



Embedded Software Systems

Lecture: RTOS Low Level Internals

Mikaël Briday



LS2N – UMR CNRS 6004

CNRS, École Centrale de Nantes, Institut Mines Telecom Atlantique, Université de Nantes

WET/CORO, 2018/2019

Contents



1 RTOS BSP

2 Basic Core Architecture

3 Processor modes

4 Task's Context

5 Context Switch Implementation

Contents



1 RTOS BSP

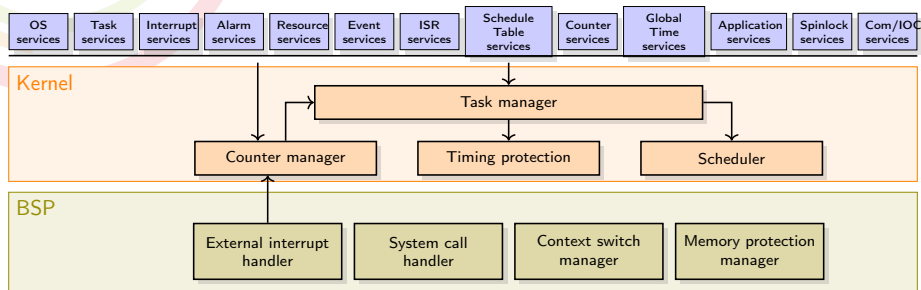
2 Basic Core Architecture

3 Processor modes

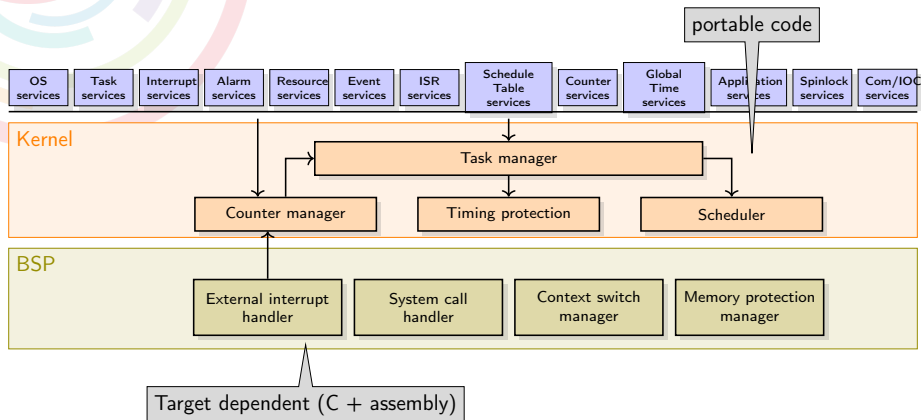
4 Task's Context

5 Context Switch Implementation

Architecture. Example of Trampoline



Architecture. Example of Trampoline





1 RTOS BSP

2 Basic Core Architecture

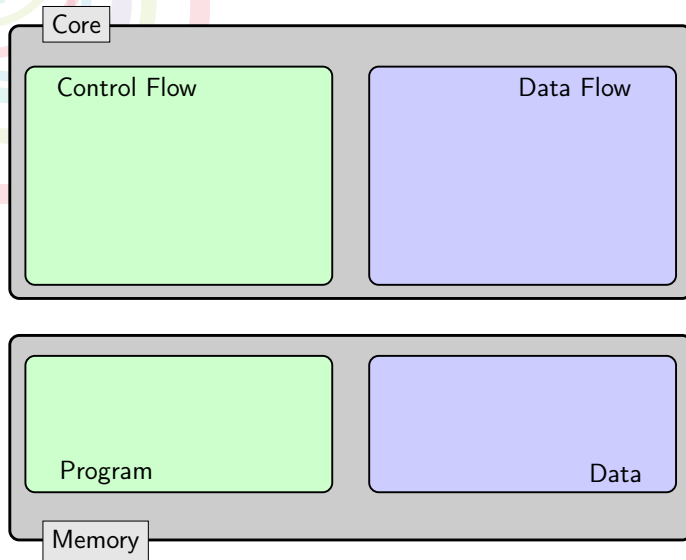
- Register...
- Application Binary Interface
- Stack

3 Processor modes

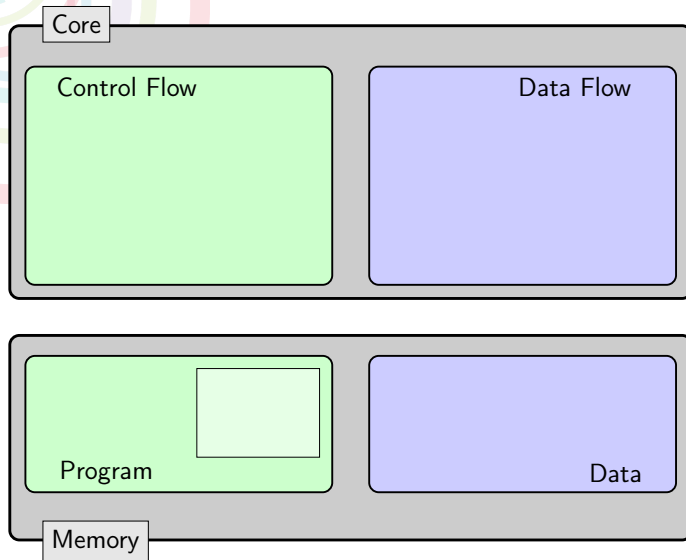
4 Task's Context

5 Context Switch Implementation

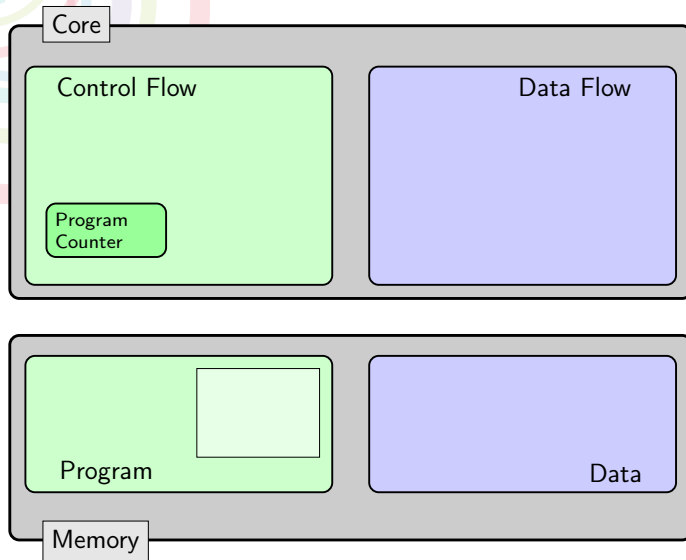
Basic Core Architecture - Operation



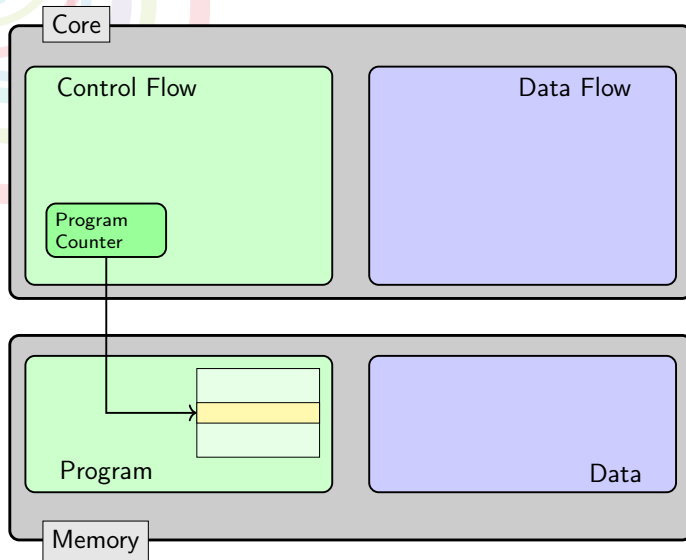
Basic Core Architecture - Operation



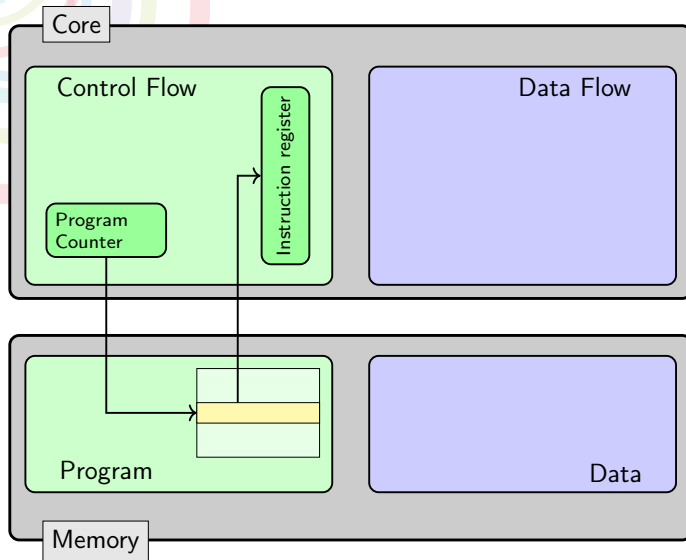
Basic Core Architecture - Operation



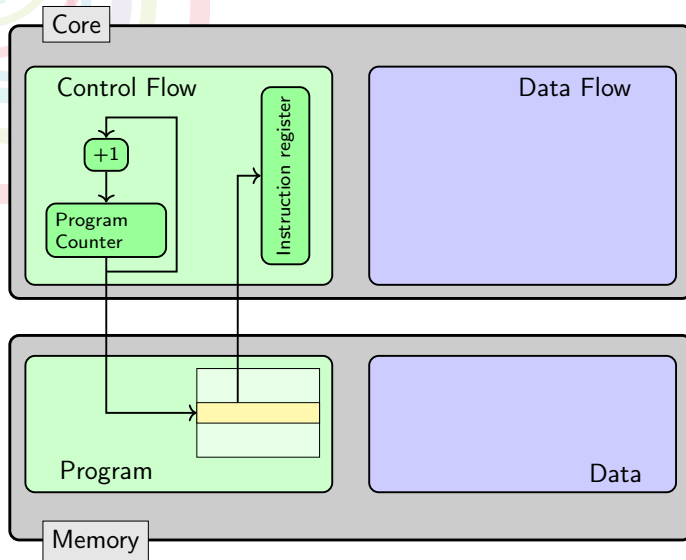
Basic Core Architecture - Operation



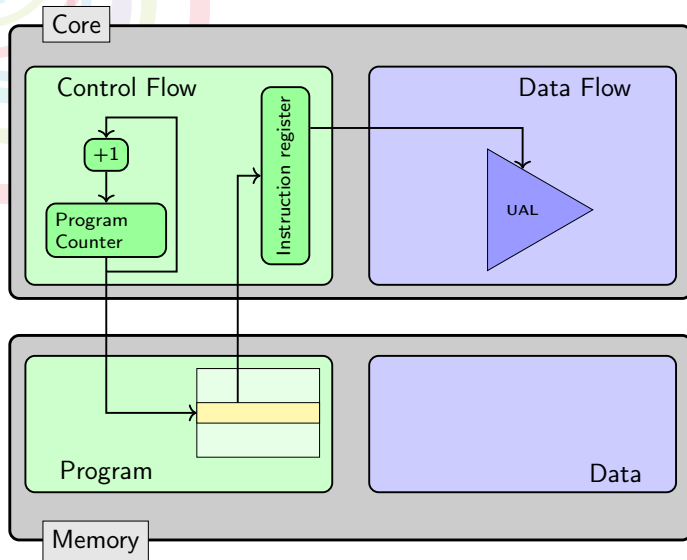
Basic Core Architecture - Operation



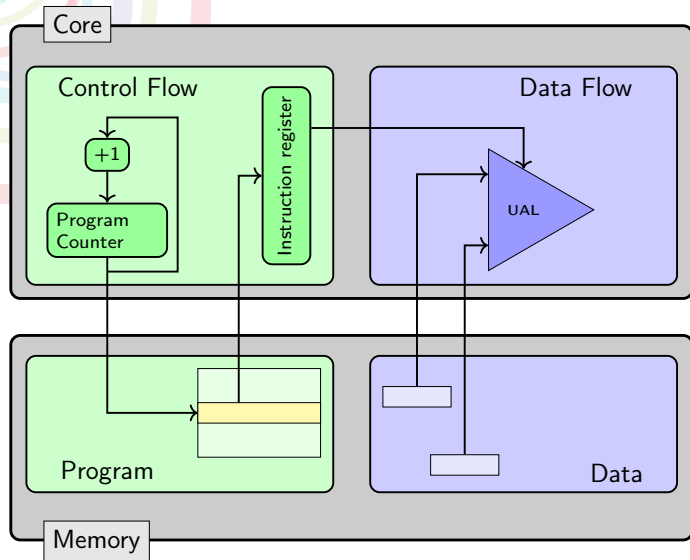
Basic Core Architecture - Operation



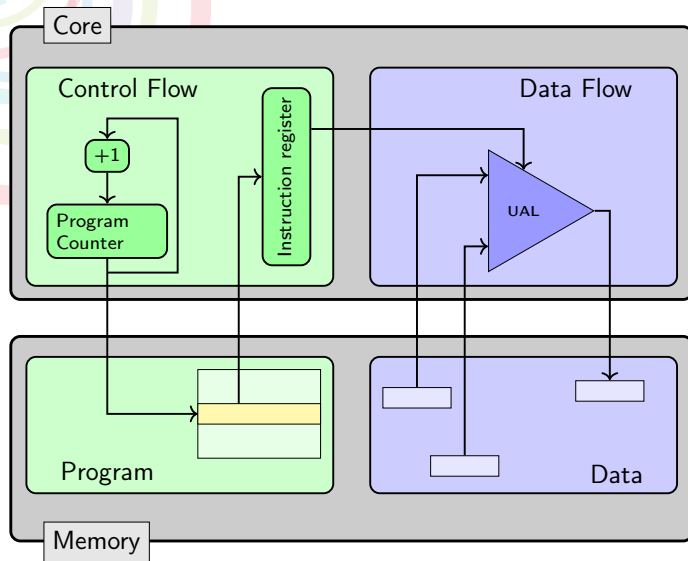
Basic Core Architecture - Operation



Basic Core Architecture - Operation



Basic Core Architecture - Operation



Registers are used to:

- » stores temporary variables to compute: *General Purpose Registers*;
- » store the address of the next instruction to execute: *Program Counter*;
- » store the address of the stack: the *Stack Pointer*
- » stores the status of UAL operations (overflow, carry, zero, negative): the *status register*.

Basic Core Architecture - Registers of the Cortex-M

The ARM Cortex-M architecture uses 16 32-bits registers r0 to r15:

- » r0 to r12 are general purpose registers
- » r13 is the *stack pointer*
- » r14 is the *link register*
- » r15 is the *program counter*

and one *status register*: psr with flags like:

- » C: the *carry* of the last arithmetic operation;
- » N: the result of the last operation is *negative*;
- » V: the last operation *overflowed*;
- » Z: the last result is *zero*.

r0
r1
r3
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
sp
lr
pc

psr

Basic Core Architecture - Example of assembly code

A C code is translated to assembly code, thanks to the compiler.
Columns in assembly listing are:

- » program memory address (hexadecimal);
- » instruction code (hexadecimal);
- » instruction mnemonic (textual representation)

```
0800049c <compute>:
800049c: 4904
ldr r1, [pc, #16] ; (80004b0 <compute+0x14>)
800049e: 2300      movs r3, #0
80004a0: 1c5a      adds r2, r3, #1
80004a2: 2a0a      cmp r2, #10
80004a4: f841 2023 str.w r2, [r1, r3, lsl #2]
80004a8: 4613      mov r3, r2
80004aa: d1f9      bne.n 80004a0 <compute+0x4>
80004ac: 2000      movs r0, #0
80004ae: 4770      bx lr
80004b0: 20000454 .word 0x20000454
```

Basic Core Architecture - Example of assembly code

A C code is translated to assembly code, thanks to the compiler.
Columns in assembly listing are:

- » program memory address (hexadecimal);
- » instruction code (hexadecimal);
- » instruction mnemonic (textual representation)

```
volatile int b[10];
int compute() {
    for(int i=0; i < 10; i++) {
        b[i] = i+1;
    }
    return 0;
}
```

```
0800049c <compute>:
800049c: 4904
ldr r1, [pc, #16] ; (80004b0 <compute+0x14>)
800049e: 2300      movs r3, #0
80004a0: 1c5a      adds r2, r3, #1
80004a2: 2a0a      cmp r2, #10
80004a4: f841 2023 str.w r2, [r1, r3, lsl #2]
80004a8: 4613      mov r3, r2
80004aa: d1f9      bne.n 80004a0 <compute+0x4>
80004ac: 2000      movs r0, #0
80004ae: 4770      bx lr
80004b0: 20000454 .word 0x20000454
```

(Embedded) Application Binary Interface

The EABI (*Embedded Application Binary Interface*) is a *set of rules* that should be respected to allow binary files from different sources to work together:

- » calling convention
 - » how arguments are given to a function call?
 - » how the function result is returned?
- » which registers should be preserved in a function?
- » much more (stack frame organization, data types, alignment, ...)

Application Binary Interface - Cortex-M

Register	Special	Role in procedure call std
r0		argument - result - scratch register 1
r1		argument - result - scratch register 2
r2		argument - scratch register 3
r3		argument - scratch register 4
r4		variable register 1
r5		variable register 2
r6		variable register 3
r7		variable register 4
r8		variable register 5
r9		platform register
r10		variable register 7
r11		variable register 8
r12	IP	Intra-procedure-call scratch register
r13	SP	Stack Pointer
r14	LR	Link Register
r15	PC	Program Counter

Registers r4-r11/ sp should be preserved in a function (using the stack...)

Basic Core Architecture - function call (reminder)

The caller:

- » stores arguments from left to the right to r0-r3 and the the stack
- » calls the procedure with b1 procname
- » gets back the result from r0-r1

The callee

- » pushes on the stack all the registers in r4-r12,r14 used in the code of the procedure
- » computes and stores the result in r0-r1
- » restore modified registers from the stack
- » return to the call point with b 1r

Basic Core Architecture - Stack

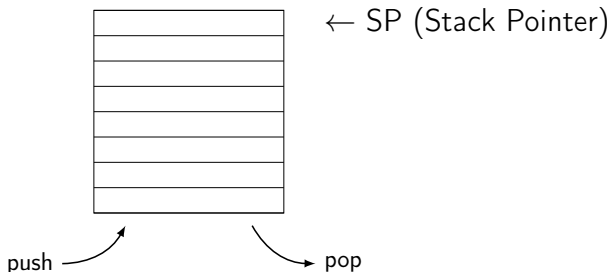
The stack is a memory zone used to store:

- » local variables of a function;
- » return address when calling a function;

The memory is managed as a *LIFO* (Last In First Out) container.

It basically uses 2 operations:

- » push: insert a data on top of stack
- » pop: get back data from stack



Basic Core Architecture - Stack

The stack is a memory zone used to store:

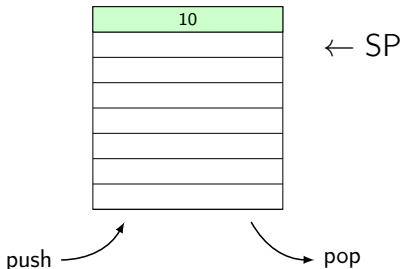
- » local variables of a function;
- » return address when calling a function;

The memory is managed as a *LIFO* (Last In First Out) container.

It basically uses 2 operations:

- » push: insert a data on top of stack
- » pop: get back data from stack

push 10



Basic Core Architecture - Stack

The stack is a memory zone used to store:

- » local variables of a function;
- » return address when calling a function;

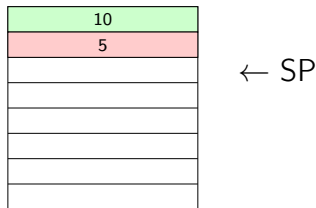
The memory is managed as a *LIFO* (Last In First Out) container.

It basically uses 2 operations:

- » push: insert a data on top of stack
- » pop: get back data from stack

push 10

push 5



Basic Core Architecture - Stack

The stack is a memory zone used to store:

- » local variables of a function;
- » return address when calling a function;

The memory is managed as a *LIFO* (Last In First Out) container.

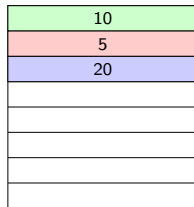
It basically uses 2 operations:

- » push: insert a data on top of stack
- » pop: get back data from stack

push 10

push 5

push 20



← SP

push

pop

Basic Core Architecture - Stack

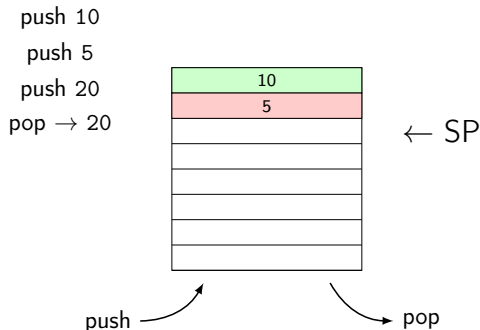
The stack is a memory zone used to store:

- » local variables of a function;
- » return address when calling a function;

The memory is managed as a *LIFO* (Last In First Out) container.

It basically uses 2 operations:

- » push: insert a data on top of stack
- » pop: get back data from stack



Basic Core Architecture - Stack

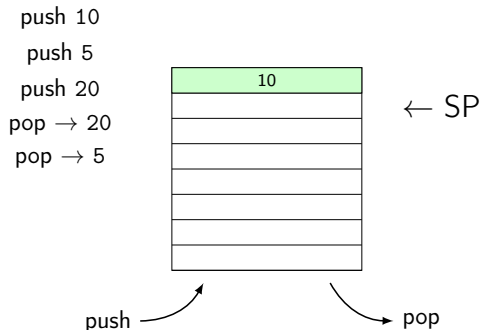
The stack is a memory zone used to store:

- » local variables of a function;
- » return address when calling a function;

The memory is managed as a *LIFO* (Last In First Out) container.

It basically uses 2 operations:

- » push: insert a data on top of stack
- » pop: get back data from stack



In fact, modern processors with a RISC architecture handles the stack using a *memory indirect addressing mode* and uses a *stack frame*.

- » It reserves memory when it enters in a function (updating SP)

```
sub sp,sp,#STACK_FRAME_SIZE;
```

- » Then it accesses to variables using a constant offset:

```
str r1, [sp, #offset] ; store r1 to [sp+offset]
```

```
ldr r1, [sp, #offset] ; load r1 from [sp+offset]
```

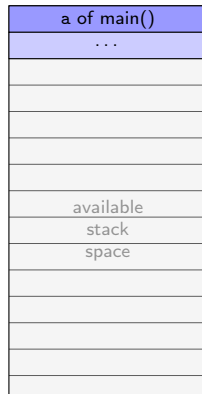
Basic Core Architecture - Stack

The stack is a memory zone used to store local variables of a function.

- » When a function is called, the stack pointer is decremented to make room for local variable;
- » when a function returns, the stack pointer is restored to its previous value.

```
void main() {           1
    int a;               2
    while(1) {           3
        loop();          4
    }                    5
}                        6
                          7
void loop() {           8
    int tab[3];          9
    ...                 10
    readSensor();       11
}                        12
                          13
int readSensor() {      14
    int result;         15
    ...                 16
    return result;      17
}                        18
```

stack frame |
main



← SP

Basic Core Architecture - Stack

The stack is a memory zone used to store local variables of a function.

- » When a function is called, the stack pointer is decremented to make room for local variable;
- » when a function returns, the stack pointer is restored to its previous value.

```
void main() {           1
    int a;              2
    while(1) {          3
        loop();         4
    }                  5
}                      6

void loop() {           7
    int tab[3];         8
    ...                9
    readSensor();      10
}                     11

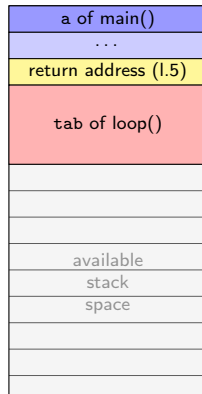
int readSensor() {     12
    int result;        13
    ...               14
    return result;     15
}                     16
                     17
                     18
```

stack frame |

main

stack frame |

loop



← SP

Basic Core Architecture - Stack

The stack is a memory zone used to store local variables of a function.

- » When a function is called, the stack pointer is decremented to make room for local variable;
- » when a function returns, the stack pointer is restored to its previous value.

```
void main() {           1
    int a;               2
    while(1) {           3
        loop();          4
    }                    5
}                        6

void loop() {           7
    int tab[3];          8
    ...                 9
    readSensor();       10
}                      11

int readSensor() {      12
    int result;         13
    ...                14
    return result;      15
}                      16
                      17
                      18
```

stack frame |

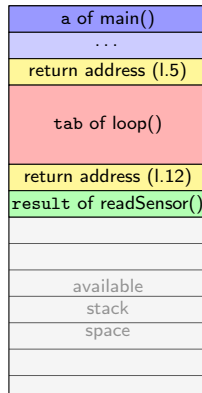
main

stack frame |

loop

stack frame |

readSensor




← SP

With a RTOS, the stack is part of the task's context, its size should be set:

- » not too high not to waste RAM (which is limited on embedded systems)
- » not too low, because a *stack overflow* will crash the system (corrupt another stack or other memory parts)


To help the developer to find a correct stack size, a static or dynamic analysis should be performed.

Recursive calls are prohibited on critical real time applications!



A *static* analysis finds an upper value of the stack usage, *for any run of the program*.

- » the `-fstack-usage` flag of `gcc`: the compiler outputs stack usage information file for the program, on a per-function basis; It is a good approach if the sources are available.
- » this information should be associated with the program control flow graph.
- » it does not work for binary provided files. . .



A *dynamic* analysis finds an upper value of the stack usage for a dedicated scenario:

- » in Trampoline, the flag `PAINT_STACK = TRUE`; in the OS config section of the `oil` file writes a pattern on the stack so that real stack usage can be retrieved;
- » the worst case scenario may be not easy to determine (... and to prove that's the worst!)

Contents



1 RTOS BSP

2 Basic Core Architecture

3 Processor modes

4 Task's Context


5 Context Switch Implementation

modes are defined to restrict the software to hardware parts. This is required if the processor have a MPU (Memory Protection Unit):

- » unrestricted mode, often called *kernel mode*. In this mode, the CPU can perform any operation allowed by the architecture.
- » restricted mode, (*user mode*) have restrictions on I/O operations. For instance, the access to the configuration of the MPU should not be allowed in user mode.

Some CPU architecture allows multiple user modes (not in embedded context), organised into *rings*. They are in particular used for virtualisation.

Processor modes on the Cortex-M

- 
- » *Handler mode*: Used to handle exceptions (kernel mode)
 - » *Thread mode*: Used to execute application software (user mode):
 - » limited access to `msr` and `mrs` (move from/to register from special function register)
 - » no access to `cps` instruction (Change Processor State)
 - » no access to *system timer*, *NVIC*, *system control block*
 - » might have restricted access to memory or peripherals.

Switch between processor modes

The switch is performed by *an interrupt*. To get into the kernel, 2 methods can be used:

- » Hardware interrupt: from an alarm (timer), or a user defined ISR (in that late case, the kernel get the control first!)
- » Software interrupt: when there is a system call. In that case, we use a specific assembly instruction *svc*.

As a consequence, the system call to the kernel should be written in assembly.

SVC instruction on Cortex-M

The SVC in the Cortex-M is defined as: `SVC #imm8` where `#imm8` is a constant not used by the processor (no effect).

The binary opcode of SVC is: `0xDFxx`, where `xx` is the `imm8` encoding. This value can be used by the programmer to get back the service id (instead of using `r3`)

exercise

Give the assembly code to retrieve the `#imm8` value in the service call handler

We can use `bic` (Bit Clear) instruction :

```
bic rd, rs, #imm ; rd <- rs & ~ imm
```


SVC instruction on Cortex-M

The SVC in the Cortex-M is defined as: `SVC #imm8` where `#imm8` is a constant not used by the processor (no effect).

The binary opcode of SVC is: `0xDFxx`, where `xx` is the `imm8` encoding. This value can be used by the programmer to get back the service id (instead of using `r3`)

exercise

Give the assembly code to retrieve the `#imm8` value in the service call handler

We can use `bic` (Bit Clear) instruction :

```
bic rd, rs, #imm ; rd <- rs & ~ imm
```

```
ldr r0, [lr, #-4]  
bic r0, r0, #0xFF00
```

Contents



1 RTOS BSP

2 Basic Core Architecture

3 Processor modes

4 Task's Context

5 Context Switch Implementation

The task's context consists in all private data required by the task:

- » working registers (GPRs, SP, PC, ...);
- » the stack zone;

A context switch consists only in replacing the context of the running task with the one of the new elected task.

Context Switch Phases

- » save the context of the task that is in the running state (if needed);
- » load the context of the task that has been chosen by the scheduler.



On Trampoline, the task's context is split in two parts:

- ») a *static part* that contains properties of the task that won't change at runtime (stored in flash)
- ») a *dynamic part* that contains runtime values (stored in RAM)

Contents



1 RTOS BSP

2 Basic Core Architecture

3 Processor modes

4 Task's Context

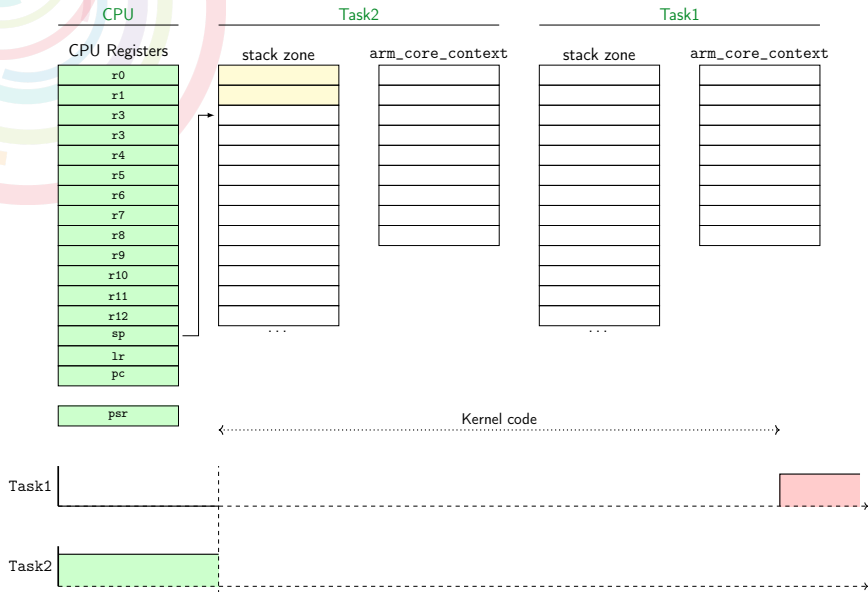
5 Context Switch Implementation

Service Call Handler implementation in Trampoline

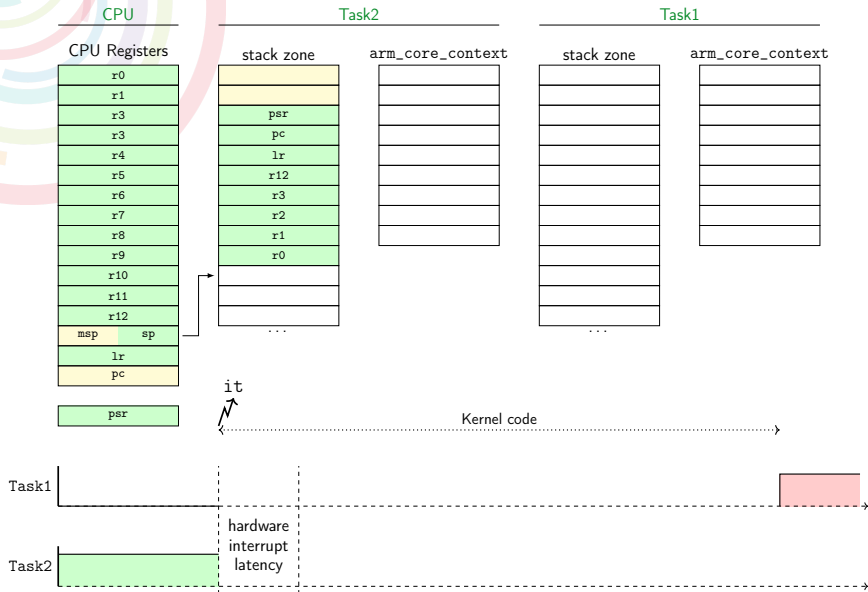
The service call handler `tpl_sc_handler` is the entry point for each service call. It:

- » *checks* that the service call is correct (identifier)
- » *resets* boolean fields `NEED_SWITCH` and `NEED_SCHEDULE` in the main kernel structure (`tpl_kern`).
- » gets the main function address for the service call, from the identifier (function pointer tabular implementation)
- » *calls the service*
- » *tests* the 2 previous fields and:
 - » save the context if required
 - » run the elected function (update internal structures in C)
 - » load the new context
- » gets back from service handler

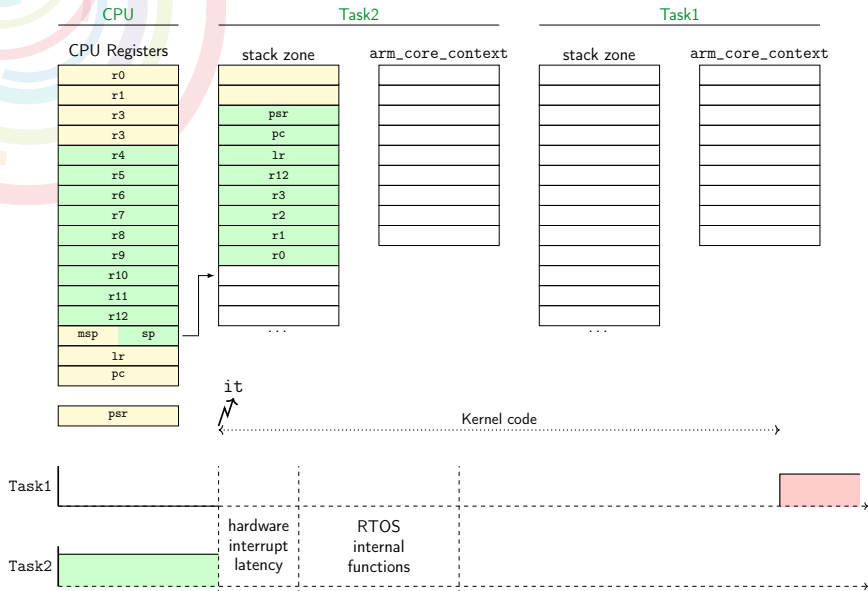
Interrupt Handling Implementation (with a context switch)



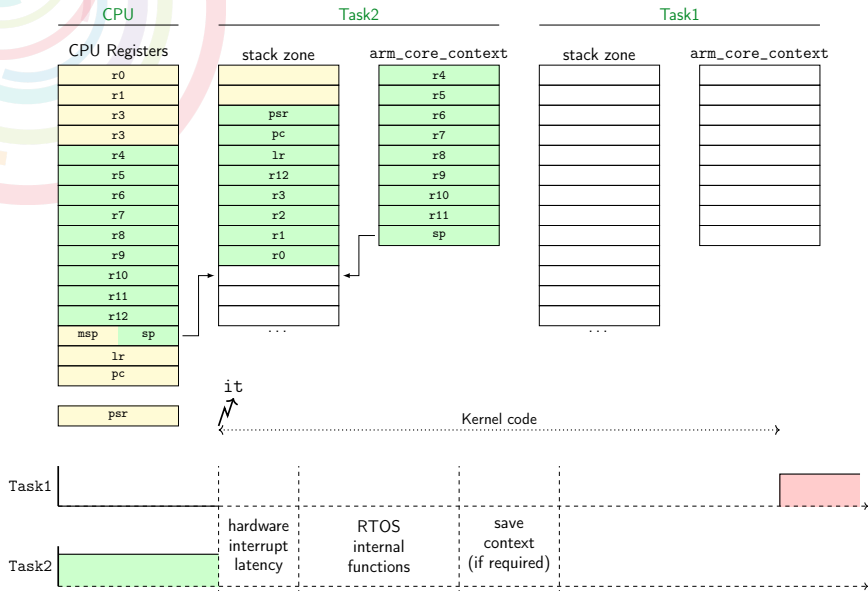
Interrupt Handling Implementation (with a context switch)



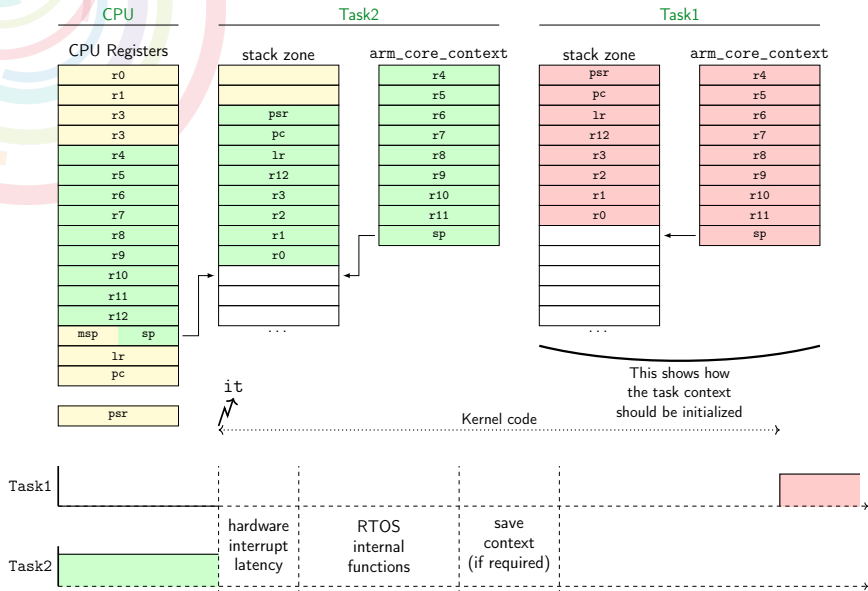
Interrupt Handling Implementation (with a context switch)



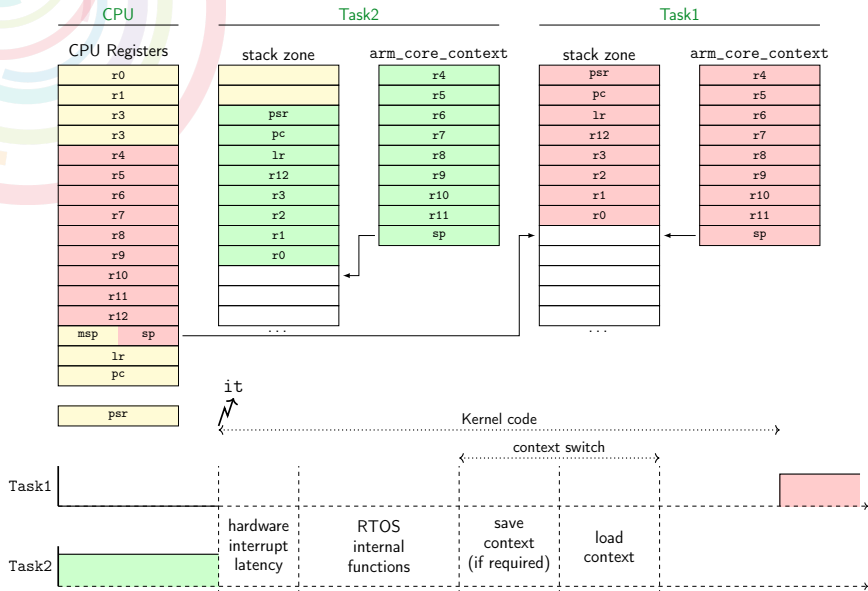
Interrupt Handling Implementation (with a context switch)



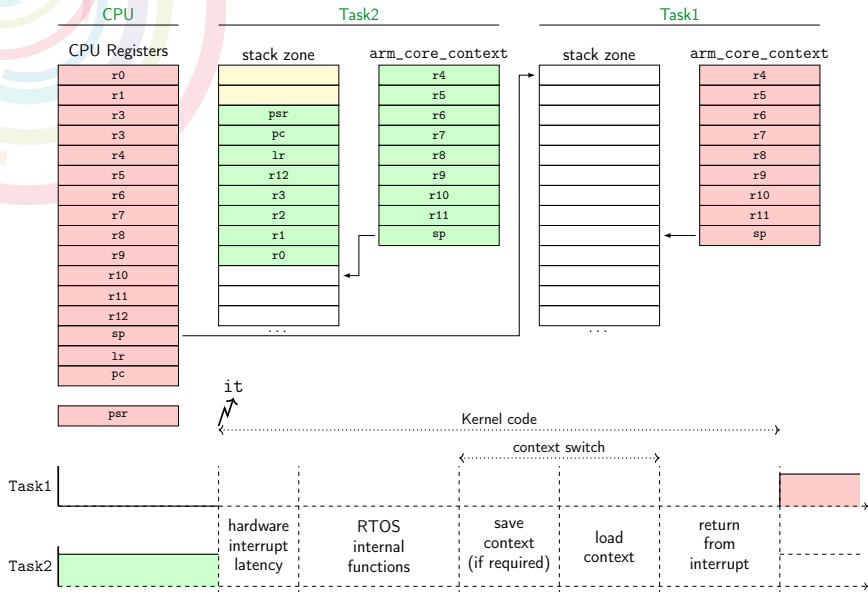
Interrupt Handling Implementation (with a context switch)



Interrupt Handling Implementation (with a context switch)



Interrupt Handling Implementation (with a context switch)



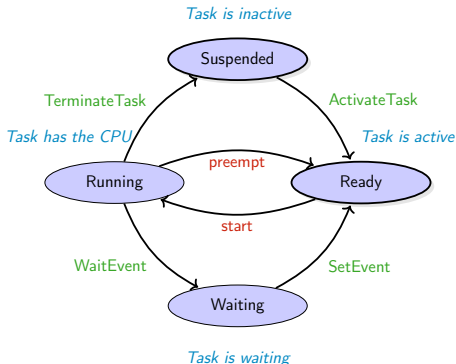
Osek - Tasks related system Calls

Basic Tasks

- » `TerminateTask()`;
- » `ActivateTask(TaskId)`;
- » `ChainTask(TaskId)`;


Extended Tasks

- » `SetEvent(TaskId, Mask)`;
- » `ClearEvent(Mask)`;
- » `GetEvent(TaskId, Mask)`;
- » `WaitEvent(Mask)`;





6 Semaphore (reminder) - lab intro

- 
- » Proposed by Edsger Dijkstra
 - » It allows to protect access to shared resources
 - » This mechanism, available in many OS (not OSEK) offers 3 functions:
 - » Init() : initialize the semaphore;
 - » P() or SemWait() to test the semaphore (Probieren);
 - » V() or SemPost() to increment the semaphore (Verhogen).

A counter is associated to the semaphore.

- » A call to `SemWait()` is used to ask for resource access:
 - » If the counter is > 0 , it is decremented and the resource may be taken.
 - » If the counter is $= 0$, the task which called `P()` is put in the waiting state until the counter became > 0 . At that time the task will be awoken and the counter will be decremented again.
- » A call to `SemPost()` is used to release a resource:
 - » The counter is incremented and a task which is waiting for the resource may be put in ready state.

Semaphore

If there is only one resource to protect a *critical section*, it is called a mutual exclusion (*mutex*):

```
TASK(Task1)
{
    ... //part 1
    SemWait(S1);
    ... //part 2
    // - critical section! -
    SemPost(S1);
    ... //part 3
}
```

```
TASK(Task2)
{
    ... //part 4
    SemWait(S1);
    ... //part 5
    // - critical section! -
    SemPost(S1);
    ... //part 6
}
```

Critical sections (part2 and part5) *should not* overlap!



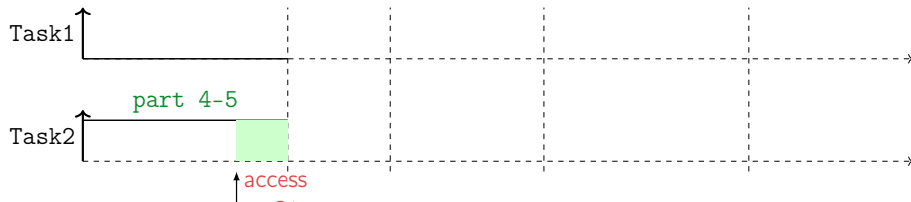
Semaphore

If there is only one resource to protect a *critical section*, it is called a mutual exclusion (*mutex*):

```
TASK(Task1)
{
    ... //part 1
    SemWait(S1);
    ... //part 2
    // - critical section! -
    SemPost(S1);
    ... //part 3
}
```

```
TASK(Task2)
{
    ... //part 4
    SemWait(S1);
    ... //part 5
    // - critical section! -
    SemPost(S1);
    ... //part 6
}
```

Critical sections (part2 and part5) *should not* overlap!



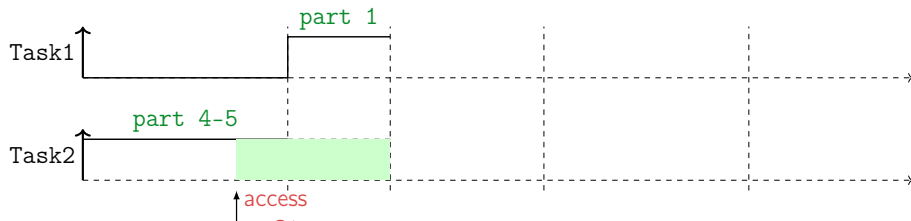
Semaphore

If there is only one resource to protect a *critical section*, it is called a mutual exclusion (*mutex*):

```
TASK(Task1)
{
    ... //part 1
    SemWait(S1);
    ... //part 2
    // - critical section! -
    SemPost(S1);
    ... //part 3
}
```

```
TASK(Task2)
{
    ... //part 4
    SemWait(S1);
    ... //part 5
    // - critical section! -
    SemPost(S1);
    ... //part 6
}
```

Critical sections (part2 and part5) *should not* overlap!



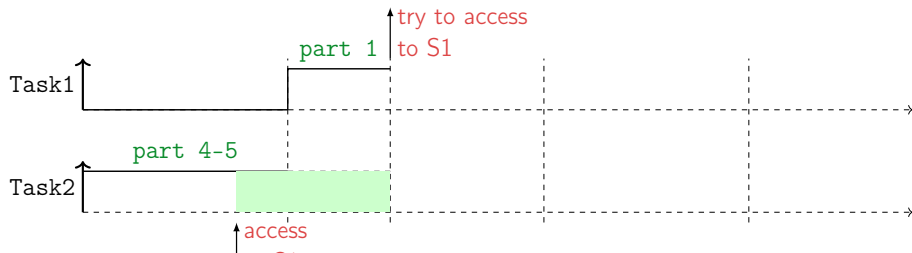
Semaphore

If there is only one resource to protect a *critical section*, it is called a mutual exclusion (*mutex*):

```
TASK(Task1)
{
    ... //part 1
    SemWait(S1);
    ... //part 2
    // - critical section! -
    SemPost(S1);
    ... //part 3
}
```

```
TASK(Task2)
{
    ... //part 4
    SemWait(S1);
    ... //part 5
    // - critical section! -
    SemPost(S1);
    ... //part 6
}
```

Critical sections (part2 and part5) *should not* overlap!



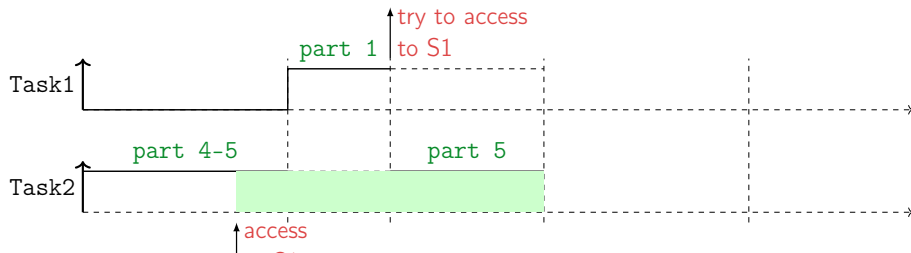
Semaphore

If there is only one resource to protect a *critical section*, it is called a mutual exclusion (*mutex*):

```
TASK(Task1)
{
    ... //part 1
    SemWait(S1);
    ... //part 2
    // - critical section! -
    SemPost(S1);
    ... //part 3
}
```

```
TASK(Task2)
{
    ... //part 4
    SemWait(S1);
    ... //part 5
    // - critical section! -
    SemPost(S1);
    ... //part 6
}
```

Critical sections (part2 and part5) *should not* overlap!



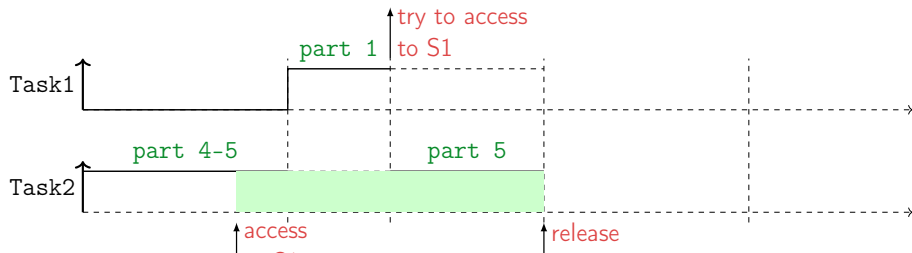
Semaphore

If there is only one resource to protect a *critical section*, it is called a mutual exclusion (*mutex*):

```
TASK(Task1)
{
    ... //part 1
    SemWait(S1);
    ... //part 2
    // - critical section! -
    SemPost(S1);
    ... //part 3
}
```

```
TASK(Task2)
{
    ... //part 4
    SemWait(S1);
    ... //part 5
    // - critical section! -
    SemPost(S1);
    ... //part 6
}
```

Critical sections (part2 and part5) *should not* overlap!



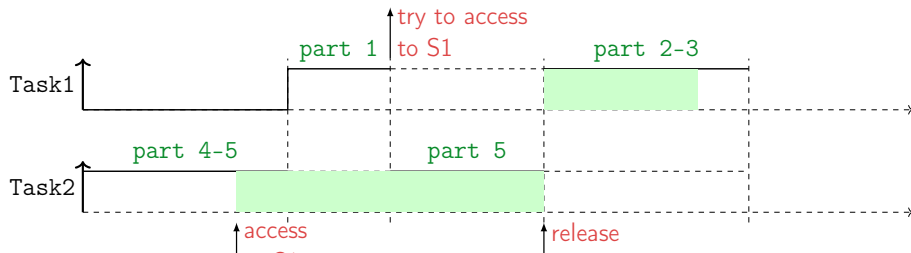
Semaphore

If there is only one resource to protect a *critical section*, it is called a mutual exclusion (*mutex*):

```
TASK(Task1)
{
    ... //part 1
    SemWait(S1);
    ... //part 2
    // - critical section! -
    SemPost(S1);
    ... //part 3
}
```

```
TASK(Task2)
{
    ... //part 4
    SemWait(S1);
    ... //part 5
    // - critical section! -
    SemPost(S1);
    ... //part 6
}
```

Critical sections (part2 and part5) *should not* overlap!



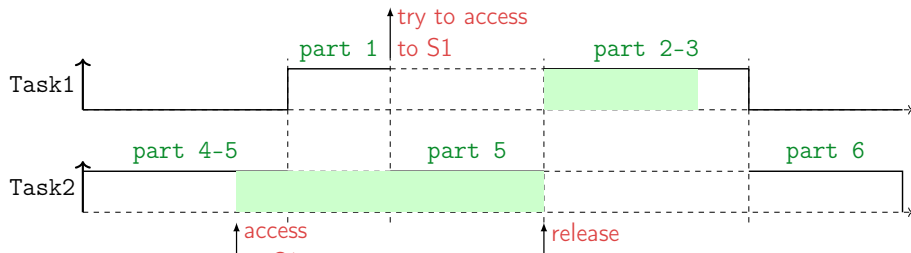
Semaphore

If there is only one resource to protect a *critical section*, it is called a mutual exclusion (*mutex*):

```
TASK(Task1)
{
    ... //part 1
    SemWait(S1);
    ... //part 2
    // - critical section! -
    SemPost(S1);
    ... //part 3
}
```

```
TASK(Task2)
{
    ... //part 4
    SemWait(S1);
    ... //part 5
    // - critical section! -
    SemPost(S1);
    ... //part 6
}
```

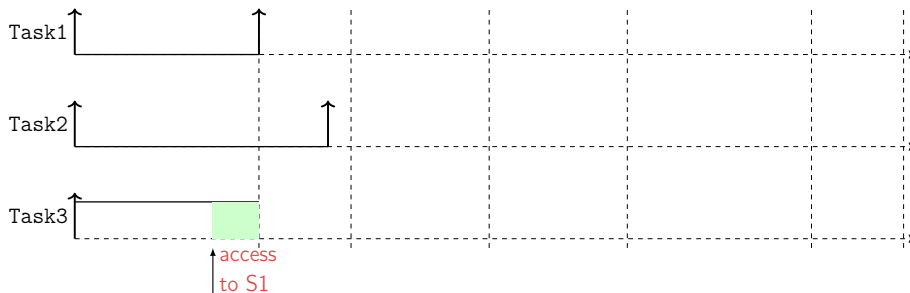
Critical sections (part2 and part5) *should not* overlap!



Semaphore - Problem of priority inversion

Classical synchronization mechanism (semaphore, mutex) may have the priority inversion problem:

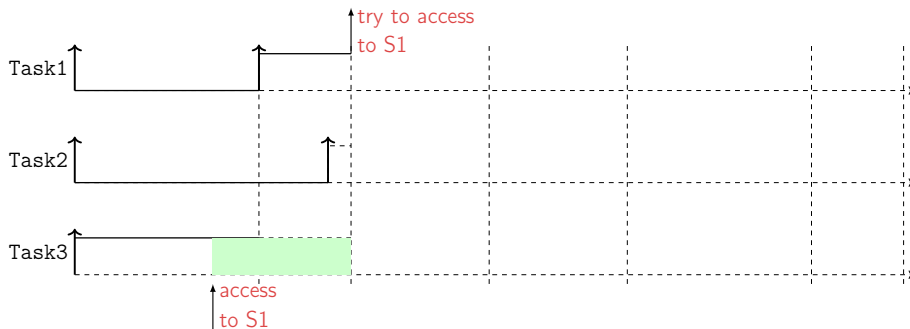
- » A task with a lower priority may delay a higher priority task.
- » The following example shows 3 preemptable tasks, T1 has the higher priority:



Semaphore - Problem of priority inversion

Classical synchronization mechanism (semaphore, mutex) may have the priority inversion problem:

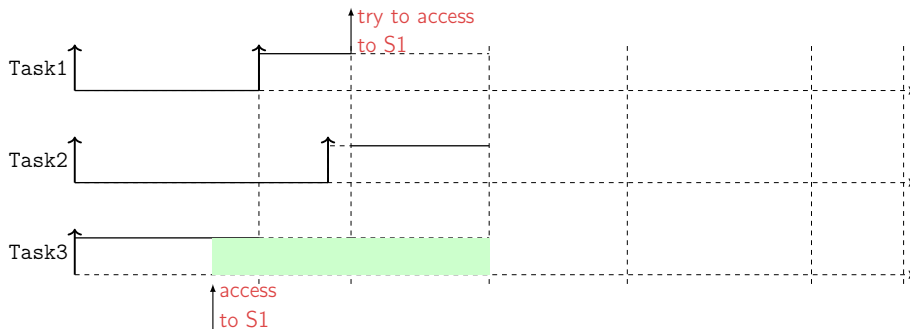
- » A task with a lower priority may delay a higher priority task.
- » The following example shows 3 preemptable tasks, T1 has the higher priority:



Semaphore - Problem of priority inversion

Classical synchronization mechanism (semaphore, mutex) may have the priority inversion problem:

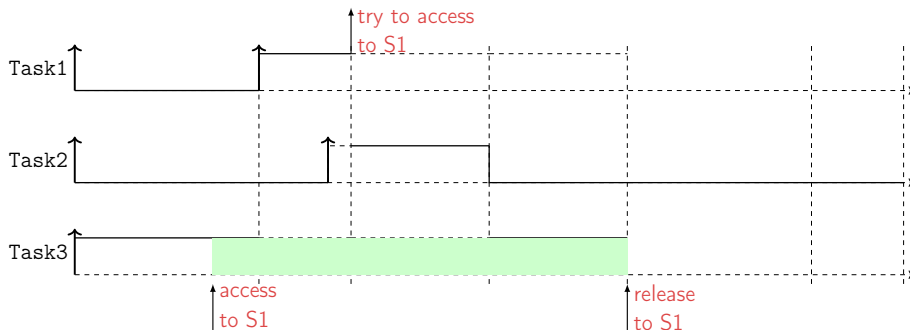
- » A task with a lower priority may delay a higher priority task.
- » The following example shows 3 preemptable tasks, T1 has the higher priority:



Semaphore - Problem of priority inversion

Classical synchronization mechanism (semaphore, mutex) may have the priority inversion problem:

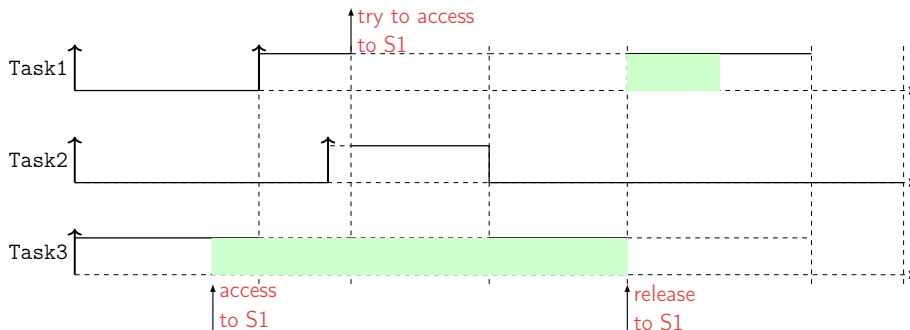
- » A task with a lower priority may delay a higher priority task.
- » The following example shows 3 preemptable tasks, T1 has the higher priority:



Semaphore - Problem of priority inversion

Classical synchronization mechanism (semaphore, mutex) may have the priority inversion problem:

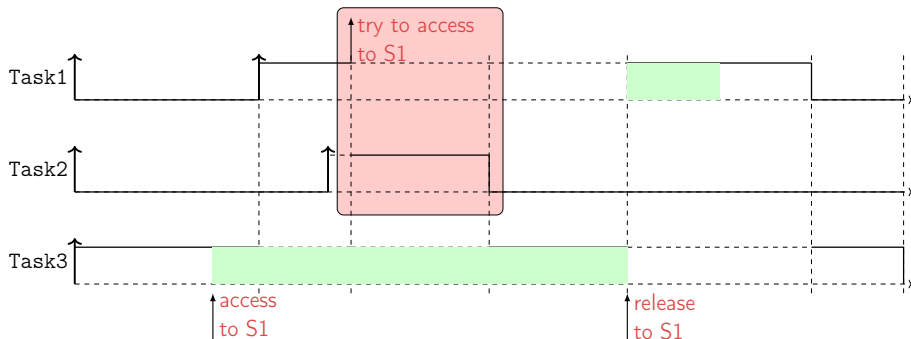
- » A task with a lower priority may delay a higher priority task.
- » The following example shows 3 preemptable tasks, T1 has the higher priority:



Semaphore - Problem of priority inversion

Classical synchronization mechanism (semaphore, mutex) may have the priority inversion problem:

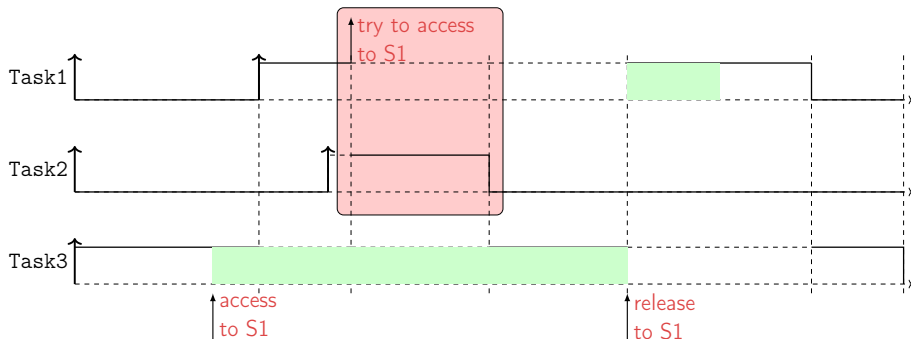
- » A task with a lower priority may delay a higher priority task.
- » The following example shows 3 preemptable tasks, T1 has the higher priority:



Semaphore - Problem of priority inversion

Classical synchronization mechanism (semaphore, mutex) may have the priority inversion problem:

- » A task with a lower priority may delay a higher priority task.
- » The following example shows 3 preemptable tasks, T1 has the higher priority:



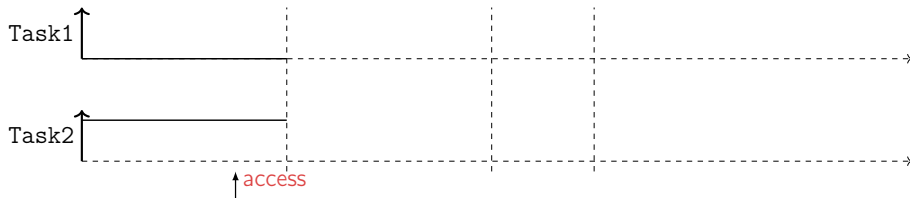
Task1 should wait for Task2 completion...

Semaphore - Deadlock

The biggest problem is the deadlock. It's a design problem

```
TASK(Task1)
{
    SemWait(S1);
    SemWait(S2);
    // - critical section! -
    SemPost(S2);
    SemPost(S1);
}
```

```
TASK(Task2)
{
    SemWait(S2);
    SemWait(S1);
    // - critical section! -
    SemPost(S1);
    SemPost(S2);
}
```

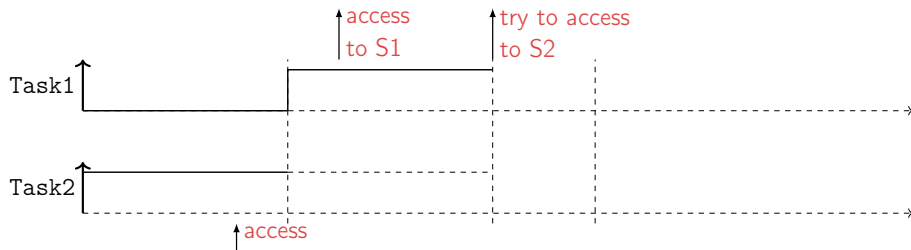


Semaphore - Deadlock

The biggest problem is the deadlock. It's a design problem

```
TASK(Task1)
{
    SemWait(S1);
    SemWait(S2);
    // - critical section! -
    SemPost(S2);
    SemPost(S1);
}
```

```
TASK(Task2)
{
    SemWait(S2);
    SemWait(S1);
    // - critical section! -
    SemPost(S1);
    SemPost(S2);
}
```

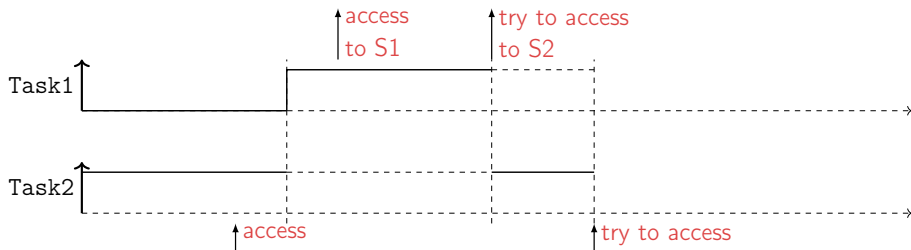


Semaphore - Deadlock

The biggest problem is the deadlock. It's a design problem

```
TASK(Task1)
{
    SemWait(S1);
    SemWait(S2);
    // - critical section! -
    SemPost(S2);
    SemPost(S1);
}
```

```
TASK(Task2)
{
    SemWait(S2);
    SemWait(S1);
    // - critical section! -
    SemPost(S1);
    SemPost(S2);
}
```



Semaphore - Deadlock

The biggest problem is the deadlock. It's a design problem

```
TASK(Task1)
{
    SemWait(S1);
    SemWait(S2);
    // - critical section! -
    SemPost(S2);
    SemPost(S1);
}
```

```
TASK(Task2)
{
    SemWait(S2);
    SemWait(S1);
    // - critical section! -
    SemPost(S1);
    SemPost(S2);
}
```

