CHALMERS

Real-Time Systems

Exercise #2

Professor Jan Jonsson

Department of Computer Science and Engineering
Chalmers University of Technology

CHALMERS

Call-back in TinyTimber

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 if 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.

CHALMERS

Call-back in TinyTimber

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 in an earlier lecture.

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

CHALMERS

Call-back in TinyTimber

Call-back functionality add-on:

- A task request access to a certain resource (object) using 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.

CHALMERS

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 must 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 must 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()`, ...) must 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).

CHALMERS

Call-back in TinyTimber

Method parameter and return-value work-arounds:

- 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).
- These work-arounds above also apply to return values.

CHALMERS

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

CHALMERS

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;
```

CHALMERS

Example: three-step solution

1. Define a data type for storage of call-back information
2. Define a data type for a queue of call-back information
3. Define a class Semaphore with associated Wait and Signal methods

CHALMERS

Example: caller queue in C

```
/*
 * Although TinyTimber offers mutual exclusion for methods, it does
 * not give any support for call-back functionality. Therefore, we
 * need to add that functionality.
 */


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 }

...
```

CHALMERS

Example: caller queue in C

```
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;
}
```

CHALMERS

Example: semaphores in C

```
// Define a new class Semaphore and declare its methods

typedef struct {
    Object    super;
    int       value;
    Caller    queue;
} 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);

...
```

CHALMERS

Example: semaphores in C

```
// Define 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);
}

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

CHALMERS

Example: semaphores in C

```
// Define two identical tasks using the same semaphore

Semaphore Sem = { initObject(), 1, 0 };    // init value = 1, empty queue

typedef struct {
    Object super;
    CallBlock cb;    // where call-back information is stored
} Task;

Task task1 = { initObject(), initCallBlock() };
Task task2 = { initObject(), initCallBlock() };

...
```

CHALMERS

Example: semaphores in C

```
...

void Critical(Task*, int);

void Non_Critical(Task *self, int unused) {
    self->cb.obj = self;           // provide call-back information
    self->cb.meth = Critical;
    ASYNC(&Sem, Wait, (int) &self->cb );   // acquire semaphore
}                          // type-cast pointer argument to 'int'

void Critical(Task *self, int unused) {
    ...                            // the critical region
    SYNC(&Sem, Signal, 0);         // release semaphore
    ASYNC(self, Non_Critical, 0);  // restart "loop"
}

...
```

CHALMERS

Example: semaphores in C

```
...

void kickoff(Task *s, int u) {      // TinyTimber's first scheduled event
  ASYNC(&task1, Non_Critical, 0);
  ASYNC(&task2, Non_Critical, 0);   // spawn two identical tasks
}

int main() {                        // we enter here after system startup,
    TINYTIMBER(&task1, kickoff, 0); // and hand over control to TinyTimber
    return 0;         // note: any object could be "host" object for
}                     // the 'kickoff()' method. We chose 'task1'.
```