Introduction to Embedded Systems

Edward A. Lee & Sanjit Seshia
UC Berkeley
EECS 124
Spring 2008

Lecture 17: Concurrency 2: Threads

Definition and Uses

Threads are sequential procedures that share memory.

Uses:

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

Thread Scheduling

Thread scheduling 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") and provide no fixed guarantees about when threads will execute.
- Real-time operating systems (RTOSs), like QNX, VxWorks, RTLinux, Windows CE, 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)

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 typically uses PThreads under the hood, but provides thread constructs as part of the programming language.

Creating and Destroying Threads

Can pass in pointers to shared variables.

```
#include <pthread.h>

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

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

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

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

Becomes arg parameter to threadFunction.
Why is it OK that this an automatic variable?

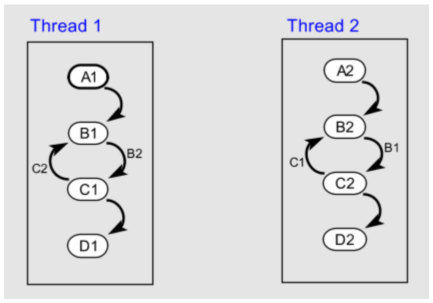
Return only after all threads have terminated.

Notes

- Threads may or may not beginning running when 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 do this next).

Modeling Threads

States or transitions represent atomic instructions



Interleaving semantics:

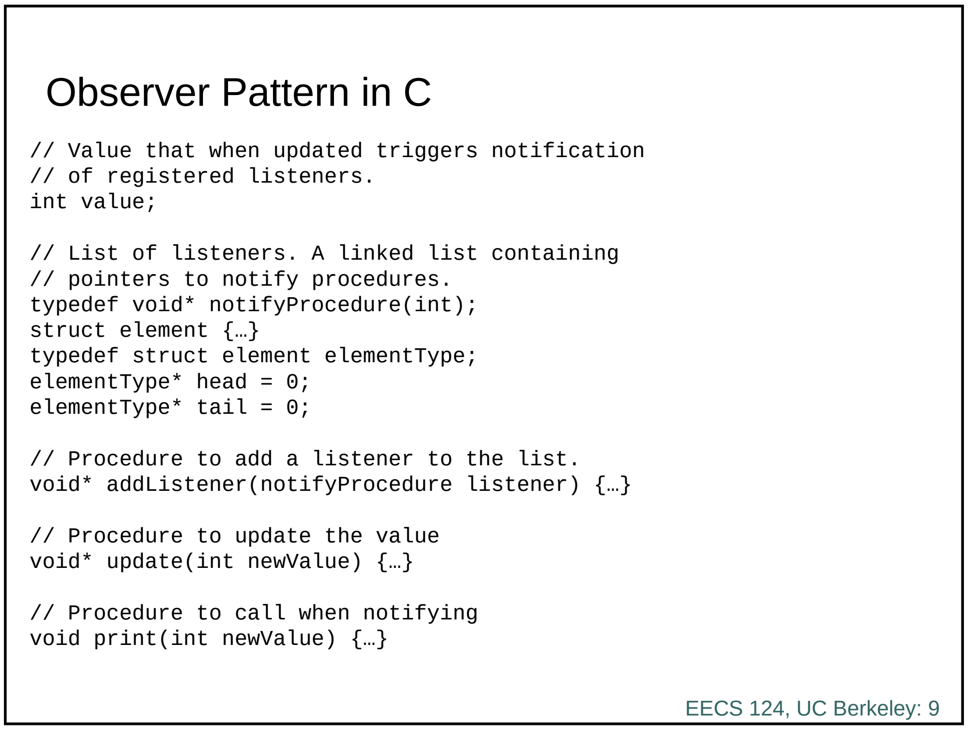
- Choose one machine at random.
- Advance to a next state if guards are satisfied.
- Repeat.

For the machines at the left, what are the reachable states?

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.”

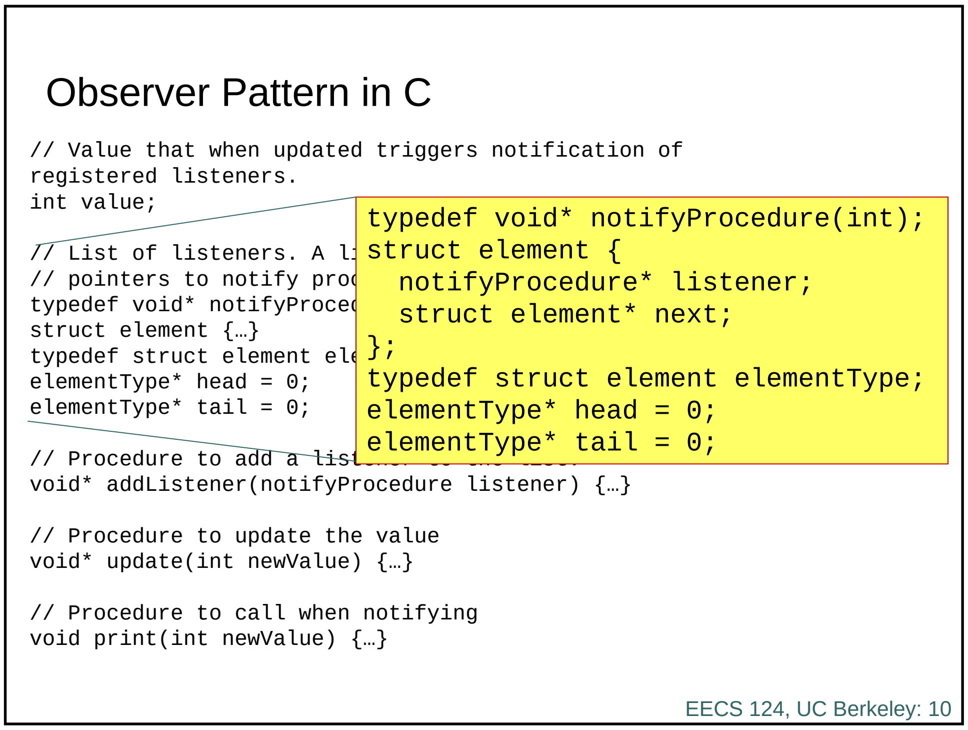
Design Patterns, Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides (Addison-Wesley Publishing Co., 1995. ISBN: 0201633612):



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) {…}
```

EECS 124, UC Berkeley: 9



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 lis
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;
```

EECS 124, UC Berkeley: 10

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* addListe

// Procedure t
void* update(i

// 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* addListe

// Procedure t
void* update(i

// 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;
  }
}
```

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?

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.

```
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.

One possible "fix"

```
#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);
  ...
}
```

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.

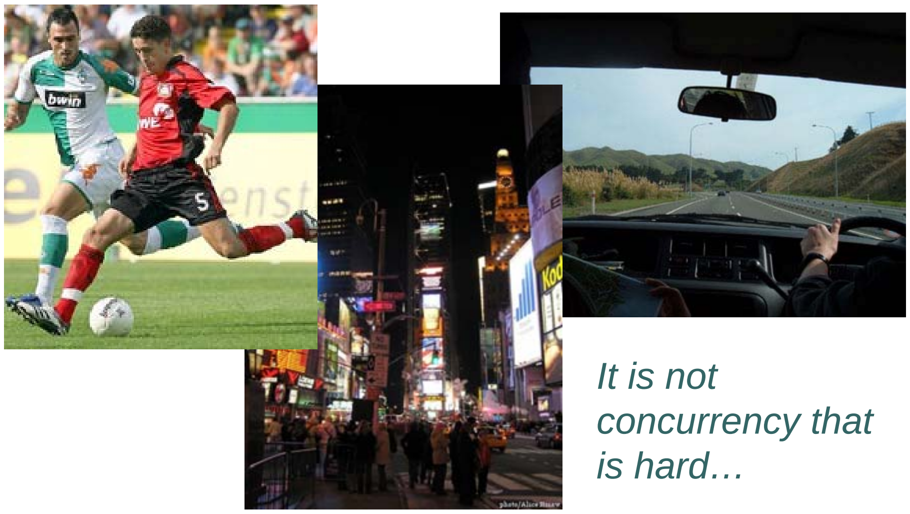
This is a very simple, commonly used design pattern. Perhaps Concurrency is Just Hard…

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…

Problems with the Foundations

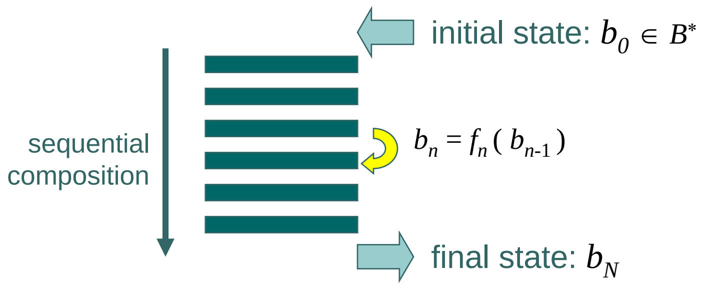
A model of computation:

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

Threads augment this model to admit concurrency.

But this model does not admit concurrency gracefully.

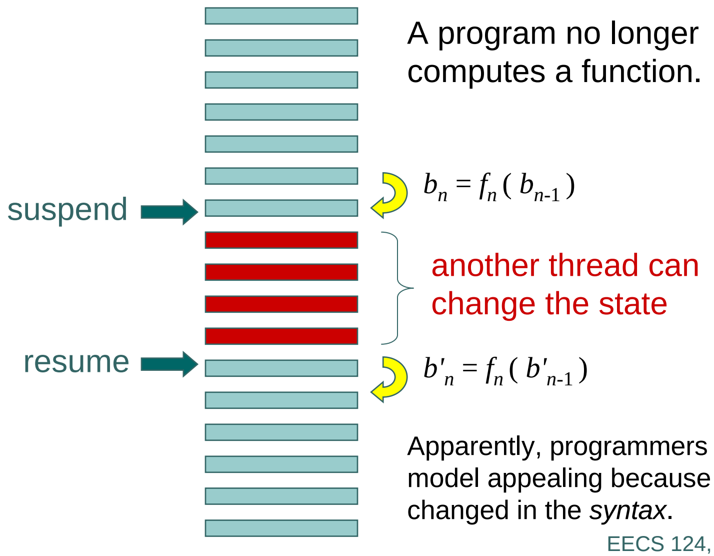
Basic Sequential Computation



Formally, composition of computations is function composition.

When There are Threads, Everything Changes

A program no longer computes a function.



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 (Blast, thread checkers, …)
- Enhanced languages (Split-C, Cilk, Guava, …)
- Enhanced mechanisms (Promises, futures, …)