Real-time systems D0003E

Lecture 6:
A model of reactive objects
(c.f. Burns&Wellings, ch. 9 & ch. 8)

Overview

- Reactive objects
  - hardware
  - software
- TinyTimer kernel
- Synchronous calls
- Asynchronous calls
- Deadlock

The reactive embedded system

reactive objects

- Each box, software or hardware, represents a reactive object
  - Maintains an internal state (variables, registers, etc)
  - Provides a set of methods as reactions to external events (ISRs, digital sequencers, etc)
  - Simply rests between reactions
  - May react in parallel with other objects (threads, chips)
- Each arrow represents an event/signal/message flow between objects, which can be either
  - asynchronous (write, sender continues in parallel)
  - synchronous (read, sender waits for a reply)

Hardware objects

- Object class
  - the kind/model of a circuit
- Object instance
  - a particular circuit on a particular board
- Object state
  - internal register/logic status of an object instance
- Provided interface
  - the set of pins on a circuit that recognize signals
- Required interface
  - the set of pins on a circuit that generate signals
- Method call
  - to raise an input signal and...
    - wait for a response (asynchronous)
    - just continue (asynchronous)

Software objects

- Object class:
  - program behavior expressed as state variable layout and method code
- Object instance:
  - a record of state variables at a particular address (object identity)
- Object state:
  - current state variable contents of a particular object
- Provided interface:
  - set of methods a class exports
- Required interface:
  - methods calls issued to other objects
- Method call:
  - to call a function with the designated object address as the first argument...
State layout in TinyTimber

- In MyClass.h:
  ```
  #include "TinyTimber.h"
  typedef struct {
    Object super;
    int x;
    char y;
  } MyClass;
  #define initMyClass(yy) \
    { initObject(), 0, yy }
  ```

- In any client file:
  ```
  #include "MyClass.h"
  MyClass a = initMyClass(13);
  ```

Compare to Java/C++ objects

- We don't allocate objects on the heap
- Our "constructors" are just preprocessor macros
- Static object structure

Method calls

```
class MyClass {
  int x;
  char y;
  ...
  void myMethod( int q ) {
    x = y + q;
  }
}
```

```
typedef struct {
  int x;
  char y;
} MyClass;

void myMethod( MyClass *this, int q ) {
  this->x = this->y + q;
}
```

```
MyClass a = ...
...
myMethod( &a, 44 );
```

Encoding method calls

We write "self" instead of "this" to remind ourselves that we're not really working with C++/Java

```
void myMethod( MyClass *this, int q ) {
  self->x = self->y + q;
}
```

Asynchronous calls

- Object A calls meth on object B
  - asynchronously
  - executes simultaneously

```
A
  ASYNC(B, meth, 73)

B
  (resting)
  meth(B, 73)
```

```
A
  ASYNC(B, meth, 73)

B
  some other method
  meth(B, 73)
```

```
A
  ASYNC(B, meth, 73)

B
  (resting)
  pseudo-parallel execution
  meth(B, 73)
```

```
A
  ASYNC(B, meth, 73)

B
  some other method
  pseudo-parallel execution
```
Synchronous call

- Object A sleeps
  - waiting for object B to finish
  - resumes after B has finished
- corresponds to function call in normal OO language

```
SYNC(B, meth, 73)
```

Implementation of SYNC

```
int sync( Object *to, Meth method, int arg ) {
    int result;
    lock( &to->mutex );
    result = method( to, arg );
    unlock( &to->mutex );
    return result;
}
```

```
define SYNC( obj, meth, arg )   sync( (Object*)obj, (Meth)meth, arg )
```

Implementing ASYNC

```
void async( Object *to, Meth meth, int arg ) {
    Msg msg = dequeue( &freeQ ); //pick new thread from queue
    msg->function = method;
    msg->arg = arg;
    msg->to = to;
    if (setjmp(msg->context) != 0) {
        sync( current->to, current->function, current->arg );
        enqueue( current, &freeQ );
    }
    STACKPTR(msg->context) = &msg->stack;
    enqueue( msg, &readyQ );
}
```

```
define ASYNC(obj,meth, arg) async( (Object*)obj, (Meth)meth, arg )
```

That is...

- Threads
  - replaced by the notion of asynchronous messages
- Old operation spawn
  - superceded by async
- Old operations lock and unlock
  - being used exclusively within sync
- New kernel interface:
  - void async( Object* to, Meth method, int arg );
  - int sync( Object* to, Meth method, int arg );
+ typedefs for Object and Meth defines for ASYNC and SYNC

Observations

- Serialization of object methods
  - just like standard mutual exclusion
- synchronous call
  - just like a mutex-protected function call
- asynchronous call
  - introduce concurrency
  - Asynchronous calls need additional temporary storage
    - if call can't execute immediately
    - have to remember to make the call later
  - Suggestion: let an asynchronous call be equivalent to a synchronous call executed by a new thread!
    - new thread is waiting to call
Implicit design rules

- All object structs must ‘inherit’ Object
  - which means they must contain an Object as the first component
- Object instantiation is done declaratively on the top level
  (static object structure), e.g.:
  ```
  ClassA a = initClassA(ival);
  ClassB b1 = initClassB();
  ClassB b2 = initClassB();
  ```
- method call goes to another object
  - either SYNC or ASYNC must be used
- All methods must take arguments "self" and an int
  - unfortunate limitation with current system

What about ASYNC to self?

- Asynchronous call to the same thread
  - thread finishes what it is doing
  - starts executing previous call
  - call "waits"
    - as usual

What about SYNC to self?

- Sunc call to same thread
  - thread waits for call to finish
  - call wont start until thread finished
- Nothing happens
  - situation called deadlock

Deadlock

- Deadlock is the result of requesting new exclusive access to something you already have
- In general, a chain of tasks may actually be involved:

```
T1
  m1
  T1 holds m1
  T1 wants m2 (waits)

T2
  m2
  T2 holds m2
  T2 wants m3 (waits)

T3
  m3
  T3 holds m3
  T3 wants m1 (waits)
```

Deadlock: Necessary conditions

- Mutual exclusion
  - only one process at a time can use resource
- Hold and wait
  - processes holding resource
    - ...waiting for another resource
- No preemption
  - resource cannot be taken away from process
  - process can only release it voluntarily
- Circular wait
  - chain of processes
    - holds resource
    - ...waits for another resource
      - which is already held by other process

Deadlock

```
car() {
    take X
    take Y
    drive...
    release Y
    release X
}
```
**Deadlock prevention**

- Remove any of the conditions
  - deadlock will not occur
- 1: disallow mutual exclusion
- 2: don’t allow wait for resource while holding resource
  - allocate everything at start, for example
    - not efficient in most cases

- 3: introduce means of taking resources back
- 4: prevent circular wait
  - impose linear ordering of resources
  - only allow allocation in order
    - the “last circular” allocation will not be allowed

**Deadlock avoidance**

- Idea: don’t allow the system to enter an “unsafe” state
  - what is “unsafe” and what is “state”?
- State:
  - # resources available
  - # allocated resources per process
  - # demanded resources per process

**Deadlock avoidance: Safe state**

- If it is possible to...
  - allocate resources to each process
  - in some order
  - not enter deadlock state
- This can be upheld by the system
- Consider example...

**Deadlock avoidance: example**

- Consider state given by table
  - Safe state
- Exists sequence
  - (P1, P0, P2)
- Such that
  - all processes can terminate

<table>
<thead>
<tr>
<th>Process</th>
<th>Has</th>
<th>Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>Total allocated:</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Units available:</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process</th>
<th>Has</th>
<th>Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0</td>
<td>done</td>
</tr>
<tr>
<td>P1</td>
<td>0</td>
<td>done</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>Total allocated:</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Units available:</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

**Deadlock avoidance: example**

- Consider state given by table
  - Safe state
- Exists sequence
  - (P1, P0, P2)
- Such that
  - all processes can terminate

<table>
<thead>
<tr>
<th>Process</th>
<th>Has</th>
<th>Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>Total allocated:</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Units available:</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process</th>
<th>Has</th>
<th>Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0</td>
<td>done</td>
</tr>
<tr>
<td>P1</td>
<td>0</td>
<td>done</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>Total allocated:</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Units available:</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process</th>
<th>Has</th>
<th>Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0</td>
<td>done</td>
</tr>
<tr>
<td>P1</td>
<td>0</td>
<td>done</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>Total allocated:</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Units available:</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

P1 allocates 2
P1 hands back 4, P0 allocates 5
P0 hands back 10, P2 allocates 7
P1 hands back 4, P0 allocates 5
P0 hands back 10, P2 allocates 7
P1 allocates 2

Deadlock avoidance: example

- Consider state given by table
  - Unsafe state
- Does Not exists sequence
- Such that
  - all processes can terminate

<table>
<thead>
<tr>
<th>Process</th>
<th>Max</th>
<th>Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>Units available: 10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Deadlock avoidance: In general

- unsafe state
  - we may get deadlock
  - not for sure
- safe
  - we will not get deadlock
  - for sure
- If more than one resource:
  - Banker’s algorithm
  - resource allocation
- Resource utilization
  - tends to be inefficient
  - we do not allow unsafe states
  - might not lead to deadlock after all...

Deadlock via SYNC

- consider possible calls
- circle in call graph
  - possible deadlock

Deadlock via SYNC

- Sufficient deadlock protection
  - insert at least one ASYNC
  - thread will not block
  - ASYNC call executed after all SYNC calls have finished

Creating cyclic structures

typedef struct {
  Object super;
  float x;
  ClassB *b;
} ClassA;

#define initClassA(b)  
{ initObject(), 0.0, b }

void ameth( ClassA *self, int q ) {
  ... 
  ASYNC( self->b, bmeth, 's' );
}

extern ClassB b;
ClassA a = initClassA(&b);

typedef struct {
  Object super;
  char *ch;
} ClassB;

#define initClassB(a)  
{ initObject(), "abc", a }

void bmeth( ClassB *self, char z ) {
  ... 
  ASYNC( self->a, ameth, 17 );
}

ClassB b = initClassB(&a);

Connecting the external world

interrupt
write to port
read from port

interrupt
write to port
read from port
The top-level object

- ... is just like any other reactive object, although
- implicitly "instantiated" when power is turned on
- state
  - all global variables
  - many will be reactive objects in their own right
- its methods are the installed interrupt handlers
- its "self" is only conceptual (no concrete pointer...)
- top-level object method scheduling
  - CPU hardware
  - not the TinyTimber kernel

Connecting interrupts

INSTALL(&obj, meth, IRQ_X)

(method meth on object obj will be invoked with arg IRQ_X as interrupt handler)

Example

```c
#include "TinyTimber.h"

typedef struct {
  Object super;
  int val;
} Counter;

#define initCounter(n)    { initObject(), n }

int inc( Counter *self, int arg ) {
  self->val = self->val + 1;
}

int reset( Counter *self, int arg ) {
  self->val = arg;
}
```

Example client

- In main.c
  - Counter counter = initCounter(0);
  - INSTALL(&counter, inc, SIG_PIN_CHANGE1);
- And also in main.c:
  - ...
  - something to reset the counter...
  - ...

Reset

Complication: the reset interrupt routine cannot return, as it hasn’t really interrupted anything...

Here’s where the active system view has its roots, seeing the reset signal as an obligation to compute until someone turns off the power...

An interrupt routine just like any other!
main()

- main() function in C
  - abstraction of the reset handler...
- ... just as a program itself
  - abstraction of the notion of “running a computer until it stops”
- In traditional programs main() does indeed return
  - request to the OS to “turn off the power”
    - to the virtual computer that was set up to run the program
- reactive system
  - don’t want power to be turned off at all
    - but we also don’t want to let main() compute forever
  - just to keep it from returning...
    (recall that a reactive system rests when it isn’t reacting)

The idle task

- Solution
  - let main() finish by literally putting the CPU to sleep until the next interrupt!
- Most architectures
  - special machine instruction that does so (called “sleep” on the AVR)
- Hence we want main() to finish by calling this code:
  ```c
  void idle() {
    ENABLE();
    while (1) SLEEP();
  }
  ```
- We capture this pattern by a macro that replaces main():
  ```c
#define STARTUP(stmt)   int main() { initialize(); stmt; idle(); }
  ```

Example client + reset

- In main.c
  ```c
  Counter counter = initCounter(0);
  INSTALL(&counter, inc, SIG_PIN_CHANGE1);
  return TINYTIMBER(&counter, reset, 0);
  ```

Sanity rules

- In a system of reactive objects,
  - methods only access variables that belong to self
  - global variables that are not objects, are considered local to the top-level object
  - method calls between objects that are wrapped within a SYNC or ASYNC shield
- Properly upheld, these rules guarantee a system that is
  - free from deadlock (provided absence of cyclic SYNC)
  - free from critical section race conditions

Next lecture

- Clocks
- timers
- periodic execution
  - scheduling