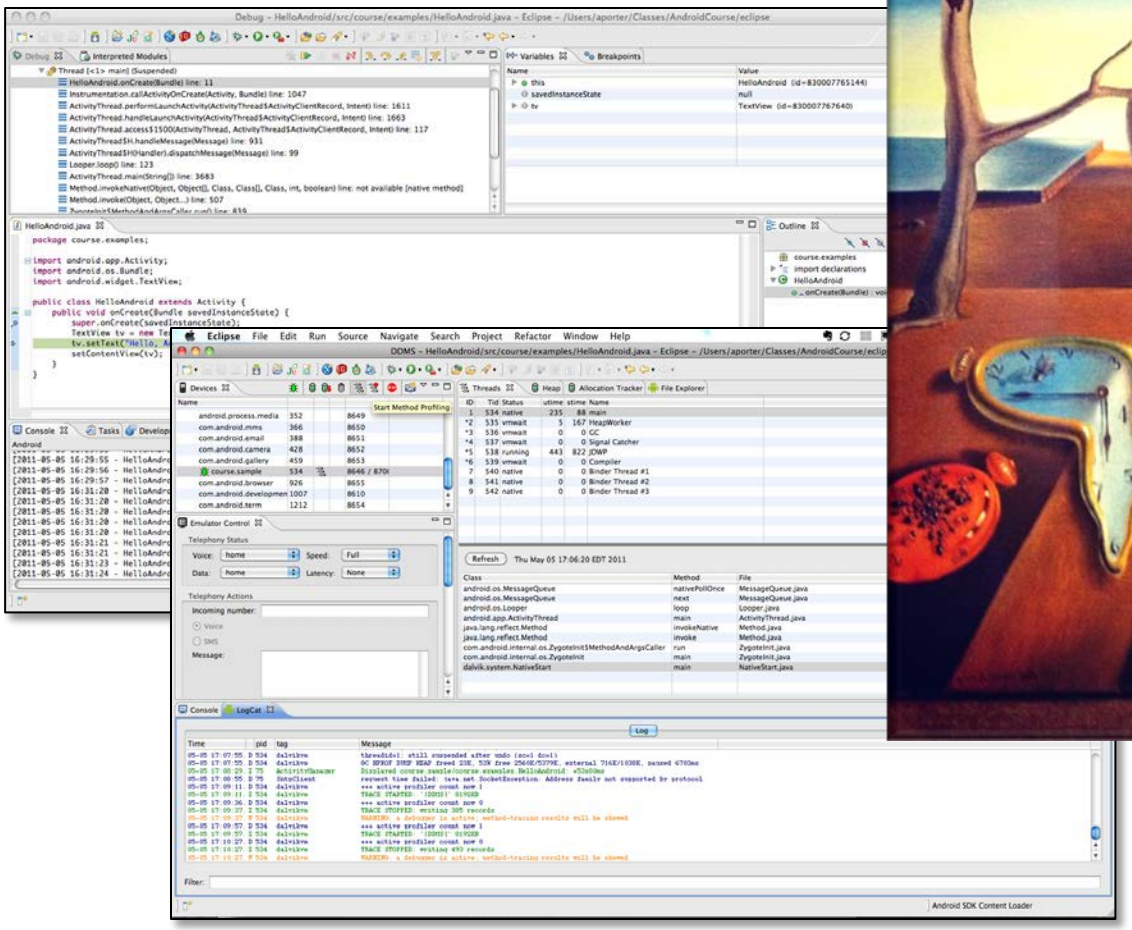
- Low-level APIs
- Limited debugging tools



en.wikipedia.org/wiki/Heisenbug has more info on Heisenbugs

Accidental Complexities for Concurrent Software

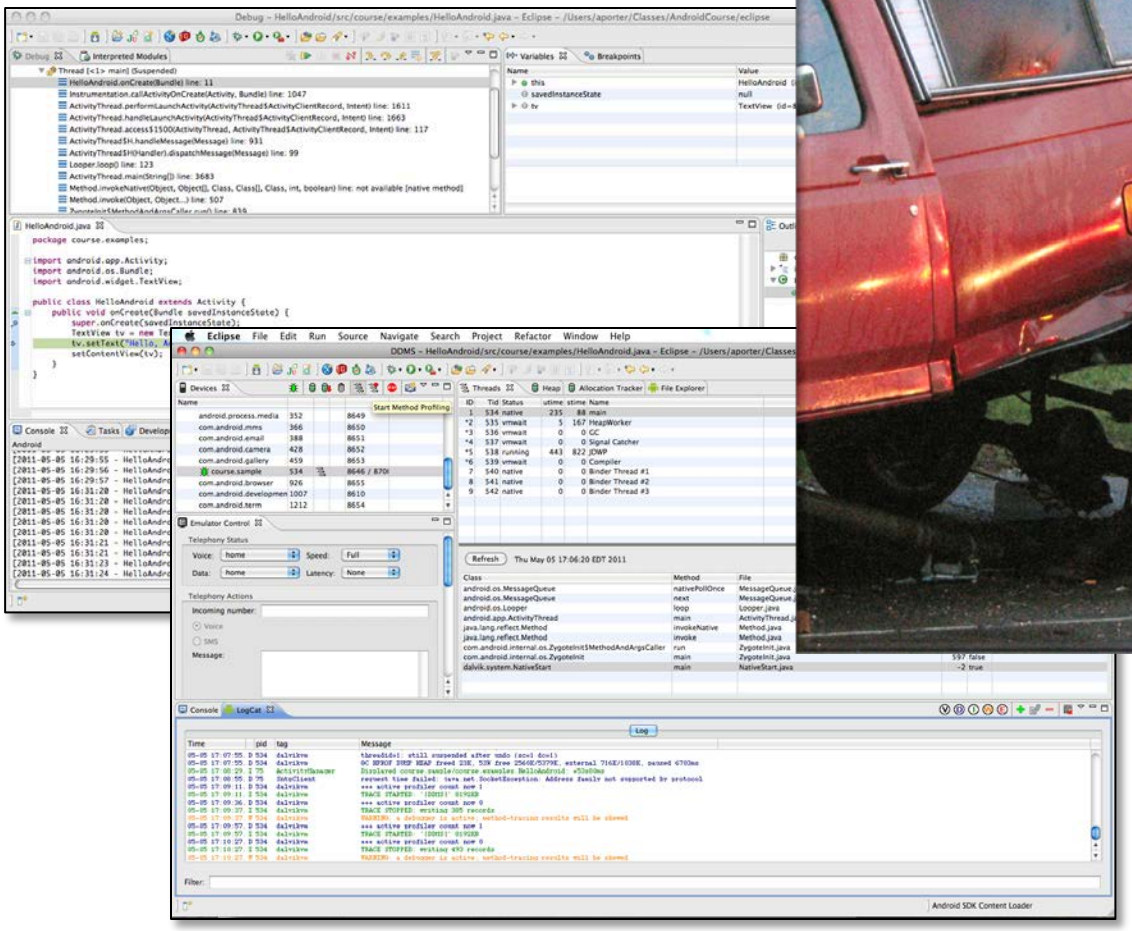
- Low-level APIs
- Limited debugging tools



See www.drdobbs.com/cpp/multithreaded-debugging-techniques/199200938

Accidental Complexities for Concurrent Software

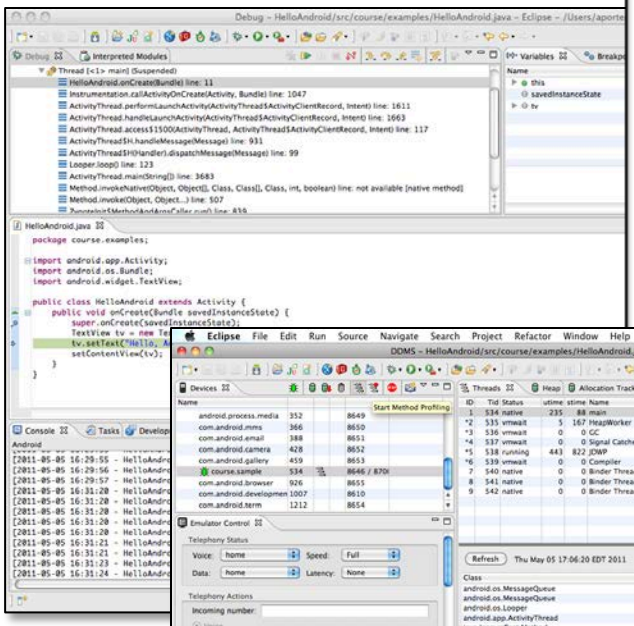
- Low-level APIs
- Limited debugging tools

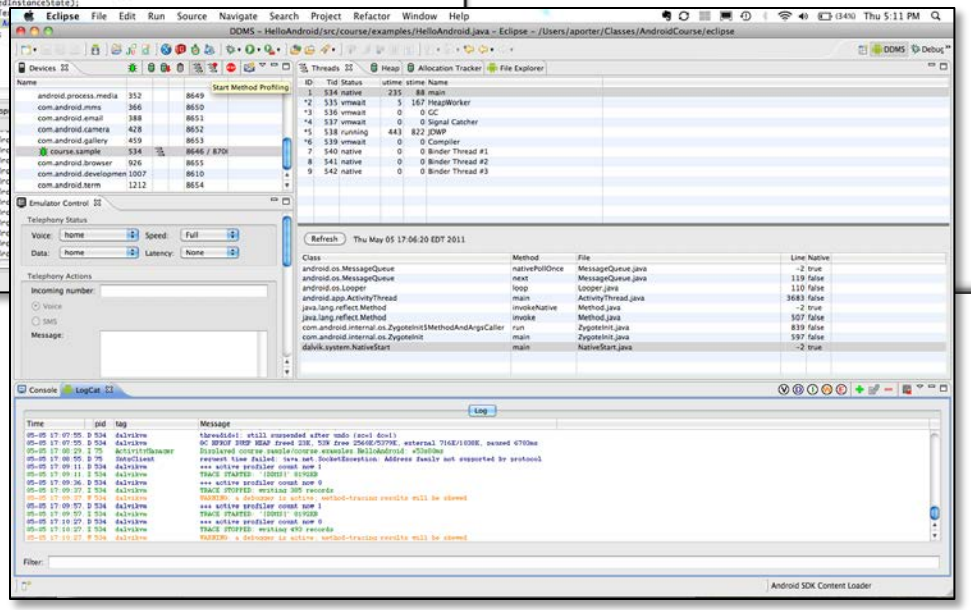


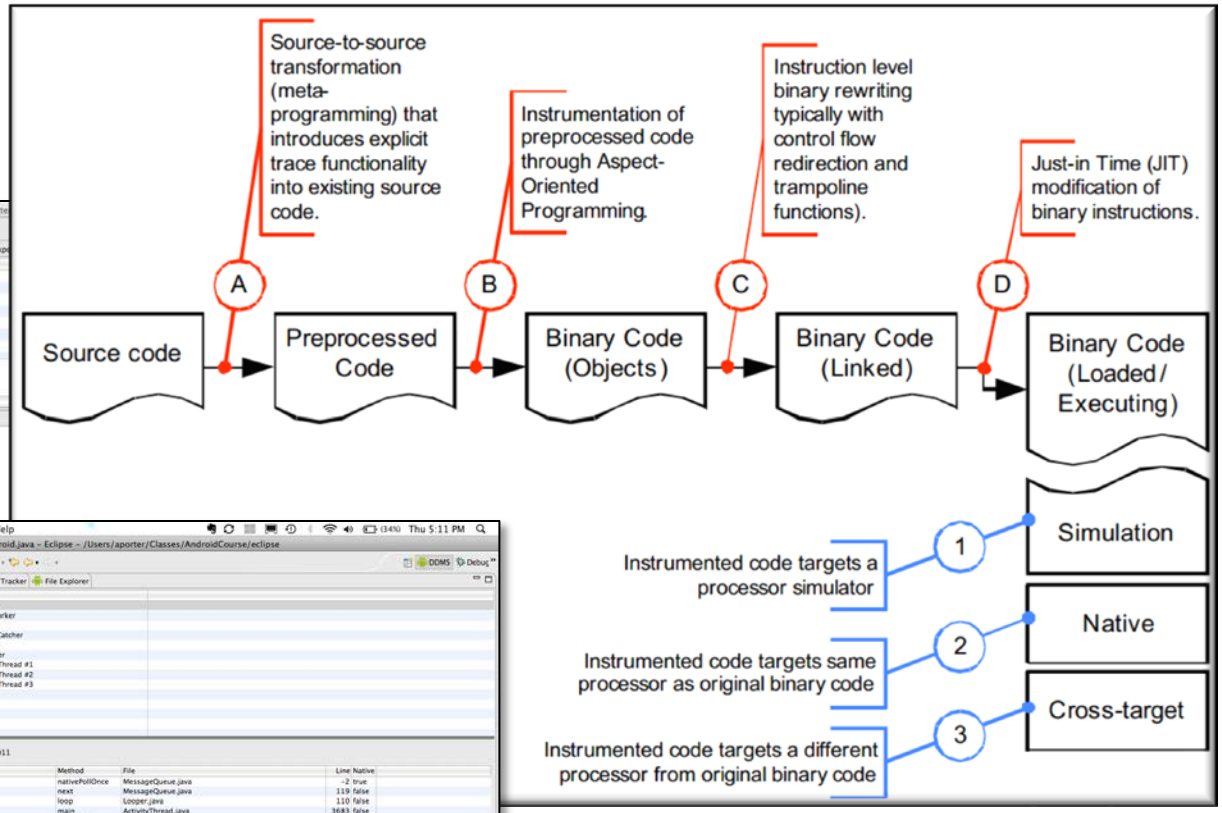
en.wikipedia.org/wiki/Race_condition has more on race conditions

Accidental Complexities for Concurrent Software

- Low-level APIs
- Limited debugging tools







See www.dre.vanderbilt.edu/~schmidt/PDF/DSIS.pdf & www.fluid.cs.cmu.edu

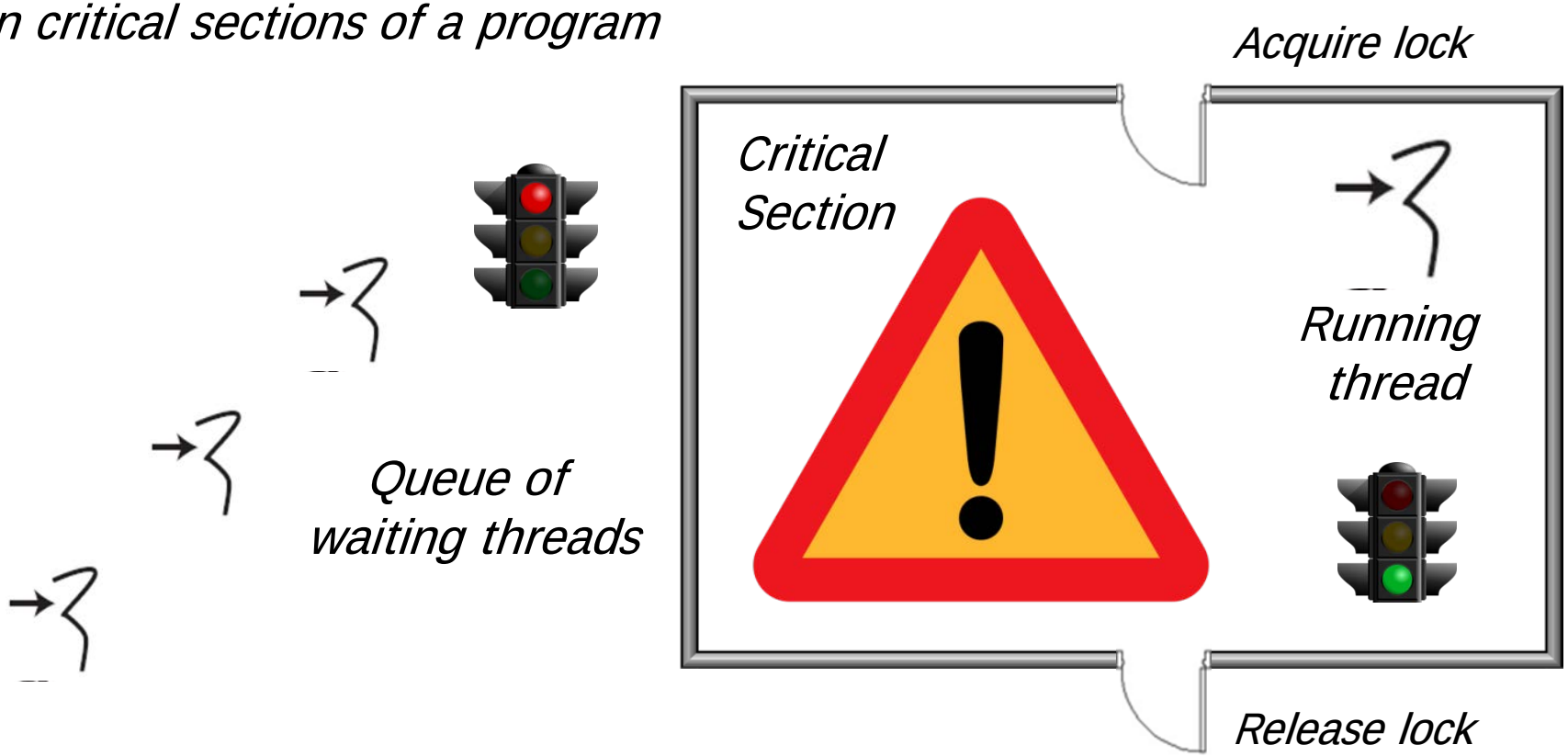
Inherent Complexities for Concurrent Software

Inherent Complexities for Concurrent Software

- Synchronization
- Scheduling

Inherent Complexities for Concurrent Software

- Synchronization
  - *Ensure that multiple concurrent threads don't simultaneously execute in critical sections of a program*

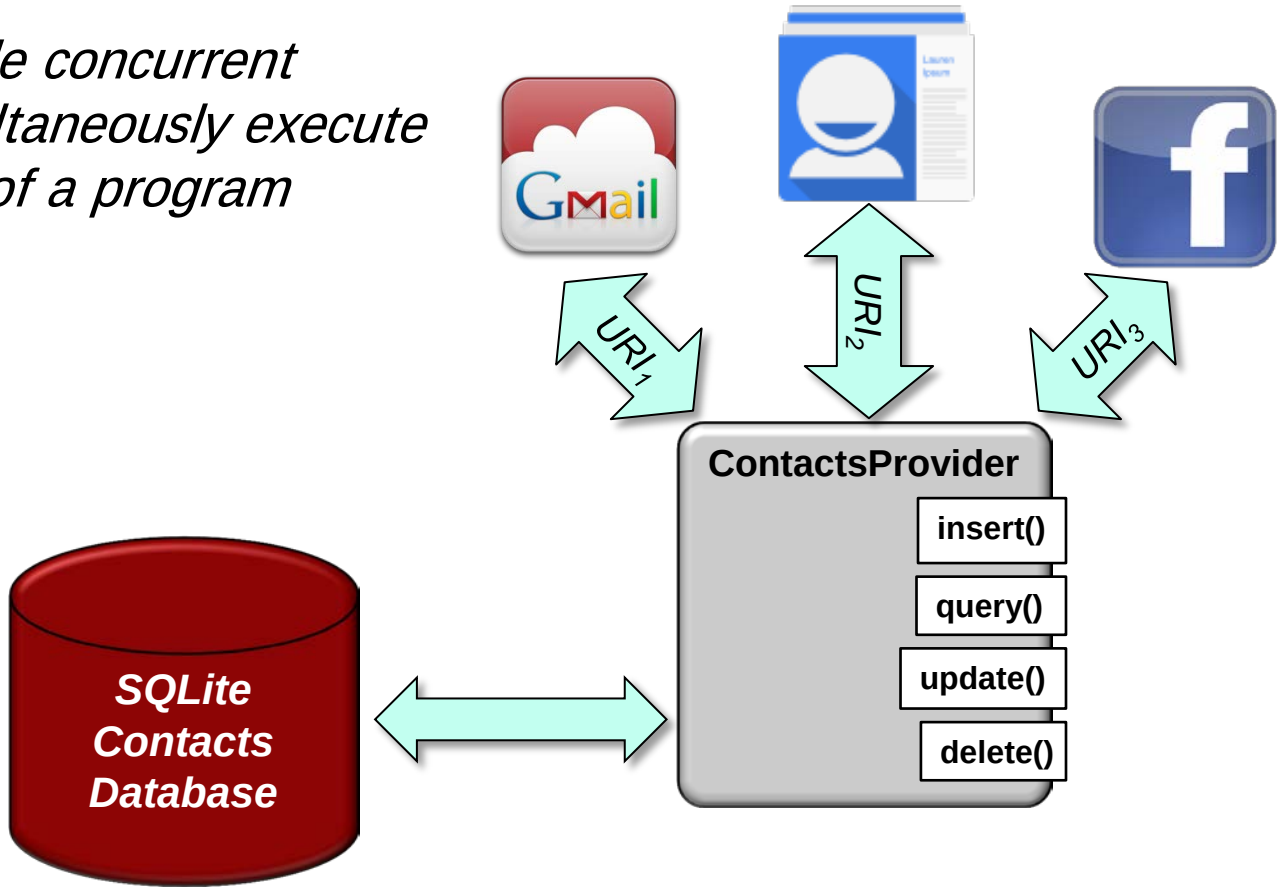


Inherent Complexities for Concurrent Software

- Synchronization
  - *Ensure that multiple concurrent threads don't simultaneously execute in critical sections of a program*

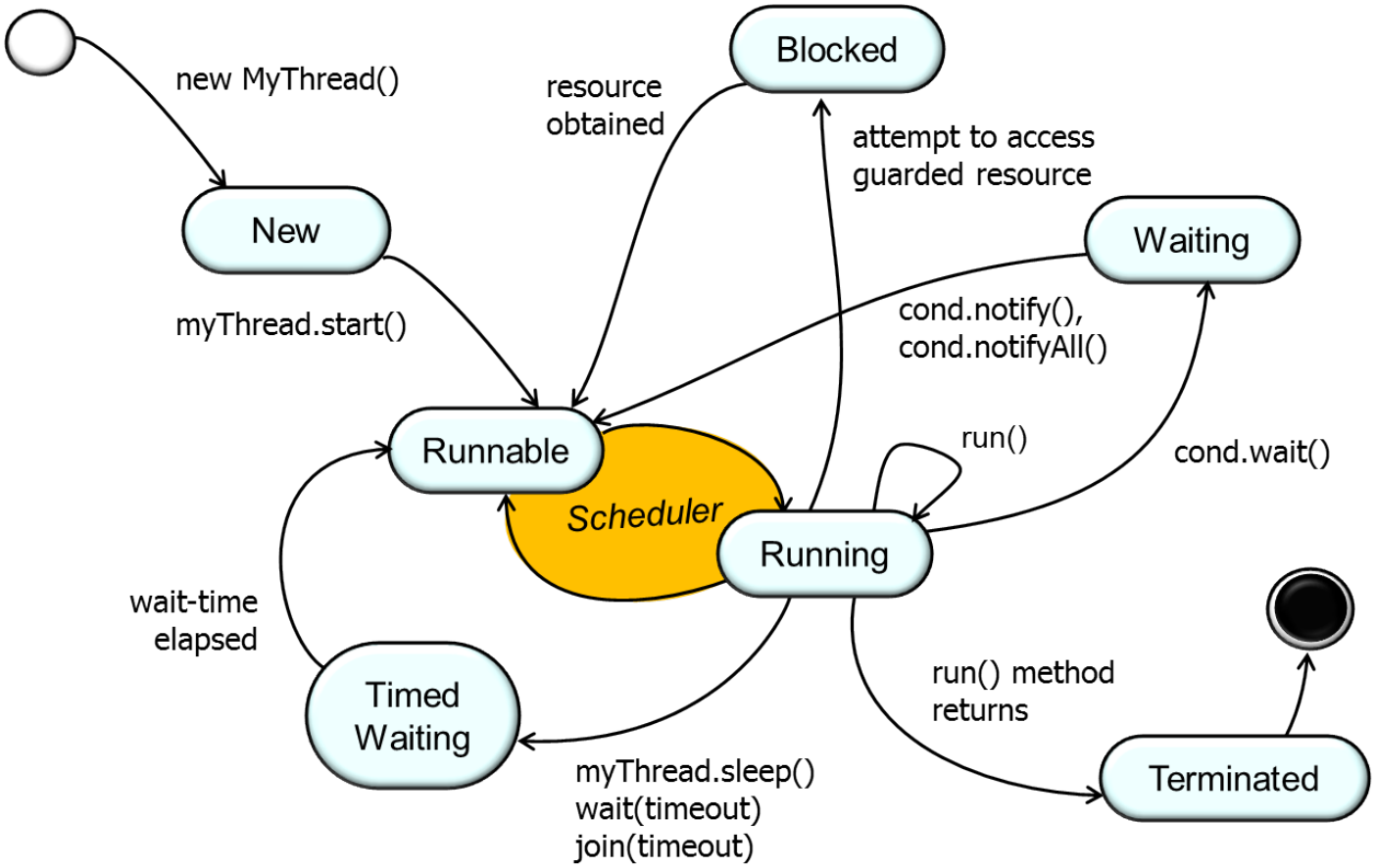
Inherent Complexities for Concurrent Software

- Synchronization
  - *Ensure that multiple concurrent threads don't simultaneously execute in critical sections of a program*



Inherent Complexities for Concurrent Software

- Synchronization
- Scheduling
  - *Ensure that threads, processes, or data flows are given proper access to system resources*

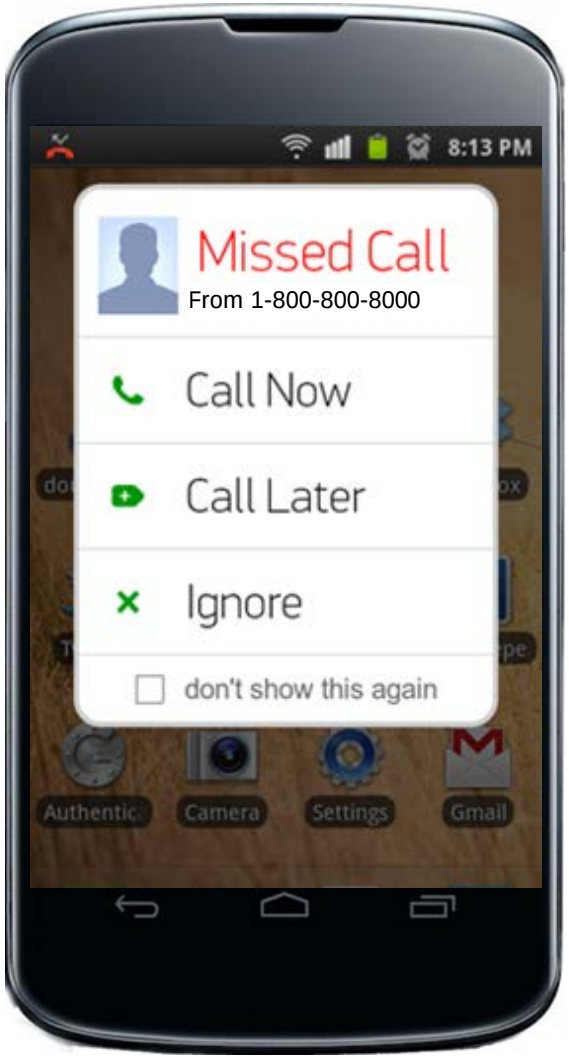


Inherent Complexities for Concurrent Software

- Synchronization
- Scheduling
  - *Ensure that threads, processes, or data flows are given proper access to system resources*

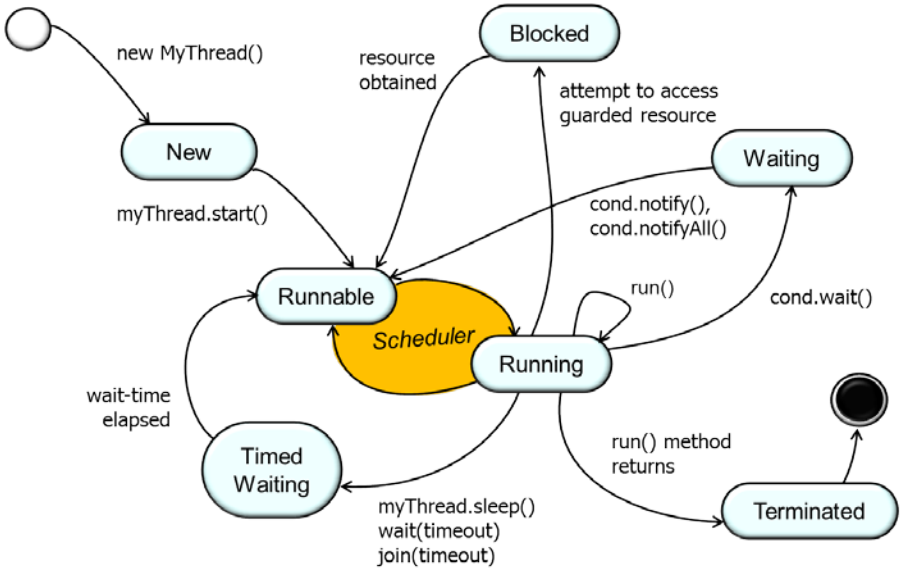
Inherent Complexities for Concurrent Software

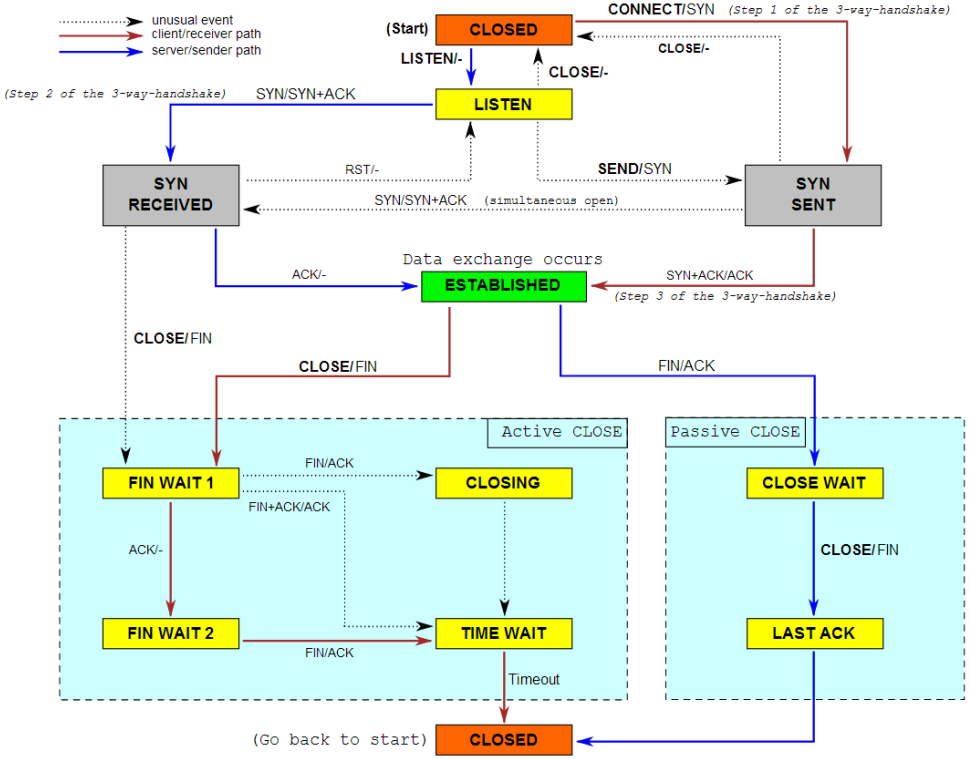
- Synchronization
- Scheduling
  - *Ensure that threads, processes, or data flows are given proper access to system resources*



Inherent Complexities for Concurrent Software

- Synchronization
- Scheduling
  - *Ensure that threads, processes, or data flows are given proper access to system resources*

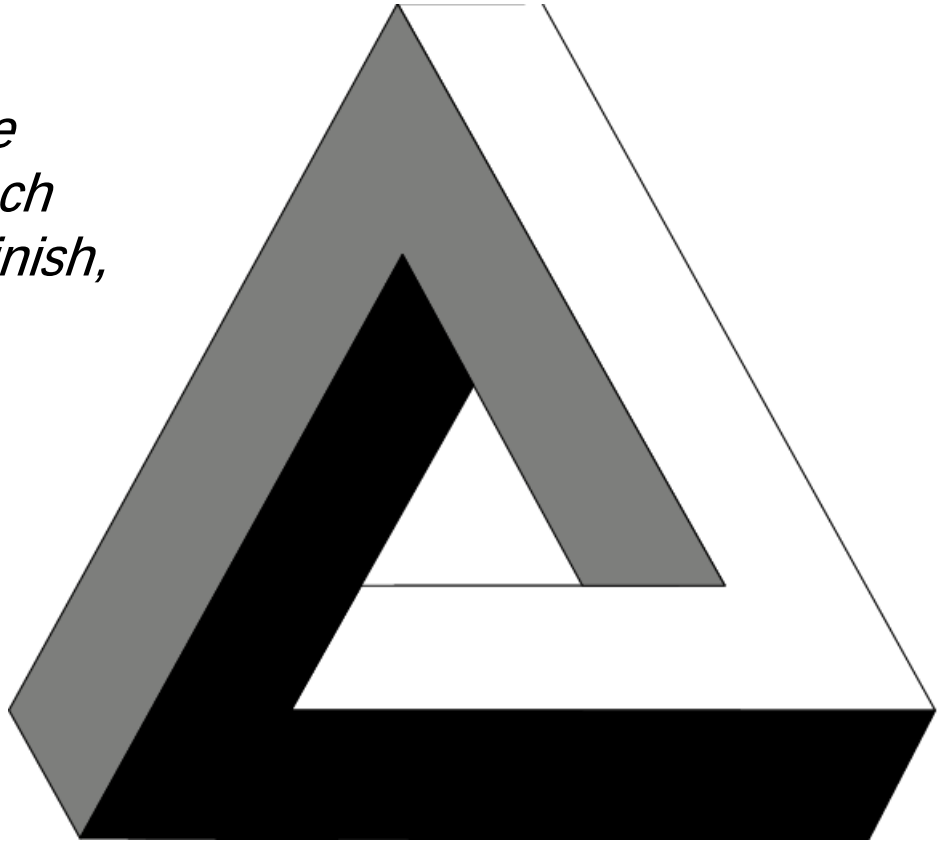




en.wikipedia.org/wiki/Scheduling_(computing) has more info

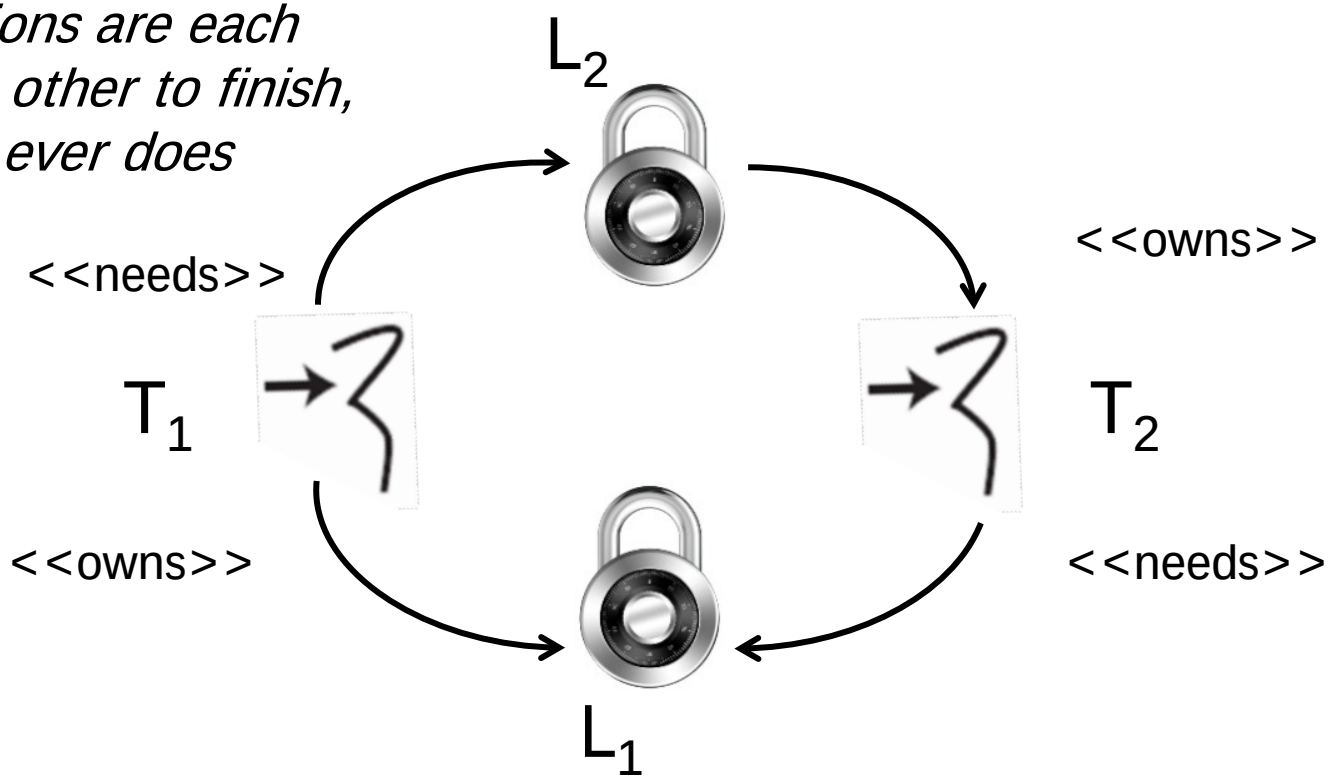
Inherent Complexities for Concurrent Software

- Synchronization
- Scheduling
- Deadlock
  - *Occurs when two or more competing actions are each waiting for the other to finish, & thus neither ever does*



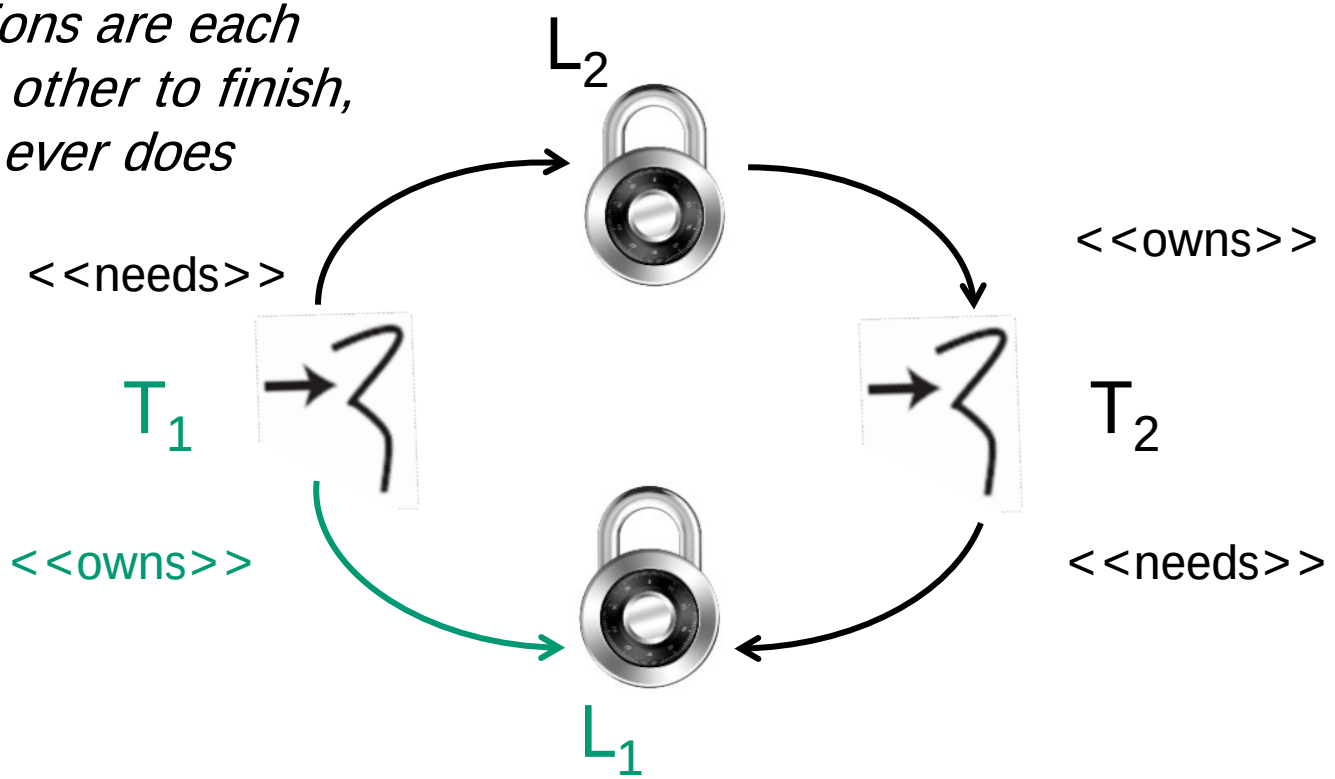
Inherent Complexities for Concurrent Software

- Synchronization
- Scheduling
- Deadlock
  - *Occurs when two or more competing actions are each waiting for the other to finish, & thus neither ever does*



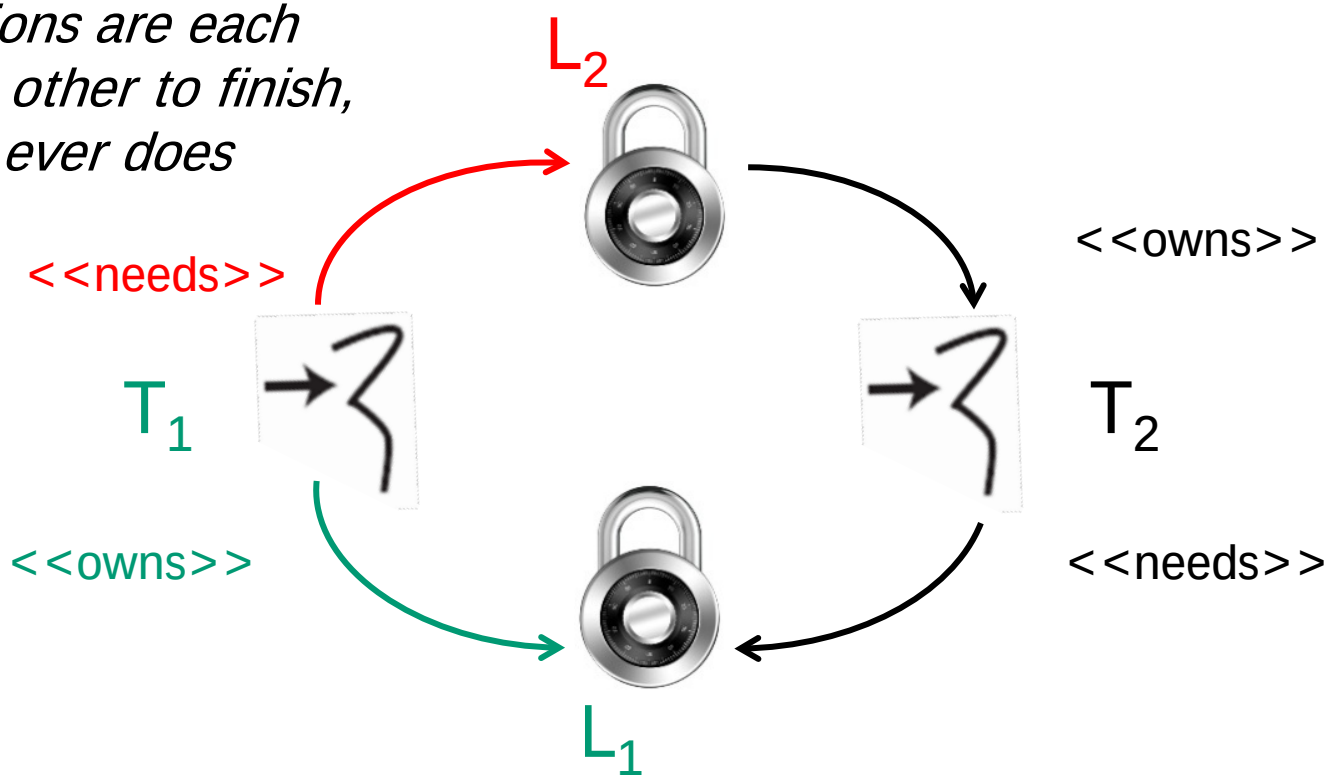
Inherent Complexities for Concurrent Software

- Synchronization
- Scheduling
- Deadlock
  - *Occurs when two or more competing actions are each waiting for the other to finish, & thus neither ever does*



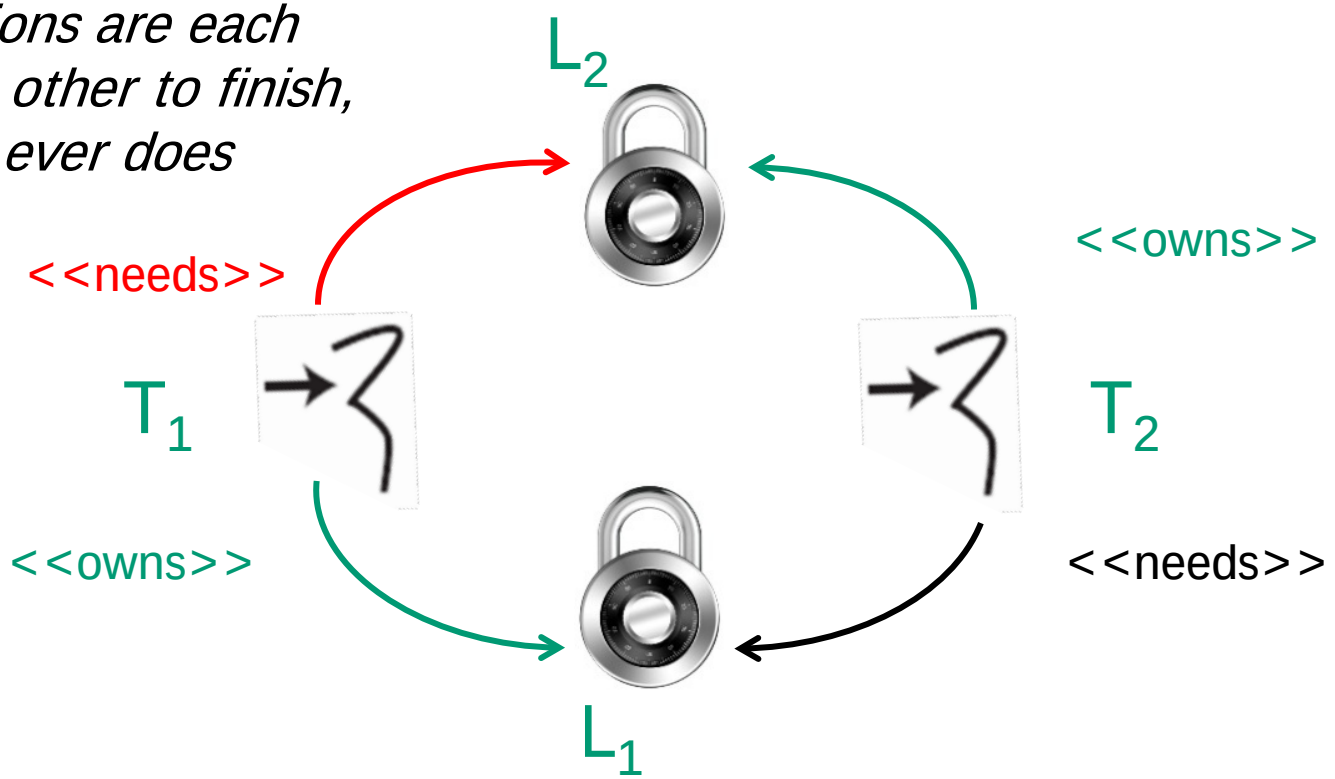
Inherent Complexities for Concurrent Software

- Synchronization
- Scheduling
- Deadlock
  - *Occurs when two or more competing actions are each waiting for the other to finish, & thus neither ever does*



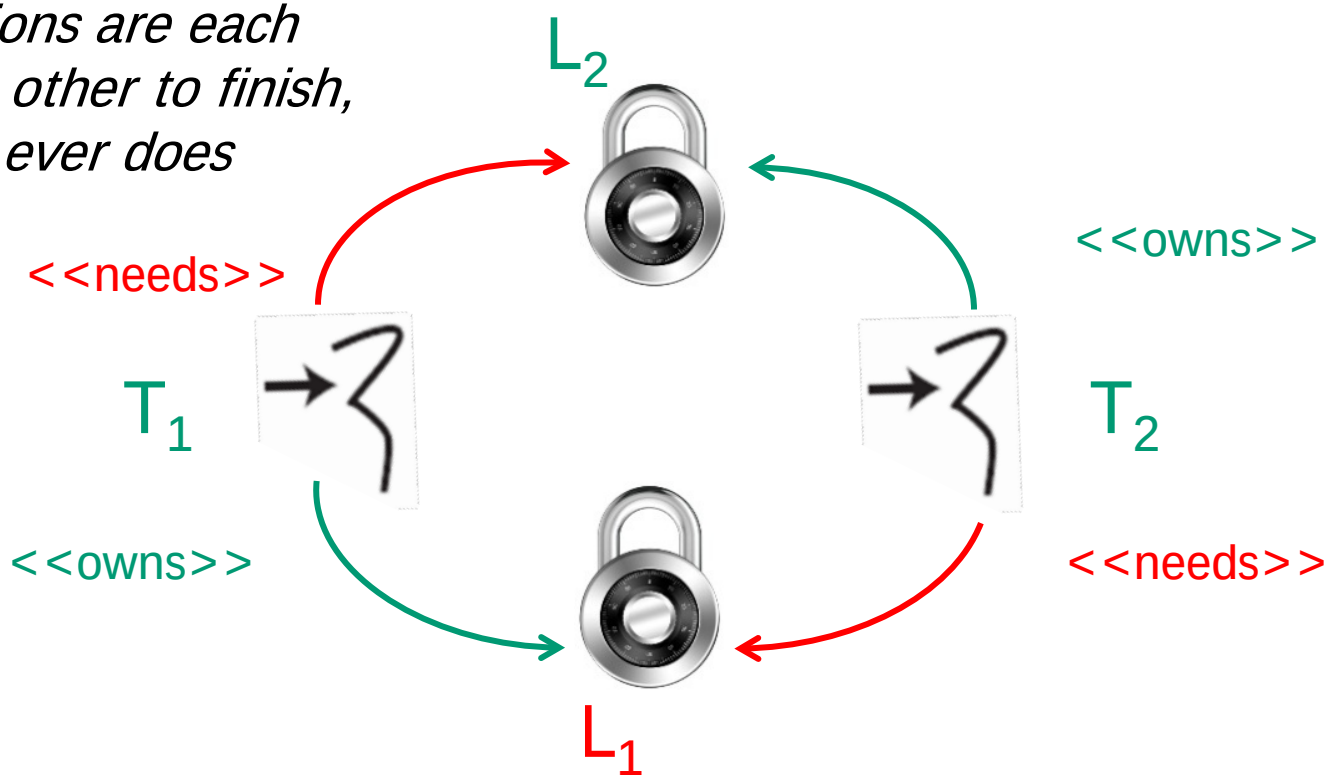
Inherent Complexities for Concurrent Software

- Synchronization
- Scheduling
- Deadlock
  - *Occurs when two or more competing actions are each waiting for the other to finish, & thus neither ever does*



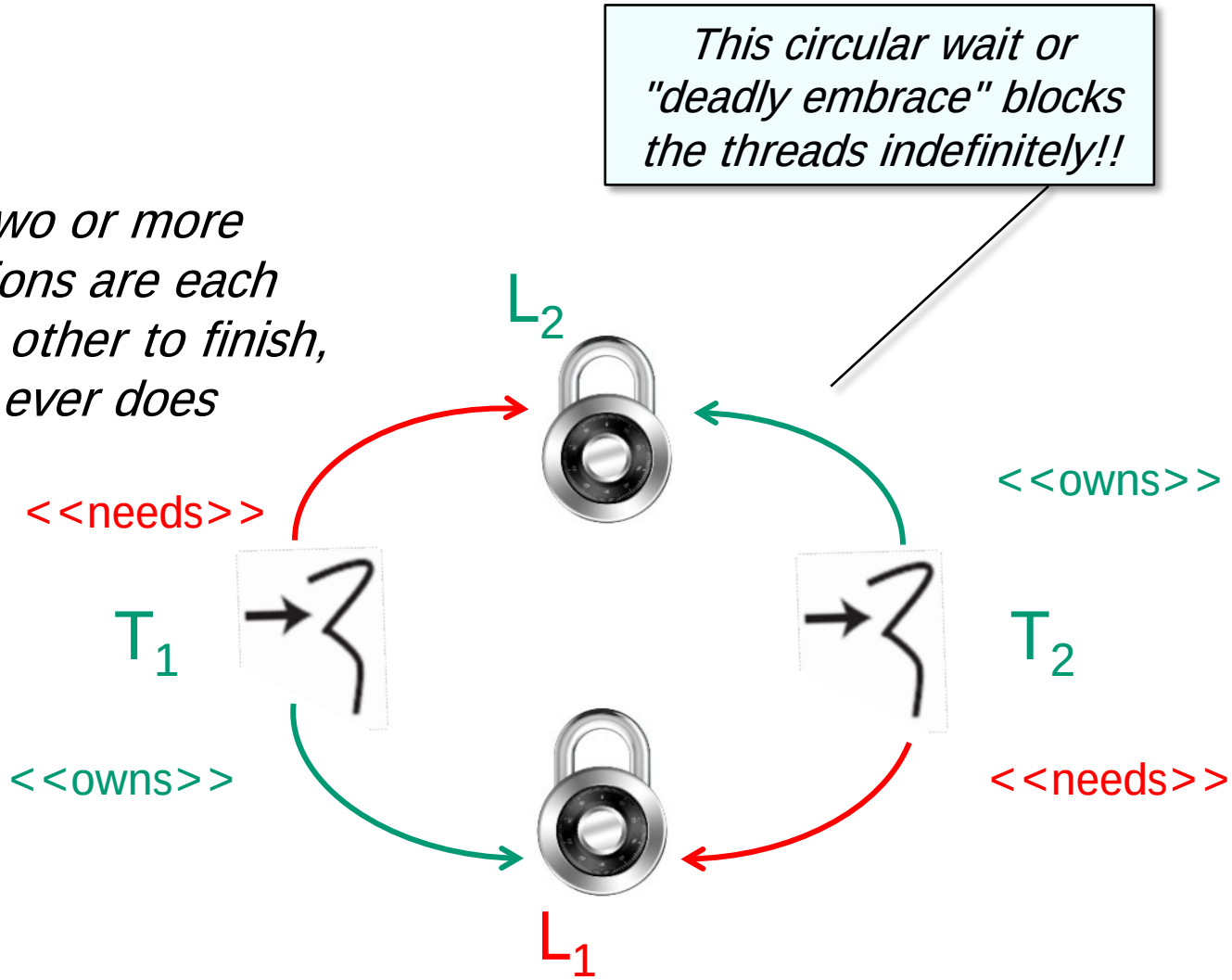
Inherent Complexities for Concurrent Software

- Synchronization
- Scheduling
- Deadlock
  - *Occurs when two or more competing actions are each waiting for the other to finish, & thus neither ever does*



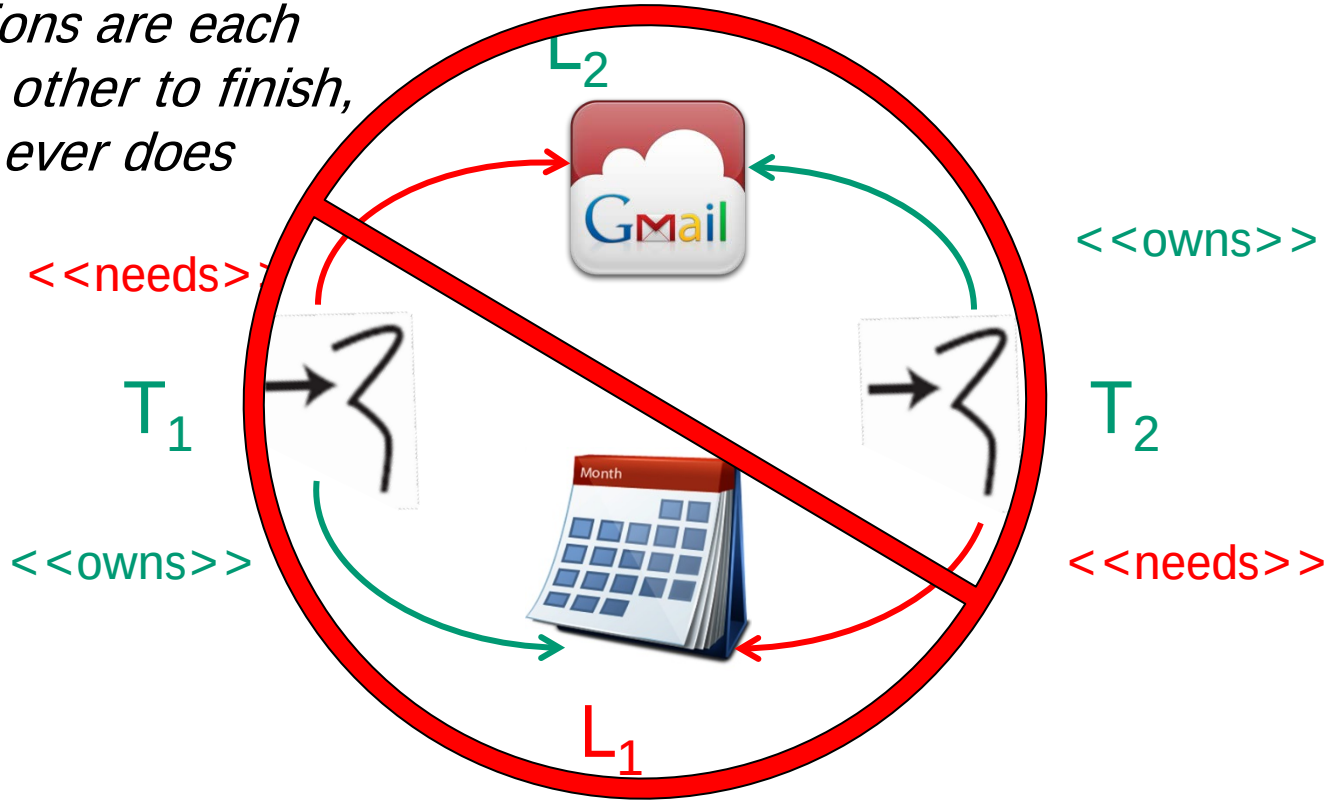
Inherent Complexities for Concurrent Software

- Synchronization
- Scheduling
- Deadlock
  - *Occurs when two or more competing actions are each waiting for the other to finish, & thus neither ever does*



Inherent Complexities for Concurrent Software

- Synchronization
- Scheduling
- Deadlock
  - *Occurs when two or more competing actions are each waiting for the other to finish, & thus neither ever does*



en.wikipedia.org/wiki/Deadlock has more info

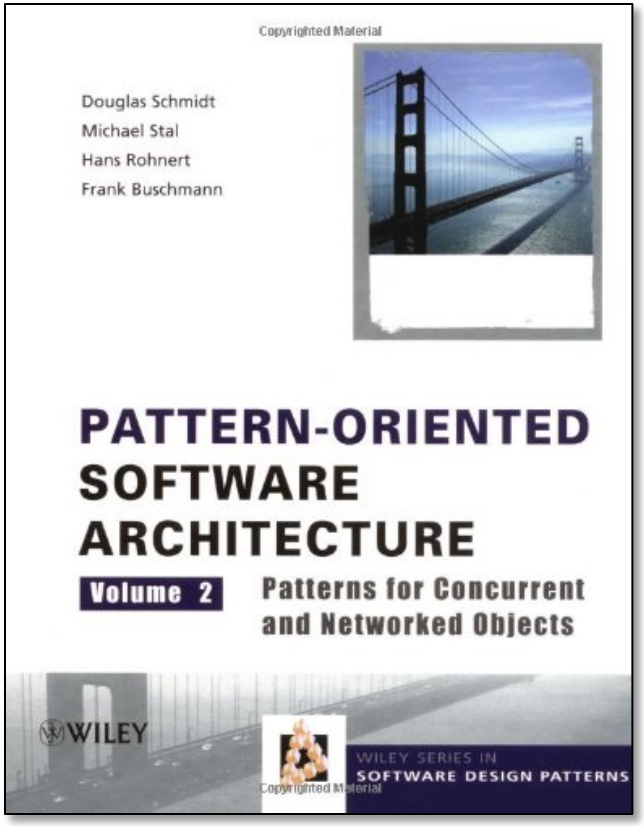
Summary

Summary

- Developers of concurrent software must address key complexities

Summary

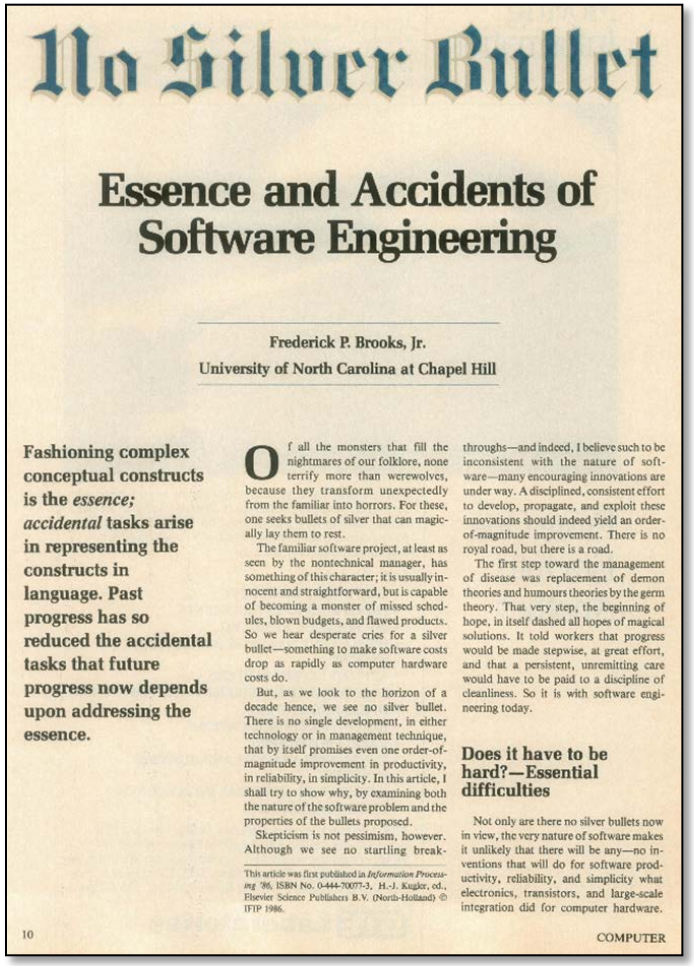
- Developers of concurrent software must address key complexities



www.dre.vanderbilt.edu/~schmidt/POSA/POSA2 has more on concurrency complexities

Summary

- Developers of concurrent software must address key complexities



No Silver Bullet

Essence and Accidents of Software Engineering

Frederick P. Brooks, Jr.
University of North Carolina at Chapel Hill

Fashioning complex conceptual constructs is the *essence;* *accidental* tasks arise in representing the constructs in language. Past progress has so reduced the accidental tasks that future progress now depends upon addressing the essence.

Of all the monsters that fill the nightmares of our folklore, none terrify more than werewolves, because they transform unexpectedly from the familiar into horrors. For these, one seeks bullets of silver that can magically lay them to rest.

The familiar software project, at least as seen by the nontechnical manager, has something of this character; it is usually innocent and straightforward, but is capable of becoming a monster of missed schedules, blown budgets, and flawed products. So we hear desperate cries for a silver bullet—something to make software costs drop as rapidly as computer hardware costs do.

But, as we look to the horizon of a decade hence, we see no silver bullet. There is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity. In this article, I shall try to show why, by examining both the nature of the software problem and the properties of the bullets proposed.

Skepticism is not pessimism, however. Although we see no startling breakthroughs—and indeed, I believe such to be inconsistent with the nature of software—many encouraging innovations are under way. A disciplined, consistent effort to develop, propagate, and exploit these innovations should indeed yield an order-of-magnitude improvement. There is no royal road, but there is a road.

The first step toward the management of disease was replacement of demon theories and humours theories by the germ theory. That very step, the beginning of hope, in itself dashed all hopes of magical solutions. It told workers that progress would be made stepwise, at great effort, and that a persistent, unremitting care would have to be paid to a discipline of cleanliness. So it is with software engineering today.

Does it have to be hard?—Essential difficulties

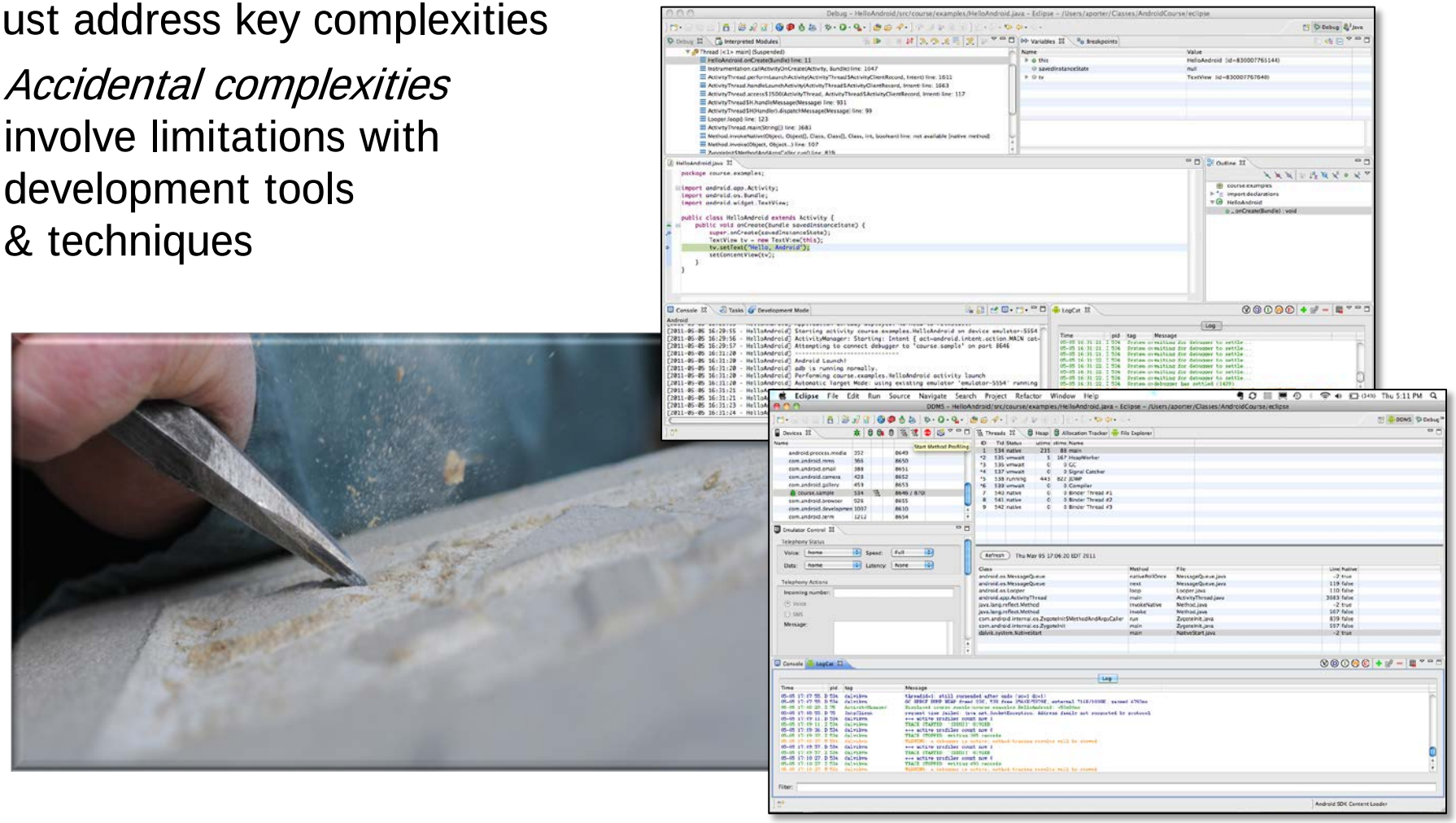
Not only are there no silver bullets now in view, the very nature of software makes it unlikely that there will be any—no inventions that will do for software productivity, reliability, and simplicity what electronics, transistors, and large-scale integration did for computer hardware.

This article was first published in *Information Processing '86*, ISBN No. 0-444-70077-3, H.-J. Kugler, ed., Elsevier Science Publishers B.V. (North-Holland) © IFIP 1986.

en.wikipedia.org/wiki/No_Silver_Bullet has more on general software complexities

Summary

- Developers of concurrent software must address key complexities
  - *Accidental complexities* involve limitations with development tools & techniques



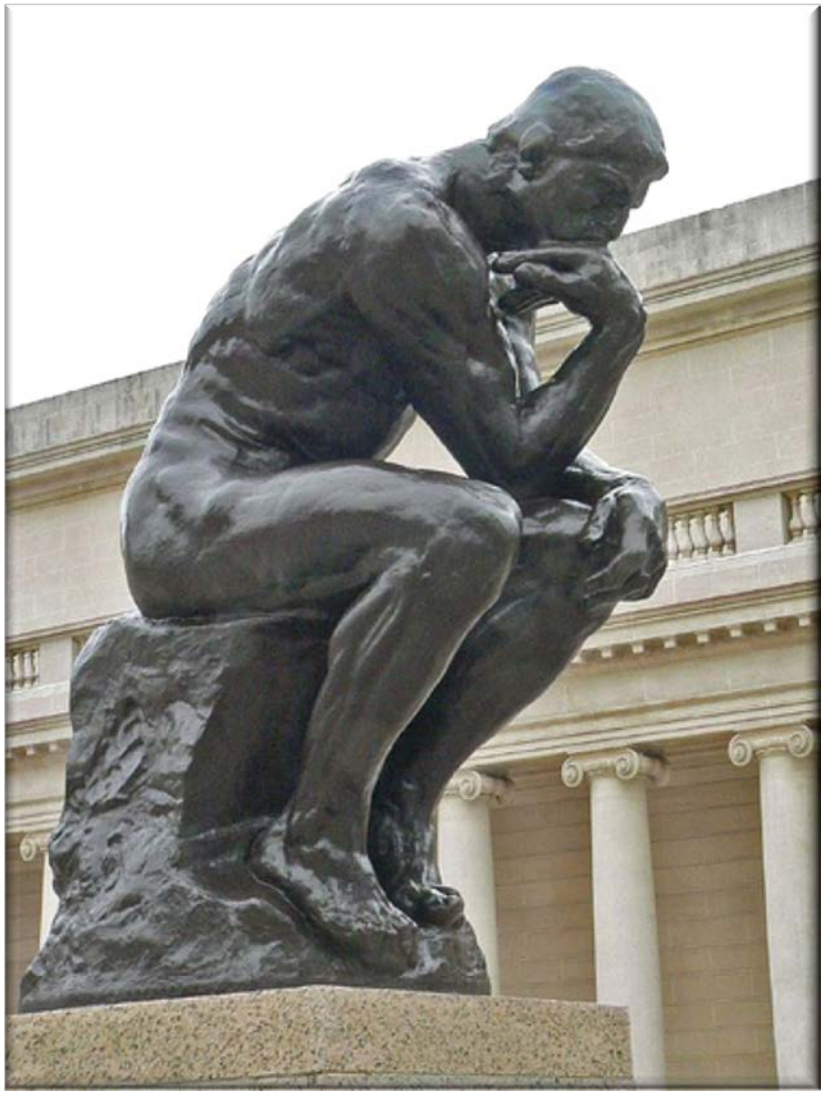
www.dre.vanderbilt.edu/~schmidt/PDF/BC-schmidt.pdf has more info

Summary

- Developers of concurrent software must address key complexities
  - *Accidental complexities* involve limitations with development tools & techniques
  - *Inherent complexities* involve fundamental domain challenges

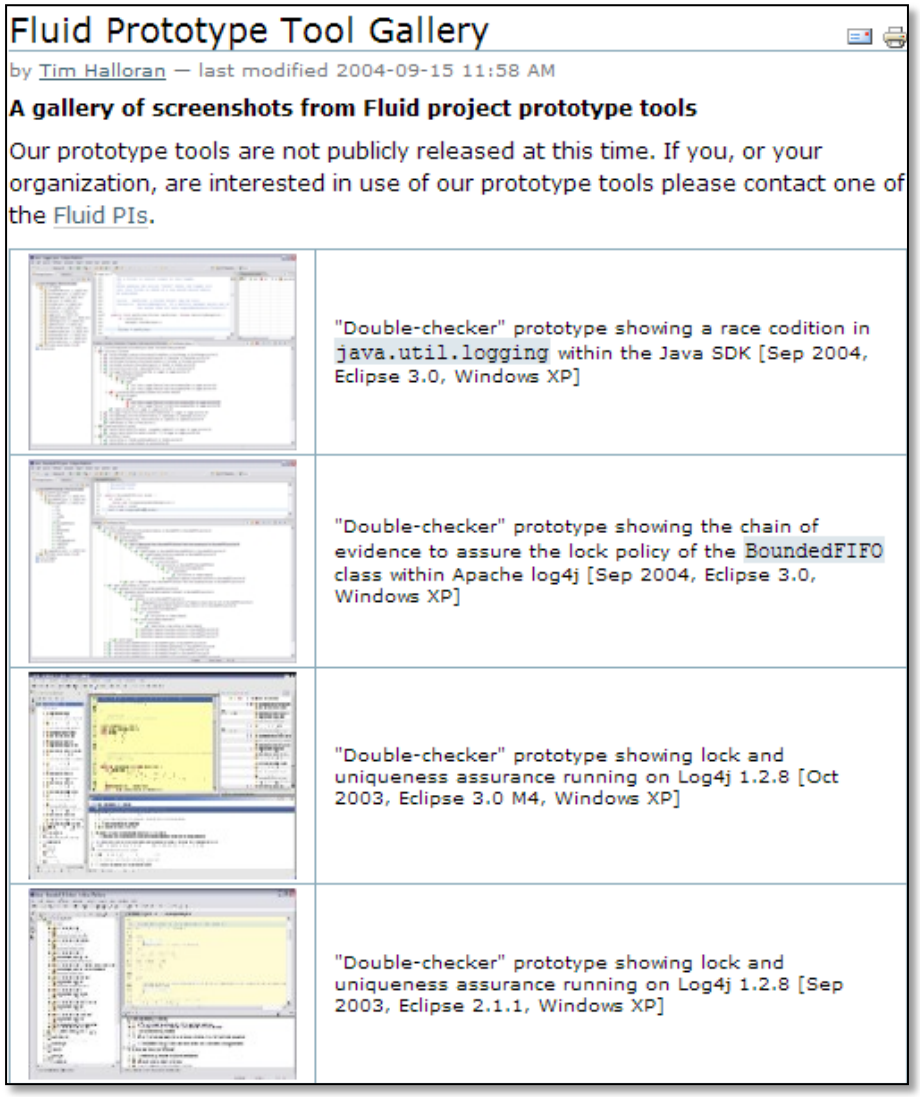
Summary

- Developers of concurrent software must address key complexities
  - *Accidental complexities* involve limitations with development tools & techniques
  - *Inherent complexities* involve fundamental domain challenges



Summary

- Developers of concurrent software must address key complexities
  - *Accidental complexities* involve limitations with development tools & techniques
  - *Inherent complexities* involve fundamental domain challenges



Fluid Prototype Tool Gallery

by Tim Halloran — last modified 2004-09-15 11:58 AM

A gallery of screenshots from Fluid project prototype tools

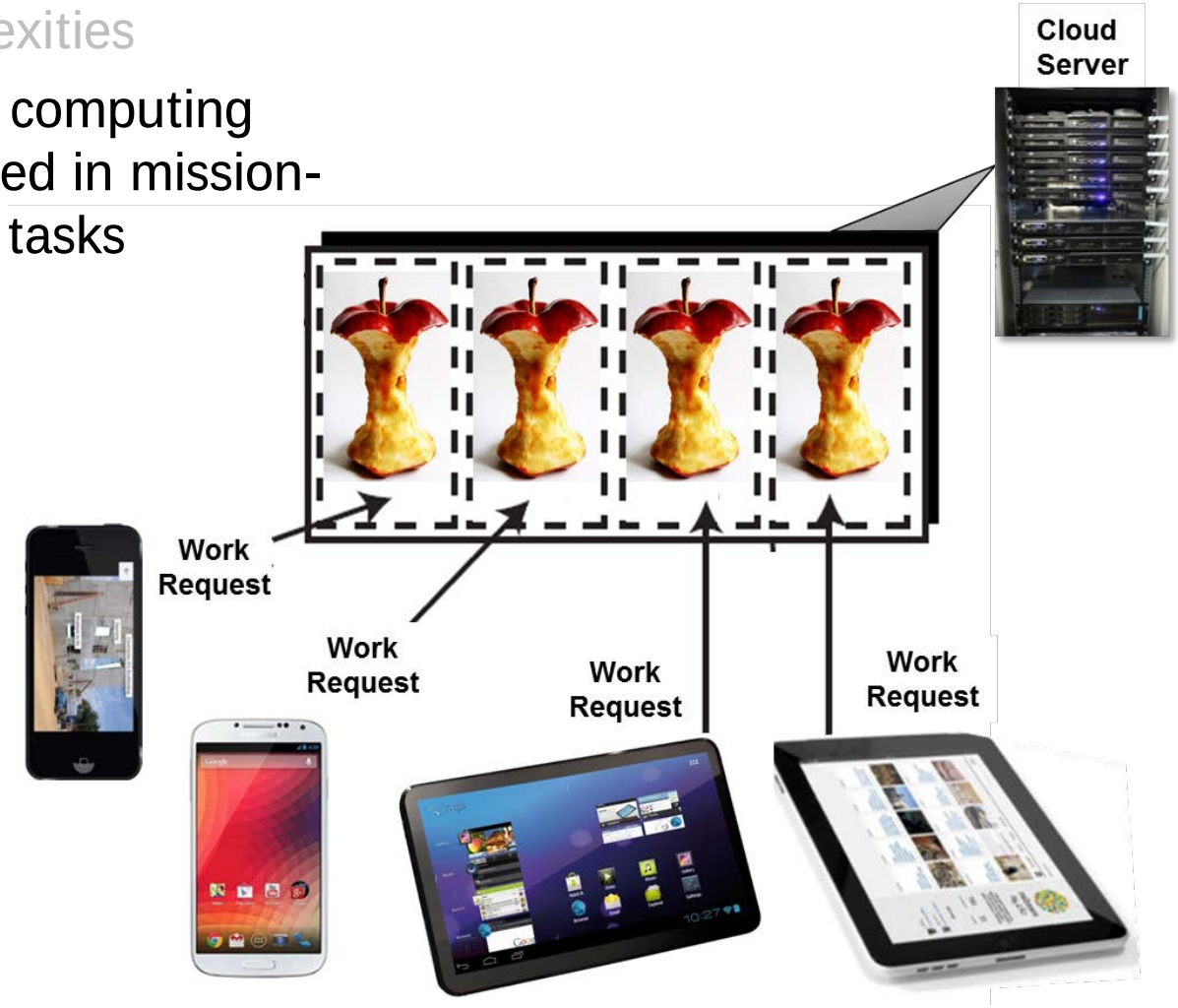
Our prototype tools are not publicly released at this time. If you, or your organization, are interested in use of our prototype tools please contact one of the Fluid PIs.

	"Double-checker" prototype showing a race codition in `java.util.logging` within the Java SDK [Sep 2004, Eclipse 3.0, Windows XP]
	"Double-checker" prototype showing the chain of evidence to assure the lock policy of the `BoundedFIFO` class within Apache log4j [Sep 2004, Eclipse 3.0, Windows XP]
	"Double-checker" prototype showing lock and uniqueness assurance running on Log4j 1.2.8 [Oct 2003, Eclipse 3.0 M4, Windows XP]
	"Double-checker" prototype showing lock and uniqueness assurance running on Log4j 1.2.8 [Sep 2003, Eclipse 2.1.1, Windows XP]

os-book.com has more info on concurrency algorithms & mechanisms

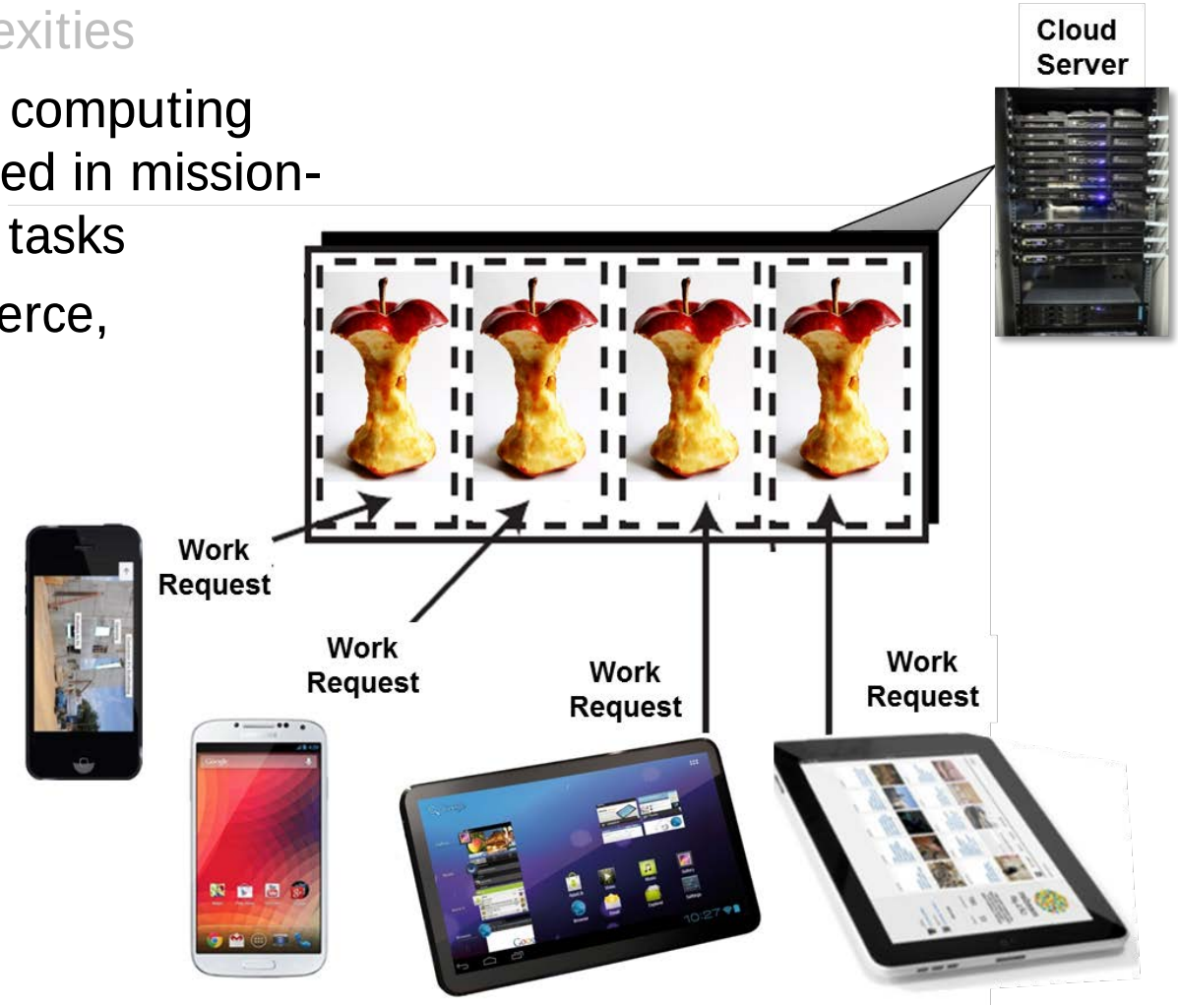
Summary

- Developers of concurrent software must address key complexities
- Concurrent mobile cloud computing software is increasing used in mission-critical & lifestyle-critical tasks



Summary

- Developers of concurrent software must address key complexities
- Concurrent mobile cloud computing software is increasing used in mission-critical & lifestyle-critical tasks
  - e.g., security, e-commerce, geo-positioning, & transportation



Summary

- Developers of concurrent software must address key complexities
- Concurrent mobile cloud computing software is increasing used in mission-critical & lifestyle-critical tasks
- Patterns & frameworks help slay the dragons of complexity

Summary

- Developers of concurrent software must address key complexities
- Concurrent mobile cloud computing software is increasing used in mission-critical & lifestyle-critical tasks
- Patterns & frameworks help slay the dragons of complexity