Lecture 17: Multitasking
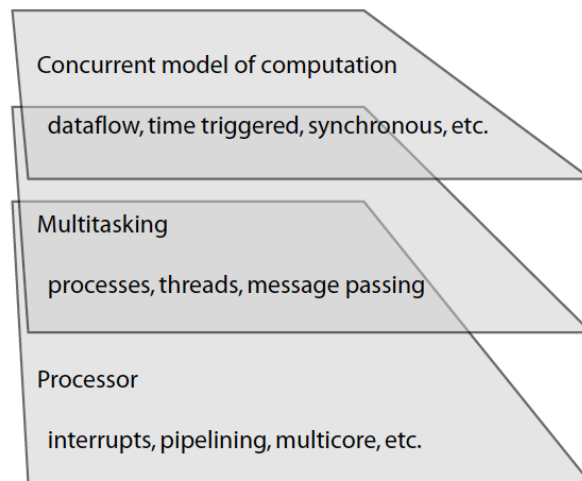
Seyed-Hosein Attarzadeh-Niaki

Based on the slides by Edward Lee

Real-Time Embedded Systems

1

Layers of Abstraction for Concurrency in Programs

Concurrent model of computation

dataflow, time triggered, synchronous, etc.

Multitasking

processes, threads, message passing

Processor

interrupts, pipelining, multicore, etc.

Real-Time Embedded Systems

2

Definition and Uses

Threads are sequential procedures that share memory.

Uses of concurrency?
- 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.

Real-Time Embedded Systems   3

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

Real-Time Embedded Systems   4

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 may use PThreads under the hood, but provides thread constructs as part of the programming language.

Real-Time Embedded Systems   5

Creating and Destroying Threads

```
#include <pthread.h>                    Can pass in pointers to shared variables.


void* threadFunction(void* arg) {
    ...
    return pointerToSomething or NULL;
}
                        Can return pointer to something.
                        Do not return a pointer to a local variable!
int main(void) {
    pthread_t threadID;
    void* exitStatus;          Create a thread (may or may not start running!)
    int value = something;
    pthread_create(&threadID, NULL, threadFunction, &value);
    ...                                        Becomes arg parameter to
    pthread_join(threadID, &exitStatus);       threadFunction.
                                               Why is it OK that this is a
    return 0;                                  local variable?
}            Return only after all threads have terminated.
```

Real-Time Embedded Systems   6

3

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.

Real-Time Embedded Systems 7

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).
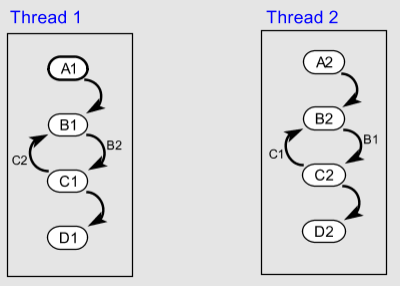
Real-Time Embedded Systems 8

Modeling Threads via Asynchronous Composition of Extended State Machines

States or transitions represent atomic instructions

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

Can Thread 1 be in C1 at the same time Thread 2 is in C2?
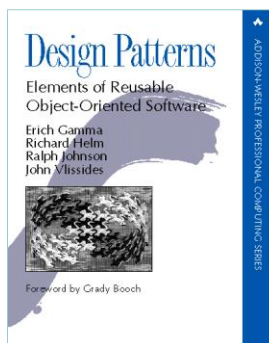
Real-Time Embedded Systems 9

A Scenario

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

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

Proper software engineering practice suggests using the observer pattern.

Real-Time Embedded Systems 10

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)
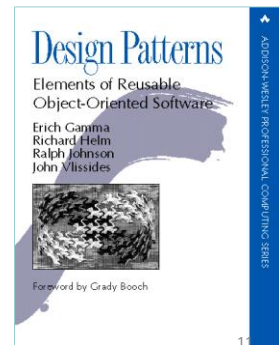
Real-Time Embedded Systems

11

Observer Pattern in C

```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) {…}
```

Real-Time Embedded Systems

12

Observer Pattern in C

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

// List of listeners. A li        typedef void* notifyProcedure(int);
// pointers to notify proc         struct element {
typedef void* notifyProce           notifyProcedure* listener;
struct element {…}                   struct element* next;
typedef struct element ele         };
elementType* head = 0;             typedef struct element elementType;
elementType* tail = 0;             elementType* head = 0;
                                   elementType* tail = 0;
// Procedure to add a 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      // Procedure to add a listener to the list.
// registered      void addListener(notifyProcedure listener) {
int value;           if (head == 0) {
                       head = malloc(sizeof(elementType));
// List of lis         head->listener = listener;
// pointers to         head->next = 0;
typedef void*          tail = head;
struct element       } else {
typedef struct         tail->next = malloc(sizeof(elementType));
elementType* h         tail = tail->next;
elementType* t         tail->listener = listener;
                       tail->next = 0;
// Procedure t       }
void addListen     }

// Procedure t
void update(in
                 }
// 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 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;
  }
}
```

Real-Time Embedded Systems   15

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  }
```
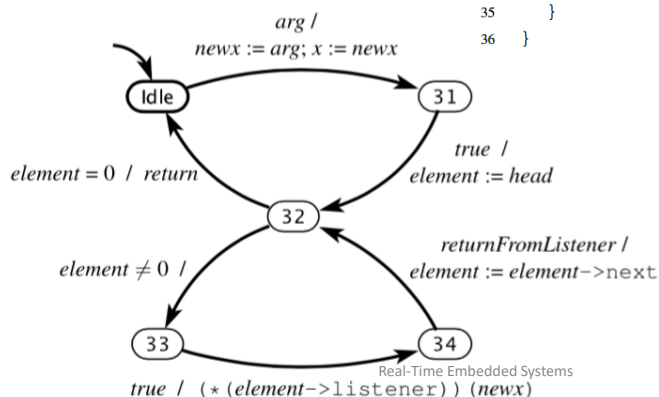
inputs: *arg*: int, *returnFromListener*: pure
outputs: *return*: pure
local variables: *newx*: int, *element*: element_t*
global variables: *x*: int, *head*: element_t*

arg /
newx := arg; x := newx

Idle → 31

element = 0 / return

true /
element := head

32

returnFromListener /
element := element->next

element ≠ 0 /

33 → 34

true / ((element->listener))(newx)*

Real-Time Embedded Systems   16

8

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?

What if addListener is called from two threads at the same time?

Real-Time Embedded Systems   17

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

Using Posix mutexes on the observer pattern in C

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.

Real-Time Embedded Systems   18

9

One possible "fix"

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

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.

Real-Time Embedded Systems   19

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.

Real-Time Embedded Systems   20

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

It is not concurrency that is hard…

Real-Time Embedded Systems 21

…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…

Real-Time Embedded Systems 22

What it Feels Like to Use Mutexes

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

Real-Time Embedded Systems 23

Message-passing programs *may* be better

```
1   void* producer(void* arg) {
2       int i;
3       for (i = 0; i < 10; i++) {
4           send(i);
5       }
6       return NULL;
7   }
8   void* consumer(void* arg) {
9       while(1) {
10          printf("received %d\n", get());
11      }
12      return NULL;
13  }
14  int main(void) {
15      pthread_t threadID1, threadID2;
16      void* exitStatus;
17      pthread_create(&threadID1, NULL, producer, NULL);
18      pthread_create(&threadID2, NULL, consumer, NULL);
19      pthread_join(threadID1, &exitStatus);
20      pthread_join(threadID2, &exitStatus);
21      return 0;
22  }
```

But there is still risk of deadlock and unexpected nondeterminism!

Real-Time Embedded Systems 24

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)*

Real-Time Embedded Systems   25

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

Real-Time Embedded Systems   26

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.

Real-Time Embedded Systems   27

Basic Sequential Computation

initial state: $b_0 \in B^*$

sequential composition

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

final state: b_N
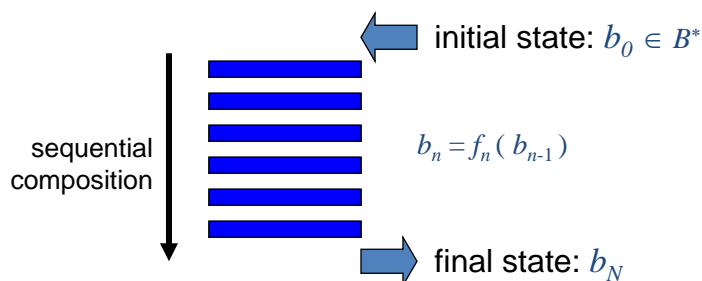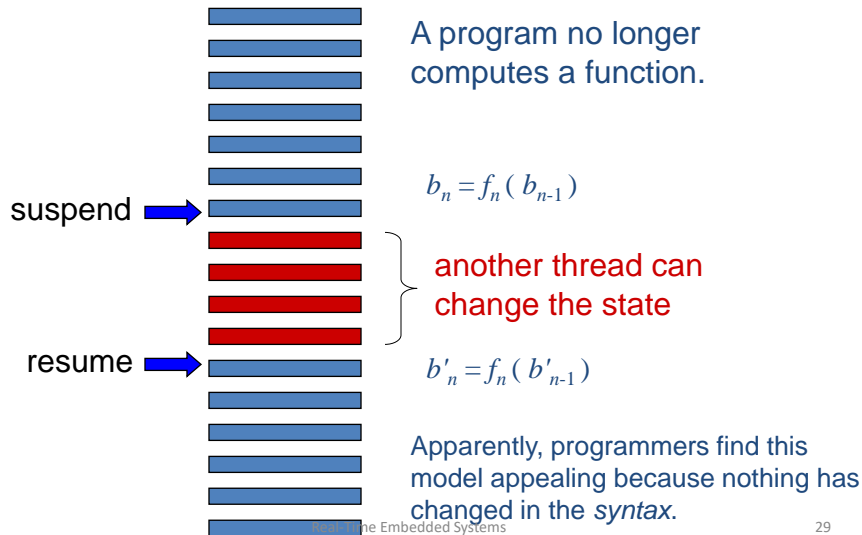
Formally, composition of computations is function composition.

Real-Time Embedded Systems   28

When There are Threads, Everything Changes

A program no longer computes a function.

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

suspend ➡

another thread can change the state

resume ➡

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

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

Real-Time Embedded Systems   29

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).

Real-Time Embedded Systems   30

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 …)

Real-Time Embedded Systems   31

IEEE Computer, May, 2006

COVER FEATURE

Future
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.

Real-Time Embedded Systems   32

16