Embedded Systems Programming

Lecture 6

www2.hh.se/staff/vero/embeddedProgramming

Verónica Gaspes

www2.hh.se/staff/vero

Center for Research on Embedded Systems

School of Information Science, Computer and Electrical Engineering

Encoding state layout

We will use a little kernel called `TinyTimber`. We will use files as modules in C.

In MyClass.h

```
#include "TinyTimber.h"

typedef struct{
    Object super;
    int x;
    char y;
} MyClass;

#define initMyClass(z) \
  { initObject ,0,z}
```

Unconstrained!

Using it

```
#include "MyClass.h"
MyClass a = initMyClass(13);
```

Encoding state layout

We will use a little kernel called `TinyTimber`. We will use files as modules in C.

In `MyClass.h`

```
#include "TinyTimber.h"

typedef struct{
    Object super;
    int x;
    char y;
} MyClass;

#define initMyClass(z) \
  { initObject ,0,z}
```

Using it

```
#include "MyClass.h"
MyClass a = initMyClass(13);
```

Encoding state layout

We will use a little kernel called `TinyTimber`. We will use files as modules in C.

In `MyClass.h`

```
#include "TinyTimber.h"

typedef struct{
    Object super;
    int x;
    char y;
} MyClass;

#define initMyClass(z) \
  {initObject,0,z}
```

- Mandatory! Specified and used by the kernel!
- Unconstrained!
- `initMyClass` corresponds to a constructor, it includes programmer defined intialization.

Using it

```
#include "MyClass.h"
MyClass a = initMyClass(13);
```

Encoding state layout

We will use a little kernel called `TinyTimber`. We will use files as modules in C.

In `MyClass.h`

```
#include "TinyTimber.h"

typedef struct{
    Object super;
    int x;
    char y;
} MyClass;

#define initMyClass(z) \
  {initObject,0,z}
```

- Mandatory! Specified and used by the kernel!
- Unconstrained!
- `initMyClass` corresponds to a constructor, it includes programmer defined intialization.

Using it

```
#include "MyClass.h"
MyClass a = initMyClass(13);
```

Encoding state layout

We will use a little kernel called `TinyTimber`. We will use files as modules in C.

In `MyClass.h`

```
#include "TinyTimber.h"

typedef struct{
    Object super;
    int x;
    char y;
} MyClass;

#define initMyClass(z) \
  { initObject ,0,z}
```

- Mandatory! Specified and used by the kernel!
- Unconstrained!
- `initMyClass` corresponds to a constructor, it includes programmer defined intialization.

Using it

```
#include "MyClass.h"
MyClass a = initMyClass(13);
```

Encoding state layout

We will use a little kernel called `TinyTimber`. We will use files as modules in C.

In `MyClass.h`

```
#include "TinyTimber.h"

typedef struct{
    Object super;
    int x;
    char y;
} MyClass;

#define initMyClass(z) \
  { initObject ,0,z}
```

- Mandatory! Specified and used by the kernel!
- Unconstrained!
- `initMyClass` corresponds to a constructor, it includes programmer defined intialization.

Using it

```
#include "MyClass.h"
MyClass a = initMyClass(13);
```

Comparing with Java

```
class MyClass{
  int x;
  char y;
  MyClass(int z){
     x=0;
     y=z;
  }
}
```

```
MyClass a = new MyClass(13);
```

In our programs we do not allocate objects in the heap (as Java does!).

Our constructors are just preprocessor macros!

Comparing with Java

```
class MyClass{
  int x;
  char y;
  MyClass(int z){
     x=0;
     y=z;
  }
}
```

```
MyClass a = new MyClass(13);
```

In our programs we do not allocate objects in the heap (as Java does!).

Our constructors are just preprocessor macros!

Comparing with Java

```
class MyClass{
  int x;
  char y;
  MyClass(int z){
     x=0;
     y=z;
  }
}
```

```
MyClass a = new MyClass(13);
```

In our programs we do not allocate objects in the heap (as Java does!).

Our constructors are just preprocessor macros!

Comparing with Java

```
class MyClass{
  int x;
  char y;
  MyClass(int z){
     x=0;
     y=z;
  }
}
```

```
MyClass a = new MyClass(13);
```

In our programs we do not allocate objects in the heap (as Java does!).

Our constructors are just preprocessor macros!

Comparing with Java

```
class MyClass{
  int x;
  char y;
  MyClass(int z){
     x=0;
     y=z;
  }
}
```

```
MyClass a = new MyClass(13);
```

In our programs we do not allocate objects in the heap (as Java does!).

Our constructors are just preprocessor macros!

Encoding methods declarations

In MyClass.h

```
typedef struct{
    Object super;
    int x;
    char y;
} MyClass;
...
int myMethod( MyClass *self , int q);
```

In MyClass.c

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

In Java

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

Encoding methods declarations

In MyClass.h

```
typedef struct{
    Object super;
    int x;
    char y;
} MyClass;
...
int myMethod( MyClass *self , int q);
```

In MyClass.c

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

In Java

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

Encoding methods declarations

In MyClass.h

```
typedef struct{
   Object super;
   int x;
   char y;
} MyClass;
...
int myMethod(MyClass *self, int q);
```

In MyClass.c

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

In Java

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

Encoding methods declarations

In MyClass.h

```
typedef struct{
   Object super;
   int x;
   char y;
} MyClass;
...
int myMethod(MyClass *self, int q);
```

In MyClass.c

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

In Java

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

Encoding function calls

In Java

```
...
MyClass a = new MyClass(13);
a.myMethod(44);
```

In our C programs

```
...
MyClass a = initMyClass(13);
myMethod( &a ,44);
```

But, we are doing all this to do something different than just function calls! We want to have the possibility of introducing the distinction between synchronous and asynchronous messages!

Encoding function calls

In Java

```
...
MyClass a = new MyClass(13);
a.myMethod(44);
```

In our C programs

```
...
MyClass a = initMyClass(13);
myMethod(&a,44);
```

But, we are doing all this to do something different than just function calls! We want to have the possibility of introducing the distinction between synchronous and asynchronous messages!

Encoding function calls

In Java

```
...
MyClass a = new MyClass(13);
a.myMethod(44);
```

In our C programs

```
...
MyClass a = initMyClass(13);
myMethod( &a ,44);
```

But, we are doing all this to do something different than just function calls! We want to have the possibility of introducing the distinction between synchronous and asynchronous messages!

Encoding function calls

In Java

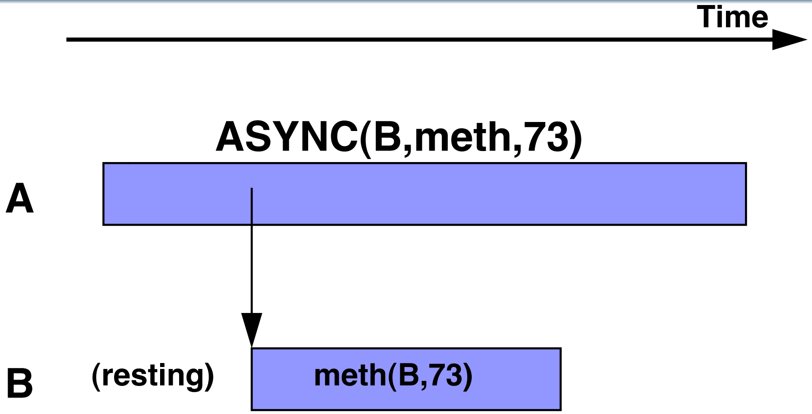
```
...
MyClass a = new MyClass(13);
a.myMethod(44);
```

In our C programs

```
...
MyClass a = initMyClass(13);
myMethod( &a ,44);
```

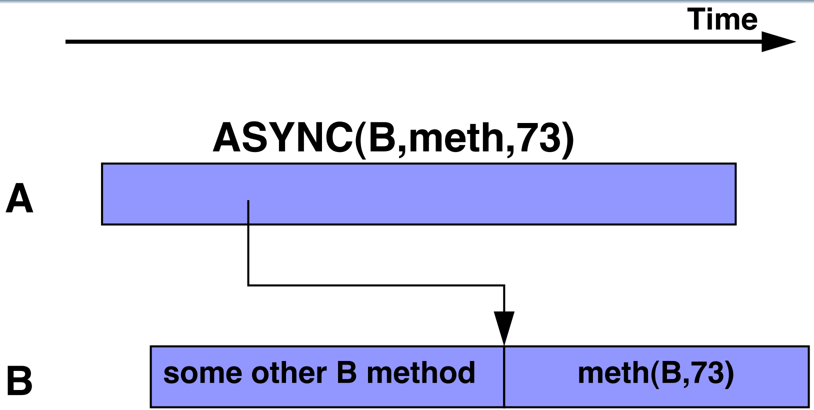
But, we are doing all this to do something different than just function calls! We want to have the possibility of introducing the distinction between synchronous and asynchronous messages!

Asynchronous calls



(Pseudo-) parallel execution!

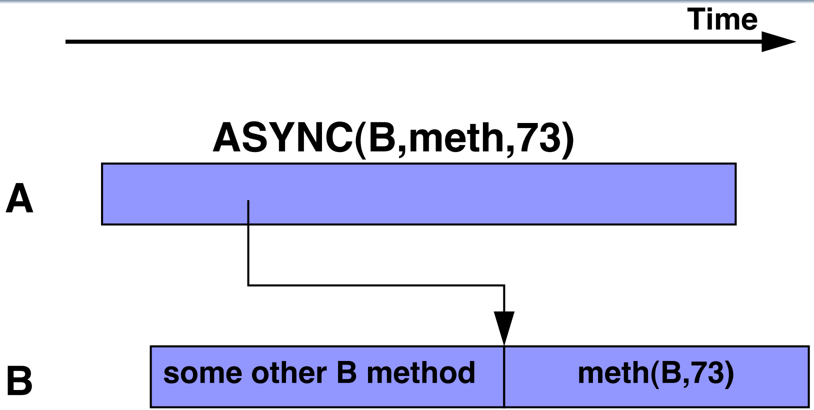
Asynchronous calls



(Pseudo-) parallel execution between A and B.

Strictly sequential execution between B's methods!

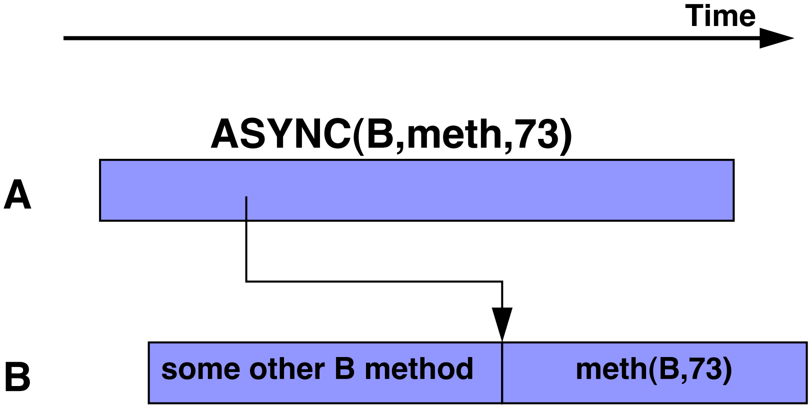
Asynchronous calls



(Pseudo-) parallel execution between A and B.

Strictly sequential execution between B's methods!

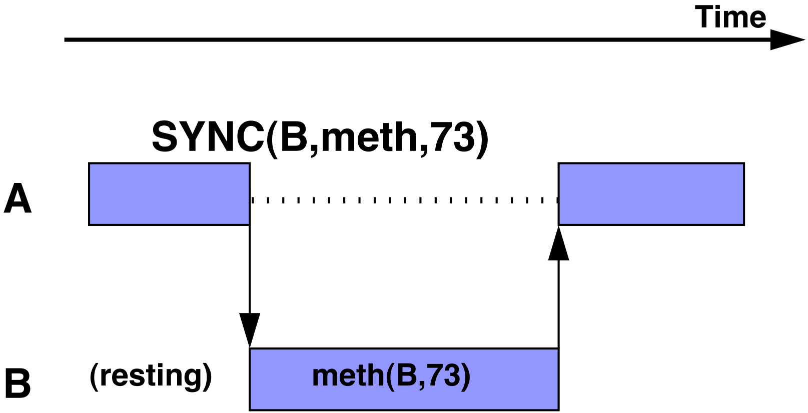
Asynchronous calls



(Pseudo-) parallel execution between A and B.

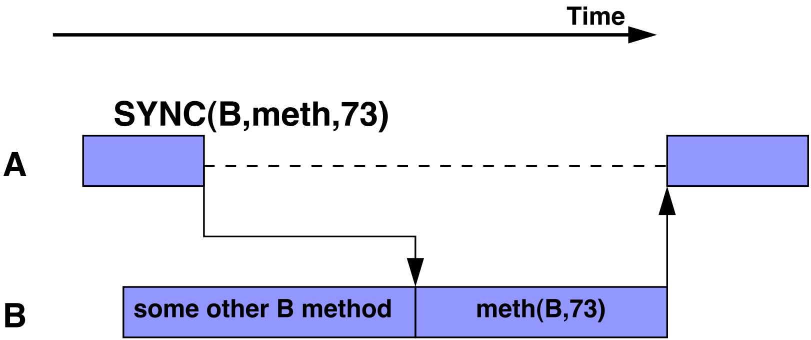
Strictly sequential execution between B's methods!

Synchronous calls



Strictly sequential execution between A and B!

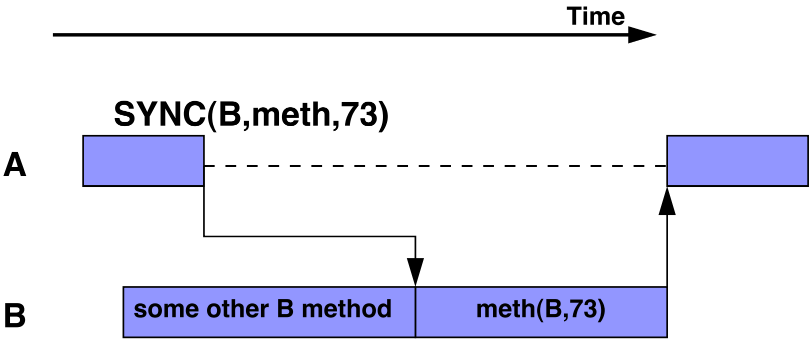
Synchronous calls



(Pseudo-) parallel execution between A and B's other method.

Strictly sequential execution between B's methods and between A the method called synchronously.

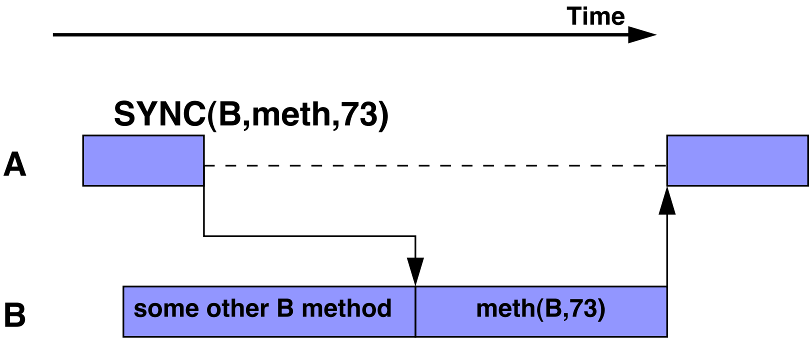
Synchronous calls



(Pseudo-) parallel execution between A and B's other method.

Strictly sequential execution between B's methods and between A and the method called synchronously.

Synchronous calls



(Pseudo-) parallel execution between A and B's other method.

Strictly sequential execution between B's methods and between A and the method called synchronously.

Observations

- Serialization of object methods looks just like standard mutual exclusion.
- A synchronous call is just like a mutex-protected function call.
- It is the asynchronous calls that introduce concurrency.
- Asynchronous calls further more need additional temporary storage (if a call cannot execute immediately)
- Suggestion: let an asynchronous call be equivalent to a synchronous call executed by a fresh thread!

Observations

- Serialization of object methods looks just like standard mutual exclusion.
- A synchronous call is just like a mutex-protected function call.
- It is the asynchronous calls that introduce concurrency.
- Asynchronous calls further more need additional temporary storage (if a call cannot execute immediately)
- Suggestion: let an asynchronous call be equivalent to a synchronous call executed by a fresh thread!

Observations

- Serialization of object methods looks just like standard mutual exclusion.
- A synchronous call is just like a mutex-protected function call.
- It is the asynchronous calls that introduce concurrency.
- Asynchronous calls further more need additional temporary storage (if a call cannot execute immediately)
- Suggestion: let an asynchronous call be equivalent to a synchronous call executed by a fresh thread!

Observations

- Serialization of object methods looks just like standard mutual exclusion.
- A synchronous call is just like a mutex-protected function call.
- It is the asynchronous calls that introduce concurrency.
- Asynchronous calls further more need additional temporary storage (if a call cannot execute immediately)
- Suggestion: let an asynchronous call be equivalent to a synchronous call executed by a fresh thread!

Observations

- Serialization of object methods looks just like standard mutual exclusion.
- A synchronous call is just like a mutex-protected function call.
- It is the asynchronous calls that introduce concurrency.
- Asynchronous calls further more need additional temporary storage (if a call cannot execute immediately)
- Suggestion: let an asynchronous call be equivalent to a synchronous call executed by a fresh thread!

Observations

- Serialization of object methods looks just like standard mutual exclusion.
- A synchronous call is just like a mutex-protected function call.
- It is the asynchronous calls that introduce concurrency.
- Asynchronous calls further more need additional temporary storage (if a call cannot execute immediately)
- Suggestion: let an asynchronous call be equivalent to a synchronous call executed by a fresh thread!

Implementing SYNC

In TinyTimber.c

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

Every object has to have its own mutex and we need a way to force every instance to have type Object!

Implementing SYNC

In `TinyTimber.c`

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

Every object has to have its own mutex and we need a way to force every instance to have type Object!

Implementing SYNC

In `TinyTimber.h`

```
typedef struct{
   mutex mutex;
} Object;

typedef int (*Meth)(Object*,int);

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

Implementing ASYNC

In TinyTimber.c

```
void async(Object* to, Method meth, int arg){
  Msg msg         = dequeue(&freeQ);
  msg->function = meth;
  msg->arg        = arg;
  msg->to         = to;

  if(setjmp(msg->context)!=0){
     sync(current->to,current->function,current->arg);
     enqueue(current,&freeQ);
     dispatch(dequeue(&readyQ));
  }

  STACKPTR(msg->context)=&msg->stack;
  enqueue(msg,&readyQ);
}
```

Implementing ASYNC

In TinyTimber.c

```
void async(Object* to, Method meth, int arg){
  Msg msg          = dequeue(&freeQ);
  msg->function = meth;
  msg->arg         = arg;
  msg->to          = to;

  if(setjmp(msg->context)!=0){
     sync(current->to,current->function,current->arg);
     enqueue(current,&freeQ);
     dispatch(dequeue(&readyQ));
  }

  STACKPTR(msg->context)=&msg->stack;
  enqueue(msg,&readyQ);
}
```

Implementing `ASYNC`

In `TinyTimber.h`

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

Summary

- Threads are replaced by asynchronous messages
- Old operation `spawn` superceeded by `async`
- Old oprations `lock` and `unlock` are only used inside `sync`
- The new kernel interface:

```
void async(Object *to, Meth meth, int arg)
int sync(Object *to, Meth meth, int arg)


typedefs for Object and Meth
defines for ASYNC and SYNC
```

Summary

- Threads are replaced by asynchronous messages
- Old operation `spawn` superceeded by `async`
- Old oprations `lock` and `unlock` are only used inside `sync`
- The new kernel interface:

```
void async(Object *to, Meth meth, int arg)
int sync(Object *to, Meth meth, int arg)

typedefs for Object and Meth
defines for ASYNC and SYNC
```

Summary

- Threads are replaced by asynchronous messages
- Old operation `spawn` superceeded by `async`
- Old oprations `lock` and `unlock` are only used inside `sync`
- The new kernel interface:

```
void async(Object *to, Meth meth, int arg)
int sync(Object *to, Meth meth, int arg)

typedefs for Object and Meth
defines for ASYNC and SYNC
```

Summary

- Threads are replaced by asynchronous messages
- Old operation `spawn` superceeded by `async`
- Old oprations `lock` and `unlock` are only used inside `sync`
- The new kernel interface:

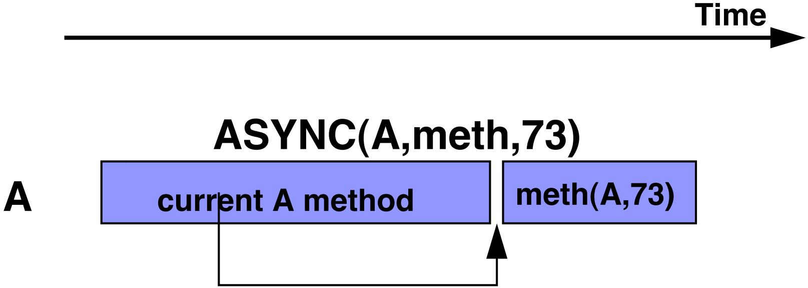
```
void async(Object *to, Meth meth, int arg)
int sync(Object *to, Meth meth, int arg)

typedefs for Object and Meth
defines for ASYNC and SYNC
```

Summary

- Threads are replaced by asynchronous messages
- Old operation `spawn` superceeded by `async`
- Old oprations `lock` and `unlock` are only used inside `sync`
- The new kernel interface:

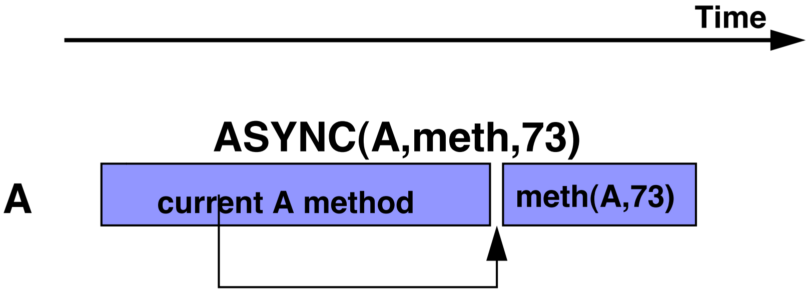
```
void async(Object *to, Meth meth, int arg)
int sync(Object *to, Meth meth, int arg)

typedefs for Object and Meth
defines for ASYNC and SYNC
```

ASYNC to self?



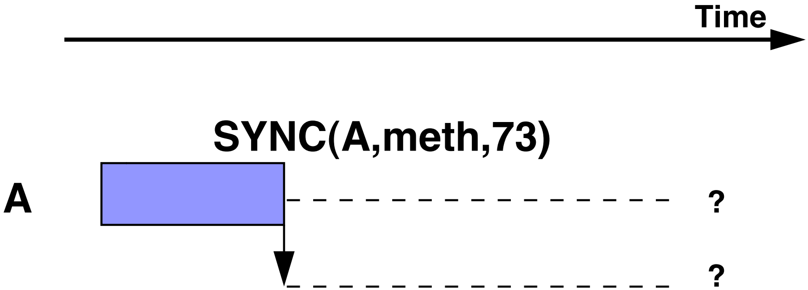
Strictly sequential execution!

ASYNC to self?



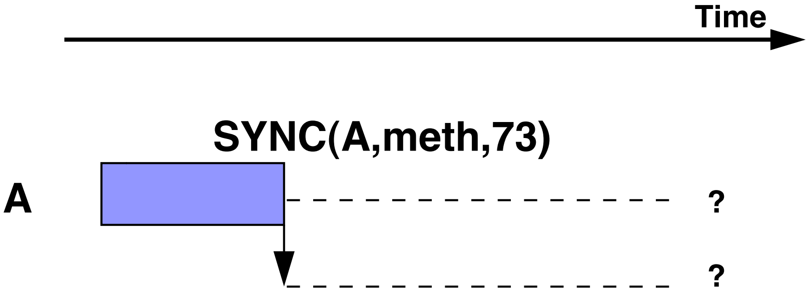
Strictly sequential execution!

SYNC to self?



DEADLOCK!

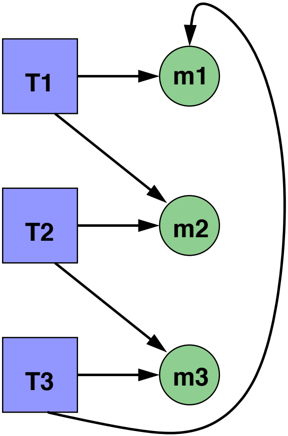
SYNC to self?



DEADLOCK!

Deadlock

Deadlock arises when requesting new exclusive access to something you already have. In general, a chain of tasks may be involved:



T1 holds **m1**
T1 wants **m2**

T2 holds **m2**
T2 wants **m3**

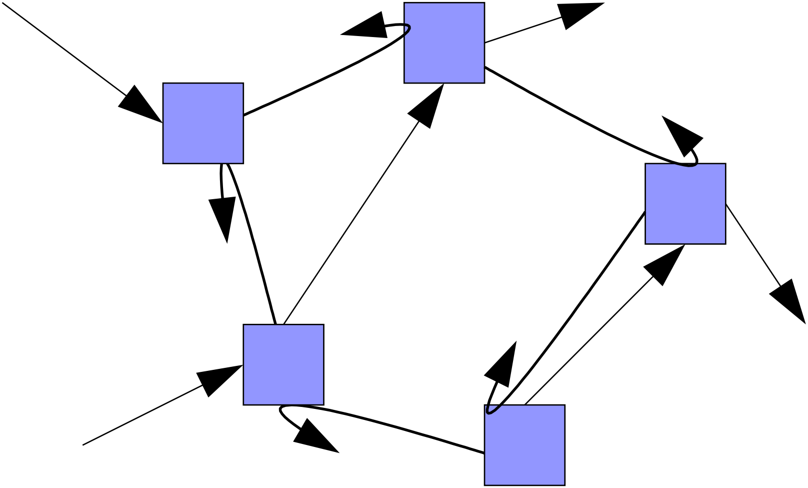
T3 holds **m3**
T3 wants **m1**

Deadlock

A system in deadlock will remain stuck, unless a thread chooses to back off from its current claim ...

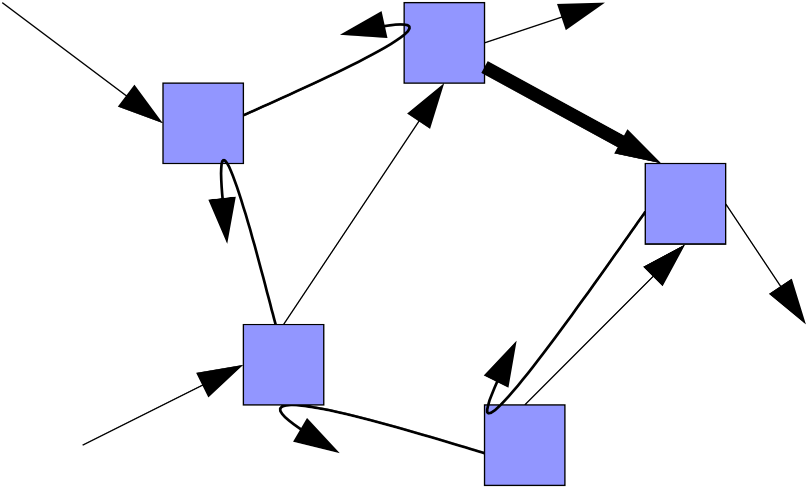
Deadlock in the real world

Deadlock via SYNC



A cycle of possible simultaneus calls to SYNC

Deadlock via SYNC



Sufficient deadlock protection: insert at least one `ASYNC`.

Programming idiom

1. Classes

All objects must *inherit* `Object`:

```
typedef struct{
  Object super;
  // extra fields
} MyClass;
```

2. Objects

Object instantiation is done declaratively on the top level (static object structure):

```
ClassA a = initClassA(ival);
ClassB b1 = initClassB();
ClassB b2 = initClassB();
```

Programming idiom

1. Classes

All objects must *inherit* `Object`:

```
typedef struct{
  Object super;
  // extra fields
} MyClass;
```

2. Objects

Object instantiation is done declaratively on the top level (static object structure):

```
ClassA a = initClassA(ival);
ClassB b1 = initClassB();
ClassB b2 = initClassB();
```

Programming idiom

1. Classes

All objects must *inherit* `Object`:

```
typedef struct{
  Object super;
  // extra fields
} MyClass;
```

2. Objects

Object instantiation is done declaratively on the top level (static object structure):

```
ClassA a = initClassA(ival);
ClassB b1 = initClassB();
ClassB b2 = initClassB();
```

Programming idiom (ctd.)

3. Method calls

Whenever a method call goes to another object, either `SYNC` or `ASYNC` must be used.

(Tiny) Limitation

All methods must take arguments `self` and an int!

Programming idiom (ctd.)

3. Method calls

Whenever a method call goes to another object, either `SYNC` or `ASYNC` must be used.

(Tiny) Limitation

All methods must take arguments `self` and an `int`!

Programming idiom (ctd.)

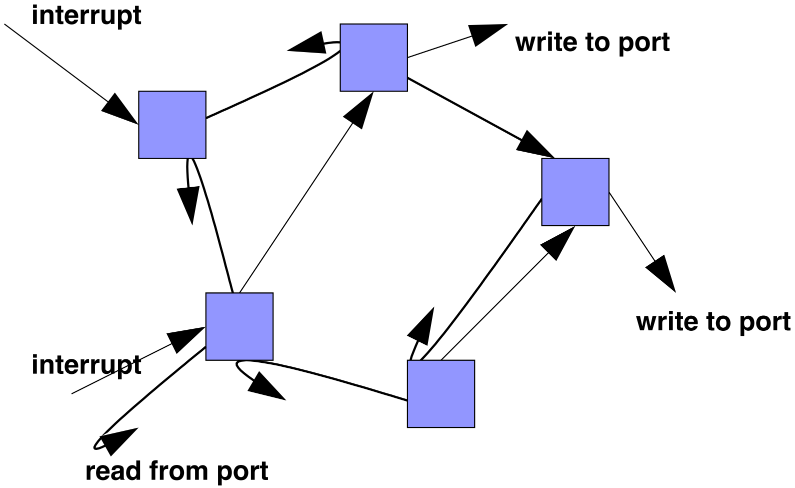
3. Method calls

Whenever a method call goes to another object, either `SYNC` or `ASYNC` must be used.

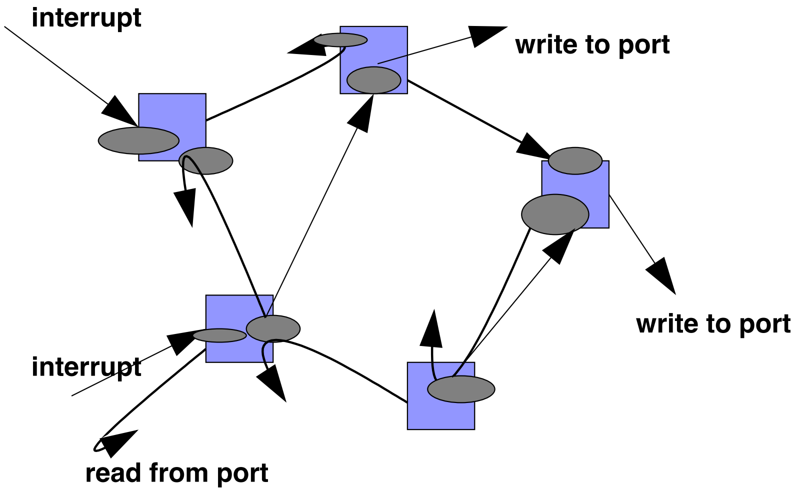
(Tiny) Limitation

All methods must take arguments `self` and an int!

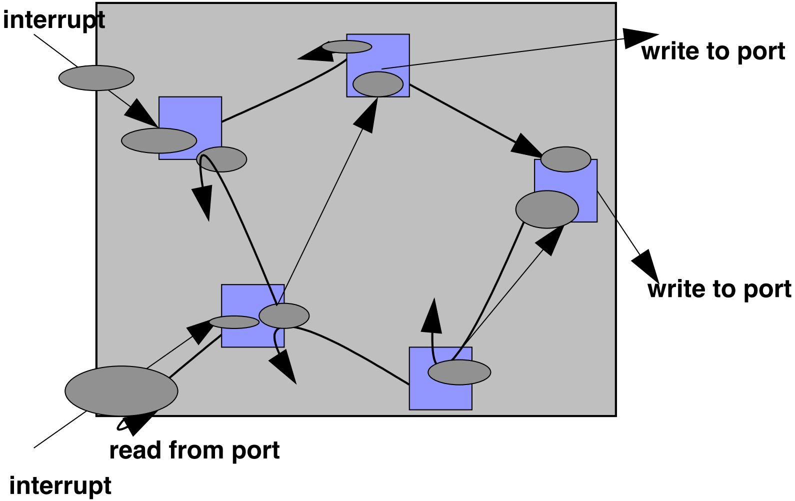
Connecting the external world



Making the methods explicit



The top-level object



Notice the interrupt handlers.

The top-level object

The microprocessor itself!

- It is just like any other reactive object!
  - it is implicitly *instantiated* when power is turned on
  - its state is all global variables, of which many will be reactive objects in their own right
  - its methods are the installed interrupt handlers
  - its *self* is only conceptual (there is no concrete pointer . . .)
- The top-level object methods are scheduled by the CPU hardware, not by the TinyTimber kernel!

The top-level object

The microprocessor itself!

- It is just like any other reactive object!
  - it is implicitly *instantiated* when power is turned on
  - its state is all global variables, of which many will be reactive objects in their own right
  - its methods are the installed interrupt handlers
  - its *self* is only conceptual (there is no concrete pointer . . .)
- The top-level object methods are scheduled by the CPU hardware, not by the TinyTimber kernel!

The top-level object

The microprocessor itself!

- It is just like any other reactive object!
  - it is implicitly *instantiated* when power is turned on
  - its state is all global variables, of which many will be reactive objects in their own right
  - its methods are the installed interrupt handlers
  - its *self* is only conceptual (there is no concrete pointer . . .)
- The top-level object methods are scheduled by the CPU hardware, not by the TinyTimber kernel!

The top-level object

The microprocessor itself!

- It is just like any other reactive object!
  - it is implicitly *instantiated* when power is turned on
  - its state is all global variables, of which many will be reactive objects in their own right
  - its methods are the installed interrupt handlers
  - its *self* is only conceptual (there is no concrete pointer . . .)
- The top-level object methods are scheduled by the CPU hardware, not by the TinyTimber kernel!

The top-level object

The microprocessor itself!

- It is just like any other reactive object!
  - it is implicitly *instantiated* when power is turned on
  - its state is all global variables, of which many will be reactive objects in their own right
  - its methods are the installed interrupt handlers
  - its *self* is only conceptual (there is no concrete pointer ...)
- The top-level object methods are scheduled by the CPU hardware, not by the TinyTimber kernel!

The top-level object

The microprocessor itself!

- It is just like any other reactive object!
  - it is implicitly *instantiated* when power is turned on
  - its state is all global variables, of which many will be reactive objects in their own right
  - its methods are the installed interrupt handlers
  - its *self* is only conceptual (there is no concrete pointer . . .)
- The top-level object methods are scheduled by the CPU hardware, not by the TinyTimber kernel!

The top-level object

The microprocessor itself!

- It is just like any other reactive object!
  - it is implicitly *instantiated* when power is turned on
  - its state is all global variables, of which many will be reactive objects in their own right
  - its methods are the installed interrupt handlers
  - its *self* is only conceptual (there is no concrete pointer . . .)
- The top-level object methods are scheduled by the CPU hardware, not by the TinyTimber kernel!

Connecting interrupts

Incoming method calls from the hardware environment correspond to interrupt signals received by the microprocessor. Apart from this special link to the outside world, interrupt handlers are ordinary methods accepting the same type of parameters as methods invoked with `SYNC` and `ASYNC`.

To install method `meth` on object `obj` as an interrupt handler for interrupt source `IRQ_X`, one writes

```
INSTALL(&obj, meth, IRQ_X);
```

Connecting interrupts

To install method `meth` on object `obj` as an interrupt handler for interrupt source `IRQ_X`, one writes

```
INSTALL(&obj, meth, IRQ_X);
```

This call, which preferably should be performed during system startup, causes `meth` to be subsequently invoked with `&obj` and `IRQ_X` as arguments whenever the interrupt identified by `IRQ_X` occurs.

The symbol `IRQ_X` is here used as a placeholder only; the exact set of available interrupt sources is captured in a platform-dependent enumeration type `Vector` defined in the TinyTimber interface.

Example

A Counter example (counter.h)

```
#include "TinyTimber.h"
typedef struct{
  Object super;
  int val;
} Counter;
#define initCounter(n) {initObject(),n}
```

A Counter example (counter.c)

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

Example

A Counter example (counter.h)

```
#include "TinyTimber.h"
typedef struct{
  Object super;
  int val;
} Counter;
#define initCounter(n) {initObject(),n}
```

A Counter example (counter.c)

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

Example

A Counter example (counter.h)

```
#include "TinyTimber.h"
typedef struct{
  Object super;
  int val;
} Counter;
#define initCounter(n) {initObject(),n}
```

A Counter example (counter.c)

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

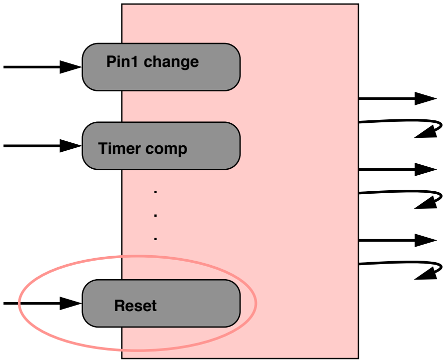
Example client

In main.c

```
Counter counter = initCounter(0);
INSTALL(&counter, inc, IRQ_PCINT1);
```

Reset

When system starts up, a reset signal is generated by the hardware. There will be an interrupt routine like any other one . . .



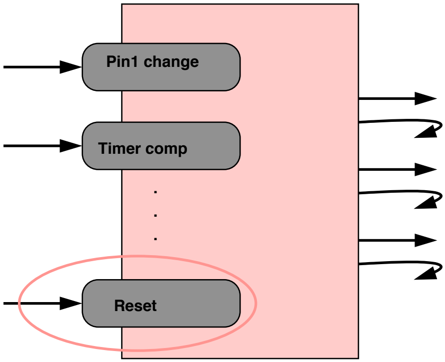
Complication

The reset routine cannot return as it has not really interrupted anything!

In the active system view this is interpreted as compute until someone turns off the power!

Reset

When system starts up, a reset signal is generated by the hardware. There will be an interrupt routine like any other one . . .



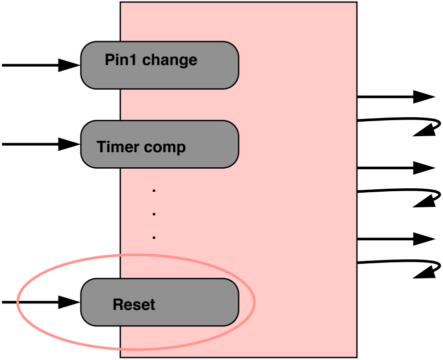
Complication

The reset routine cannot return as it has not really interrupted anything!

In the active system view this is interpreted as compute until someone turns off the power!

Reset

When system starts up, a reset signal is generated by the hardware. There will be an interrupt routine like any other one . . .



Complication

The reset routine cannot return as it has not really interrupted anything!

In the active system view this is interpreted as compute until someone turns off the power!

main()

The `main()` function in C is an abstraction of the reset handler . . .

. . . just as a program is an abstraction of the notion of *running a computer until it stops*

In *traditional programs* `main()` does indeed return, which can be understood as a request to the OS to *turn off the power* to the virtual computer that was set up to run the program!

In a *reactive system* we do not want power to be turned off at all, but we also do not want to let `main()` compute forever just to keep it from returning . . . a reactive system *rests* when it is not reacting

main()

The `main()` function in C is an abstraction of the reset handler . . .

. . . just as a program is an abstraction of the notion of *running a computer until it stops*

In *traditional programs* `main()` does indeed return, which can be understood as a request to the OS to *turn off the power* to the virtual computer that was set up to run the program!

In a *reactive system* we do not want power to be turned off at all, but we also do not want to let `main()` compute forever just to keep it from returning . . . a reactive system *rests* when it is not reacting

main()

The `main()` function in C is an abstraction of the reset handler . . .

. . . just as a program is an abstraction of the notion of *running a computer until it stops*

In *traditional programs* `main()` does indeed return, which can be understood as a request to the OS to *turn off the power* to the virtual computer that was set up to run the program!

In a *reactive system* we do not want power to be turned off at all, but we also do not want to let `main()` compute forever just to keep it from returning . . . a reactive system *rests* when it is not reacting

main()

The `main()` function in C is an abstraction of the reset handler . . .

. . . just as a program is an abstraction of the notion of *running a computer until it stops*

In *traditional programs* `main()` does indeed return, which can be understood as a request to the OS to *turn off the power* to the virtual computer that was set up to run the program!

In a *reactive system* we do not want power to be turned off at all, but we also do not want to let `main()` compute forever just to keep it from returning . . . a reactive system *rests* when it is not reacting

The idle task

Solution

Let `main()` finish by literally *putting the CPU to sleep* until the next interrupt! (Most architectures have a special machine instruction that does so!)

We want `main()` to finish by calling this instruction:

```
void idle(){
  ENABLE();
  while(1)SLEEP();
}
```

The idle task

Solution

Let `main()` finish by literally *putting the CPU to sleep* until the next interrupt! (Most architectures have a special machine instruction that does so!)

We want `main()` to finish by calling this instruction:

```
void idle(){
  ENABLE();
  while(1)SLEEP();
}
```

The idle task

Solution

Let `main()` finish by literally *putting the CPU to sleep* until the next interrupt! (Most architectures have a special machine instruction that does so!)

We want `main()` to finish by calling this instruction:

```
void idle(){
  ENABLE();
  while(1)SLEEP();
}
```

main in a tinytimber program

This is achieved by invoking the non-terminating primitive `TINYTIMBER` as the last `main` statement:

```
int main() {
   INSTALL(&obj1, meth1, IRQ_1);
   INSTALL(&obj2, meth2, IRQ_2);
   return TINYTIMBER(&obj3, meth3, val);
}
```

The scheduler

In TinyTimber:

```
int tinytimber(Object *obj, Method m, int arg) {
    DISABLE();
    initialize();
    ENABLE();
    if (m != NULL)
        m(obj, arg);
    DISABLE();
    idle();
    return 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 the absence of cyclic `SYNC`)
- free from critical section race conditions

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 the absence of cyclic `SYNC`)
- free from critical section race conditions

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 the absence of cyclic `SYNC`)
- free from critical section race conditions

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 the absence of cyclic `SYNC`)
- free from critical section race conditions

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 the absence of cyclic `SYNC`)
- free from critical section race conditions