Exception Handling

Involves close interaction between hardware and software.

Exception handling is similar to a procedure call with important differences:

- processor prepares exception handling: save* part of the current processor state before execution of the software exception handler
- assigned to each exception is an exception number, the exception handler's code is accessible via some exception table that is configurable by software
- exception handlers run in a different processor mode with unrestricted access to the system resources.

* in special registers or on the stack – we **will** go into the details for some architectures

Recall: ARM Processor Modes

ARM from v5 has (at least) seven basic operating modes

- Each mode has access to **own stack** and a different subset of registers
- Some operations can only be carried out in a privileged mode

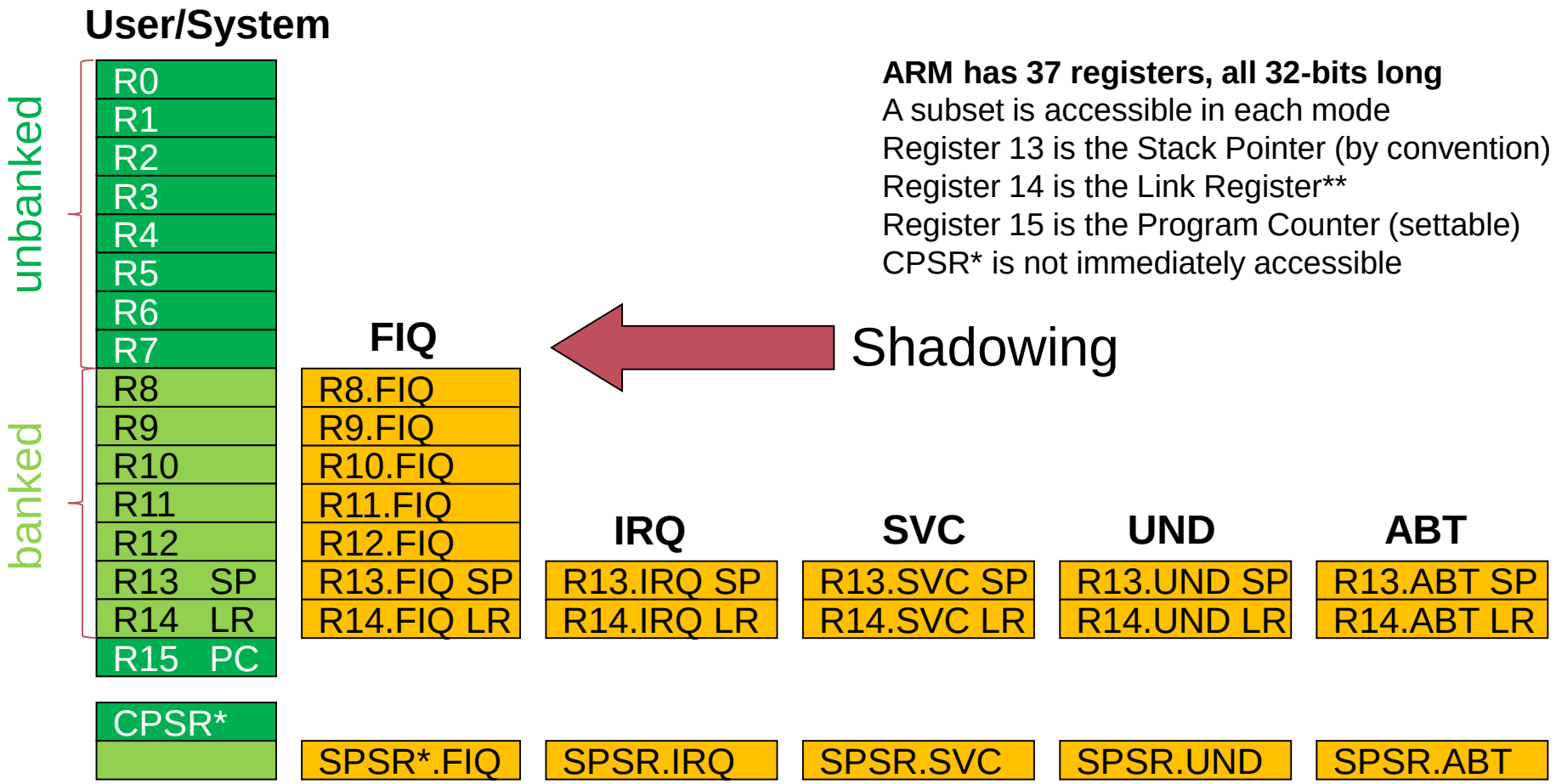
Mode	Description / Cause
Supervisor	Reset / Software Interrupt
FIQ	Fast Interrupt
IRQ	Normal Interrupt
Abort	Memory Access Violation
Undef	Undefined Instruction
System	Privileged Mode with same registers as in User Mode
User	Regular Application Mode

privileged

exceptions

normal execution

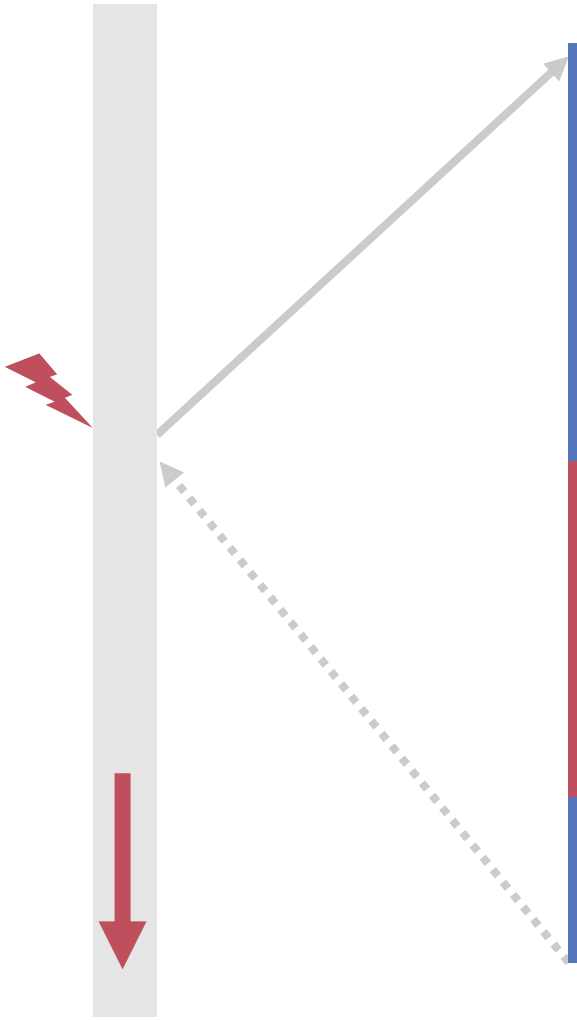
Recall: ARM Register Set



ARM has 37 registers, all 32-bits long
A subset is accessible in each mode
Register 13 is the Stack Pointer (by convention)
Register 14 is the Link Register**
Register 15 is the Program Counter (settable)
CPSR* is not immediately accessible

* current / saved processor status register, accessible via MSR / MRS instructions
** more than a convention: link register set as side effect of some instructions

Exception handling on ARM



Hardware

Hardware action at entry (invoked by exception)

R14(exception_mode):= return link
SPSR(exception_mode) := CPSR
CPSR[4:0] := exception_mode number
CPSR[5] := 0 (* execute in ARM state *)
If exception_mode = Reset or FIQ then CPSR[6]=1 (* disable fast IRQ *)
CPSR[7]=1 (* disable normal interrupts *)
PC=exception vector address

Software

```
STMDB SP!, {R0 .. R11, FP, LR} (* store all non-banked registers on stack *)

... (* exception handler *)

LDMIA SP! {R0..R11,FP,LR} (* read back all non-banked registers from stack*)
SUBS PC,LR, #ofs (* return from interrupt instruction *)
```

HW

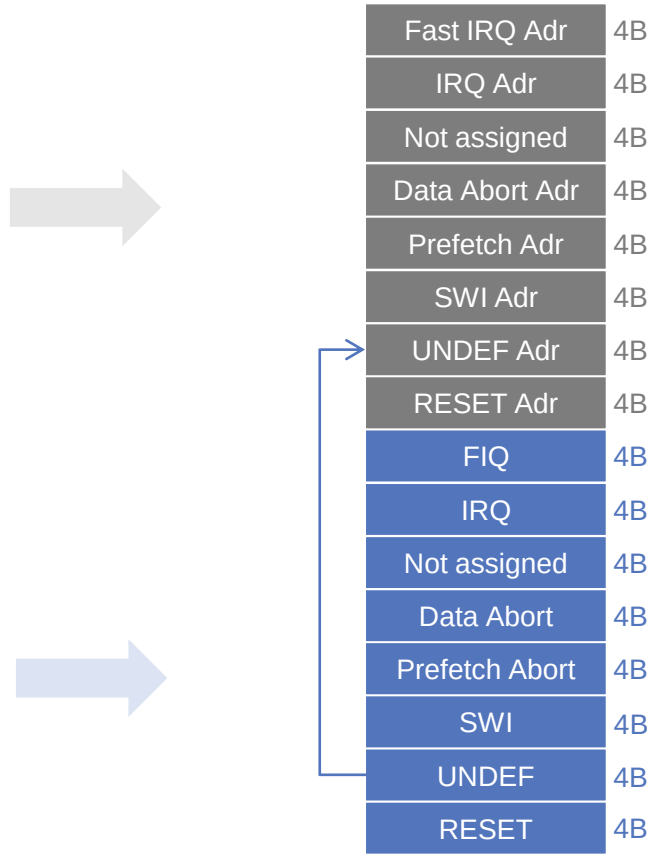
Hardware action at exit (invoked by MOVS or SUBS instruction)

CPSR := SPSR(exception mode) (* includes a reset of the irq/fiq flag *)
PC := LR – ofs

Initialization of Exceptions

```
InstallHandler(SWITrap, Platform.SWI);
InstallHandler( .... );
....
```

```
FOR i := 0 TO 7 DO
   SYSTEM.PUT32(ExceptionVectorBase + 4*i,
      0E59FF018H);
END;
```



Enable/Disable IRQs

```
PROCEDURE EnableIRQs*;
VAR cpsr: SET32;
BEGIN
   SYSTEM.STPSR(0, cpsr);
   cpsr :=cpsr - {7};
   SYSTEM.LDPSR(0, cpsr)
END EnableIRQs;
```

```
PROCEDURE DisableIRQs*;
VAR cpsr: SET32;
BEGIN
   SYSTEM.STPSR(0, cpsr);
   cpsr := cpsr + {7, 8};
   SYSTEM.LDPSR( 0, cpsr)
END DisableIRQs;
```

* reverse cmp/sub meaning compared with x86

Install Timer

```
Platform.WriteWord(Platform.STC1,
       Platform.ReadWord(Platform.STCLO)+Platform.TimerInterval);

Platform.WriteBits (Platform.STCS, {1});

nextTimerInterrupt := Platform.ReadWord(Platform.STC1);

EnableIRQ(Platform.SystemTimerIRQ, TRUE);
```

Sets bit in IRQEnable registers
cf. BCM2835 ARM Peripherals document, Chapter 7, p. 109ff

The System Timer on RPI

- The system timer on RPI is described in BCM2835 document, chapter 12
- It provides 4 timer match registers, where two of them (1 and 3) are available to the ARM Core.
- The system timer is either driven by the APB (peripheral bus), running with 1/2 cpu frequency or from the crystal (19.2 MZh).
- Default source on the RPI is the APB with a timer divider register (described in Quad A7 control document) initialized with 0x06aaaab corresponding to a divider of about 19.2 => 1 MHz timer frequency.
- The System timer IRQs are GPU#1 1 and 3
- The ARM Timer (chapter 14) is driven from the (variable) GPU frequency

ARM peripherals interrupts table.

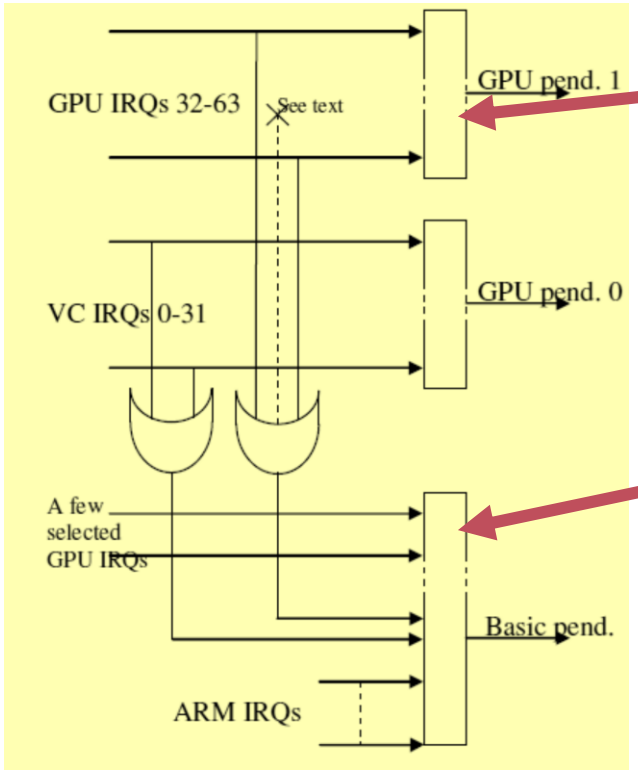
#	IRQ 0-15	#	IRQ 16-31	#	IRQ 32-47	#	IRQ 48-63
0		16		32		48	smi
1	system timer match 1	17		33		49	gpio_int[0]
2		18		34		50	gpio_int[1]
3	system timer match 3	19		35		51	gpio_int[2]
4		20		36		52	gpio_int[3]
5		21		37		53	i2c_int
6		22		38		54	spi_int
7		23		39		55	pcm_int
8		24		40		56	
9	USB controller	25		41		57	uart_int
10		26		42		58	
11		27		43	i2c_spi_slv_int	59	
12		28		44		60	
13		29	Aux int	45	pwa0	61	
14		30		46	pwa1	62	
15		31		47		63	

The table above has many empty entries. These should not be enabled as they will interfere with the GPU operation.

ARM peripherals interrupts table.

0	ARM Timer
1	ARM Mailbox
2	ARM Doorbell 0
3	ARM Doorbell 1
4	GPU0 halted (Or GPU1 halted if bit 10 of control register 1 is set)
5	GPU1 halted
6	Illegal access type 1
7	Illegal access type 0

Install IRQ (Uart)




```
Kernel.EnableIRQ( Platform.UartInstallIrq , TRUE );
```

57

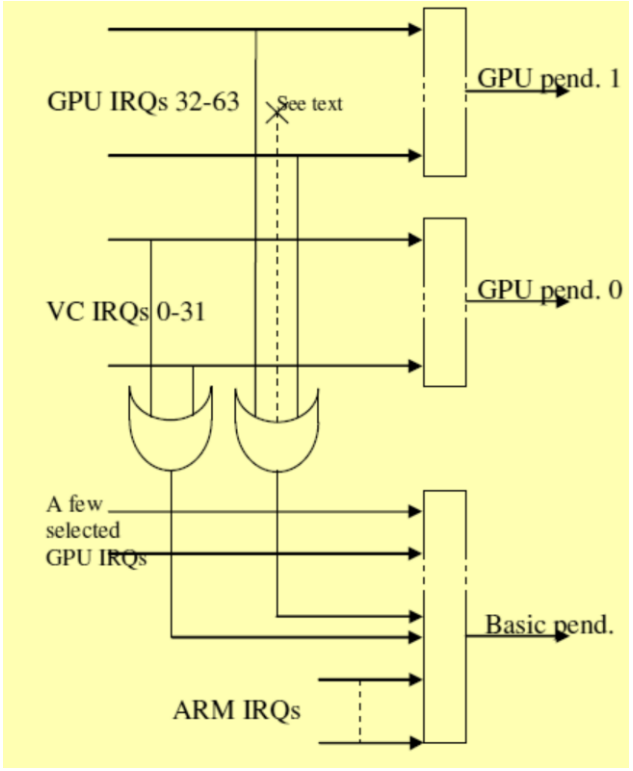
```
Kernel.InstallIrqHandler(Platform.UartEffectiveIrq ,
UartHandler0);
```

19

cf. BCM2835 ARM Peripherals document, Chapter 7,
basic pending register (p. 110 + 113)

IRQ Trap Handler

```
PROCEDURE {INTERRUPT, PCOFFSET=4} IRQTrap;
VAR i, j, spsr: SIZE;  basicPending, pending1, pending2: SET32;
BEGIN
  SYSTEM.STPSR( 1, spsr );   (* store SPSR *)
  (* read pending bits *)
  ...
  (* disable corresponding device interrupts *)
  ...
  (* cf BCM2835 Manual, Section 7.5 *)
  (* process pending bits and call irq handler*)
  ...
  SYSTEM.LDPSR( 1, spsr ); (* SPSR := old *)
END IRQTrap;
```



DataAbort handler

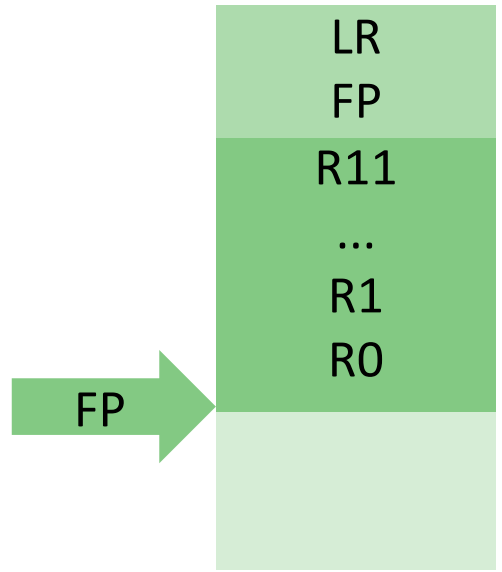
```
(*page fault*)
PROCEDURE {INTERRUPT, PCOFFSET=8} DataAbort;
VAR lnk, fp: LONGINT;
BEGIN
  (* The location that trapped was lnk - 8 *)
  lnk := SYSTEM.LNK - 8;
  fp := SYSTEM.FP;
  IF trapHandler # NIL THEN
    trapHandler(Platform.DataAbort, lnk, fp)
  ELSE
   (* diagnostics output and halt *)
  END
END DataAbort;
```

SWITrap handler

```
PROCEDURE {INTERRUPT, PCOFFSET=0} SWITrap;
(* software interrupt (e.g. failed ASSERT) *)
  VAR lnk, fp: LONGINT;
BEGIN
  (* The location that trapped was lnk - 4 *)
  lnk := SYSTEM.LNK - 4;
  fp := SYSTEM.FP;
  IF trapHandler # NIL THEN
    trapHandler(Platform.SWI, lnk, fp) (* stack trace *)
  END
END SWITrap;
```

SWITrap handler

```
PROCEDURE {INTERRUPT, PCOFFSET=0} SWITrap;
(* software interrupt (e.g. failed ASSERT) *)
  VAR lnk, fp: LONGINT;
BEGIN
  (* The location that trapped was lnk - 4 *)
  lnk := SYSTEM.LNK - 4;
  fp := SYSTEM.FP;
  IF trapHandler # NIL THEN
    trapHandler(Platform.SWI, lnk, fp) (* stack trace *)
  END
END SWITrap;
```



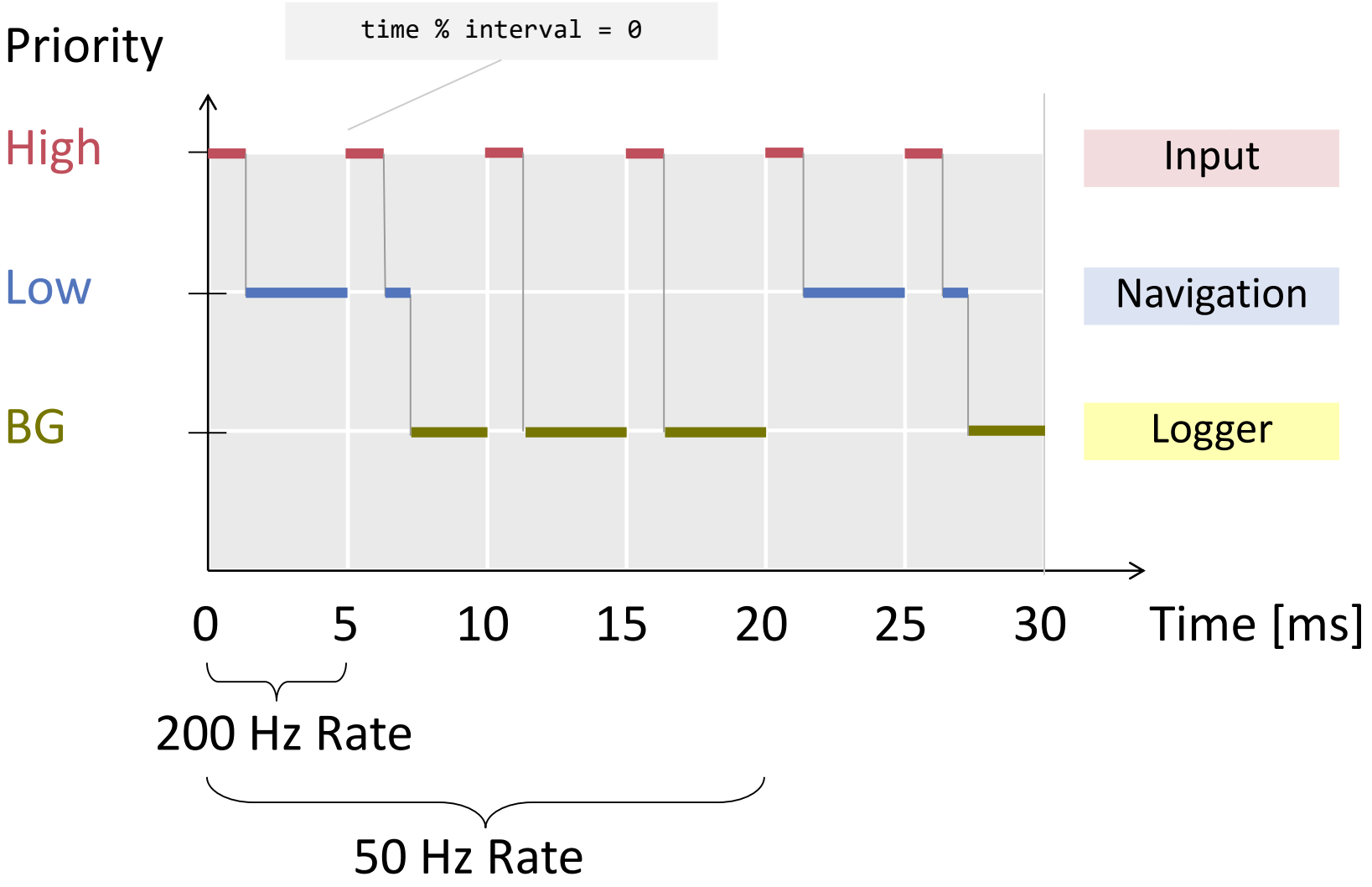
All registers are saved during entering the trap. Get the original FP (reg12 – not banked) from the local stack and traverse the stack.

1.4. TASK SCHEDULING

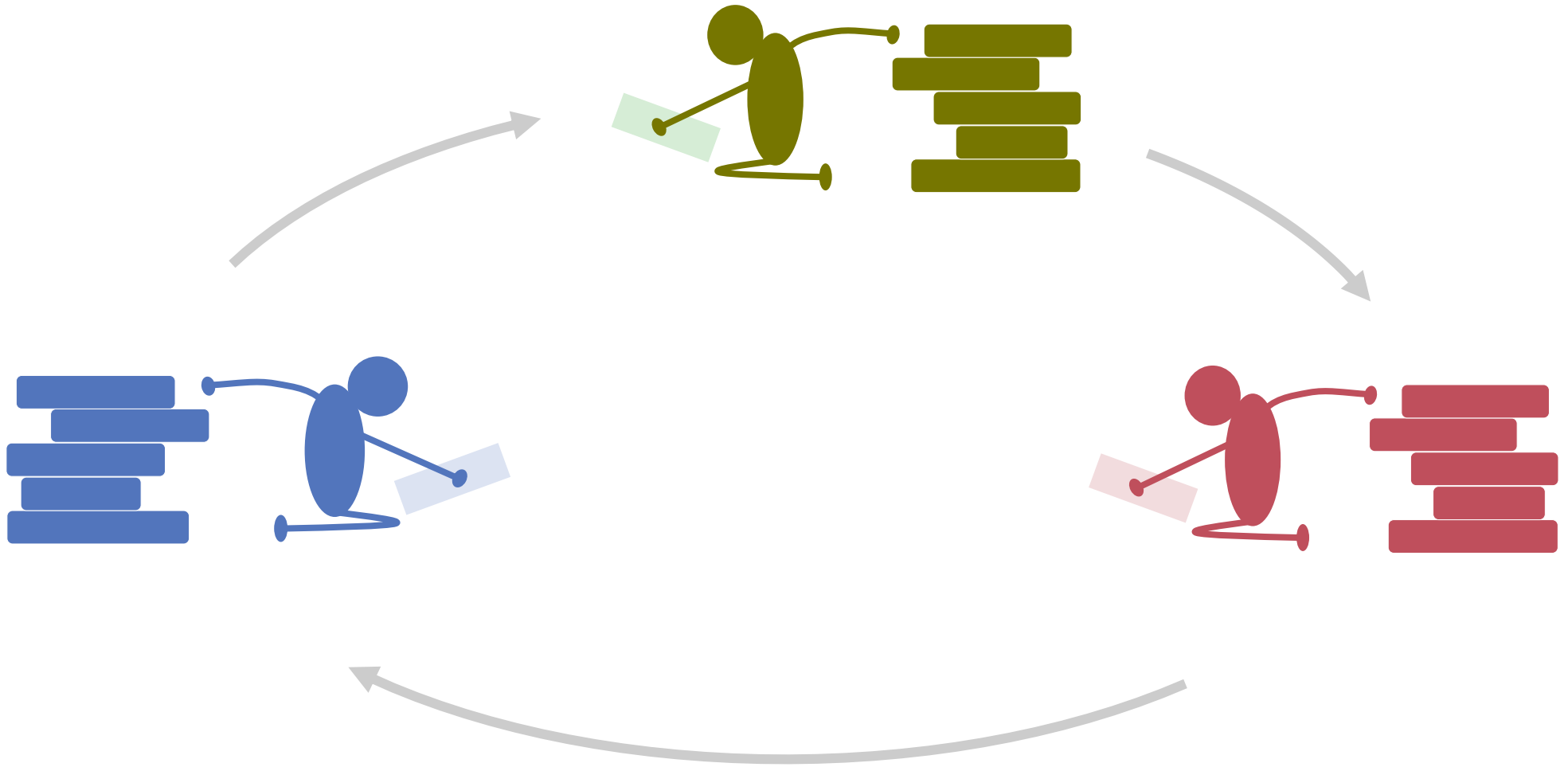
Scheduling Strategy

- Task types
  - High priority synchronous tasks (scheduled each 5 ms)
  - Low priority synchronous tasks (scheduled each 20 ms)
  - Background tasks
- Rules of preemption
  - High priority tasks preempt all others
  - Low priority tasks preempt background tasks
  - Background tasks don‘t preempt

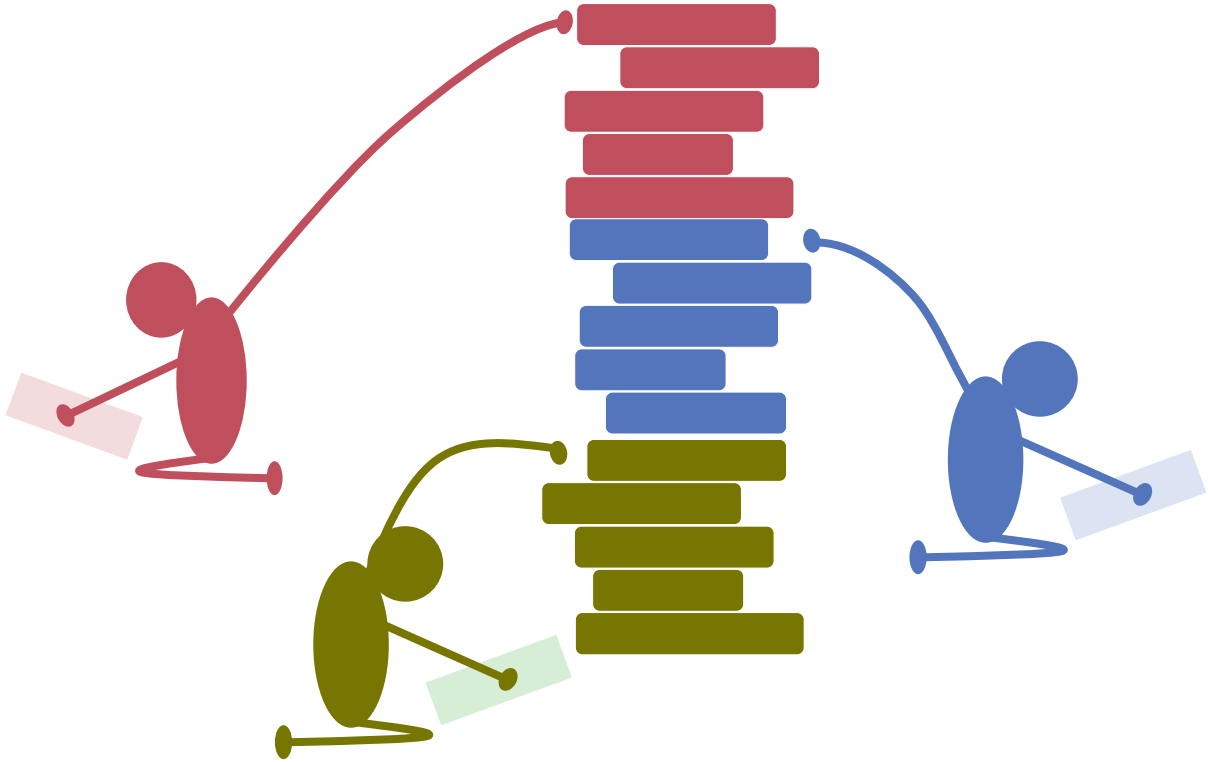
Scheduling Example



A Stack for each Process?

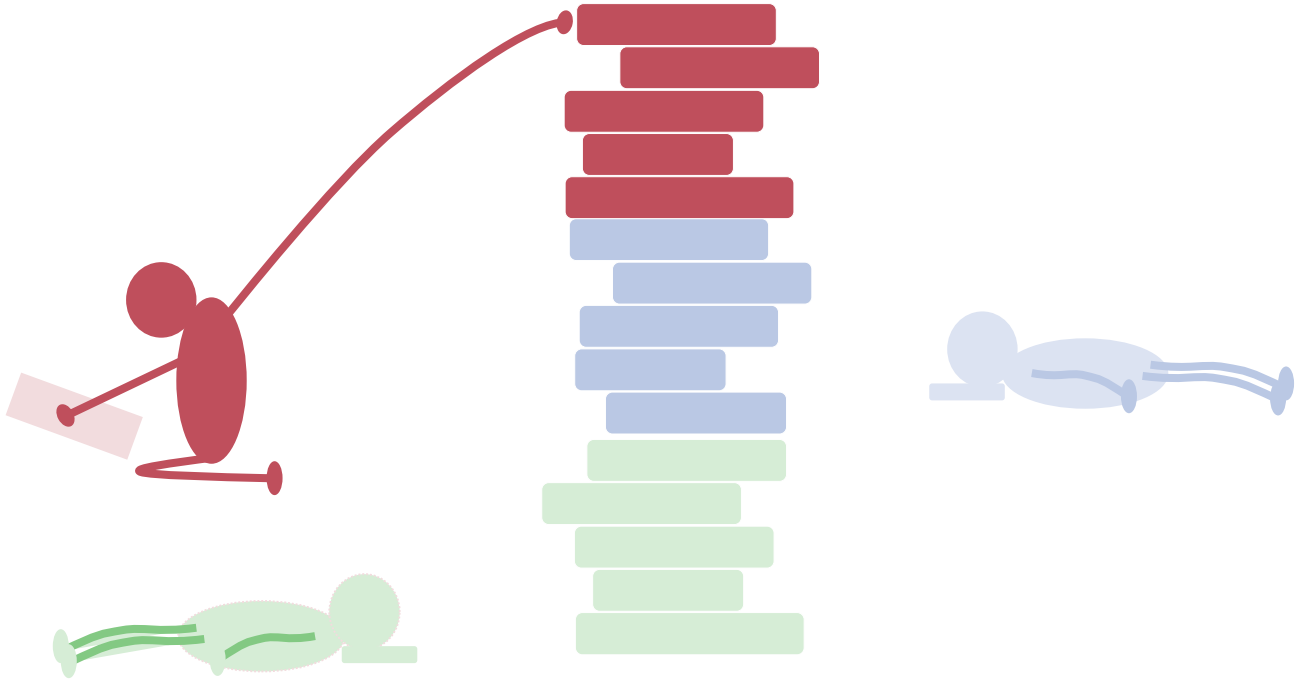


One Stack for All?

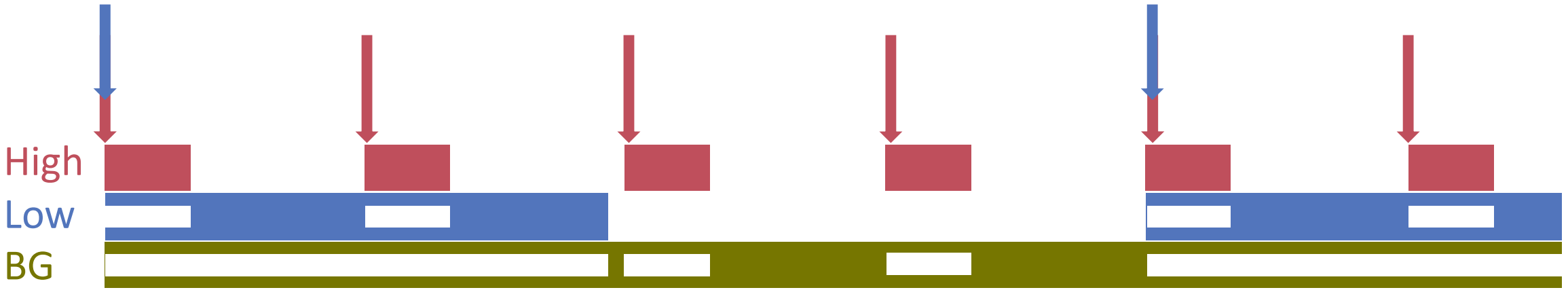


When, How ?

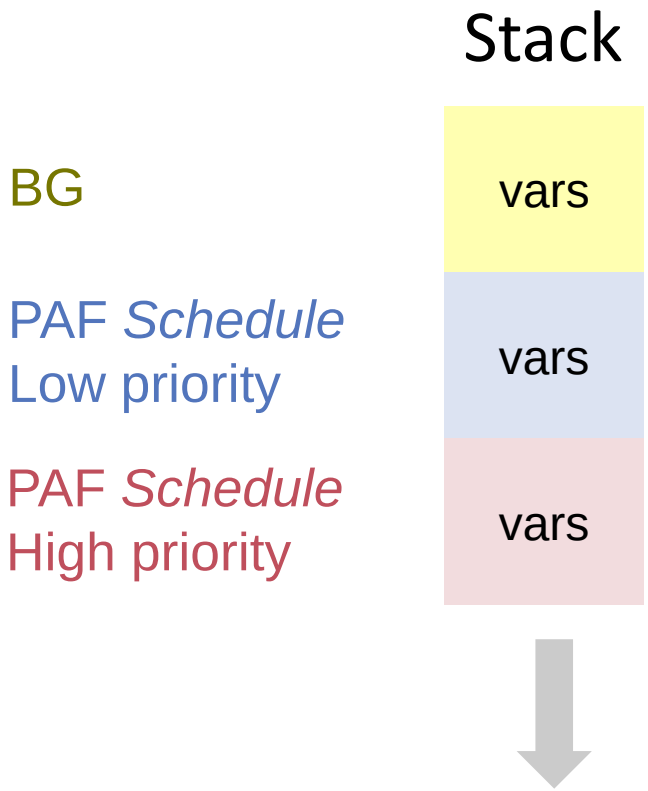
Run to completion!



Preemption



Stack Organisation



Where is the process context ?

Tasks

- Descriptors for *asynchronous* (background) tasks

```
Task*= POINTER TO TaskDesc;
TaskDesc* = RECORD
   next: Task;
   proc: TaskCode;
   name: ARRAY 32 OF CHAR;
END;
```

PROCEDURE (me: Task)

- Descriptors for *synchronous* (periodic) tasks

```
PeriodicTaskDesc* = RECORD (TaskDesc)
    interval: LONGINT;
    subPriority: LONGINT;
    nextTime: LONGINT;
END;
```

Scheduler

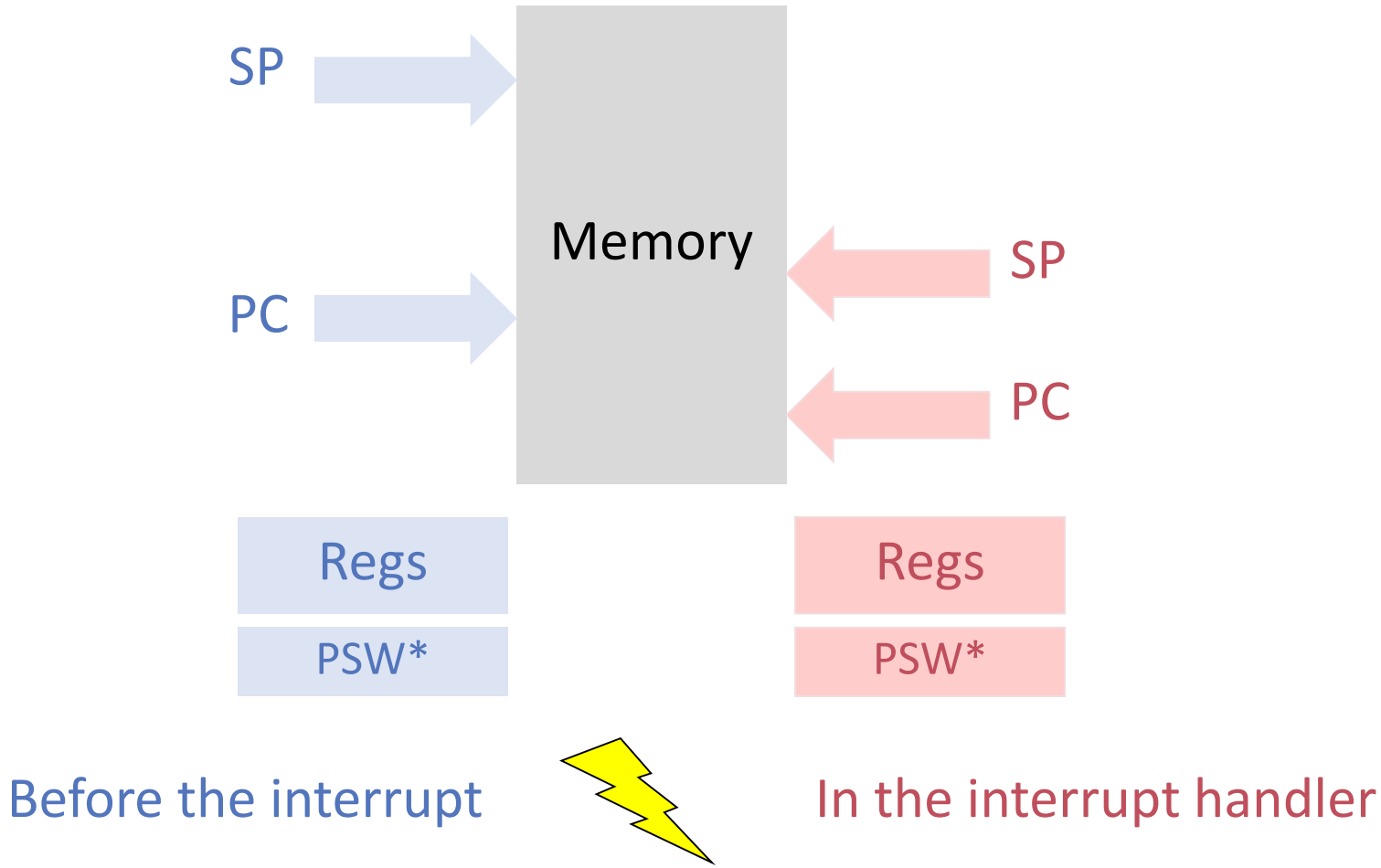
- Recursive interrupt procedure

```
Prolog (Interrupts masked)
Scheduling (Interrupts allowed)
Epilog(Interrupts masked)
```

- Must be reentrant
  - Register values on stack
  - Private variables

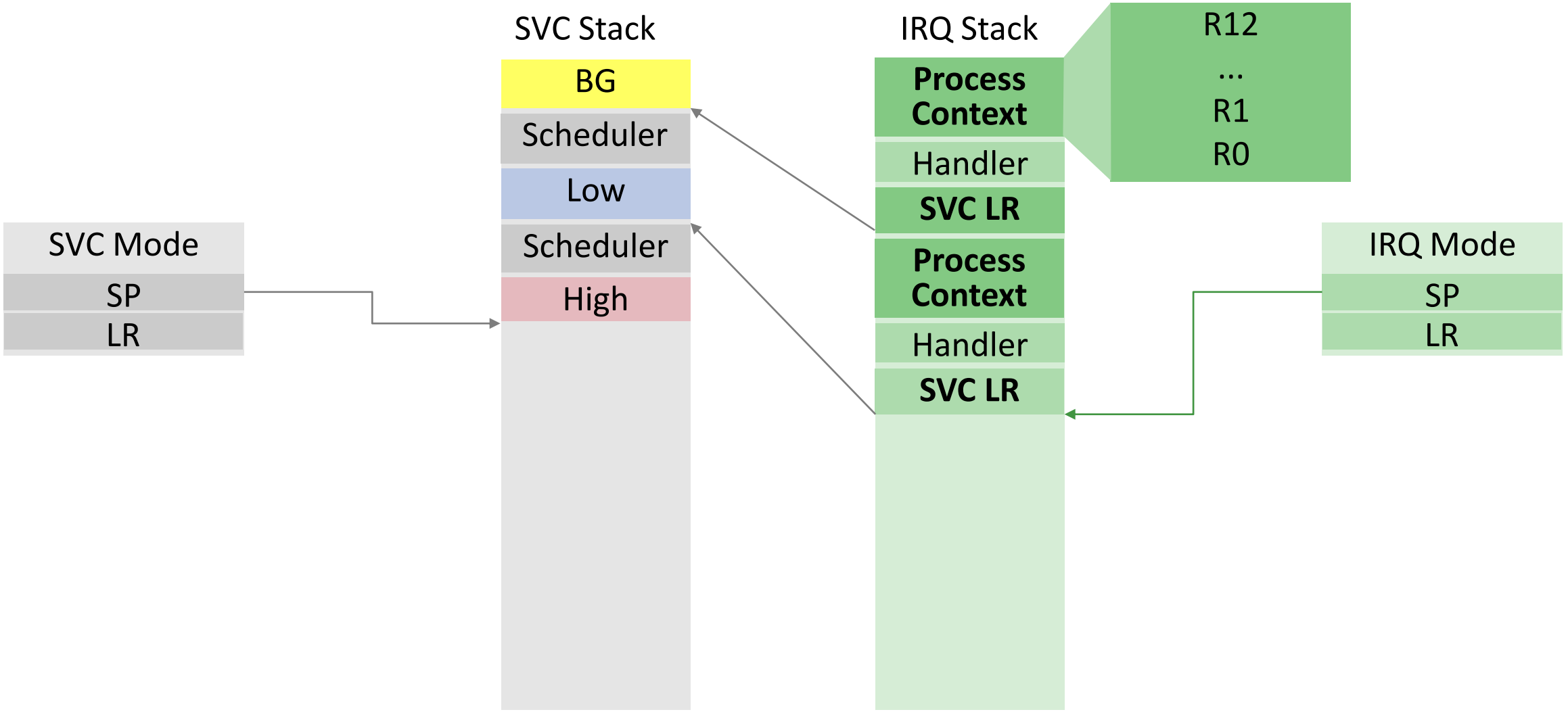
- Assume that Interval(low) is a multiple of Interval(high)

Context change, schematic



*Processor Status Word

Process Context

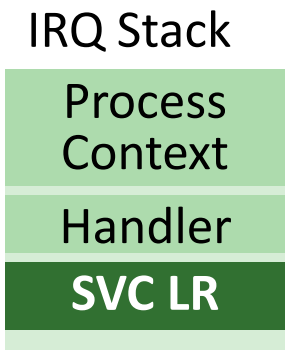


Some tricks required …

Kernel.TimerIrqHandler

```
VAR lr: INTEGER;
BEGIN
   INC( timer, Platform.UNIT );
   IF timerHandler # NIL THEN
      SYSTEM.LDPSR( 0, SVCMode + IRQDisabled );
      globalLR := SYSTEM.LNK();
      SYSTEM.LDPSR( 0, IRQMode + IRQDisabled );
      lr := globalLR;
      SYSTEM.LDPSR( 0, Platform.SVCMode );
      timerHandler;
      SYSTEM.LDPSR( 0, IRQMode + IRQDisabled );
      globalLR := lr;
      SYSTEM.LDPSR( 0, SVCMode + IRQDisabled );
      SYSTEM.SETLNK(globalLR);
      SYSTEM.LDPSR( 0, IRQMode + IRQDisabled );
END;
```

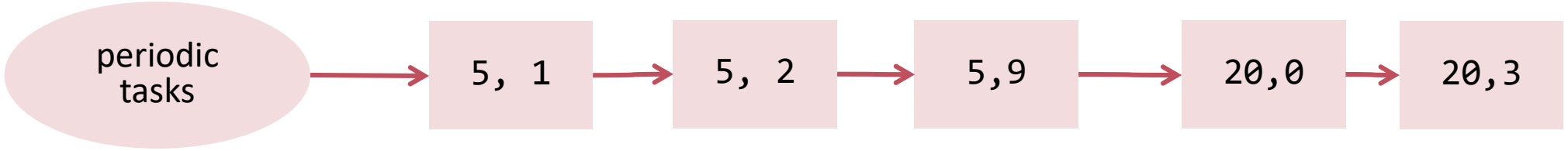
Switch to SVC mode, no IRQs



Scheduler Code

Assumptions:

- linked list stores tasks sorted by period / priority
- tasks run to completion within given period



Rate Monotonic Scheduling

Minos.Scheduler

```
currentTime := Kernel.GetTime();
current := periodicTasks;
WHILE current # NIL DO
   IF currentTime MOD current.interval = 0 THEN
      current.proc( current )
   END;
   current := current.next(PeriodicTask);
END;
```