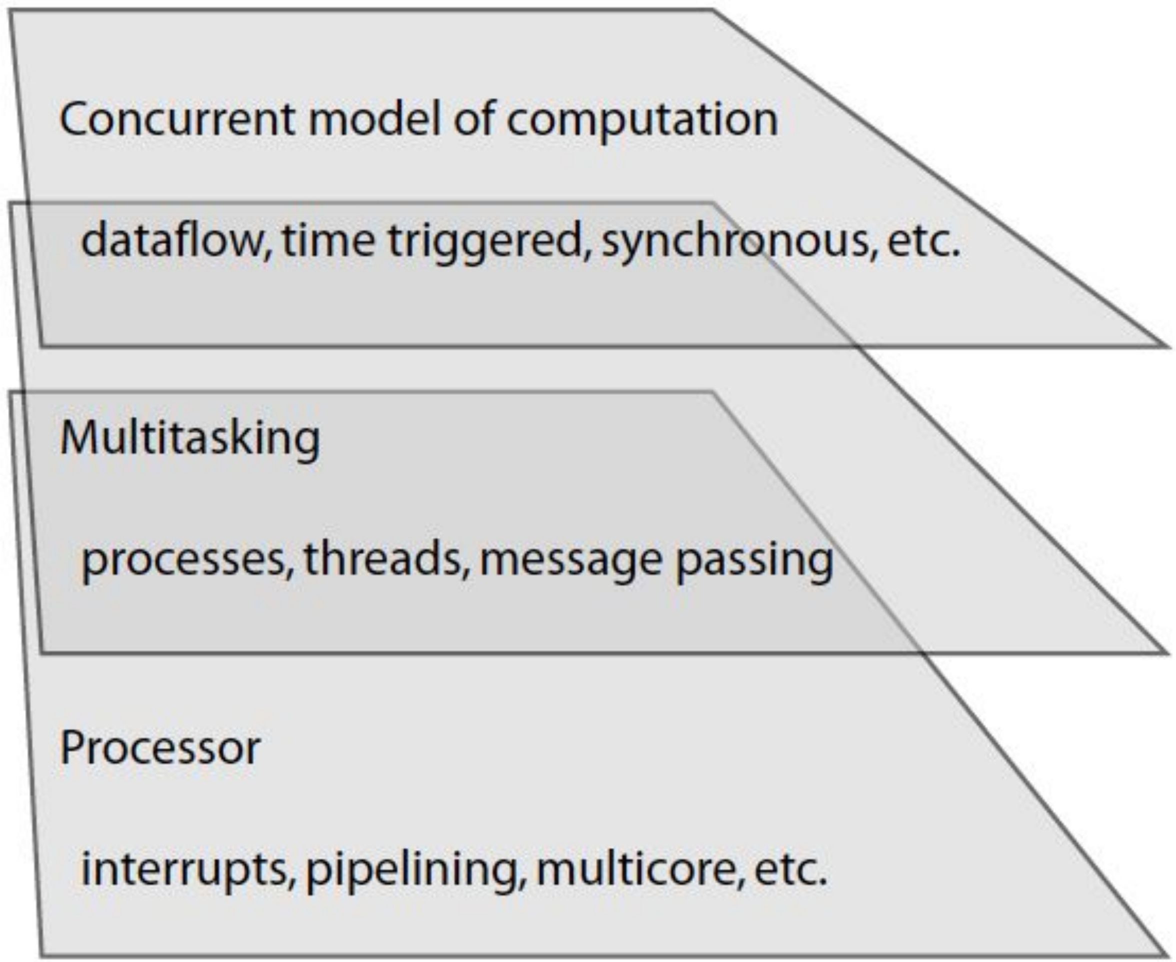
Multitasking & Threads Programming in Embedded Systems

PSI 3442 2020
Capítulo 11 do Livro Texto

Layers of Abstraction for Concurrency in Programs



Definition and Uses

Threads are sequential procedures that share (concurrent dispute) memory.

Uses of concurrency:

- Reacting to external events (interrupts)
- Exception handling (software interrupts)
- Creating the **illusion** of simultaneously running different programs (multitasking)
- parallelism in the hardware (real simultaneously) (e.g. multicore machines).
- Dealing with real-time constraints.

Thread Scheduling

Predicting the thread schedule is an iffy proposition.

- Without an OS, multithreading is achieved with interrupts. Timing is determined by external events.
- Generic OSs (Linux, Windows, OSX, …) provide thread libraries (like "pthreads" POSIX Threads IEEE Standard) and provide no fixed guarantees about when threads will execute. absolute non-determinism
- Real-time operating systems (RTOSs), like FreeRTOS, QNX, VxWorks, RTLinux, MBED, NutOS, support a variety of ways of controlling when threads execute (priorities, preemption policies, deadlines, …).
- Processes are collections of threads with their own memory, not visible to other processes. **Segmentation faults** are attempts to access memory not allocated to the process. Communication between processes must occur via OS facilities (like pipes or files).

Posix Threads (PThreads) IEEE 1988

PThreads is an API (Application Program Interface) implemented by many operating systems, both real-time and not. It is a library of C procedures.

Standardized by the IEEE in 1988 to unify variants of Unix. Subsequently implemented in most other operating systems.

An alternative is Java, which may use PThreads under the hood, but provides thread constructs as part of the programming language.

Creating and Destroying Threads

```
#include <pthread.h>

void* threadFunction(void* arg) {
    ...
    return pointerToSomething or NULL;
}

int main(void) {
    pthread_t threadID;
    void* exitStatus;
    int value = something;
    pthread_create(&threadID, NULL, threadFunction, &value);
    ...
    pthread_join(threadID, &exitStatus);
    return 0;
}
```

Can pass in pointers to shared variables.

Can return pointer to something.
Do not return a pointer to an local variable!

Create a thread (may or may not start running!)

Becomes arg parameter to threadFunction.
Why is it OK that this is a local variable?

Return only after all threads have terminated.

Difficulties in thread programming

you must:

a) know how to programs in C;
b) details of the microprocessor
c) details of the operating system

and how the IEEE 1988 Pthreads works

What’s Wrong with This?

```
#include <pthread.h>
#include <stdio.h>
void *myThread() {
 int ret = 42;
 return &ret;
}

int main() {
 pthread_t tid;
 void *status;
 pthread_create(&tid, NULL, myThread, NULL);
 pthread_join(tid, &status);
 printf("%d\n",*(int*)status); return 0;
}
```

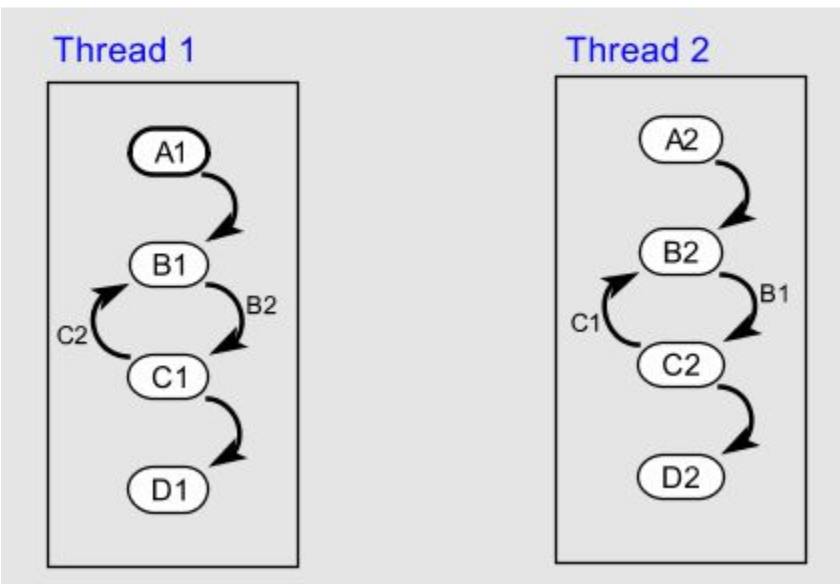
Don’t return a pointer to a local variable, which is on the stack.

Notes

- Threads can (and often do) share variables
- Threads may or may not begin running immediately after being created.
- A thread may be suspended between any two *atomic* instructions (typically, assembly instructions, not C statements!) to execute another thread and/or interrupt service routine.
- Threads can often be given *priorities*, and these may or may not be respected by the thread scheduler.
- Threads may *block* on semaphores and mutexes (we will do this later in this lecture).

Modeling Threads via Asynchronous Composition of Extended State Machines

States or transitions represent atomic instructions



Can Thread 1 be in C1 at the same time Thread 2 is in C2?

Interleaving semantics:

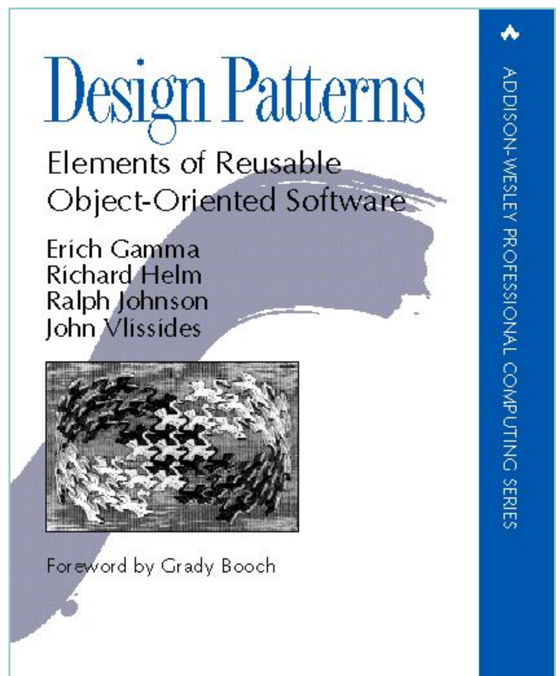
- Choose one machine, arbitrarily.
- Advance to a next state if guards are satisfied.
- Repeat.

Need to compute reachable states to reason about correctness of the composed system

A Scenario

Under Integrated Modular Avionics, software in the aircraft engine continually runs diagnostics and publishes diagnostic data on the local network.

Proper software engineering practice suggests using the observer pattern.

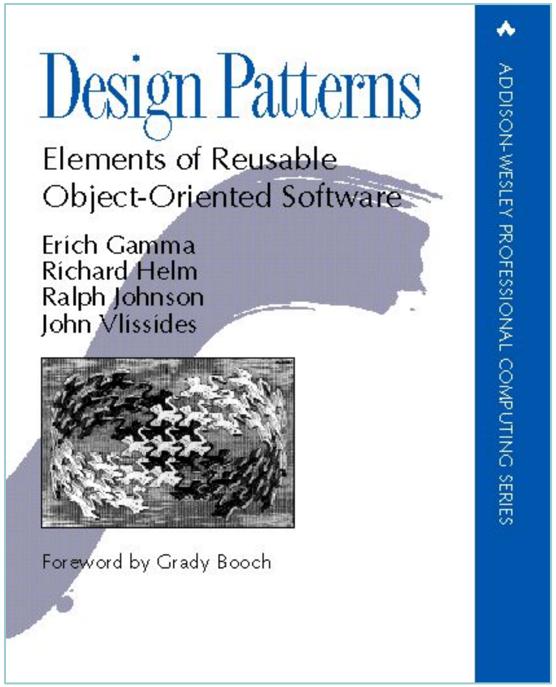


An observer process updates the cockpit display based on notifications from the engine diagnostics.

Typical thread programming problem

"The *Observer pattern* defines a one-to-many dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated automatically."

Design Patterns, Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides (Addison-Wesley, 1995)



Observer Pattern in C

```
// Value that when updated triggers notification
// of registered listeners.
int value;

// List of listeners. A linked list containing
// pointers to notify procedures.
typedef void* notifyProcedure(int);
struct element {…}
typedef struct element elementType;
elementType* head = 0;
elementType* tail = 0;

// Procedure to add a listener to the list.
void addListener(notifyProcedure listener) {…}

// Procedure to update the value
void update(int newValue) {…}

// Procedure to call when notifying
void print(int newValue) {…}
```

Observer Pattern in C

```
// Value that when updated triggers notification of
// registered listeners.
int value;

// List of listeners. A li
// pointers to notify proc
typedef void* notifyProced
struct element {…}
typedef struct element ele
elementType* head = 0;
elementType* tail = 0;

// Procedure to add a listener to the list.
void addListener(notifyProcedure listener) {…}

// Procedure to update the value
void update(int newValue) {…}

// Procedure to call when notifying
void print(int newValue) {…}
```

```
typedef void* notifyProcedure(int);
struct element {
  notifyProcedure* listener;
  struct element* next;
};
typedef struct element elementType;
elementType* head = 0;
elementType* tail = 0;
```

Observer Pattern in C

```
// Value that
registered lis
int value;

// List of lis
// pointers to
typedef void*
struct element
typedef struct
elementType* h
elementType* t

// Procedure t
void addLister

// Procedure t
void update(in

// Procedure to call when notifying
void print(int newValue) {…}
```

```
// Procedure to add a listener to the list.
void addListener(notifyProcedure listener) {
  if (head == 0) {
    head = malloc(sizeof(elementType));
    head->listener = listener;
    head->next = 0;
    tail = head;
  } else {
    tail->next = malloc(sizeof(elementType));
    tail = tail->next;
    tail->listener = listener;
    tail->next = 0;
  }
}
```

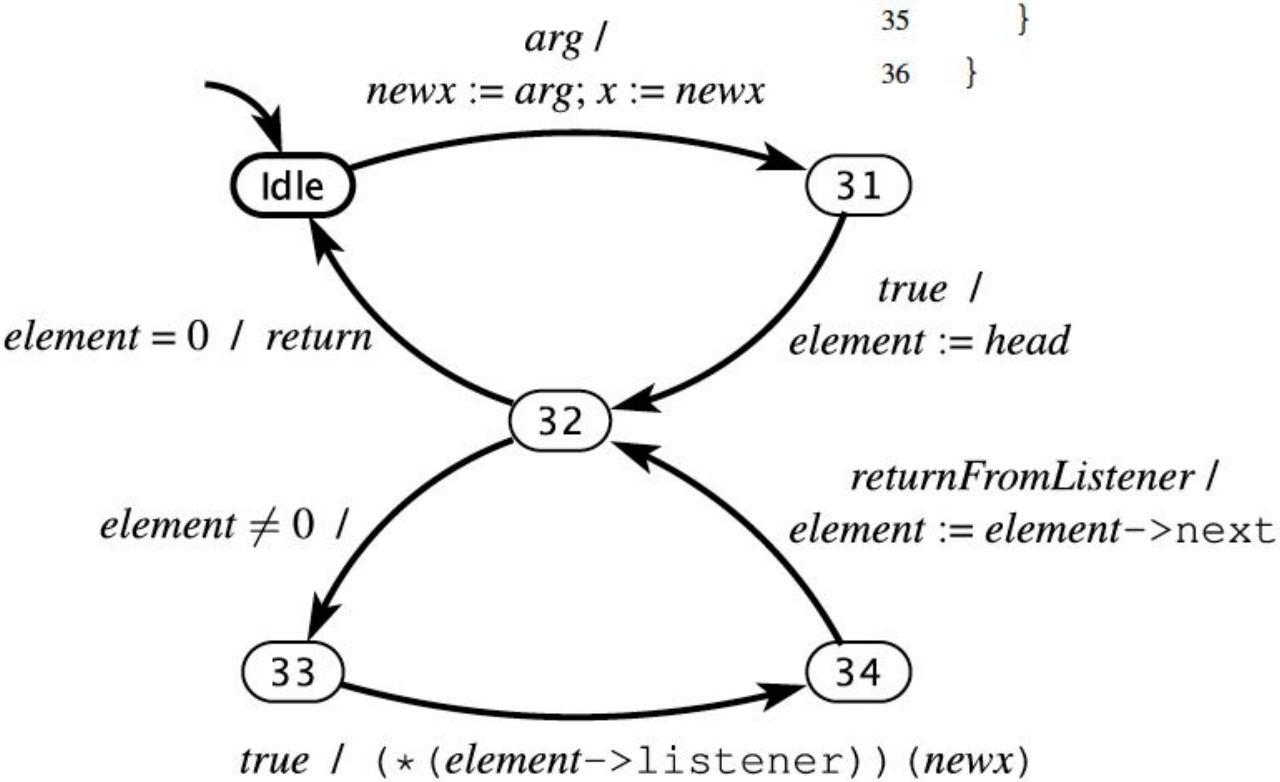
Observer Pattern in C

```
// Value that when updated triggers notification of
registered listeners.
int value;

// List of listeners. A linked list containing
// pointers to notify procedures.
typedef void* notifyProcedure(int);
struct element {…}
typedef struct
elementType* h
elementType* t

// Procedure t
void addListen

// Procedure t
void update(in

// Procedure t
void print(int
```

```
// Procedure to update the value
void update(int newValue) {
  value = newValue;
  // Notify listeners.
  elementType* element = head;
  while (element != 0) {
    (*(element->listener))(newValue);
    element = element->next;
  }
}
```

Model of the Update Procedure

```
27 // Procedure to update x.
28 void update(int newx) {
29   x = newx;
30   // Notify listeners.
31   element_t* element = head;
32   while (element != 0) {
33     (*(element->listener))(newx);
34     element = element->next;
35   }
36 }
```



Observer Pattern in C

```
// Value that when updated triggers notification of registered listeners.
int value;

// List of listeners. A linked list containing
// pointers to notify procedures.
typedef void* notifyProcedure(int);
struct element {…}
typedef struct element elementType;
elementType* head = 0;
elementType* tail = 0;

// Procedure to add a listener to the list.
void addListener(notifyProcedure listener) {…}

// Procedure to update the value
void update(int newValue) {…}

// Procedure to call when notifying
void print(int newValue) {…}
```

Will this work in a multithreaded context?

Will there be unexpected/undesirable behaviors?

Using Posix mutexes on the observer pattern in C

```
#include <pthread.h>
...
pthread_mutex_t lock;

void addListener(notify listener) {
  pthread_mutex_lock(&lock);
  ...
  pthread_mutex_unlock(&lock);
}

void update(int newValue) {
  pthread_mutex_lock(&lock);
  value = newValue;
  elementType* element = head;
  while (element != 0) {
    (*(element->listener))(newValue);
    element = element->next;
  }
  pthread_mutex_unlock(&lock);
}

int main(void) {
  pthread_mutex_init(&lock, NULL);
  ...
}
```

However, this carries a significant deadlock risk. The update procedure holds the lock while it calls the notify procedures. If any of those stalls trying to acquire another lock, and the thread holding that lock tries to acquire this lock, deadlock results.

```java
public synchronized void addChangeListener(ChangeListener listener) {
    NamedObj container = (NamedObj) getContainer();
    if (container != null) {
        container.addChangeListener(listener);
    } else {
        if (_changeListeners == null) {
            _changeListeners = new LinkedList();
            _changeListeners.add(0, listener);
        } else if (!_changeListeners.contains(listener)) {
            _changeListeners.add(0, listener);
        }
    }
}
```

After years of use without problems, a Ptolemy Project code review found code that was not thread safe. It was fixed in this way. Three days later, a user in Germany reported a deadlock that had not shown up in the test suite.

```c
#include <pthread.h>
...
pthread_mutex_t lock;

void addListener(notify listener) {
  pthread_mutex_lock(&lock);
  ...
  pthread_mutex_unlock(&lock);
}

void update(int newValue) {
  pthread_mutex_lock(&lock);
  value = newValue;
  ... copy the list of listeners ...
  pthread_mutex_unlock(&lock);
  elementType* element = headCopy;
  while (element != 0) {
    (*(element->listener))(newValue);
    element = element->next;
  }
}

int main(void) {
  pthread_mutex_init(&lock, NULL);
  ...
}
```

One possible "fix"

What is wrong with this?

Notice that if multiple threads call update(), the updates will occur in some order. But there is no assurance that the listeners will be notified in the same order. Listeners may be mislead about the "final" value.

This is a very simple, commonly used design pattern. Perhaps Concurrency is Just Hard…

Sutter and Larus observe:

"Humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings among even simple collections of partially ordered operations."

H. Sutter and J. Larus. Software and the concurrency revolution. ACM Queue, 3(7), 2005.

If concurrency were intrinsically hard, we would not function well in the physical world

It is not concurrency that is hard…

…It is Threads that are Hard!

Threads are sequential processes that share memory. From the perspective of any thread, the *entire state of the universe can change between any two atomic actions* (itself an ill-defined concept).

Imagine if the physical world did that…

What it Feels Like to Use Mutexes

Image “borrowed” from an Iomega advertisement for Y2K software and disk drives, Scientific American, September 1999.

Message-passing programs *may* be better

```
void* producer(void* arg) {
    int i;
    for (i = 0; i < 10; i++) {
        send(i);
    }
    return NULL;
}
void* consumer(void* arg) {
    while(1) {
        printf("received %d\n", get());
    }
    return NULL;
}
int main(void) {
    pthread_t threadID1, threadID2;
    void* exitStatus;
    pthread_create(&threadID1, NULL, producer, NULL);
    pthread_create(&threadID2, NULL, consumer, NULL);
    pthread_join(threadID1, &exitStatus);
    pthread_join(threadID2, &exitStatus);
    return 0;
}
```

But there is still risk of deadlock and unexpected nondeterminism!

Claim

Nontrivial software written with threads, semaphores, and mutexes is incomprehensible to humans.

- *Need better ways to program concurrent systems (we will see some later in the course)*
- *Better tools to analyze and reason about concurrency (e.g. model checking)*

Do Threads Have a Sound Foundation?

If the foundation is bad, then we either tolerate brittle designs that are difficult to make work, or we have to rebuild from the foundations.

Note that this whole enterprise is held up by threads

Problems with the Foundations

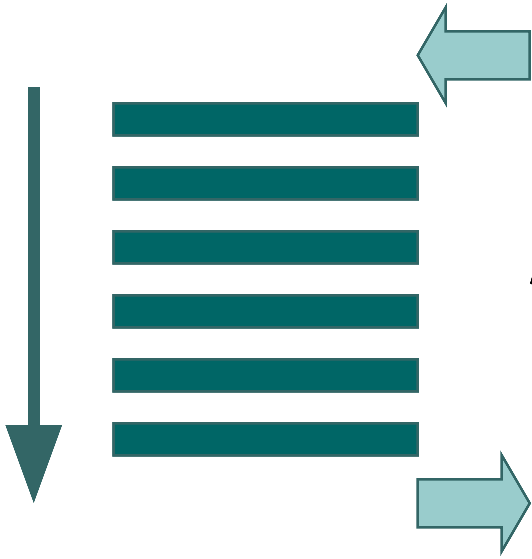
A model of computation:

- Bits: $B = \{0, 1\}$
- Set of finite sequences of bits: B^*
- Computation: $f : B^* \rightarrow B^*$
- Composition of computations: $f \cdot f'$
- Programs specify compositions of computations

Threads augment this model to admit concurrency.

But this model does not admit concurrency gracefully.

Basic Sequential Computation



initial state: $b_0 \in B^*$

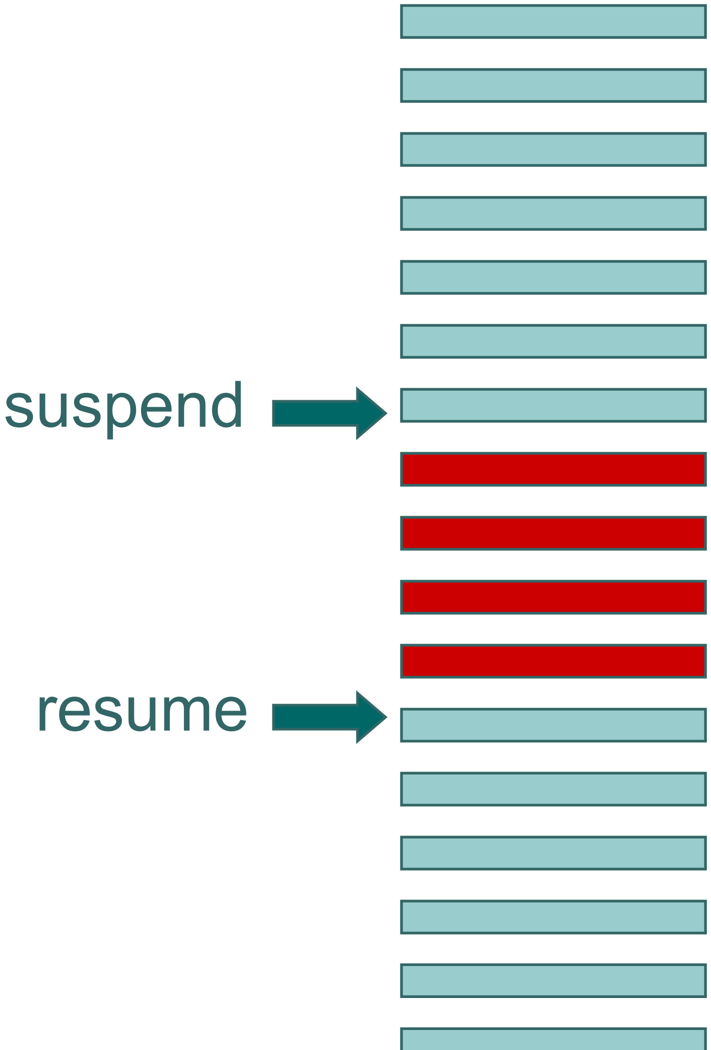
sequential composition

$$b_n = f_n(b_{n-1})$$

final state: b_N

Formally, composition of computations is function composition.

When There are Threads, Everything Changes



A program no longer computes a function.

$$b_n = f_n(\, b_{n\text{-}1}\,)$$

another thread can change the state

$$b'_n = f_n(\, b'_{n\text{-}1}\,)$$

Apparently, programmers find this model appealing because nothing has changed in the *syntax*.

Succinct Problem Statement

Threads are wildly nondeterministic.

The programmer’s job is to prune away the nondeterminism by imposing constraints on execution order (e.g., mutexes) and limiting shared data accesses (e.g., OO design).

Incremental Improvements to Threads

- Object Oriented programming
- Coding rules (Acquire locks in the same order…)
- Libraries (Stapl, Java >= 5.0, …)
- Transactions (Databases, …)
- Patterns (MapReduce, …)
- Formal verification (Model checking, …)
- Enhanced languages (Split-C, Cilk, Guava, …)
- Enhanced mechanisms (Promises, futures, asynchronous atomic callbacks …)

IEEE Computer, May, 2006

COVER FEATURE

The Problem with Threads

Edward A. Lee
University of California, Berkeley

For concurrent programming to become mainstream, we must discard threads as a programming model. Nondeterminism should be judiciously and carefully introduced where needed, and it should be explicit in programs.