CHALMERS | UNIVERSITY OF GOTHENBURG

Real-Time Systems

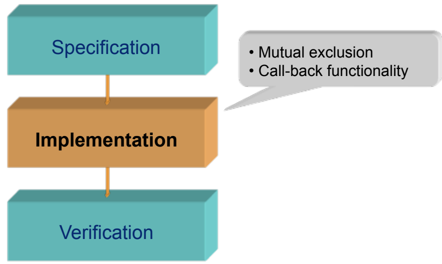
Lecture #5

Professor Jan Jonsson

Department of Computer Science and Engineering
Chalmers University of Technology

CHALMERS | UNIVERSITY OF GOTHENBURG

Real-Time Systems



CHALMERS | UNIVERSITY OF GOTHENBURG

Mutual exclusion

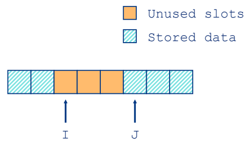
Systems with cooperating concurrent tasks often work with shared data structures.

- A problem that has to be solved is then how to guarantee that the data structure is always kept in a consistent state.
  Data structures such as queues, lists and data bases will not work as intended if their state becomes inconsistent.
- A working solution is achieved if one makes sure that only one task at a time receive access to the data structure.
- Exclusive access to a data structure can be achieved by making sure that the program code (i.e., the critical region) that manipulates the data structure can execute without being preempted in the most critical moment.

CHALMERS | UNIVERSITY OF GOTHENBURG

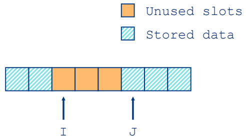
Example: circular buffer in TinyTimber

```
// Define a class Circular_Buffer with space for 8 natural numbers (≥ 0)

#define BSize 8

typedef struct {
    Object     super;
    int        count;
    int        I;
    int        J;
    int        A[BSize];
} Circular_Buffer;

// If the buffer is full, Put should return the value -1.
// If the buffer is empty, Get should return the value -1.

int Put(Circular_Buffer*, int); // Insert new element
int Get(Circular_Buffer*, int); // Remove old element

// Define an instance of the buffer

Circular_Buffer Buf = { initObject(), 0, 0, 0 };  // empty buffer
```



Example: circular buffer in TinyTimber

```
int Put(Circular_Buffer *self, int data) {
    if (self->count < BSize) {
        self->A[self->I] = data;
        self->I = (self->I + 1) % BSize;
        self->count = self->count + 1;
        return 0;
    }
    else
        return -1;
}

int Get(Circular_Buffer *self, int unused) {
    if (self->count > 0) {
        int data = self->A[self->J];
        self->J = (self->J + 1) % BSize;
        self->count = self->count - 1;
        return data;
    }
    else
        return -1;
}
```



Mutual exclusion

In TinyTimber the methods `Put` or `Get` must be called using `SYNC()` in order to guarantee mutual exclusion.

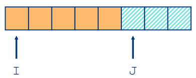
If `Put` or `Get` would be called as regular functions in C, mutual exclusion can not be guaranteed.

In the latter case, the buffer data structure could very easily become corrupt and give rise to data inconsistencies.

The following example demonstrates one such case ...

Mutual exclusion

Assume that the buffer has the following state:



Now, investigate what happens if `Put` is called as a regular function by two concurrent tasks:

```
void T1(App *self, int c) {
    ...
    Put(&Buf,X);
    ...
    ASYNC(self,T1,c);
}
```

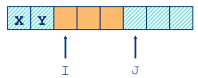
```
void T2(App *self, int c) {
    ...
    Put(&Buf,Y);
    ...
    ASYNC(self,T2,c);
}
```


Mutual exclusion

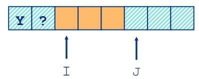
The following execution order causes data inconsistency:

Put(&Buf,X):	Put(&Buf,Y):	Comment:
A(I) = X;		
	A(I) = Y; I = (I + 1) % BSize; count = count + 1;	// X is overwritten
I = (I + 1) % BSize; count = count + 1;		// old value remains // in last data slot

What we want is consistent data:

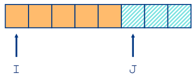


What we get is inconsistent data:



Mutual exclusion

Again, assume that the buffer has the following state:



But this time observe what happens when `Put` is called using `SYNC()` by the two concurrent tasks:

```
void T1(App *self, int c) {
    ...
    SYNC(&Buf,Put,X);
    ...
    ASYNC(self,T1,c);
}
```

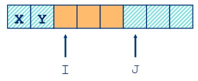
```
void T2(App *self, int c) {
    ...
    SYNC(&Buf,Put,Y);
    ...
    ASYNC(self,T2,c);
}
```


Mutual exclusion

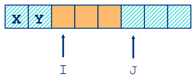
With `SYNC()` we get data consistency:

```
SYNC(&Buf,Put,X):          SYNC(&Buf,Put,Y):          Comment:

A(I) = X;                                             // data store is done
I = (I + 1) % BSize;                                  // correctly
count = count + 1;
                           A(I) = Y;                  // data store is done
                           I = (I + 1) % BSize;       // correctly
                           count = count + 1;
```

What we want is consistent data:



What we get is consistent data:



Machine-level mutual exclusion

To guarantee mutual exclusion in the critical regions of e.g. semaphore operations or mutex methods some even more fundamental support is needed.

For this purpose there are two mechanisms offered at the lowest (machine-code) level:

- Disabling the processor's interrupt service mechanism
  - Should involve any interrupt that may lead to a task switch
  - Only suitable for single-processor systems
- Atomic processor instructions
  For example: the test-and-set instruction
  - Variables can be tested and updated in one operation
  - Necessary for systems with two or more processors

Disabling processor interrupts

In single-processor systems, the mutual exclusion is guaranteed by disabling the processor's interrupt service mechanism ("interrupt masking") while the critical region is executed.

This way, unwanted task switches in the critical region (caused by e.g. timer interrupts) are avoided. However, <u>all other</u> tasks are unable to execute during this time.

Therefore, critical regions should only contain such instructions that really require mutual exclusion (e.g., code that handles the operations `wait` and `signal` for semaphores).

Note: this method is not used in multi-processor systems since interrupt management is typically not synchronized between the processors.

CHALMERS | UNIVERSITY OF GOTHENBURG

Disabling processor interrupts

```
task A;
task B;
task body A is
begin
  Disable_Interrupts;       -- turn off interrupt handling
  ...                       -- critical region
  Enable_Interrupts;        -- A leaves critical region
  ...                       -- remaining program code
end A;
task body B is
begin
  Disable_Interrupts;       -- turn off interrupt handling
  ...                       -- critical region
  Enable_Interrupts;        -- B leaves critical region
  ...                       -- remaining program code
end B;
```

CHALMERS | UNIVERSITY OF GOTHENBURG

Atomic processor instruction

In multi-processor systems with shared memory, a test-and-set instruction is used for handling critical regions.

A test-and-set instruction is a processor instruction that reads from and writes to a variable in one atomic operation.

The functionality of the test-and-set instruction can be illustrated by the following Ada procedure:

```
procedure testandset(lock, previous : in out Boolean) is
 begin
   previous := lock;          -- lock is read and its value saved
   lock := true;              -- lock is set to "true"
 end testandset;
```

The combined read and write of `lock` must be atomic. In a multi-processor system, this is guaranteed by locking (disabling access to) the memory bus during the entire operation.

CHALMERS | UNIVERSITY OF GOTHENBURG

Atomic processor instruction

```
lock : Boolean := false;             -- shared flag
task A, B;

task body A is
  previous : Boolean;
begin
  loop
    testandset(lock, previous);      -- A waits if critical region is busy
   exit when not previous;
  end loop;
  ...                                -- critical region
  lock := false;                     -- A leaves critical region
  ...                                -- remaining program code
end A;

task body B is
  previous : Boolean;
begin
  loop
    testandset(lock, previous);      -- A waits if critical region is busy
   exit when not previous;
  end loop;
  ...                                -- critical region
  lock := false;                     -- A leaves critical region
  ...                                -- remaining program code
end B;
```

CHALMERS | UNIVERSITY OF GOTHENBURG

Atomic processor instruction

```
lock : Boolean := false;             -- shared flag
task A, B;

task body A is
  previous : Boolean;
begin
  loop
    testandset(lock, previous);      -- A waits if critical region is busy
   exit when not previous;
  end loop;
  ...                                -- critical region
  lock := false;                     -- A leaves critical region
  ...                                -- remaining program code
end A;

task body B is
  previous : Boolean;
begin
  loop
    testandset(lock, previous);      -- A waits if critical region is busy
   exit when not previous;
  end loop;
  ...                                -- critical region
  lock := false;                     -- A leaves critical region
  ...                                -- remaining program code
end B;
```

CHALMERS | UNIVERSITY OF GOTHENBURG

Call-back functionality

Operations for resource management:

- `acquire`: to request access to a resource
- `release`: to release a previously acquired resource

The `acquire` operation can be either blocking or non-blocking:

- Blocking: the task that calls `acquire` is blocked if the resource is not available. Blocked tasks are stored in a queue, in FIFO or priority order. When the requested resource becomes available one of the blocked tasks is unblocked and is activated via a *call-back functionality*.
- Non-blocking: `acquire` returns a status code to the calling task indicating whether access to the resource was granted or not.

To support the reactive programming paradigm (that is, no “busy waiting” code) we should use the blocking approach.

CHALMERS | UNIVERSITY OF GOTHENBURG

Call-back functionality

Protected objects:

```
protected type Exclusive_Resource is
  entry Acquire;
  procedure Release;
private
  Busy : Boolean := false;
end Exclusive_Resource;

protected body Exclusive_Resource is
  entry Acquire when not Busy is
  begin
    Busy := true;
  end Acquire;
  procedure Release is
  begin
    Busy := false;
  end Release;
end Exclusive_Resource;

...
```

If task blocks here, call-back information must be saved in order to wake up the task later.

CHALMERS | UNIVERSITY OF GOTHENBURG

Call-back functionality

Monitors:

```
monitor body Exclusive_Resource is
  Busy : Boolean := false;
  notBusy: condition_variable;
  procedure Acquire is
  begin
    if Busy then Wait(notBusy); end if;
    Busy := true;
  end Acquire;
  procedure Release is
  begin
    Busy := false;
    Send(notBusy);
  end Release;
end Exclusive_Resource;
```

If task blocks here, call-back information must be saved in order to wake up the task later.

CHALMERS | UNIVERSITY OF GOTHENBURG

Call-back functionality

Semaphores:

```
protected type Semaphore (Initial : Natural := 0) is
    entry Wait;
    procedure Signal;
private
    Value : Natural := Initial;
end Semaphore;

protected body Semaphore is
    entry Wait when Value > 0 is
    begin
      Value := Value - 1;
    end Wait;
    procedure Signal is
    begin
      Value := Value + 1;
    end Signal;
end Semaphore;
...
```

If task blocks here, call-back information must be saved in order to wake up the task later.

Call-back functionality

Call-back information:

- As shown in the previous examples, the implementation of resource management mechanisms such as protected objects, monitors and semaphores make use of call-back information to be able to wake up a blocked task when the requested resource becomes available.
- Since multiple tasks may want to request access to a resource that is currently unavailable, call-back information for each of these tasks must be stored in a suitable data structure, e.g., a queue.

Call-back functionality

Call-back functionality in TinyTimber:

- TinyTimber has inherent call-back functionality for *object methods*, via the `SYNC()` call, in its implementation of the object (with its internal state) as an exclusive resource.
- If a generic `acquire/release` type of mechanism for shared resources, such as semaphores, is to be added to TinyTimber a separate call-back functionality must be implemented for that mechanism.

  In this week's exercise session we show how a semaphore object type (with call-back) is implemented in TinyTimber.
- TinyTimber also has call-back functionality in the *device drivers* for the serial port and CAN interfaces, in support of the reactive programming paradigm.

Call-back functionality

Device driver programming:

- A device driver is a software module that allows the user to interact with peripheral devices, such as serial ports or network interfaces, in a hardware-independent fashion.
- The device driver conceals the details in the cooperation between software and hardware by defining a set of operations on the device, e.g., *initialize*, *read*, and *write*.
- The device driver also contains handler code for any hardware interrupt that may be associated with the peripheral device. If a task may block while waiting for an event to happen on the device, e.g., data becomes available, the interrupt handler will require call-back information from the user of the device.

Interrupt handlers in TinyTimber

Guidelines for interrupt handling in TinyTimber:

- Interrupts must be handled using objects.
- An interrupt handler must be written as a method in the object.
- Data being processed by the interrupt handler must be stored in state variables in the object.
- Reading and writing such data from the user's program code must be done via synchronous calls to methods in the object, i.e., `SYNC()` calls.

We will now study the device driver for the serial port (SCI) in more detail.

Interrupt handlers in TinyTimber

Example: implementing an SCI interrupt handler:

1. Define class `Serial`, and add state variables for:
   - the hardware base address of the device
   - call-back information for a method if data received by the handler needs to be taken care of by the user-level code (the call back should be done using an `ASYNC()` call)
   - necessary local storage (buffers, queues, etc)
2. Define a symbol `SCI_PORT0` representing the hardware base address of the device.

   ```
   #define SCI_PORT0 device_hardware_address
   ```
3. Create an object `sci0` of class `Serial`, and initialize it with:
   - the hardware base address `SCI_PORT0`
   - any possible call-back information

Interrupt handlers in TinyTimber

Example: implementing an SCI interrupt handler (cont'd):

In file 'application.c':

```
App app = { initObject(), 0, 'X' };

void reader(App*, int);

Serial sci0 = initSerial(SCI_PORT0, &app, reader);

void reader(App *self, int c) {    // call-back function
    SCI_WRITE(&sci0, "Rcv: \'");
    SCI_WRITECHAR(&sci0, c);
    SCI_WRITE(&sci0, "\'\n");
}
```


Interrupt handlers in TinyTimber

Example: implementing an SCI interrupt handler (cont'd):

4. Write an interrupt handler as a method `sci_interrupt` and associate it with the object.
5. Declare a symbol `SCI_IRQ0` and assign to it the TinyTimber kernel's logical number of the hardware interrupt:

   ```
   #define SCI_IRQ0 interrupt_logical_number
   ```
6. Inform the TinyTimber kernel that the method is a handler for interrupt `SCI_IRQ0`, by making a call to

   ```
   INSTALL(&sci0, sci_interrupt, SCI_IRQ0);
   ```

   This should be done <u>before</u> the call to `TINYTIMBER()`

Interrupt handlers in TinyTimber

Example: implementing an SCI interrupt handler (cont'd):

7. Provide an operation `SCI_INIT()` that takes care of performing any remaining initialization of the device.
8. Call `SCI_INIT()` in the "kick-off" method that was supplied as argument to the `TINYTIMBER()` call.

CHALMERS | UNIVERSITY OF GOTHENBURG

Interrupt handlers in TinyTimber

Example: implementing an SCI interrupt handler (cont'd):

In file 'application.c':

```
void startApp(App *self, int arg) {
    SCI_INIT(&sci0);
    SCI_WRITE(&sci0, "Hello, hello...\n");
    …
}

int main() {
    INSTALL(&sci0, sci_interrupt, SCI_IRQ0);
    TINYTIMBER(&app, startApp, 0);
}
```