CHALMERS
UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG

Real-Time Systems

Exercise #2

Victor Wallsten

Department of Computer Science and Engineering
Chalmers University of Technology

Call-back in TinyTimber

Device-driver call-back functionality in TinyTimber:

- The device drivers in TinyTimber contain interrupt handler code for any hardware event that may be associated with the device.
- For example, when pressing a key on the console keyboard the interrupt handler of the SCI device driver is activated.

  The SCI interrupt handler will then call the method provided by the user (via the call-back information) when creating the corresponding SCI device object.

  The typed keyboard character will be provided as the second parameter in the call-back method call.

Example: SCI integer parser

Problem: Input decimal integer values from the keyboard and print the values to the console using the TinyTimber kernel.

- The call-back and print-out functionality of the SCI device driver should be used.
- The typed individual characters of the integer value should be stored within a suitable object.
- The input of an integer value should be ended by typing the delimiter character ‘e’.
- The input integer value character string should be converted to C integer data type before being printed out.

Call-back in TinyTimber

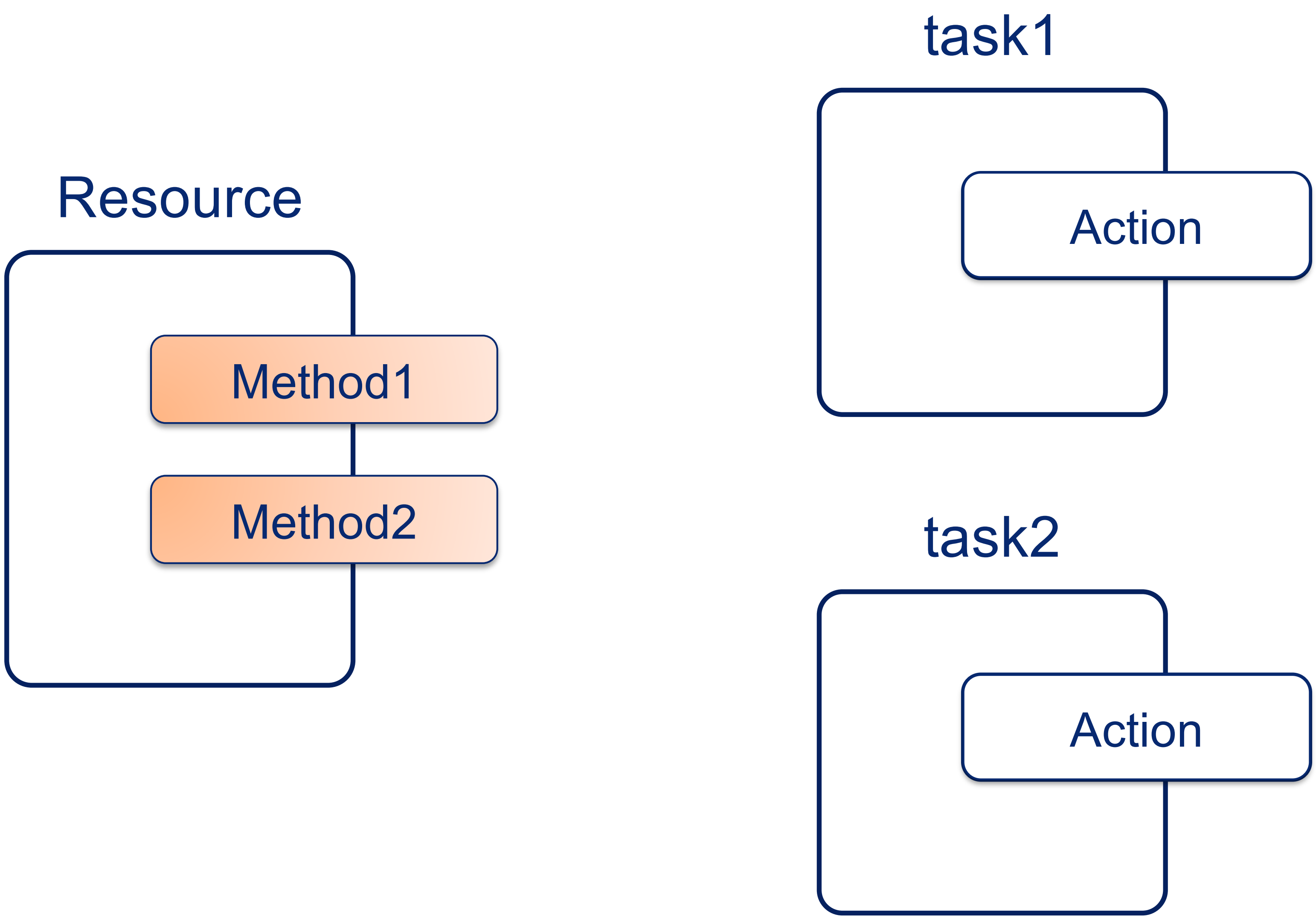
Object call-back functionality in TinyTimber:

- TinyTimber guarantees that an object is handled like an exclusive resource during the execution of a method that belongs to the object <u>if</u> that method is called using `SYNC()`.

  If multiple concurrent calls, using `SYNC()`, are made to methods belonging to the same object, only one call will be granted access to the object. The other calls will be blocked (put in a waiting queue.)

  When the object is available again, one of the blocked calls will be unblocked and the corresponding method is executed by means of a basic call-back functionality in TinyTimber.

Call-back in TinyTimber



Call-back in TinyTimber

Object call-back functionality in TinyTimber (cont'd):

- Although this basic call-back functionality is sufficient in many cases, TinyTimber lacks one powerful property that protected objects, monitors and semaphores have:

  The basic call-back functionality in TinyTimber cannot account for conditions relating to the contents of an object.

  Note: this prevents us from implementing blocking versions of the `Get/Put` methods in the circular buffer example used in Lecture #5.

- Thus, in order to use advanced resource management with TinyTimber we must provide a call-back functionality add-on.

Call-back in TinyTimber

Call-back functionality add-on:

- A task requests access to a certain resource (object) with a call to a method belonging to that resource (object).
- If access is not granted (because a condition regarding the object state prevents this) the method call will be blocked.
- If the calling task used `ASYNC()` to request the resource the task itself is not blocked but continues executing code.
- Implementing call-back functionality means that a calling task supplies `ASYNC()` with information about a method to wake up (call back) when the resource becomes available.
- Since multiple tasks may want to request the resource, the provided call-back information must be stored in a queue.

Call-back in TinyTimber

Method parameter and return-value convention:

- TinyTimber uses a uniform approach to method definitions: all methods must have two parameters of specific types
  - The first parameter <u>must</u> be a pointer to an object of the class to which the method belongs. This pointer (often named '`self`') allows the methods to access the state variables of the object.
  - The second parameter <u>must</u> be of type '`int`' and can be used as an input parameter to the method (but can also be ignored).
- For this reason calls to method operations in the kernel (`TINYTIMBER()`, `ASYNC()`, `SYNC()`, `AFTER()`, ...) <u>must</u> include these parameters in addition to a method reference.
- The return value of a method must be of type '`int`', unless no value is returned (in which case type '`void`' is used).

Call-back in TinyTimber

Method parameter and return-value work-around:

- If an input parameter of type '`xxx`' (different than '`int`') is needed for the method, type casting the argument to type '`int`' must be performed at call time; then the parameter is type-cast back to type '`xxx`' within the method itself.
- If multiple input parameters are needed, they should be stored in a struct, and a pointer to the struct should be passed as the argument at call time (with appropriate type casting).
- This work-around is also applicable to return values.

Example: semaphores in C

Problem: Implement a class `Semaphore` in C using the TinyTimber kernel.

- The object should receive an initial value when it is created.
- The object should have two methods, `Wait` and `Signal`, that work in accordance with the definition of semaphores.
- The methods should have support for call-back functionality

Example: semaphores in C

Solution overview:

1. Show example application code that uses a semaphore
2. Define a class Semaphore with Wait and Signal methods, as well as an initialization macro
3. Implement the Wait and Signal methods
4. Define a data type for call-back information, that can also be stored as an element in a queue (Appendix)
5. Implement functions for manipulating a queue containing elements of the call-back information data type (Appendix)

Example: semaphores in C

A semaphore `s` is an integer variable with value domain ≥ 0

Atomic operations on semaphores:

```
Init(s,n):   assign s an initial value n

Wait(s):     if s > 0 then
                s := s - 1;
             else
                ”block calling task”;

Signal(s):   if ”any task that has called Wait(s) is blocked”
             then
                ”allow one such task to execute”;
             else
                s := s + 1;
```

Example: semaphores in C

Solution overview:

1. Show example application code that uses a semaphore
2. Define a class Semaphore with Wait and Signal methods, as well as an initialization macro
3. Implement the Wait and Signal methods
4. Define a data type for call-back information, that can also be stored as an element in a queue (Appendix)
5. Implement functions for manipulating a queue containing elements of the call-back information data type (Appendix)

Example: semaphores in C

```
// Template code for one task using semaphore Sem

void Non_Critical(Task *self, int unused) {
    Do_Non_Critical_Work();

    self->cb.obj = self;            // store call-back info for this task
    self->cb.meth = Critical;
    ASYNC(&Sem, Wait, (int) &self->cb ); // acquire semaphore
}                                        // (note ‘int’ type-cast)

void Critical(Task *self, int unused) {
    Do_Critical_Work();

    SYNC(&Sem, Signal, 0);          // release semaphore
    ASYNC(self, Non_Critical, 0);   // restart “loop”
}
```

Example: semaphores in C

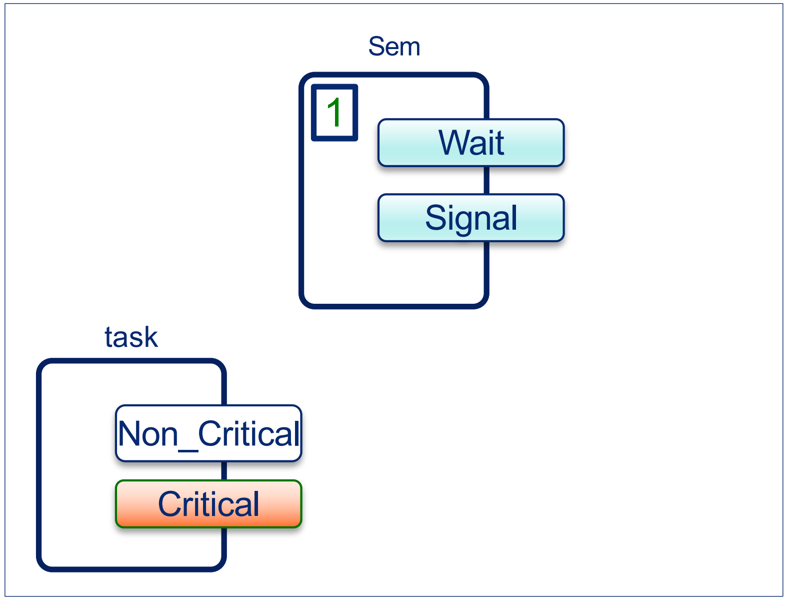
```
// Template code for one task using semaphore Sem

void Non_Critical(Task *self, int unused) {
    Do_Non_Critical_Work();

    self->cb.obj = self;
    self->cb.meth = Critical;
    ASYNC(&Sem, Wait, (int) &self->cb );
}

void Critical(Task *self, int unused) {
    Do_Critical_Work();

    SYNC(&Sem, Signal, 0);
    ASYNC(self, Non_Critical, 0);
}
```



Example: semaphores in C

```
// Program code for two identical tasks using the same semaphore

typedef struct {
    Object super;
} Application;

Application app = { initObject() };

// Define Task class with two methods, Non_Critical() and Critical()

typedef struct {
    Object super;
    CallBlock cb;     // stored call-back info (see Appendix)
} Task;

void Non_Critical(Task*, int);
void Critical(Task*, int);

#define initTask() { initObject(), initCallBlock() }
```

Example: semaphores in C

```
// Create the semaphore and the two identical tasks

Semaphore Sem = initSemaphore(1);   // binary semaphore

Task task1 = initTask();
Task task2 = initTask();

void Non_Critical(Task *self, int unused) {
    Do_Non_Critical_Work();

    self->cb.obj = self;
    self->cb.meth = Critical;
    ASYNC(&Sem, Wait, (int) &self->cb ); // acquire semaphore
}

void Critical(Task *self, int unused) {
    Do_Critical_Work();

    SYNC(&Sem, Signal, 0);          // release semaphore
    ASYNC(self, Non_Critical, 0);
}
```

Example: semaphores in C

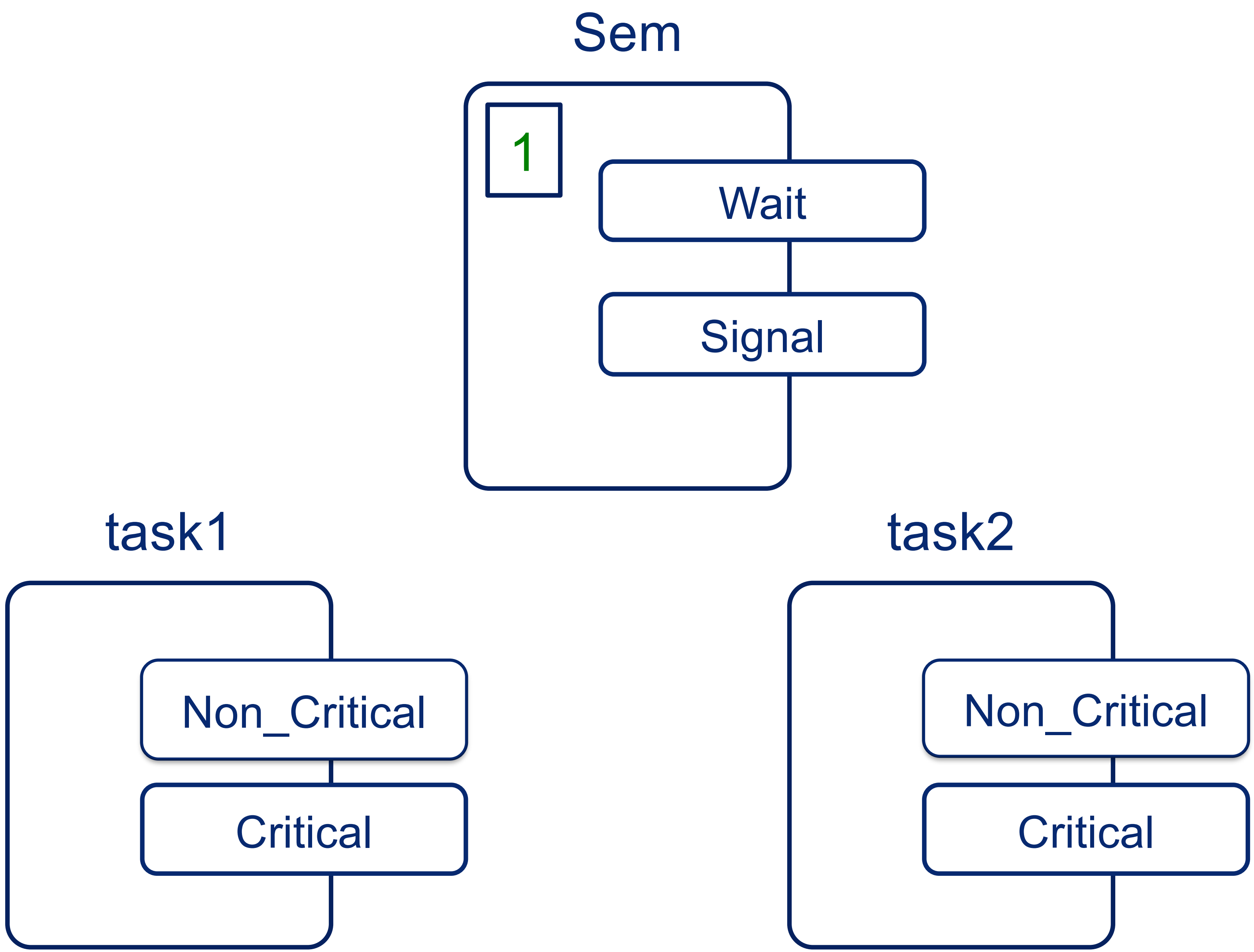
```
// TinyTimber first scheduled event

void kickoff(Application *self, int unused) {
    ASYNC(&task1, Non_Critical, 0);
    ASYNC(&task2, Non_Critical, 0);    // spawn two identical tasks
}

// Start the TinyTimber run-time system

// We enter main() after system startup
// and hand over control to TinyTimber

int main() {
    TINYTIMBER(&app, kickoff, 0);
}
```


Sem
1
Wait
Signal
task1
Non_Critical
Critical
task2
Non_Critical
Critical

Example: semaphores in C

Solution overview:

1. Show example application code that uses a semaphore
2. Define a class Semaphore with Wait and Signal methods, as well as an initialization macro
3. Implement the Wait and Signal methods
4. Define a data type for call-back information, that can also be stored as an element in a queue (Appendix)
5. Implement functions for manipulating a queue containing elements of the call-back information data type (Appendix)

Example: semaphores in C

```
// Define a class Semaphore with Wait and Signal methods,
// as well as an initialization macro

typedef struct {
    Object    super;
    int       value;
    Caller    queue; // queue of stored call-back info (see Appendix)
} Semaphore;

// Note that TinyTimber methods only accept type 'int' for the second
// parameter. This means that, if we want to send a parameter of another
// scalar type (i.e. a pointer), we will have to trick the system by
// "type casting" to 'int' before a call, and then back to the original
// type within the method.

void Wait(Semaphore*, int);
void Signal(Semaphore*, int);

#define initSemaphore(n) { initObject(), n, 0 }
```

Example: semaphores in C

Solution overview:

1. Show example application code that uses a semaphore
2. Define a class Semaphore with Wait and Signal methods, as well as an initialization macro
3. Implement the Wait and Signal methods
4. Define a data type for call-back information, that can also be stored as an element in a queue (Appendix)
5. Implement functions for manipulating a queue containing elements of the call-back information data type (Appendix)

Example: semaphores in C

A semaphore `s` is an integer variable with value domain ≥ 0

Atomic operations on semaphores:

`Init(s,n)`: assign `s` an initial value `n`

```
Wait(s):     if s > 0 then
                s := s - 1;
             else
                "block calling task";

Signal(s):   if "any task that has called Wait(s) is blocked"
             then
                "allow one such task to execute";
             else
                s := s + 1;
```

Example: semaphores in C

```
// Implement the methods Wait and Signal

void Wait(Semaphore *self, int c) {
    Caller wakeup = (Caller) c;  // type-cast back from 'int'
    if (self->value > 0) {
        self->value--;
        ASYNC(wakeup->obj, wakeup->meth, 0);
    }
    else
        c_enqueue(wakeup, &self->queue); // (see Appendix)
}

void Signal(Semaphore *self, int unused) {
    if (self->queue) {
        Caller wakeup = c_dequeue(&self->queue); // (see Appendix)
        ASYNC(wakeup->obj, wakeup->meth, 0);
    }
    else
        self->value++;
}
```

Appendix: the call-back queue

Solution overview:

1. Show example application code that uses a semaphore
2. Define a class Semaphore with Wait and Signal methods, as well as an initialization macro
3. Implement the Wait and Signal methods
4. Define a data type for call-back information, that can also be stored as an element in a queue (Appendix)
5. Implement functions for manipulating a queue containing elements of the call-back information data type (Appendix)

Appendix: the call-back queue

```
// Define a data type for call-back information, that can also
// be used as an element in a queue

struct call_block;
typedef struct call_block *Caller;

typedef struct call_block {
    Caller     next;   // for use in linked lists
    Object     *obj;
    Method     meth;
} CallBlock;

#define initCallBlock() { 0, 0, 0 }
```

Appendix: the call-back queue

```
// Implement functions for manipulating a queue containing
// elements of the call-back information data type

void c_enqueue(Caller c, Caller *queue) {
    Caller prev = NULL, q = *queue;
    while (q) { // find last element in queue
        prev = q;
        q = q->next;
    }
    if (prev == NULL)
        *queue = c;   // empty queue: put 'c' first
    else
        prev->next = c; // non-empty queue: put 'c' last
    c->next = NULL;
}

Caller c_dequeue(Caller *queue) {
    Caller c = *queue;
    if (c)
        *queue = c->next; // remove first element in queue
    return c;
}
```