Real-Time Systems

Exercise #3

Course Assistant Elena Marzi

Department of Computer Science and Engineering
Chalmers University of Technology

Periodic tasks in C

Today:

- event-triggered vs time-triggered systems
- periodicity in TinyTimber: `AFTER()`, `BEFORE()`, `SEND()`

Exercise:

Implement two periodic tasks with a shared object in C using the TinyTimber kernel.

The two tasks will be implemented:

1. Without deadline
2. With deadline
3. With a limited lifetime

Time-vs. Event-triggered system

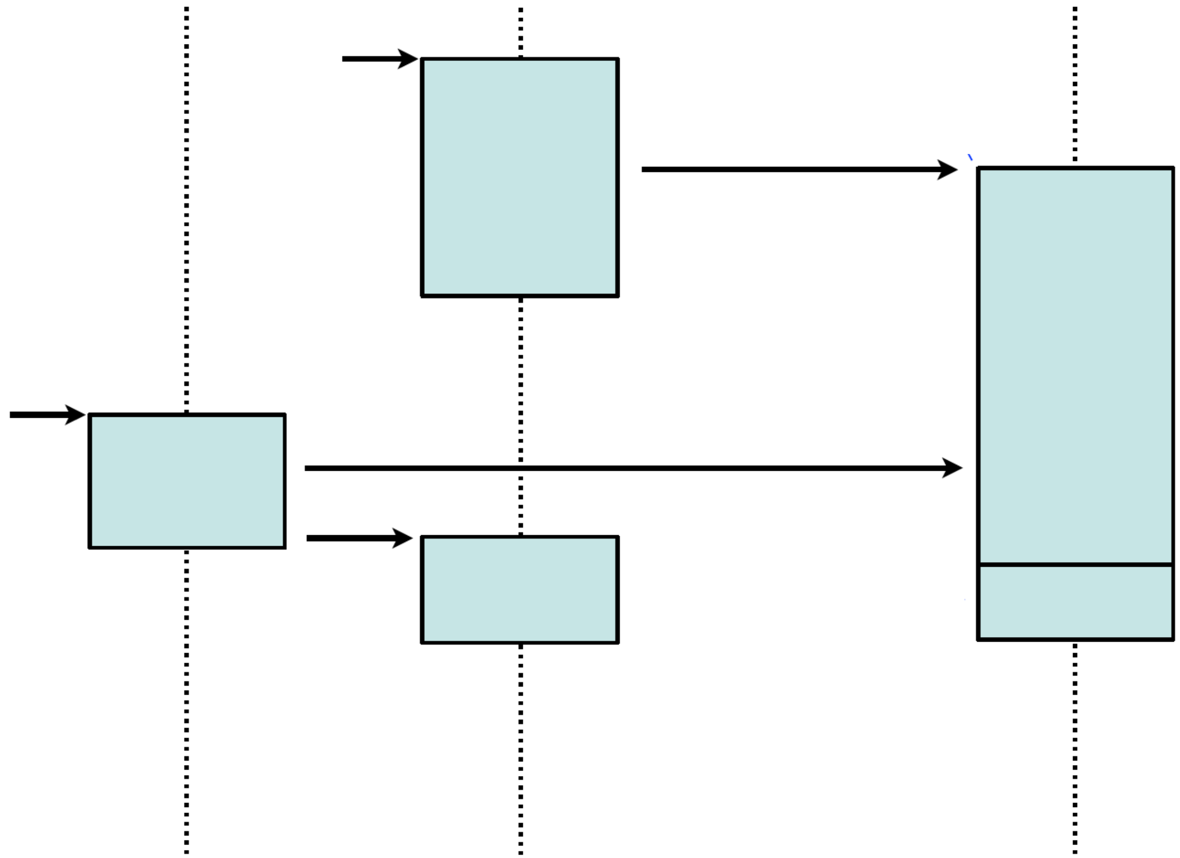
Time-triggered

- Tasks are released at predetermined points in time
- Deterministic

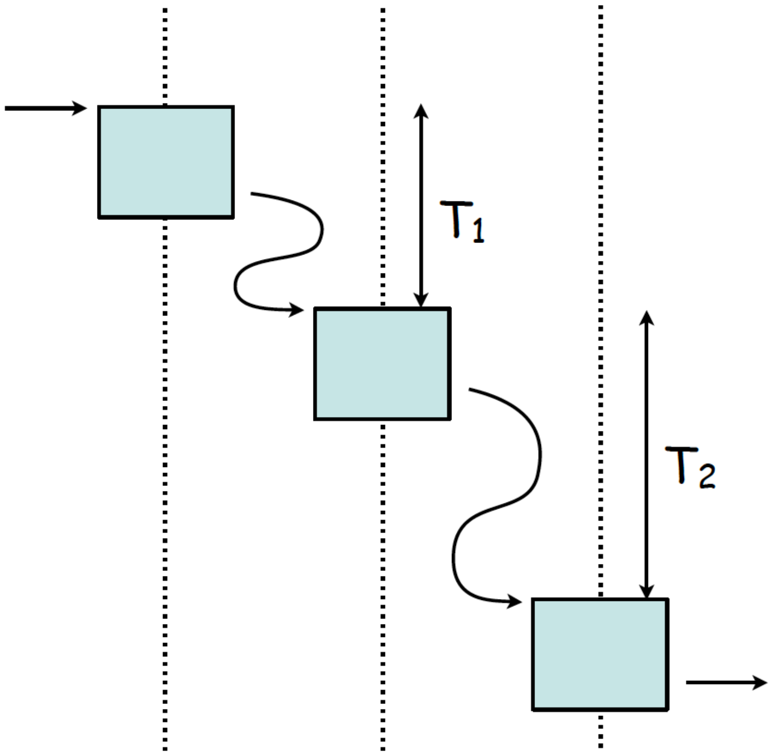
Event-triggered

- System reacts to events in the environment
- Flexible

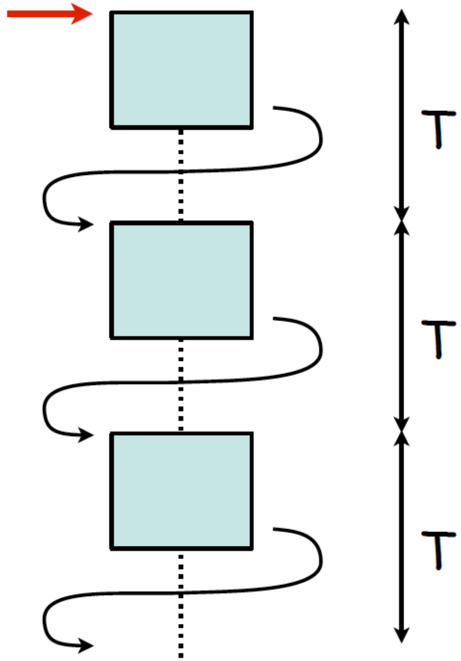
TinyTimber: Event-triggered runtime



TinyTimber: Event-triggered runtime

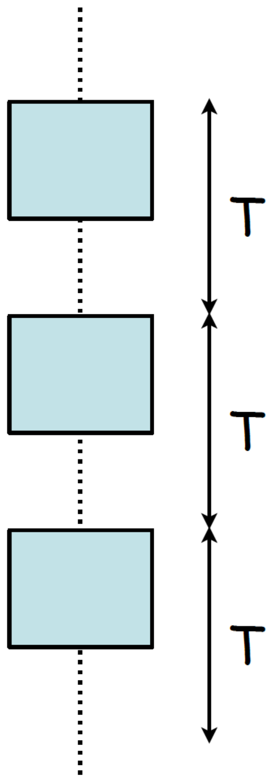


An event-triggered system with offsets

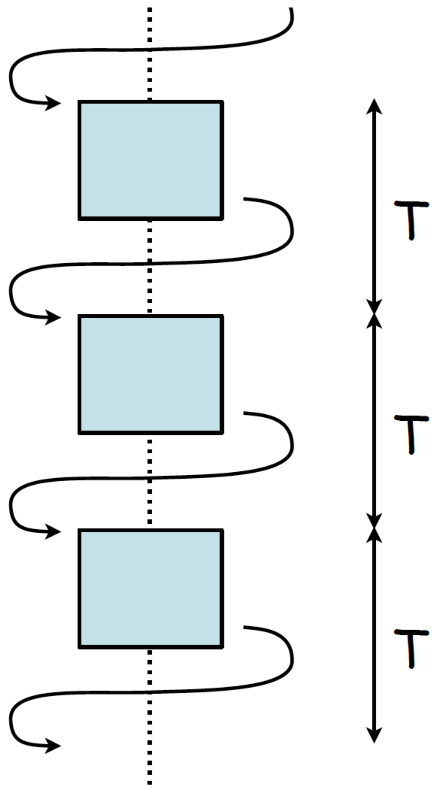


A self-referencing event-triggered system with offsets

TinyTimber: Event-triggered runtime

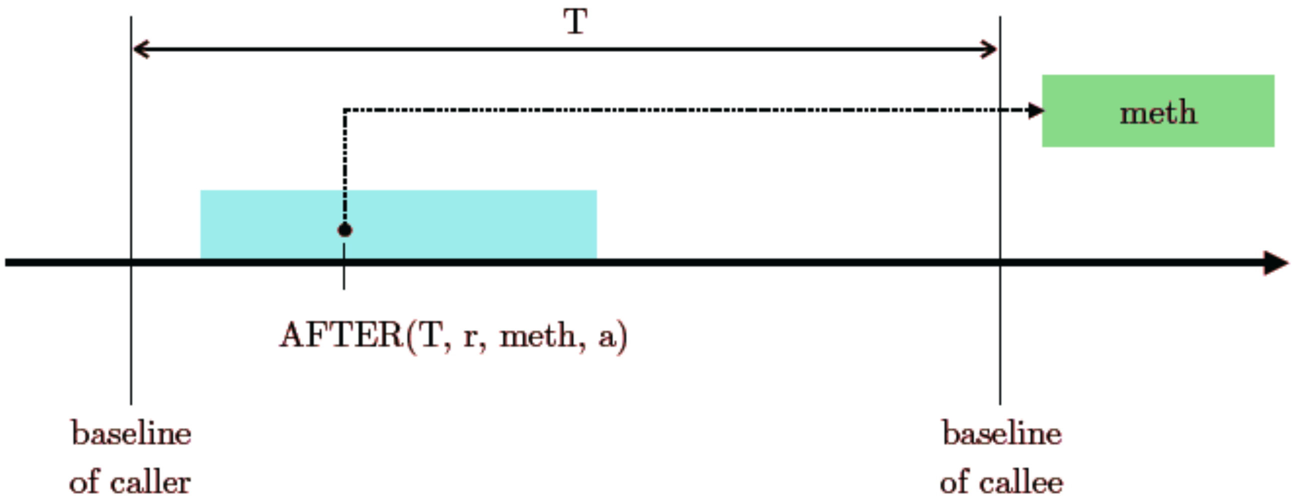


A time-triggered system



A self-referencing event-triggered system with offsets

Some more about `AFTER()`



An `AFTER()` call with a baseline of 0 means that the called method runs with the same baseline as the caller.

```
ASYNC(&obj,meth,12) == AFTER(0,&obj,meth,12);
```

Some more about `AFTER()`

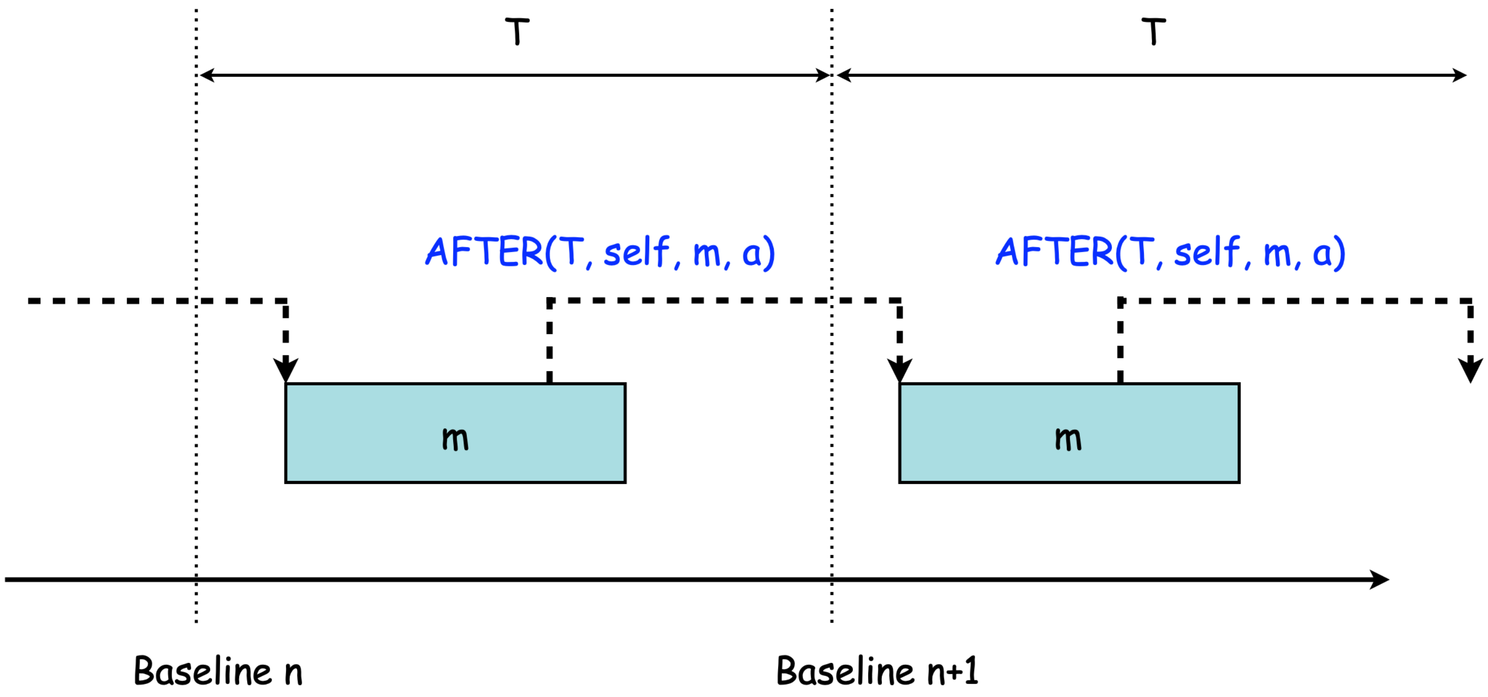
Using the baseline to derive time offsets makes the actual time the `AFTER()` call is made less critical!

```
int work1(MyObject *self, int arg) {
   ... // do some work
   AFTER(SEC(T),&obj,do_more_work,0);
}
```

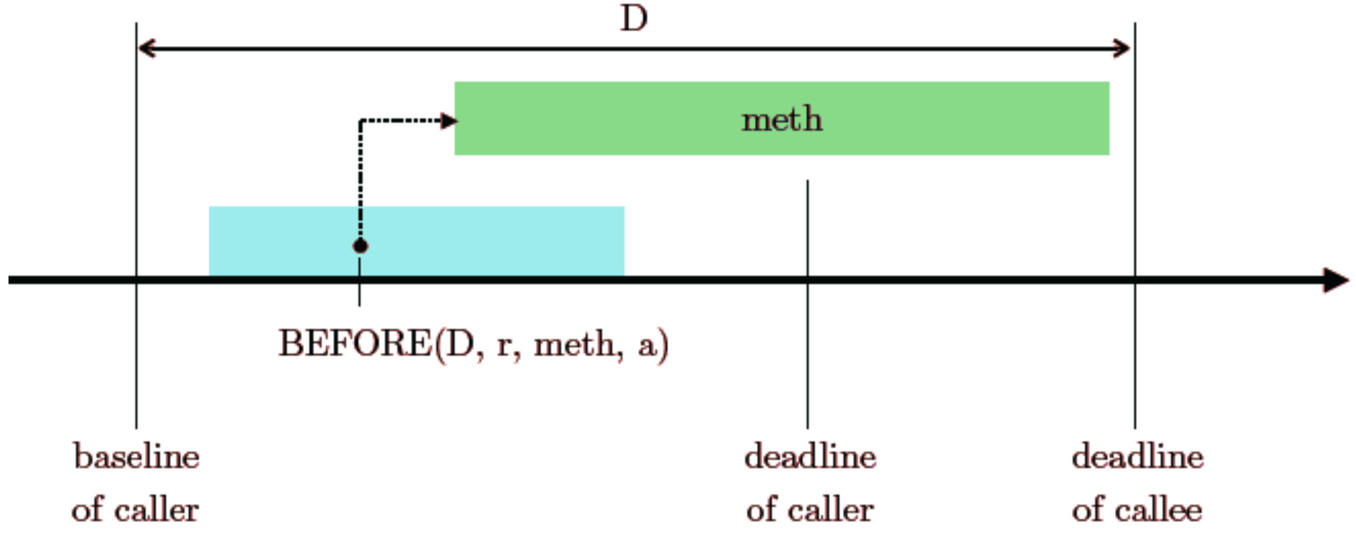
has the same behavior as

```
int work1(MyObject *self, int arg) {
   AFTER(SEC(T),&obj,do_more_work,0);
   ... // do some work
}
```

Periodicity with AFTER



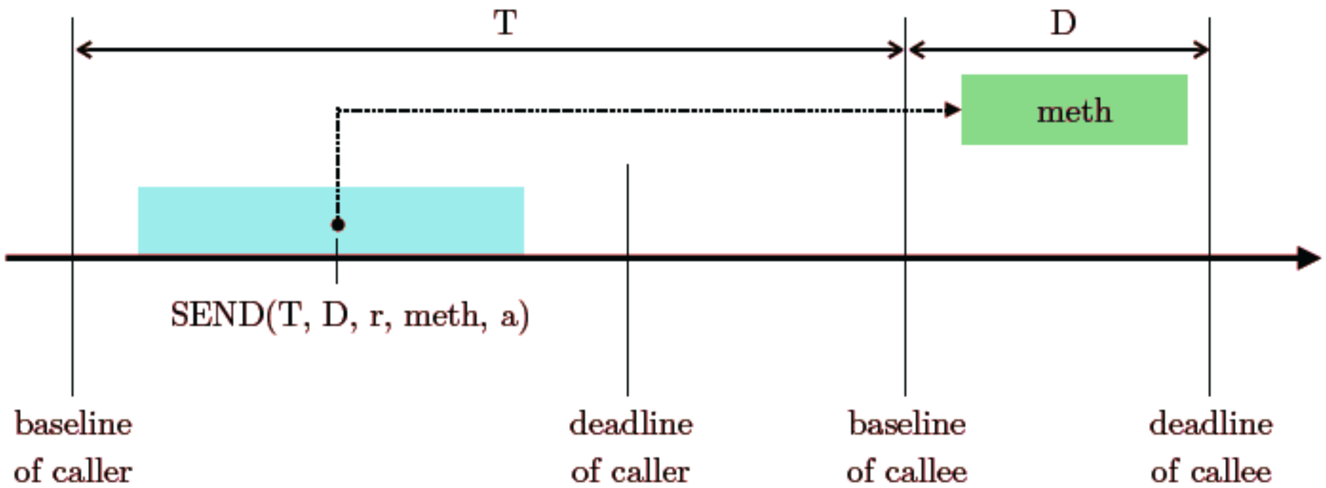
Some more about **`BEFORE()`**



The `BEFORE()` call has an implicit baseline of 0, i.e., the called method runs with the same baseline as the caller.

To assign a deadline to a delayed method call, you need to use the `SEND()` call.

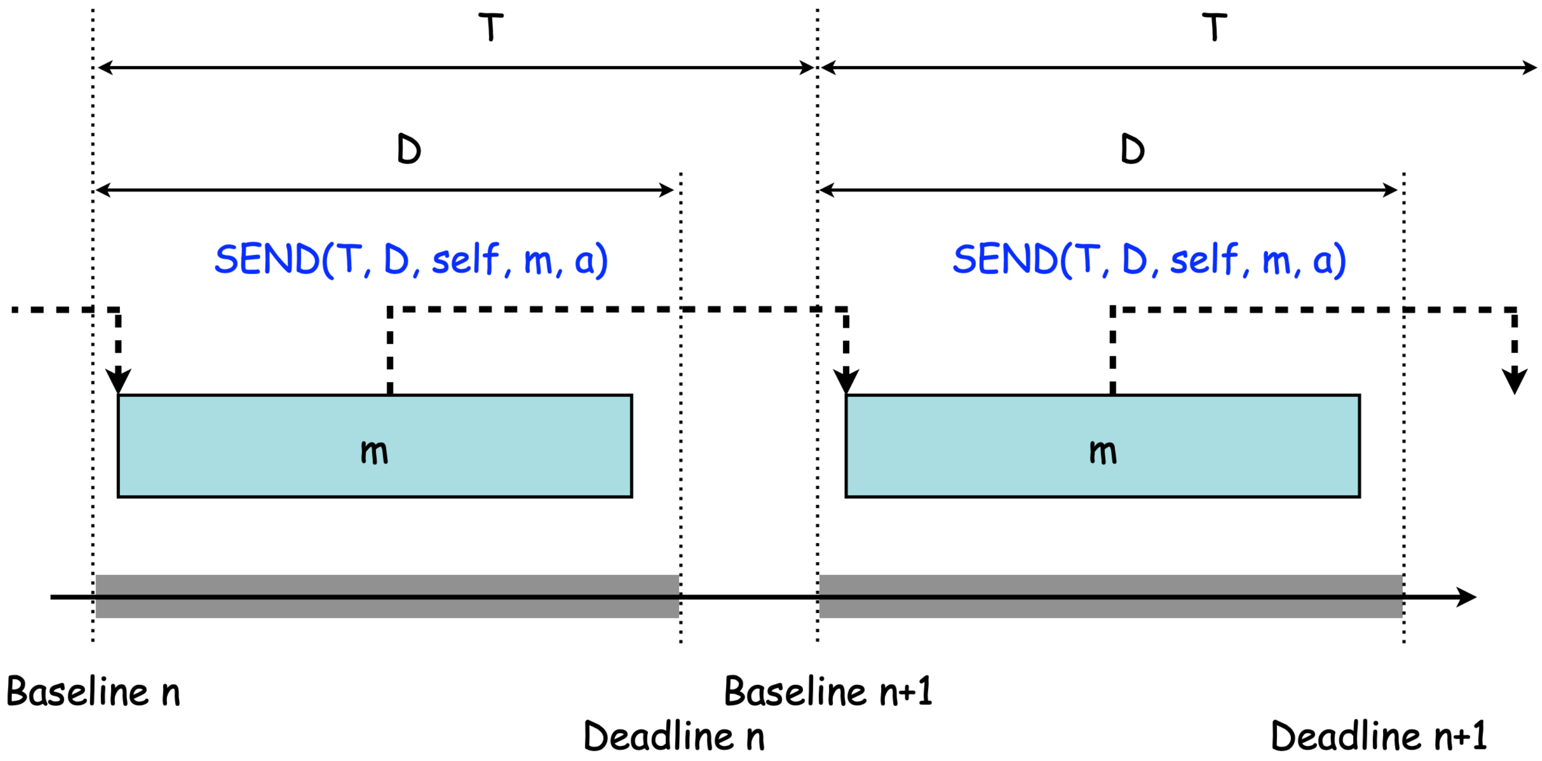
Some more about `SEND()`



The `SEND()` call is the fundamental building block for the `AFTER`, `BEFORE` and `ASYNC` calls.

```
AFTER(T,&obj,meth,12)  == SEND(T,0,&obj,meth,12);
BEFORE(D,&obj,meth,12) == SEND(0,D,&obj,meth,12);
ASYNC(&obj,meth,12)    == SEND(0,0,&obj,meth,12);
```

Periodicity with SEND



Example: periodic tasks in C

Problem: Implement two periodic tasks with a shared object in C using the TinyTimber kernel.

- Assume that an object `actobj` of type `Actuator` is shared by two periodic tasks `task1` and `task2` with periods 300 μs and 500 μs, respectively.
- Both tasks may concurrently call a method `update` of object `actobj` with an initial value 10 and 20, respectively.
- The old value of object `actobj` should be returned by the `update` method, to be used by the tasks.

Example: periodic tasks in C

```
typedef struct{
   Object super;
   int state;
} Actuator;

int update(Actuator *self , int new_value){
   int old_value = self->state;
   self->state = new_value;
   ...        // code updating the actuator hardware
   return old_value;
}

Actuator actobj = { initObject(), 0 }; //an object of Actuator
   class

...
```

Example: periodic tasks in C

```
typedef struct{
   Object super;
   Time period;


} TaskObject;

TaskObject task1 = { initObject(), USEC(300)     };
TaskObject task2 = { initObject(), USEC(500)     };

void task1code(TaskObject *self, int n);
void task2code(TaskObject *self, int n);


// Q: Why do we need one object for each task?
// A: To make sure the tasks can execute concurrently.
```

Example: periodic tasks in C

```
// Task1 and Task2 methods

void task1code(TaskObject *self, int value){

   int old_state = SYNC(&actobj, update, value);
   ...       // do something with returned value
   AFTER(self->period, self, task1code, value);

}

void task2code(TaskObject *self, int value){

   int old_state = SYNC(&actobj, update, value);
   ...       // do something else with returned value
   AFTER(self->period, self, task2code, value);

}
```

Example: periodic tasks in C

```
// How to begin the initial invocation?

void kickoff(TaskObject *self , int unused) {
   ASYNC(&task1, task1code, 10);
   ASYNC(&task2, task2code, 20);
}

int main() {
   TINYTIMBER(&task1, kickoff, 0);
   return 0;
}
```

Example: periodic tasks in C

Problem: Implement two periodic tasks with a shared object in C using the TinyTimber kernel.

- Assume that an object `actobj` of type `Actuator` is shared by two periodic tasks `task1` and `task2` with periods 300 μs and 500 μs, respectively.
- Both tasks may concurrently call a method `update` of object `actobj` with an initial value 10 and 20, respectively.
- The old value of object `actobj` should be returned by the `update` method, to be used by the tasks.
- Add deadlines of 100 μs and 150 μs to `task1` and `task2`, respectively.

Example: periodic tasks in C

```
typedef struct{
   Object super;
   Time period;
   Time deadline;

} TaskObject;

TaskObject task1 = { initObject(), USEC(300), USEC(100)    };
TaskObject task2 = { initObject(), USEC(500), USEC(150)    };
```

Example: periodic tasks in C

```
// Task1 and Task2 methods

void task1code(TaskObject *self, int value){
   int old_state = SYNC(&actobj, update, value);
   ...       // do something with returned value
   SEND(self->period, self->deadline, self, task1code, value);
}

void task2code(TaskObject *self, int value){
   int old_state = SYNC(&actobj, update, value);
   ...       // do something else with returned value
   SEND(self->period, self->deadline, self, task2code, value);
}
```

Example: periodic tasks in C

```
// How to begin the initial invocation?

void kickoff(TaskObject *self , int unused) {
   BEFORE(USEC(100), &task1, task1code, 10);
   BEFORE(USEC(150), &task2, task2code, 20);
}

int main() {
   TINYTIMBER(&task1, kickoff, 0);
   return 0;
}
```

Example: periodic tasks in C

Problem: Implement two periodic tasks with a shared object in C using the TinyTimber kernel.

- Assume that an object `actobj` of type `Actuator` is shared by two periodic tasks `task1` and `task2` with periods 300 µs and 500 µs, respectively.
- Both tasks may concurrently call a method `update` of object `actobj` with an initial value 10 and 20, respectively.
- The old value of object `actobj` should be returned by the `update` method, to be used by the tasks.
- Add deadlines of 100 µs and 150 µs to `task1` and `task2`, respectively.
- Stop the execution of `task1` and `task2` after 100 ms and 200 ms, respectively.

Example: periodic tasks in C

```
// How to make conditional invocations?

typedef struct{
   Object super;
   Time period;
   Time deadline;
   int running;
} TaskObject;

TaskObject task1 = { initObject(), USEC(300), USEC(100), 1 };
TaskObject task2 = { initObject(), USEC(500), USEC(150), 1 };
```

Example: periodic tasks in C

```
// Q: How to set the state variable 'running' to 0 after a
   delay?
// A: Define a method to be called after the delay

void stop(TaskObject *self, int unused){
   self->running = 0;
}
```

Example: periodic tasks in C

```
// How to make conditional invocations?

void task1code(TaskObject *self, int value){
   if (self->running) {
       int old_state = SYNC(&actobj, update, value);
       ...          // do something with returned value
       SEND(self->period, self->deadline, self, task1code,
   value);
   }
}

void task2code(TaskObject *self, int value){
   if (self->running) {
       int old_state = SYNC(&actobj, update, value);
       ...          // do something else with returned value
       SEND(self->period, self->deadline, self, task2code,
   value);
   }
}
```

Example: periodic tasks in C

```
// How to begin the initial invocation?

void kickoff(TaskObject *self , int unused) {
   BEFORE(USEC(100), &task1, task1code, 10);
   BEFORE(USEC(150), &task2, task2code, 20);
   AFTER(MSEC(100), &task1, stop, 0);
   AFTER(MSEC(200), &task2, stop, 0);
}

int main() {
   TINYTIMBER(&task1, kickoff, 0);
   return 0;
}

// Q: Why do we need two different objects in the AFTER calls
   to 'stop'?
// A: To make sure that the correct task is terminated by each
   call.
```